Systems/C C Library
Version 2.30

Systems/C C Library
Version 2.30

Contents

How to use this book

This book describes the Systems/C C run-time library.

The Systems/C run-time library provides functions that implement most of the ANSI-C standard library on OS/390 and z/OS. Using the Systems/C library, you can build stand-alone programs that run on OS/390 and z/OS.

For information on the Systems/C C compiler, refer to the *Systems/C C Compiler* manual.

Systems/C also includes several utility programs used to manage the process of building OS/390 and z/OS programs. For more information regarding these utilities, see the *Systems/C Utilities* manual.

For further information, contact Dignus, LLC at (919) 676-0847, or visit http://www.dignus.com.

Using the Systems/C C library

This section describes how to link with the Systems/C C library and how to execute the resulting programs.

Linking with the Systems/C C run-time library on OS/390 and z/OS

Once the compiler generated assembly source has been assembled, the disparate objects can be linked into an executable load module. If the Systems/ASM assembler was used to cross-assemble the assembly source, the object decks should be transferred to OS/390 via FTP or some other binary-mode transfer mechanism.

Systems/C contains two versions of the Systems/C library - the RENT version for generating re-entrant programs and the non-rent version for generating non-re-entrant programs.

If the source were compiled with the *–frent* option, the RENT library should be employed to produce a re-entrant load module. This will require using the Systems/C pre-linker **PLINK** during the link step.

If no source was compiled with the *–frent* option, then the non-rent library should be used. In that case, it is not necessary to use the Systems/C pre-linker, **PLINK**.

A note on re-entrant (RENT) programs

Re-entrant (RENT) programs are programs which can safely be linked with the RENT option applied to the IBM LINKER, and can be placed in the OS/390 LINKLST, etc... They are, generally speaking, programs which do not modify their own loaded sections, but instead allocate memory to contain program variables at program start-up.

When a C source file is compiled with the *–frent* option, the compiler will place all of the `extern` and `static` variables in the pseudo-register vector, the **PRV**. These variables are referred to by `Q-CON` references in the generated assembly source.

The IBM linker gathers all of the `Q-CON` references together allocating an entry for each in the **PRV**.

The Systems/C library, at start-up, allocates the appropriate space for the **PRV**, and retains a pointer to the **PRV** at a known location.

At run-time, a reference to a variable in the **PRV** uses the **PRV** pointer and the value the linker has substituted for the `Q-CON`, adding them together to produce the run-time offset for the variable.

An issue arises because of variable initialization allowed by the ANSI C standard. For example, the address of a variable in the **PRV** isn't known until run-time, when the **PRV** is allocated, but is a valid file-scoped initialization value.

Because of this, the Systems/C compiler, **DCC** produces run-time initialization scripts which the Systems/C library processes at program start up, after the **PRV** has been allocated. It is the job of the Systems/C pre-linker, **PLINK**, to locate the start of these scripts in each object and gather them together. **PLINK** then places a list of these at the end of the resulting object, in a known section. The run-time library walks the list, interpreting the scripts it finds.

Thus, RENT programs must be processed with the Systems/C pre-linker, **PLINK**, to ensure proper run-time initialization of variables located in the **PRV**.

Using PLINK

PLINK gathers the input objects together, performing AUTOCALL resolution where appropriate, producing a single file which can then be processed by the IBM BINDER or older IEWL linker.

As **PLINK** gathers objects, it examines the defined symbols, looking for a Systems/C initialization script section and other object file processing that may need to be performed.

The output of **PLINK** is then processed by the IBM BINDER to produce the executable load module.

For detailed information on **PLINK**, see the **PLINK** section in the *Systems/C Utilities* manual.

On cross-hosted platforms (Windows and UNIX), **PLINK** is typically executed with the object files listed on the command line; and a *–S* option or library names to locate any required library objects.

For example, on a Windows platform the command:

```
plink "-SC:\sysc\lib\objs_rent\&M" prog.obj
```

will read the initial input file, `prog.obj` and examine the `C:\sysc\lib\objs_rent` directory for any AUTOCALL references. Because no -o option was specified, the resulting object file is writting to the file `p.out`.

This command, on UNIX platforms:

```
plink t1.obj t2.obj libone.a -L../mylibs -ltwo
```

will read the two primary input objects `t1.obj` and `t2.obj`. It will try and resolve references from the **DAR** archive `libone.a` and then the second **DAR** archive `../mylibs/libtwo.a`

On OS/390 or z/OS, **PLINK** operates similar to the IBM pre-linker. The resulting gathered object is written to the file `//DDN:SYSMOD` unless otherwise specified. **PLINK** has a default library template of `-S//DDN:SYSLIB(%M)` which causes it to look in the `SYSLIB` PDS for autocall references. Other input objects, `-S` library templates or **DAR** archives may be added in the PARMS option on the **PLINK** step. **PLINK** reads the file `//DDN:SYSIN` as the initial input file. Typically, this file contains `INCLUDE` cards to include the primary objects for the program. Other primary input files may be included in the PARMS for **PLINK**. For example, the following JCL reads the object `INDD(PROG)` and uses `DIGNUS.LIBCR.OBJ` as the autocall library:

```
//PLINK EXEC PGM=PLINK
//STDERR DD SYSOUT=A
//STDOUT DD SYSOUT=A
//SYSLIB DD DSN=DIGNUS.LIBCR.OBJ,DISP=SHR
//INDD DD DSN=mypds,DISP=SHR
//SYSIN DD *
 INCLUDE INDD(PROG)
//SYSMOD DD DSN=myoutput.obj,DISP=NEW
```

Note that the `STDERR` and `STDOUT` DDs were specified for **PLINK**'s message output. Also, the `ARLIBRARY` control card could have been used to add additional **DAR** archive files for resolving external references.

For more detailed information regarding **PLINK** and the other Systems/C utilities, see the *Systems/C Utilities* manual.

Linking under the OpenEdition shell

Systems/C programs can be linked under the OpenEdition shell; to create load modules that reside in the Hierarchical File System (HFS).

To create an HFS load-module, the output from **PLINK** can be linked using the OpenEdition `cc` command. The `-e //` option should be added the `cc` command to indicate that the entry-point is not the default Language Environment entry point expected by `cc`. The Systems/C runtime library will specify its own entry-point.

For example, to pre-link and link the object `myfunc.o` and produce the `HFS` load-module `myprog` under the OpenEdition shell (assuming `/usr/local/dignus` is the installation location), simply run **PLINK**:

```
plink -omyprog.o myfunc.o "-S/usr/local/dignus/objs_rent/&m"
```

then use the OpenEdition `cc` command:

```
cc -e // -omyprog myprog.o
```

to produce the `myprog` load-module. `myprog` can then be invoked as any other OpenEdition program.

Other useful utilities

Systems/C provides other useful utilities. More details and examples of their use can be found in the *Systems/C Utilities* manual.

DAR — the Systems/C Archive utility

The Systems/C archive utility, **DAR**, creates and maintains groups of files combined into an archive. Once an archive has been created, new files can be added and existing files can be extracted, deleted or replaced. Files gathered together with DAR can be used to resolve AUTOCALLed references from **PLINK**.

DRANLIB — the Systems/C Archive index utility

DRANLIB is used to index a Systems/C archive to allow for AUTOCALL references to longer names, or to names which are not dependent on the archive member name. **DRANLIB** will create a `__SYMDEF` member in the Systems/C archive which **PLINK** will consult when looking for symbolic resolutions.

GOFF2XSD — Convert GOFF format objects to XSD format

GOFF2XSD is used to convert GOFF format objects to XSD format. Typically, GOFF format objects are created by the IBM HLASM assembler when the `XOBJECT` option is enabled. The **PLINK** linker can read GOFF format natively. This utility is no longer required for using **PLINK** and is provided only for back-level support.

DCCPC — the Systems/C CICS Command Processor

DCCPC is used to convert `EXEC CICS` commands in C source into plain C code for compilation. It is especially useful in cross environments where IBM's translators cannot be used.

D2S — the Systems/C DSECT to struct conversion tool

D2S extracts assembly DSECTs from assembler-generated ADATA information and generates C-style `struct` definitions. This is intended to allow C code to work seamlessly with data structures from your assembly code.

Linking programs on OS/390 and z/OS

Before execution, programs must be prepared, optionally using the Systems/C prelinker, **PLINK**, and then linked using either **PLINK** or the IBM LINKER or BINDER.

Systems/C provides two versions of the Systems/C C library, one for RENT programs and one for non-RENT programs. If you are using the Systems/C library, it is important to link with the appropriate version. If any source programs reference variables found in the Systems/C library (e.g. `errno`) and that program was compiled with the *–frent* option, then the re-entrant version of the Systems/C library should be used. Using the incorrect version of the library will cause strange run-time errors. The installation instructions for your particular host platform will detail where to find the correct Systems/C library. Normally the Systems/C library is specified as the last library to use for AUTOCALL resolution in the **PLINK** step. Furthermore, **PLINK** must be used for re-entrant programs that use the Systems/C library or to take advantage of **DAR** archive libraries for external reference resolution.

In the following example JCL, there are three objects to link together to form the resulting executable, *MAIN*, *SUB1*, and *SUB2*, representing a main module and two supporting sub-modules. These are found in the PDS *MY.PDS.OBJ.* The resulting executable is written to *MY.PDS.LOAD(MPROG).*

```
//LINK JOB
//PLINK EXEC PGM=PLINK,REGION=2048K
//STEPLIB DD DSN=DIGNUS.LOAD,DISP=SHR
//STDOUT  DD SYSOUT=*
//STDERR  DD SYSOUT=*
//SYSLIB  DD DSN=DIGNUS.LIBCR.OBJ,DISP=SHR
//SYSMOD  DD DSN=&&PLKDD,UNIT=VIO,DISP=(NEW,PASS),
//        SPACE=(32000,(30,30)),
```

```
//        DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//INDD    DD DSN=MYPDS.OBJ,DISP=SHR
//SYSIN   DD *
  INCLUDE INDD(MAIN)
  INCLUDE INDD(SUB1)
  INCLUDE INDD(SUB2)
//STDIN   DD *
//LINK EXEC PGM=IEWL,REGION=2M,PARM=('LIST',
//  'MAP,XREF,LET',
//  'ALIASES=NO,UPCASE=NO,MSGLEVEL=4,EDIT=YES')
//SYSPRINT DD SYSOUT=*
//SYSUT1   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
```

First, the Systems/C pre-linker, **PLINK** is invoked, specifying the inclusion of the three object modules and the Systems/C C reentrant library. This step could have been performed on a cross-platform host, running **PLINK** there. Then the IBM BINDER is invoked for final linking and generation of the resulting load module.

Executing programs

Once a program has been successfully linked, it is a typical OS/390 or zOS load module and may be executed via JCL, TSO CALL command or, via the OpenEdition `exec()` linkage.

By default, the Systems/C library contains no modules that are loaded during program execution, meaning it is "all-resident." As such, there are no run-time library concerns, and no particular modules which must be present in a `STEPLIB` concatendation.

For traditional (non-POSIX) programs, the Systems/C C library's default behavior is to open file descriptors descriptors #0, #1 and #2 using the names `//DDN:STDIN`, `//DDN:STDOUT` and `//DDN:STDERR`. Thus, the DD-names `STDIN`, `STDOUT` and `STDERR` must be properly allocated. The open(2) description contains more information regarding file descriptors and file I/O.

For standard Systems/C library uses, where the Direct-CALL feature is not employed, arguments specified in the TSO CALL command, or via the PARM option of JCL are processed and presented to the program in the `argv[]` array passed to the `main()` function. The Systems/C library uses a comma (,) as the argument delimiter, similar to other option processes used in TSO and batch environments.

For example, if the resulting program was named "`PROG`" in the `MY.PROGS` PDS, the following JCL would execute the program, passing the argument strings "`arg1`", "`arg2`", and "`arg3`" in the `argv[]` array.

```
    ...
//PROG EXEC PGM=PROG,PARM="arg1,arg2,arg3"
//STEPLIB DD DSN=MY.PROGS,DISP=SHR
//STDOUT  DD SYSOUT=*
//STDERR  DD SYSOUT=*
```

Note the definition of the `STDOUT` and `STDERR` DD statements to provide the necessary output path for the Systems/C library.

For programs invoked via the `exec()` service (POSIX programs), the first 3 file descriptors are inherited from the parent process. Also, the arguments are processed by the Unix Systems Services environment and are presented to the program in the typical UNIX style.

Systems/C C Library Features

The Systems/C C Library contains several features to aid in the development of OS/390 and z/OS programs.

This include Systems/C C compiler support for in-line expansion of certain commonly used C functions, support for using Systems/C programs in almost any runtime environment, and enhanced support for z/Architecture 64-bit programs.

Special "built-in" implementations for common C library functions.

The Systems/C compiler, **DCC**, provides built-in implementations for some of the more common C library functions. Built-in functions are used when the Systems/C C library header file `<string.h>` system header file is included. The following functions have built-in implementations:

memcpy()
memset()
memcmp()
memchr()
strcpy()
strlen()
strcmp()
strcat()
strchr()
strncat()
strncmp()
strncpy()
strrchr()

`#include <string.h>` to take advantage of the built-in versions of these functions.

Using the Systems/C Direct-CALL interface

The Systems/C C library is implemented using the Systems/C entry and exit macros which assume a Systems/C environment is extent at run time.

The Systems/C environment includes items such as the local stack frame used for automatic variables in your C code, the Systems/C run-time heap, I/O data blocks, etc...

Thus, in order to call a Systems/C function which uses the Systems/C entry and exit linkage macros, this environment must be established and accessible.

For typical Systems/C programs, where the initial function is a C `main()` function; the Systems/C library handles creation of this environment.

However, there are circumstances where there is no Systems/C `main()` function. For example, calling Systems/C routines from COBOL or directly from assembler source in a system exit.

For this situation, Systems/C provides the Direct-CALL (DCALL) interface, where a Systems/C function can be directly called from any environment. This interface can be employed to either automatically create and destroy a Systems/C environment, or to create and re-use, then destroy a Systems/C environment.

Automatic Creation/Destruction of the Systems/C environment

To use the Direct-CALL to automatically create an destroy a Systems/C environment, particular Systems/C functions are indicated as being "directly called". These functions establish a Systems/c environment, so that normal System/C library functions can be employed until the "directly called" function has ended. At the end of execution of the "directly called" function, the System/C library environment is destroyed and all resources are returned to the operating system.

To indicate that a particular function is to be "directly called", the `DCALL=YES` keyword is added to the functions prologue macro with a `#pragma prolkey` control statement:

```
#pragma prolkey(funcname,"DCALL=YES")
```

where *funcname* is the name of the "directly called" function. This causes the prologue generated for the named function to establish the System/C environment, and destroy it on return from the function.

For more information regarding `#pragma prolkey`, see the *Systems/C C Compiler* manual.

For example, if MYFUNC was to be "directly called" from assembler language; and MYFUNC would further use the Systems/C library you might have in your assembler source:

```
   L    2,=F'1'
   ST   2,PARMS
   L    2,=F'2'
   ST   2,PARMS+4
   LA   1,PARMS
   L    15,=V(MYFUNC)
   BALR 14,15           Call 'MYFUNC'
*   The return value from MYFUNC is in R15.
        ...
   PARMS DS  2F
```

which invokes MYFUNC with a standard parameter list, passing the values #1 and #2.

For the C definition of MYFUNC:

```
#pragma prolkey(MYFUNC,"DCALL=YES")

int
MYFUNC(int one, int two)
{
   printf("In MYFUNC - arg #1 is %d\n", one);
   printf("            arg #2 is %d\n", two);

    ...
   return (0); /* return 0 to the caller */
}
```

In this example, when MYFUNC is invoked, a Systems/C environment will be created, and MYFUNC can then invoke Systems/C library functions (e.g. printf() above.)

On return from MYFUNC - the Systems/C environment will be destroyed and any resources will be returned to the operating system. The return value from MYFUNC will be in `R15` as it is declared to be a function returning an `int`.

Notice also that two parameters were passed to MYFUNC in this example. The Systems/C linkage follows standard linkage conventions, so Systems/C "direct call" functions interoperate well with most environments. When invoking "direct call" Systems/C functions from some high-level languages, such as PL/I and COBOL, be sure to declare any parameters as pointers to their data types, as these other languages pass parameters by-reference, instead of the C by-value approach.

Register uses across DCALL executions

When a Dignus environment is created and destroyed, the Dignus runtime saves and restores the registers as per normal linkage rules.

In the 31-bit runtime environment, the Dignus library assumes R13 points to a 72-byte save area and will save and restore R2 through R14.

In the 64-bit runtime environment, if the DCALL creation routine is invoked in AMODE 31, then the runtime only assumes R13 points to a 72-byte save area. However, the full 64-bit registers values for R2 through R14 are saved and restored. If the creation routine is invoked in AMODE 64, then the runtime assumes R13 points to a 144-byte save area, and the full 64-bit values of R2 through R14 are saved and restored.

The runtime does not guarantee the preservation of R0, R1 or R15 across a DCALL function call.

Creating, re-using and destroying a Systems/C environment

The Direct-CALL interface can also be used to create a Systems/C environment which is not destroyed. The created environment may be used multiple times until it is explicitly destroyed. This can save run-time cost, as the creation of a Systems/C environment involves some overhead. Reusing a previously created environment avoids the creation problem for functions called many times.

Creating a Systems/C environment

To create a Systems/C environment, the DCALL prologue key is altered to indicate that the environment should be created when the named function is invoked, but not destroyed when the function returns. To indicate this, use

```
DCALL=ALLOCATE
```

on the prologue key statement. On return from the C function, an environment pointer is returned in general register one (R1). This value should be saved, and may be re-used on subsequent Systems/C function calls to re-use the created environment. Note that the creation function can call other Systems/C functions, alter global data in the environment, etc... providing for a nice location to accomplish any particular initialization that may be needed.

For example:

```
#pragma prolkey(create,"DCALL=ALLOCATE")
void
create()
{
   /* This function is called to create a */
   /* Systems/C environment. */

   /* On return, the created environment address */
   /* is returned in R1. */
}
```

Another approach to saving the environment pointer for R1 is to use the **__dcall_env()** function. **__dcall_env()** returns the same environment pointer that will be returned in the R1 register.

Thus, **__dcall_env()** can be used to save the environment pointer in a parameter passed to the

```
DCALL=ALLOCATE
```

function. For example:

```
#include <machine/dcall.h>

void
create(void **env_ptr)
{
    /* Set the *env_ptr value to the created environment */
    /* pointer*/
       *env_ptr = __dcall_env();
}
```

Using this approach, the environment pointer can be saved in a parameter which can then later be passed to other functions which need it.

Reusing a created environment

To reuse a previously created Systems/C environment, a different DCALL prologue key is provided, indicating that an environment should not be established, but can be found in the supplied location. To indicate this use:

```
DCALL=SUPPLIED
```

on the prologue key statement. When this is specified, the Systems/C library will use the environment address specified in register zero (R0). Before invoking the function, load register 0 with the address previously returned in R1 by a DCALL=ALLOCATE function call.

For example:

```
#pragma prolkey(func1,"DCALL=SUPPLIED")
func1()
{
   /* calls to func1() assume a Systems/C */
   /* environment is passed in R0 */
   printf("in func1\n");
}

#pragma prolkey(func2,"DCALL=SUPPLIED")
func2()
{
   /* calls to func2() assume a Systems/C */
   /* environment is passed in R0 */
   printf("in func2\n");
}
```

A `DCALL=SUPPLIED` function cannot invoke another `DCALL=SUPPLIED` function using the same environment address. That is, while the environment is being used by a `DCALL=SUPPLIED` function, a separately invoked `DCALL=SUPPLIED` function will restart the stack point and corrupt the program. If two `DCALL=SUPPLIED` interfaces are provided, and need to share function code, then a 3rd non-DCALL function is the best approach.

The Systems/C library also provides for an exit to locate the desired environment when a function is called, the FINDENV=*exitname* option for DCALL function. FINDENV specifies an entry point which will be invoked for DCALL=SUPPLIED function calls. When FINDENV is present, the Systems/C library will invoke the exit before any other processing. The exit should return with a

```
L R15,=V(CRT9A)
BR 15
```

having located the Systems/C environment and placing that address into register 0 (R0.)

The FINDENV exit must preserve register nine through thirteen (R9-R13) and does not have an available save area.

For example,

```
#pragma prolkey(func3,"DCALL=SUPPLIED,FINDENV=FINDME")
```

When `func3()` is invoked, the exit `FINDME` will be driven to load the Systems/C environment pointer into R0.

One example of using the FINDENV option is to pass the environment pointer in the parameter block. For example, if the SUPPLIED function was:

```
#progma prolkey (SUPPEX, "DCALL=SUPPLIED,FINDENV=@@FNDENV")
SUPPEX(void **env_ptr, int *parm1)
{
  /* env_ptr is used by the @@FNDENV assembler piece */
  /* to set R0 to the environment pointer. */
}
```

then the @@FNDENV assembly function might be:

```
@@FNDENV CSECT
@@FNDENV AMODE ANY
@@FNDENV RMODE ANY
         USING @@FNDENV,15
         L  2,0(0,1)
         L  0,0(0,2)  Get environment ptr into R0
         L  15,=V(@CRT9A)
         BR 15
         LTORG
         END
```

Note that if the name specified in `FINDENV` is an external label, not present in the current compilation, then it should be made visible to the assembler via an `EXTRN` statement. For example:

```
__asm {  EXTRN FARAWAY }
#pragma prolkey(func4,"DCALL=SUPPLIED,FINDENV=FARAWAY")
```

The `DCCPRLG` macro will reference the `FINDENV` specified label via an address-constant (A-CON), and the label needs to be appropriately defined for proper reference. Note that the `EXTRN` statement should only appear once in the generated assembly. Multiple `EXTRN` statements for the same label are flagged as errors by the assembler.

Reusing an existing PRV

For a DCALL=ALLOCATE or DCALL=YES function, it is possible to indicate that the previously established PRV should be employed instead of re-allocating the PRV and performing global initialization functions.

Normally, when a DCALL=ALLOCATE or DCALL=YES function is invoked, the runtime will acquire space for the PRV and run global initialization functions (including global constructor functions.)

However, there are instances where only a new stack frame is required, but no PRV should be allocated. For example; to implement multi-threading via the ATTACH macro. The environment requires its own stack, but wants to share the global state. This only applies to re-entrant data as non-re-entrant data is in the load module proper and thus would be shared amongst all environments.

In this case, the PRV=0 option on the DCALL=ALLOCATE or DCALL=YES can be used.

When PRV=0 is specified, the Dignus runtime will create a new stack frame environment for the environment but will not create or initialize a new PRV and will not invoke global constructor functions. Instead, the PRV is taken from the value found in register zero (`R0`) at the start of the DCALL=ALLOCATE or DCALL=YES function.

The Dignus library function __getprv(2) can be used in the primary environment to retrieve the current PRV value before invoking a PRV=0 function.

When re-using an existing PRV, the environment start up does not reinitialize global or static reentrant variables and does not execute global initializers on start up. On completion, the runtime does not execute global destructors or free the specified PRV. In this fastion, the PRV=0 environment will "share" the global re-entrant variables with the environment specified in `R0`.

Differences from main()

The Systems/C Direct-CALL environment start up does not provide all the same function that a normal main() start up provides.

There is no argument processing in the Direct-CALL environment, the `DCALL` function is simply invoked directly and processes parameters as any other C function. The caller should create a normal parameter list, with R1 pointing at the parameter block.

Furthermore, the Direct-CALL start up does not initialize the `TZ` environment variable. The `TZ` environment variable is used by the **localtime()** function to determine the current timezone and is initialized when a normal `main()` function is invoked.

However, this initialization is operating-system specific and is avoided in the DirectCALL start up.

To have **localtime()** present a locale time zone, the `TZ` environment variable should be set appropriately. See the tzset(3) description for more information on the format of the `TZ` environment variable.

Destroying the Systems/C environment

After the Systems/C environment is no longer needed, it should be destroyed to return resources to the operating system. To destroy an environment created with DCALL=ALLOCATE, use the DCALL=DESTROY prologue key. The address of the environment to destroy should be placed in register 0 (`R0`.) To indicate this use:

```
DCALL=DESTROY
```

on a prologue key pragma. The specified function will be invoked with the given environment, and may invoke any other Systems/C functions. On return from that function, global destructors will be invoked, and the the environment will be destroyed, returning resources to the operating system. After calling the function, the environment address is no longer valid.

For example:

```
#pragma prolkey(destroy,"DCALL=DESTROY")
destroy()
{
   /* perform any clean-up that needs to happen */
   /* On return from this function, the         */
   /* Systems/C environment specified in R0     */
   /* will be destroyed. */
}
```

Note that the Pseudo-Register-Vector (PRV) is created when a DCALL=ALLOCATE function is invoked, and is used when any SUPPLIED or DESTROY functions are subsequently invoked. Thus, all DCALL=ALLOCATE/SUPPLIED/DESTROYed functions that use the same global variables must be in the same bound load module, so they will have the same PRV. That is, environments created with DCALL=ALLOCATE cannot be passed to functions linked in different load modules, unless `__remote` function pointers are employed to switch PRVs.

Saving environment memory by avoiding I/O

By default, when a Systems/C environment is created, with either DCALL=YES or DCALL=ALLOCATE, the Systems/C library initializes the I/O functions so file I/O can occur. This initialization consumes some overhead in both run-time and memory. If the Systems/C functions do not make use of any of the Systems/C I/O facilities, this can be avoided by adding

```
NOSTDIO=1
```

to the DCALL statements in the prologue keys for the library creation. When NOSTDIO=1 is specified, the Systems/C library will not initialize its I/O functions. Any calls to I/O functions in that environment will ABEND, so care must be taken to ensure none exist.

For example:

```
#pragma prolkey(noio,"DCALL=YES,NOSTDIO=1")
noio()
{
   /* This function, or any function it calls, */
   /* does no I/O */
}
```

Systems/C z/Architecture Library

Systems/C supports programs for the z/Architecture system, providing the complete Systems/C library to the z/Architecture environment.

This includes full support for 64-bit addresses, bringing the power of the Systems/C library to this environment.

The Systems/c z/Architecture library uses the LP64 programming model, `long` and pointer data types are 64-bits wide.

z/Architecture library features

The Systems/C library contains extentions to the memory management facilities helpful in a 64-bit programming environment, **__malloc31()**, **__free31()**, **__malloc24()**, **__free24()**. These allow for the management of allocated space that is guaranteed to be 31-bit or 24-bit addressable as appropriate.

z/Architecture programs can use the Systems/C and Systems/C++ `__ptr31` pointer qualifier to define and use 31-bit addresses in z/Architecture mode. For more information regarding the `__ptr31` qualifier, see the *Systems/C C Compiler* manual.

The Systems/C z/Architecture library provides full support for all of the functions in the 31-bit library, including TCP/IP, memory allocation, and file I/O. For many applications, simply recompiling and relinking with the z/Architecture library will enable programs on the new z/Architecture hardware.

The Systems/C z/Architecture library has no restrictions on program data, all data can reside above the 2-gigabyte "bar".

z/Architecture data and code locations

The Systems/C z/Architecture library allows loading of data above the 2-gigabyte "bar". There are no restrictions in the Systems/C z/Architecture library for data to

reside anywhere in particular. The default location for the runtime heap, stack and re-entrant data in the z/Architecture library is above the 2-gigabyte "bar", freeing up lower instructions for.

Currently, the z/OS program loader will not load instruction code above the 2-gigabyte "bar", thus the Systems/C library assumes that program code is located within the first 2-gigabytes of the address space.

Determining addressing mode

The z/Architecture library has no restrictions on the addressing mode. It will operate correctly if the AMODE is 64, 31 or 24.

At program start-up, the z/Architecture library determines the proper addressing mode based on flags present in the definition of the `main()` function. If `main()` was compiled with the *–mlp64* option, and the *–famode* option was not used to specify otherwise, the z/Architecture library will switch to `AMODE=64` before beginning the program.

If the *–famode* option was used to indicate an AMODE other than 64, the z/Architecture library will not change the AMODE to 64.

The *Systems/C C Compiler* manual has more information on the *–mlp64* and *–famode* options.

Linking with the Systems/C z/Architecture Library

To produce z/Architecture programs, the program must be linked with the z/Architecture libraries. The procedure is only slightly different that linking with the non-z/Architecture library.

Systems/C provides a reentrant and non-reentrant z/Architecture libraries. On cross-platform hosts, these objects are in the `objs_rent_z` and `objs_norent_z` directories. On OS/390 and z/OS, these are in the `LIBCRZ` and `LIBCNZ` PDSes. To use the Systems/C z/Architecture library, simply specify these directories/PDSs in place of the non-zArchitecture versions.

For example, JCL to execute the `PLINK` pre-linker with the Systems/C z/Architecture reentrant library would be similar to the following:

```
        ...
//PLINK EXEC PGM=PLINK
//STDERR DD SYSOUT=A
//STDOUT DD SYSOUT=A
//SYSLIB DD DSN=DIGNUS.LIBCRZ.OBJ,DISP=SHR
```

```
//INDD DD DSN=mypds,DISP=SHR
//SYSIN DD *
 INCLUDE INDD(PROG)
//SYSMOD DD DSN=myoutput.obj,DISP=NEW
```

The same command on a UNIX or Windows platform might be:

```
plink -omyoutput.obj prog.obj "-SC:\sysc\objs_rent_z\&M"
```

assuming Systems/C was installed in the `C:\sysc` directory.

z/Architecture and OpenEdition services

All of the OpenEdition (POSIX) functions available to 31-bit programs operate with the z/Architecture library. The Systems/C library uses the 64-bit z/OS interfaces for this, and thus all pointers will be 64-bits in size.

The 64-bit z/OS interfaces were only made available after z/OS 1.5, and thus the z/Architecture library requires z/OS 1.5 or later for OpenEdition services.

Direct-CALL extensions

Systems/C Direct-CALL programs linked with the z/Architecture library can be invoked from a 64-bit or 31-bit execution environment. The Direct-CALL library will automatically switch to AMODE 64 when a z/Architecture DCALL entry point is invoked.

Note that the environment pointer returned in register `R0` with a `DCALL=CREATE` invocation is allocated in the `AMODE` of the calling function. Thus, for 31-bit programs invoking z/Architecture `DCALL=SUPPLIED` entry points, even if the entry pointer is running with `AMODE=64`, the environment pointer will be a 31-bit address. For z/Architecture programs running with AMODE=64, the environment pointer for a `DCALL=SUPPLIED` invocation is be a complete 64-bit value.

Mixing z/Architecture and non-z/Architecture functions

With Systems/C, each load module is either linked with the z/Architecture versions of the Systems/C library, or non-z/Architecture versions. This allows for a complete library in both environments without issues in clashing names or varying pointer sizes, or other considerations.

However, programs linked with the z/Architecture library may invoke `DCALL` programs created with the non-z/Architecture library, and vice-versa. The Systems/C

`DCALL` environment initialization will automatically switch `AMODE`s as appropriate, allowing for a seamless transition between the `AMODE=64` and `AMODE=24`/`AMODE=31` environments.

Also, Systems/C and Systems/C++ provide extensions which allow the programmer to declare explicit 64 and 31-bit pointers which facilitates the transition between the two environments. See the Systems/C Compiler manual and Systems/C++ Compiler manual for more information.

Programming for TSO and BATCH

Systems/C programs can be executed from either TSO or BATCH (JCL) environments.

Running programs under TSO

Systems/C programs started via a TSO address space are typically invoked via the `CALL` command.

For example:

```
READY
call 'my.progs(prog)' 'my parms'
```

would invoke the program `prog` in the `my.progs` PDS, passing the single parm string "`my parms`".

The double-quote character can be used to group together characters including a comma. Within a double-quoted string, the back-slash character can be used to represent the double quote. For example, to produce the argv[] strings `"my,parm"` and `"parm2"`, the parm string would be `'"my,parm",parm2'`.

Note that TSO, by default, will upper-case parameter strings. If lower-case letters are needed in the parm string, be careful to add the `ASIS` option on the `CALL` command.

If a Systems/C program is in a PDS that is in the `JOBLIB` or `STEPLIB` concatentations, it can be executed just as any other system program, without directly using CALL.

argv processing under TSO

When a program is executed via the TSO `CALL` interface, the argument string is parsed looking for argument delimitor character, which defaults to a comma (`,`). Each delimitor character separates an argument value.

Unlike UNIX (USS) systems, the parm string is not parsed for spaces, or quotes. It is simply broken at each instance of the delimitor character.

An alternate character for argument delimiters can be specified by defining the `char __argvc` variable. If that is defined, the runtime uses the specified character as the delimiter. If the character specified is a space, the runtime will skip multiple spaces.

Thus, the parm string `'my,parms'` would produce two values passed in the `argv` array on the invocation of `main()`. The first would be the string `"my"`, the second is the string `"parms"`.

Similarly, the parm string `'my space,parms'` would produce two `argv` elements, `"my space"` and `"parms"`, because spaces are not a parameter delimiter.

Running programs under BATCH JCL

Systems/C programs can be executed under normal JCL via the typical `EXEC` JCL statement.

For example, if the PDS `MY.PDS` contained a Systems/C program named `MYPROG` then the JCL statement:

```
//RUN EXEC PGM=MY.PDS(MYPROG),PARM='parm string'
```

would execute `MYPROG` passing the parameter string `'parm string'`.

argv processing under BATCH

Similar to argument processing under TSO, the Systems/C runtime looks for the argument delimitor character to separate arguments. Unlike UNIX or POSIX systems, BATCH mainframe programs typically use commas as a parameter delimiter. The default argument delimitor character is a comma, but can be overriden by defining the `char __argvc` variable. If the `char __argvc` is defined to be a space, then multiple spaces are skipped. To better integrate into existing environments, the Systems/C runtime defaults to a comma as the argument delimitor character.

Each delimitor character in the incoming `PARM` value is taken as a separator to separate the resulting `argv` values passed to the `main()` function.

The `char __argvc` variable can indicate a different character to use as the argument separator. If `char __argvc` is set to a space, the runtime environment will skip adjacent spaces, considering them as one.

Thus, if the delimitor character is using the default value of a comma, the `PARM` value `'my,parms'` would produce two values passed in the `argv` array on the invocation of `main()`. The first would be the string `"my"`, the second is the string `"parms"`.

Similarly, if a comma is the delimitor character, the `PARM` string `'my space,parms'` would produce two `argv` elements, `"my space"` and `"parms"`, because spaces are not a parameter delimiter.

If needed, the double-quote character can be used to group together characters, including the delimitor charactor. Within a double-quoted string, the back-slash character can be used to represent the double quote. For example, to produce the argv[] strings `"my,parm"` and `"parm2"`, the `PARM` string would be `'"my,parm",parm2'`, assuming comma is the delimitor character.

Programming for OpenEdition

The Systems/C library supports programs executed under OpenEdition MVS (Unix Systems Servies - USS). Programs can be executed under the USS shell, or take advantage of the facilities provided by OpenEdition services, including the various POSIX functions and Hierarchical File System (HFS.)

Note that all of the POSIX functions also operate in 64-bit mode.

As noted in the individual function descriptions, many of the POSIX functions are only supported for HFS files. A POSIX file funtion applied to a non-POSIX file will fail with an appropriate error code.

If Unix System Services are unavailable, the functions will fail with error return codes when possible.

More recent versions of OpenEdition require re-entrant programs; thus the compiler option *–frent* must be specified when compiling, and the objects should be linked with the re-entrant Systems/C library.

Linking programs under the OpenEdition Shell

Systems/C programs can be linked under the OpenEdition shell; to create load modules that reside in the Hierarchical File System (HFS).

To create an HFS load-module, the output from **PLINK** can be linked using the OpenEdition `cc` command. The `-e //` option should be added the `cc` command to indicate that the entry-point is not the default Language Environment entry point expected by `cc`. The Systems/C runtime library will specify its own entry-point.

For example, to pre-link and link the object `myfunc.o` and produce the `HFS` load-module `myprog` under the OpenEdition shell (assuming `/usr/local/dignus` is the installation location), simply run **PLINK**:

```
plink -omyprog.o myfunc.o "-S/usr/local/dignus/objs_rent/&m"
```

then use the OpenEdition `cc` command:

```
cc -e // -omyprog myprog.o
```

to produce the `myprog` load-module. `myprog` can then be invoked as any other OpenEdition program.

Copying programs from a PDS to the OpenEdition Shell

To copy a program from a PDS or PDSE to the HFS file system, the program must be re-linked into the HFS. Unfortunately, the IBM linker will not determine the proper entry-point in this case, and so the Systems/C entry point must be specified on the `cc` command.

To re-link a program from a PDS or PDSE into the HFS, the Systems/C entry-point `@crt0` (lower-case) must be specified in the `cc` command.

For example, if the PDS `MYNAME.T.LOAD` contained the Systems/C program named `MYPROG`, it could be copied into the HFS executabled named `myprog` with the command:

```
cc -e @crt0 -omyprog "//'MYNAME.T.LOAD(TEST)'"
```

Running programs under the OpenEdition Shell

Programs running under the OpenEdition shell are started via the `BPX1EXC exec` service. Systems/C programs residing in the HFS can simply be run as any other OpenEdition program.

More recent versions of OpenEdition require re-entrant programs; thus the compiler option *–frent* must be specified when compiling, and the objects should be linked with the re-entrant Systems/C library.

The Systems/C runtime recognizes when the program is started via the `exec` service, and processes the incoming argument and environment parameters appropriately. The arguments will be presented to the program in the `argv` array; and the environment variables will be available via the standard `getenv()` functions.

Furthermore, when started via `exec`, the first three file descriptors will be inherited from the invoking process. The Systems/C I/O functions will make these directly available to the program as file descriptors #0, #1 and #2, which are then also associated with `FILE *` variables `stdin`, `stdout` and `stderr`. Also in this case, the default filename style will be set to "`//HFS:`" so that file names will, by default, refer to files within the Hierarchical File System.

Programming for CMS

The Systems/C library supports a limited CMS environment, taking advantage of the OSRUN facility on CMS. The library does not support TCP/IP, or the SFS, but is a basic port of the existing I/O and memory management library used on z/OS.

Linking programs for CMS

To produce an object deck that is eventually linked on CMS, the CMS runtime objects must be present on the **PLINK** command line before the normal object library specification. This will insert the CMS runtime ahead of the normal runtime. These are the `cmsutil`, `@@ddndec`, `@@tyqsac` and `@@ddncms` object decks found in the `objs_rent` and `objs_norent` directories on cross-platform hosts, or the `LIBCR` and `LIBCN` PDSs on z/OS.

The Systems/C runtime also makes reference to the `DMSSTKR` symbol when linking, thus during the **PLINK** step the *–allow_ref=DMSSTKR* option should be used to account for this unresolved reference in the **PLINK** step.

Once the **PLINK** step is performed, the resulting object deck can be copied to CMS, and placed in an FB 80 file with the `.TEXT` file mode.

The CMS linker, `LKED` can then be used to create a member of a `LOADLIB` that can be executed with `OSRUN`.

Note that Systems/C programs for CMS are currently limited to `RMODE=24` execution, because of restrictions in the OS/390 emulation routines.

Using PLINK to create CMS programs

PLINK performs several important tasks for CMS programs.

The CMS linkage editor (`LKED`) is limited to only 4096 bytes for PRV (Psuedo Register Vector) processing. **PLINK** addresses this issue by performing all PRV processing, so that the object deck presented to `LKED` has no PRV references.

LKED also does not handle XSD or GOFF style input. **PLINK** when the *–(px)* option will properly adjust the resulting object deck to only be ESD-style, shortening long names and converting the input object decks appropriately.

To create programs for CMS, the CMS runtime support must be specified before the normal z/OS runtime libraries.

For example, on a Windows platform, if the typical **PLINK** command for pre-linking looked like (where the Systems/C installation was in the C:\sysc directory):

```
plink -px -omy_prog.obj t1.obj t2.obj "-SC:\sysc\objs_rent\&M"
```

then to link for execution on CMS, we need to insert the CMS runtime objects and specify the DMSSTKR is allowed to be an unresolved reference:

```
plink -px -omy_prog.obj t1.obj t2.obj
          -allow_ref=DMSSTKR
          C:\sysc\objs_rent\cmsutil
          C:\sysc\objs_rent\@@ddndec
          C:\sysc\objs_rent\@@tyqsac
          C:\sysc\objs_rent\@@ddnms
               "-SC:\sysc\objs_rent\&M"
```

(note that if the command line becomes too long for Windows, the –@ option can be used to place command line options in a file. See the **PLINK** section of the *Systems/C Utilities* manual for more information.)

To implement the same task when running **PLINK** on OS/390 or z/OS, simply adjust the SYSIN stream to specify the CMS runtime object decks.

If the typical **PLINK** step in the JCL looked like:

```
      ...
//PLINK EXEC PGM=PLINK,PARM='-px'
//STDERR DD SYSOUT=A
//STDOUT DD SYSOUT=A
//SYSLIB DD DSN=DIGNUS.LIBCR.OBJ,DISP=SHR
//INDD DD DSN=mypds,DISP=SHR
//SYSIN DD *
 INCLUDE INDD(PROG)
//SYSMOD DD DSN=myoutput.obj,DISP=NEW
```

then, to pre-link this program for CMS, adjust the PARM value to include the -allow_ref=DMSSTKR and specify the CMS objects in the SYSIN stream, as in:

```
      ...
//PLINK EXEC PGM=PLINK,PARM='-px,allow_ref=DMSSTKR'
//STDERR DD SYSOUT=A
//STDOUT DD SYSOUT=A
//SYSLIB DD DSN=DIGNUS.LIBCR.OBJ,DISP=SHR
//INDD DD DSN=mypds,DISP=SHR
//SYSIN DD *
 INCLUDE INDD(PROG)
 INCLUDE SYSLIB(CMSUTIL)
 INCLUDE SYSLIB(@@DDNDEC)
 INCLUDE SYSLIB(@@TYQSAC)
 INCLUDE SYSLIB(@@DDNCMS)
//SYSMOD DD DSN=myoutput.obj,DISP=NEW
```

(note that the *–px* option was also specified in the `PARM` string when executing **PLINK**.)

Using LKED to link CMS programs

After the **PLINK** step has been executed the resulting object deck should be copied to CMS and placed into an FB 80 dataset with the `TEXT` file mode. This can be accomplished using FTP or any other binary transfer.

Once the **PLINK** output has been placed on CMS, the `LKED` command will link it and produce a `LOADLIB` member which can be executed with `OSRUN`.

The `VMLIB TXTLIB` must be `GLOBAL`'d to resolve references that the Systems/C runtime requires. Also, this library should be specified as the `SYSLIB` so `LKED` can resolve those references.

The **PLINK** generated object deck should be specified as the `SYSLIN` input to `LKED`.

Because of limitations in the `OSRUN` environment, Systems/C programs must be linked with `RMODE=24` specified.

For example, if the result of **PLINK** was placed on the `A` disk with the file name `PROG TEXT A`, and the resulting program should reside in the `MYLOAD LOADLIB A` load library, these commands would execute `LKED` to accomplish the linking:

```
FILEDEF SYSLMOD DISK MYLOAD LOADLIB A (RECFM U
GLOBAL TXTLIB VMLIB
FILEDEF SYSLIB DISK VMLIB TXTLIB S (PERM
FILEDEF SYSLIN DISK PROG TEXT A
LKED PROG (RMODE 24 AMODE 31
FILEDEF SYSLMOD CLEAR
FILEDEF SYSLIB  CLEAR
FILEDEF SYSLIN  CLEAR
```

Note the RMODE 24 was specified on the LKED command.

Consult the IBM VM/CMS documentation for further information about these commands.

Executing programs on CMS

To execute Systems/C programs on CMS, the OSRUN command is used. Appropriate FILEDEFs should be specified as the program may require. Each DD the program opens should be FILEDEF'd so that the open() may succeed.

The PARM option of the OSRUN command specifies any parameters passed to the program.

For example, if we intend to execute the program PROG with the parameters, "any,parms", from the MYLOAD LOADLIB A library, and the program read from the STDIN DD and wrote to the STDOUT and STDERR DDs, these commands would be employed:

```
GLOBAL LOADLIB MYLOAD
FILEDEF STDIN TERMINAL
FILEDEF STDOUT TERMINAL (LRECL 133 BLKSIZE 133
FILEDEF STDERR TERMINAL (LRECL 133 BLKSIZE 133
OSRUN PROG PARM='any,parms'
```

Note that the LRECL and BLKSIZE values must be specified on the FILEDEF for CMS files. For TERMINAL type files, the LRECL and BLKSIZE should be the same to avoid any block level buffering.

Systems/C programs are limited to the environment supported by OSRUN.

Programming for MVS 3.8

The Systems/C library supports programs for the MVS 3.8 operating system. Generally, the full support of the Systems/C library is available, with the restrictions inherent in the MVS 3.8 environment.

Linking programs for MVS 3.8

Systems/C supports executing C programs on MVS 3.8 by inserting MVS 3.8 specific objects in the link step before the normal library objects. These objects replace the normal library objects, providing MVS 3.8 low-level operating system support. The modules can be found in the `MVS38_objs_rent` and `MVS38_objs_norent` directories on cross-platform hosts, or as PDS members in the `LIBCR38` and `LIBCN38` PDS libraries on OS/390 and z/OS hosts.

To create MVS 3.8 executables, simple place these directories (or PDSs) in the **PLINK** search order ahead of the normal library.

Also, for support of long names in external identifiers, the Systems/C library is delivered in extended object (`XSD`) form. The MVS 3.8 linker does not support this form of object deck. Thus, the Systems/C pre-linker, **PLINK**, must be used and the *–px* option of **PLINK** must be enabled to process these objects and produce an object deck that is suitable for linking with the MVS 3.8 linker.

If IBM's HLASM assembler is used to produce the objects, and the `XOBJECT` parameter to HLASM is enabled, HLASM will produce object files in the GOFF object file format. **PLINK** can directly process these input files and produce objects that can be handled by the MVS 3.8 linker. **PLINK**, with the *–px* option, will properly convert these files into objects suitable for the MVS 3.8 linker. With this approach, HLASM-produced objects with support for long identifier names can be used to create MVS 3.8 programs. For more detailed information about **PLINK** see the *Systems/C Utilities* manual.

Using PLINK to create MVS 3.8 programs

As mentioned above, **PLINK** must be used to pre-link the input objects and place them in a format suitable for use on MVS 3.8. **PLINK**'s primary function in this regard is to convert any long names and or XSD cards in the generated objects to short names and produce an object file that the MVS 3.8 linker will process. Thus, the *–px* option should be used on the **PLINK** command, which instructs **PLINK** to perform this processing.

Furthermore, for re-entrant programs, **PLINK** will process all `PRV`-related operations, processing `PR` and `XD` symbols internally. The MVS 3.8 linker cannot handle `PRV` vectors larger than 4K bytes. The **PLINK** *–prem* option, which supports this function, is enabled by default, and should not be disabled when creating MVS 3.8 executables.

PLINK is also used to ensure the MVS 3.8 objects are used instead of the normal library objects. On cross-platform systems, simply specify the MVS 3.8 object directories ahead of the normal object directoris on the **PLINK** command line. On OS/390 or z/OS, specify the MVS 3.8 library PDS ahead, in a concatenation with the normal library PDS.

On OS/390 or z/OS to link with the MVS 3.8 re-entrant libraries, add the `LIBCR38` PDS to the `SYSLIB` concatenation, otherwise use the `LIBCN38` PDS. On cross-platform hosts, to link with the re-entrant MVS 3.8 library, add the `MVS38_objs_rent` directory to the search list, ahead of the normal library specification. To link with the non-re-entrant MVS 3.8, on cross-platform hosts, add the `MVS38_objs_norent` directory.

For example, on a Windows platform, if the typical **PLINK** command for pre-linking looked like (where the Systems/C installation was in the `C:\sysc` directory):

```
plink -omy_prog.obj t1.obj t2.obj "-SC:\sysc\objs_rent\&M"
```

then to link for execution on MVS 3.8, insert another search template which specifies the MVS 3.8 directory, as in:

```
plink -omy_prog.obj t1.obj t2.obj "-SC:\sysc\MVS38_objs_rent\&M"
                "-SC:\sysc\objs_rent\&M"
```

(note that if the command line becomes too long for Windows, the *–@* option can be used to place command line options in a file. See the **PLINK** section of the *Systems/C Utilities* manual for more information.)

To implement the same task when running **PLINK** on OS/390 or z/OS, simply adjust the `SYSLIB` DD statement to provide the proper concatenation.

If the typical **PLINK** step in the JCL looked like:

```
      ...
//PLINK EXEC PGM=PLINK,PARM='-px'
//STDERR DD SYSOUT=A
//STDOUT DD SYSOUT=A
//SYSLIB DD DSN=DIGNUS.LIBCR.OBJ,DISP=SHR
//INDD DD DSN=mypds,DISP=SHR
//SYSIN DD *
 INCLUDE INDD(PROG)
//SYSMOD DD DSN=myoutput.obj,DISP=NEW
```

then, to pre-link this program for MVS 3.8, adjust the `SYSLIB` DD statement to add the MVS 3.8 PDS, as in:

```
       ...
//PLINK EXEC PGM=PLINK,PARM='-px'
//STDERR DD SYSOUT=A
//STDOUT DD SYSOUT=A
//SYSLIB DD DSN=DIGNUS.LIBCR38.OBJ,DISP=SHR
//       DD DSN=DIGNUS.LIBCR.OBJ,DISP=SHR
//INDD DD DSN=mypds,DISP=SHR
//SYSIN DD *
 INCLUDE INDD(PROG)
//SYSMOD DD DSN=myoutput.obj,DISP=NEW
```

(note that the *–px* option was specified in the `PARM` string when executing **PLINK**.)

MVS 3.8 runtime restrictions

In general, beyond the environmental constraints of an MVS 3.8 system, there are no issues with Systems/C programs. Except for the following noted differences, the entire Systems/C run-time library and all of the Systems/C programming features operate as they would on a more recent operating system.

MVS 3.8 only supports 24-bit addresses, thus Systems/C programs are limited by MVS 3.8's memory size restrictions.

Dynamic allocation of files via the `O_CREAT` flag on open(2) calls is not supported.

MVS 3.8 does not provide TCP/IP, thus the TCP/IP related functions in the Systems/C library will not operate.

MVS 3.8 does not provide the `BPX` family of services, thus POSIX functions are not available and will fail with an error return code.

MVS 3.8 programs can be executed on OS/390 or z/OS. If the program objects are re-linked on OS/390 or z/OS, the `AMODE=24` and `RMODE=24` options should be specified on the IBM link step, or to the **PLINK** command if **PLINK** is creating a TSO TRANSMIT module. The MVS 3.8 low-level operating system interfaces provided in the MVS 3.8 objects will not operate correctly on OS/390 or z/OS if the program is not linked `AMODE=24` and `RMODE=24`.

Controlling the runtime environment

Runtime Options specified in the program arguments

Runtime options can be specified in the program arguments in a BATCH or TSO program, or in the `_DIG_RUNOPTS` environment variable in a program running under OpenEdition.

Runtime options are not examined in DCALL environments.

By default, runtime options are disabled in the BATCH and TSO environments, to enable them set the global `int __runopt` variable to `1`.

The following runtime options are supported:

`ENVAR(`*`name=val`*`)` specifies that the environment variable *name* should be set to the value *val* in the start-up runtime environment.

`TRAP(`*`trap-setting`*`)` Specifies that a runtime ESTAE should be established for the receipt of signals generated by hardware interrupts. *trap-setting* is either `ON` or `OFF`.

Unrecognized options are silently ignored.

Runtime Options in TSO and Batch

In the TSO and BATCH environment, the incoming argument string is examined. All text up to the first backslash ('\') is examined to look for runtime options. The actual program arguments follow this first backslash. If no backslash is present at all, then the entire string is taken to be the program arguments. Note that the backslash character can be changed via the declaration of the global variable `__rochar`, as in:

```
char __rochar = '/'; /* set / as the runtime options delimiter */
```

Previous versions of the Dignus Systems/C runtime used the slash ('/') character as the delimiter; but Dignus Systems/C programs use the slash in file names which frequently appear as command line arguments (i.e. `//DSN:MY.FILE.NAME`). Because of this the default runtime options delimiter character was changed to backslash ('\').

Furthermore, to avoid other potential issues with older programs, runtime options processing is off by default in the TSO and BATCH environments. If your program wants to take advantage of runtime options processing in the TSO or BATCH environments, it needs to be specifically enabled by declaring the `__runopt` integer as in:

```
int __runopt = 1;
```

Runtime Options in OpenEdition

When running in the OpenEdition environment, the Systems/C library looks for runtime options in the environment variable "`_DIG_RUNOPTS`". Any runtime options are specified there. The incoming argument list is not examined in this environment.

Disabling/Enabling runtime options in TSO and Batch

If you define an integer named `__runopt` at global scope, and give it the value 0; then runtime options processing is disabled, and the entire parameter string will be used.

This is the default behavior.

That is:

```
int __runopt = 0;
```

will defeat runtime options processing in TSO and Batch.

To enable runtime options processing in TSO and Batch, set the value of `__runopt` to 1 as in:

```
int __runopt = 1;
```

By default, the backslash (
) character is used to delimit the end of the runtime options and the start of the argument string. You can change this default character by declaring the character variable `__rochar` at file scope and initializing it with the character to use. For example:

```
char __rochar = '|'; /* use | to mark end of runtime options */
```

stdin, stdout and stderr

According to the C standard definition, three standard streams are initilized and opened when program begins execution, `stdin` for input, and `stdout` and `stderr` for output.

In Systems/C, streams are implemented in terms of lower-level file descriptors, `stdin` is associated with file descriptor #0, `stdout` is associated with file descriptor #1 and `stderr` is associated with file descriptor #2.

Initially, the Systems/C library opens these file descriptors with the names `"//DDN:STDIN"`, `"//DDN:STDOUT"` and `"//DDN:STDERR"`. `"//DDN:STDOUT"` and `"//DDN:STDERR"` are opened with an LRECL=133 and BLKSIZE=1330 by default.

Changing standard filenames at execution time

The standard approach of using the freopen(3) function to reassociate the standard file streams operates as expected with the Systems/C runtime.

For example, to close and re-open the `stdin` stream to the `SYSIN` DD, a program can simply:

```
if(!(freopen("//DDN:SYSIN", "r", stdin)) {
   perror("couldn't re-open stdin");
}
```

See the freopen(3) function description for more information.

Changing standard filenames and attributes at compile time

A program can specify alternate strings to change the names the library uses to open the first three file descriptors.

To change the name, initialize a `char *` variable with the replacement file name to be used, as described in the following table:

```
stdin   char * __fd0nm = "name";
stdout  char * __fd1nm = "name";
stderr  char * __fd2nm = "name";
```

When `__fd0nm`, `__fd1nm`, or `__fd2nm` is defined, the library uses the name defined there as the initial name for file descriptors #0, #1 and #2 respectively.

For example, the following declaration causes the library to associate the `SYSIN` DD with file descriptor #0, making it the `stdin` stream at program start-up:

```
char *__fd0nm = "//DDN:SYSIN";
```

To change the default attribute used to open a file, specify an attribute string in the `__fd0atr`, `__fd1atr` or `__fd2atr` global variables. The string specified will be passed as the attribute parameter to the open(2) invocation at library start-up.

For example, the following declaration will cause the `stdout` stream, file descriptor #1, to be opened as an FB80 file with a blocksize of 800:

```
char *__fd1atr = "recfm=fb,lrecl=80,blksize=800";
```

See the open(2) function description for more information about file attribute strings.

Note that these names are only used when a program is not executed via the BPX `execve` interface. If a program is executed from the USS shell, or via the `BPXCALL` interface in batch mode, the first 3 file descriptors are inherited from the environment and the Systems/C library does not invoke open(2) to provide them.

Choosing the TCP/IP interface

By default, the Systems/C runtime library uses the BPX socket interface for implementation of the various TCP/IP-related functions.

Older versions used the `EZASMI` interface.

You can choose to use the `EZASMI` interface by defining the `__bpxso` integer variable at a global scope and initializing it with the value `0`, as in:

```
int __bpxso = 0;
```

Note that if you use the `EZASMI` interface, socket file descriptors will not be inherited across a **fork()** function call and the file descriptor will be close()'d in the child.

Changing argv delimiters for BATCH and TSO

By default, when a Systems/C program is executed under BATCH or TSO, the delimiter that separates arguments is the comma, which is typical of these programs.

However, it can be changed to any character by defining the `char __argvc` variable in a program.

When `__argvc` is defined, the runtime library uses the character value specified there as the argument delimiter.

Furthermore, if `__argvc` is defined as a single space, the runtime will consider adjacent spaces as one.

Thus, if the program had:

```
char __argvc = ' ';
```

then the parm string, `'a parm string'`, under TSO or BATCH would generate the `argv[]` array:

```
argv[1]  "a"
argv[2]  "parm"
argv[3]  "string"
argv[4]  NULL
```

Disabling runtime options for BATCH and TSO

Normally, when executing in a TSO or BATCH environment, any value in the PARM string up to the first right slash is examined for runtime options.

If needed, you can disable this check by defining the `__runopt` integer variable at global scope, with a value of 0, as in:

```
int __runopt = 0;
```

If `__runopt` is defined, and it has a zero value, then the initial runtime startup processing will not look for any runtime options, and the entire string will be used to produce the `argc` and `argv` values passed to the `main()` function.

Controlling stack space allocation

The Systems/C runtime library allocates space used at runtime for per-function areas. This space is the runtime "stack". The runtime doesn't allocate a separate space for each function, instead allocating and managing this space in blocks of storage.

The initial block of storage is called the `ISTK` (initial stack allocation.) If this block is sufficiently large for the entire run of the program, no other memory will need to be allocated. Creating a sufficiently large block can greatly improve runtime performance.

The initial stack allocation can be specified in the `#pragma prolkey` of the `main()` or other entry-point function. When the Systems/C library begins execution, it uses the value specified in the `ISTK=`*n* prologue key for the initial allocation size.

For example,

```
#pragma prolkey(main,"ISTK=4096")
main()
{
```

specifies that the initial stack allocated when this program is begun is 4096 bytes.

As a program runs, more stack space may be dynamically required. The Systems/C runtime system automatically allocates and manages that space as needed. This space is called the extension stack, or `ESTK`.

However, small and frequent allocations can result in degraded performance. If indicated, it may be prudent to specify a particular stack extension on a function, to cause the library to pre-allocate a larger extension should one be needed. This can be done using the `ESTK=`*n* prologue key.

For example:

```
#pragma prolkey(lotsofstack,"ESTK=16384")
lotsofstack()
{
```

specifies that when `lotsofstack()` is invoked, should a stack extension be required, the allocated space will be at least 16384 bytes in size.

Specifying the runtime storage SUBPOOL

By default, the Systems/C runtime allocates stack and heap memory in the default subpool.

However, this can be altered by specifing `#pragma prolkey` setting `SP=`*`n`* on the `main()` or other entry-point function.

The subpool value is a numeric.

For example,

```
#pragma prolkey(main,"DCALL=YES,SP=123")
myfunc()
{
```

specifies that memory allocated by the Direct-CALL function `myfunc()` be allocated from sub-pool #123.

Specifying the runtime KEY

Systems/C programs begin execution in the default key setting. Specifying the `#pragma prolkey` setting `KEY=`*`val`* will cause the runtime library to switch to the specified key and begin execution. When the runtime is complete, for example, a Direct-CALL environment terminates, the key will be reset to the value that was present on entry to the function.

val can be either the keyword `ENTRY` or a numeric key value.

For example:

```
#pragma prolkey(func,"DCALL=YES,KEY=8")
func()
```

specifies that just before `func()` is invoked, the current hardware protection key saved, and the set to 8. On return from `func()`, the key will be restored to the saved value.

Controlling access to Unix System Services

On z/OS, when a Unix System Service is required the runtime library invokes the appropriate assembler interface. For example, to `open` a file in the Hierarchical File System (HFS), the runtime library would invoke the `BPX1OPN` service.

When a Unix System Service is invoked, it causes the z/OS task to become 'dubbed'; which is undesirable in some cases, and simply not allowed in others. In particular, this is not allowed in a CICS environment.

To provide control over this, the runtime library defines an `int` variable named `__NOBPX`.

`__NOBPX` is initialized to `0` indicating that Unix System Service functions are allowed.

Assigning any non-zero value to `__NOBPX` will cause the runtime library to disallow Unix System Service functions, failing the function with `errno` set to `ENOSYS`.

This can be particularly useful in a Direct-CALL CICS environment, where Unix System Service functions are not allowed.

Note that the runtime library may attempt to invoke Unix System Services due to runtime library requirements, even if the user's program does not directly invoke these services.

Signal Handling

A Systems/C program supports several style of signal management. Using an library-established ESTAE exit to catch signal-producing hardware interrupts, as well as supporting the OpenEdition "SIR" style of signal handling.

When a Systems/C program is initiated via an exec(2) function, the OpenEdition style of signals is assumed, as this would be in a POSIX environment. Also, if the pthread_create(3) function is called, that requires the POSIX environment, and thus OpenEdition signals are used.

Otherwise, the use of an ESTAE to recognize hardware interrupts and generate a signal is controlled by the `TRAP` setting on the prologue macro of the **main()** function. `TRAP` defaults to `OFF` so no ESTAE is established, and a hardware interrupt is simply processed as z/OS usually does, typically an ABEND. Specifying the `#pragma prolkey` setting `TRAP=ON`, or specifying the TRAP(ON) run option, causes the Systems/C library to establish an ESTAE exit to handle hardware interrupts and raise an appropriate signal.

You can also use the runtime option of "`TRAP(ON)`" or "`TRAP(OFF)`" to override the program setting on the prologue macro.

As previously mentioned, in a POSIX environment (execution was initiated via exex(2)), an ESTAE exit is required for proper signal delivery and the setting of the `TRAP` value will be forced to `ON`. Also the ESTAE exit is required for POSIX pthread processing if pthread functions are used.

Considerations for SIGABND processing

In the Dignus runtime on z/OS when an ESTAE exit is established, a SIGABND signal is raised when an ABEND is issued. The SIGABND can be handled with a SIGABND signal handler established by either the signal(3) or sigaction(2) functions.

The signal hander can use the __abendcode(3) and __rsncode(3) functions to query the abend and reason codes associated with the ABEND.

Returning from the signal handler causes the program state to be restored to the instruction that caused the ABEND. As that is the instruction that caused the ABEND, the ABEND will then be re-issued. If the SIGABND signal handler remains active, then it will be re-invoked, essentially causing a loop. This is consistent with signal processing in UNIX environments.

If the signal state for SIGABND is SIG_DFL on return from the signal handler normal z/OS abend processing will processing will occur. In z/OS parlance, the ABEND will be "percolated".

SIG_IGN is invalid for SIGABND signals; as a SIGABEND cannot be ignored.

The Dignus stack management routines issue an ABEND 978 for the out-of-stack situation. To establish a signal handler for that it is necessary to use the sigaltstack(3) and sigaction(2) functions to provide a separate stack area for executing of the handler. Otherwise, an ABEND 978 is immediately percolated as there is no stack on which to execute the signal handler.

An example of how to catch an ABEND 978 is provided in the appendix.

OpenEdition

Linking under OpenEdition

A Systems/C program can either be linked into a PDS, PDSE or into the Hiearchical File System (HFS.

Under USS, to link a Systems/C program, first the program is processed with the Systems/C pre-linker (**PLINK**, to produce an output object module. Then, the final linking can use the USS cc command to link. To use the cc command link into the HFS, the "-e //" option should be added.

For example,

```
plink -o prog1.obj prog1.o -S"objs_rent/&M"
cc -e // -o prog1 prog1.obj
```

would pre-link the program `prog1.o` including the re-entrant Systems/C library objects, then the "`cc -e //`" command would complete the link, producing the program `prog1`.

Note that if the library objects need not be present in the HFS; the **PLINK** command can reference the library PDS, as in:

```
plink -o prog1.obj prog1.o -S"//DSN:DIGNUS.LIBCR.OBJ(&M)"
```

Alternatively, a batch linking process can specify that the final output from the IBM binder reside in the `HFS`. Simply adjust the `SYSLMOD DD` definition appropriately. For example, to cause the IBM binder to write to the HFS file `/users/employee/prog1`, an appropriate `SYSLMOD` definition might be:

```
//SYSLMOD  DD PATH='/u/rivers/PLINK20/test/prog',
//            PATHOPTS=(OWRONLY,OCREAT),
//            PATHMODE=(SIRWXO,SIRWXG,SIRWXU)
```

Running under OpenEdition

When a Systems/C program is executed via the `exec()` function, the Systems/C runtime start-up processes the command-line arguments in a typical UNIX-style fashion. Also, the Systems/C runtime correctly initializes the environment values from the environment pointers specified in the `exec()` invocation.

The isPosixOn(2) function can be used to determine if the program was started via `exec()` and thus, mostly likely, under the OpenEdition shell. For example:

```
if(isPosixOn()) {
     printf("I was started under USS\n");
} else {
     printf("I was started under BATCH or TSO\n");
}
```

When programs are started via `exec()`, the default file "style" (`_style`) is set to `"//HFS:"`. So that any file name which doesn't explicitly specifically specify a different style is assumed to be an `//HFS:`-style file.

Programs started via `exec()` inherit the first 3 file descriptors from the parent process. So the values of `__fd0nm`, `__fd1nm`, `__fd2nm`, `__fd0atr`, `__fd1atr` and `__fd2atr` do not apply. If a program needs to adjust these default file descriptors, the standard freopen(3) approach should be used. See the freopen(3) function description for more information.

Programs executing in the OpenEdition environment use BPX-style signal handling.

Data locations

Systems/C data address ranges are not restricted in any fashion, data can reside above the 2-gigabyte "bar", or above the 16-megabyte "line" or below the 16-megabyte "line".

Data in Systems/C programs is in three general areas, the "HEAP", which is data allocated via `malloc()` function calls, the "STACK", which is local data allocated to each function, and the "PRV", which contains file-scoped re-entrant data. Also note that the library provides a separate mechanism for dynamic management of "heap" memory in a particular address range, via `__malloc31()` and `__malloc24`.

Each of these locations can be specified by additional `#pragma prolkey` statements on program entry points, either the `main()` function entry point, or a DCALL entry point.

The default locations for the HEAP, STACK and PRV in z/Architecture programs (programs where the `main()` function was compiled with the *–march=zarch* option enabled) is above the 2-gigabyte "bar". Otherwise, the default is below the 2-gigabyte "bar".

If the *–famode=any*, or *–famode=31* or *–famode=24* option was specified on the compilation of the `main()` function for z/Architecture programs, the default locations will be below the 2-gigabyte "bar".

Each of the HEAP, STACK and PRV locations can be specifically defined by adding `LOCHEAP=`*loc*, `LOCSTACK=`*loc* and `LOCPRC=`*loc* to the entry-point's prologue macro using `#pragma prolkey`. The values for *loc* are `24`, `31`, `ANY` and `64`. `ANY` is equivalent to `31`.

For example, if the following `#pragma prolkey` was specified for a `main()` compiled with *–march=zarch*, and no *–famode* option was specified, then the Systems/C runtime could allocate the PRV and HEAP data above the 2-gigabyte "bar", but STACK data would be restricted to below.

```
#pragma prolkey(main,"LOCSTACK=31")
int main(int argc, char *argv[])
{
  ...
```

Stand alone function

The Systems/C and Systems/C++ compilers can be used to create programs that use your own entry and exit linkage. Such code cannot be linked with the Systems/C runtime because the Systems/C runtime assumes that the Systems/C linkage has been employed.

However, some functions can be safely linked into such an environment.

Compiler invoked routines

When not compiling in z/Arch mode (*–march=zarch* is not specified), the C and C++ compilers use "helper" functions to accomplish some 64-bit arithmetic. These functions are:

`@@MULU64`	64-bit unsigned multiply
`@@MULI64`	64-bit signed multiply
`@@DIVU64`	64-bit unsigned divide/modulus
`@@DIVI64`	64-bit signed divide/modulus

These function only assume that register 13 points to a typical save area (80 bytes) and can safely be used in a typical environment. These functions can be found in the Systems/C library and can be linked with your own code (provided R13 points to a typical save area in your own linkage.)

Initializing re-entrant data

When compiling with the *—frent* option, or more generally, in the presence of any `__rent` data, the Systems/C and Systems/C++ compilers generate a re-entrant initialization section which is gathered together by the pre-linker **PLINK**. **PLINK** places its information in a `CSECT` named `@@RINIT#`. If that symbol is not resolved, then the program required no re-entrant initializations.

At initial program start-up, the Systems/C runtime examines this section and, after allocating the re-entrant data, performs the various initializations indicated.

If it is desired to have initialized re-entrant data within your own runtime environment, then this same operation needs to be performed.

The Systems/C runtime provides the `@@SARNTI` function (stand-alone re-entrant initialization) to accomplish this task. After allocating re-entrant data, the program should initialize the data area to zeros and then invoke `@@SARNTI` with the address of the re-entrant data, the **PLINK**-generated re-entrant initialization section and some other parameters as described below.

The size of the re-entrant data can be obtained using a `CXD` relocation.

`@@SARNTI` is "stand alone" in that it requires no other runtime and uses standard OS linkage. When linked from the 31-bit runtime library `@@SARNTI` assumes that R13 points to a typical 80-byte save area, and further assumes that R1 points to a typical OS-linkage parameter block. When linked from the 64-bit runtime library `@@SARNTI` assumes that R13 is a 64-bit pointer that points to a Format-4 64-bit style save area, and that R1 is a 64-bit pointer that points to a parameter block that contains 64-bit pointers.

The parameters for `@@SARNTI` are:

`stack`	address of at least 1024 bytes used for stack space
`prv`	address of allocated rent data
`entries`	address of **PLINK**-generated initializers

The first parameter (the stack space used by `@@SARNTI`) must be a least 1024 bytes long.

The second parameter is the address of the memory allocated to contain the PRV. That memory must be initialized to all zeros before invoking `@@SARNTI`.

The third parameter is the address of the `@@RINIT#` section containing the `PLINK`-generated initialization data.

`@@SARNTI` returns a zero (`0`) in register R15 if it is successful, otherwise it returns a non-zero value in R15. The current return codes are:

`0`	successful initialization
`4`	invalid/unsupported initializers

On return from `@@SARNTI` the temporary stack space passed as the first parameter is unused by the Systems/C runtime and may be released or reused.

For example, the following snippet of code shows how to allocate the re-entrant data section for a 31-bit environment, initialize it to zero and then invoke `@@SARNTI`. The 64-bit environment would be similar except that the parameters addressed by R1 would be 64-bit pointers.

```
WXTRN  @@RINIT#
...
L  6,=A(@@RINIT#)
LTR  6,6
BZ  NONE         nothing to do?
L  4,RENT_SIZE
GETMAIN RC,LV=(4)   allocate reentrant data
LR 5,1
LR 0,1
L  1,RENT_SIZE
LA 14,0(0,0)
LA 15,0(0,0)
MVCL 0,14      zero-out rent data (assuming it's not too large)
LA 1,PARMS
LA 7,STACK     stack space
ST 7,0(0,1)    first parm
LR 7,5         addr of rent data
ST 7,4(0,1)    2nd parm
LR 7,6         addr of initializers
ST 7,8(0,1)    3rd parm
```

```
     L  15,=V(@@SARNTI)
     BALR 14,15
     LTR 15,15
     BZ DONE
     ... problems; initialization failed

NONE DS 0H
DONE DS 0H
     ...
RENT_SIZE CXD
STACK  DS 0D
       DS CL1024 stack space
PARMS  DS 0D
       DS  1A   address of stack
       DS  1A   address of PRV
       DS  1A   address of rent
```

User ABEND codes issued by the runtime

The Systems/C library can issue a small number of user ABENDs, due to various start-up and other situations where it is impossible to continue running and no other diagnostic facility is available.

These diagnostics are issued using the `ABEND` macro in z/OS, with the given ABEND number.

These values should not be employed by user code as they indicate issues particular to the Systems/C runtime environment.

The following list describes the ABEND number and the situation where it arises:

135 A dd number could not be allocated in the table of DD names; this is deprecated and will be eliminated in a future release.

136 The table for managing DD names cannot be allocated. This is likely due to insufficent available memory. This is deprecated and will be eliminated in a future release.

178 The initial memory allocation for a dynamic storage area (DSA) set-up failed, likely due to insufficient available memory, the program cannot run.

278 The first memory segment allocation failed, likely due to insufficient available memory, the program cannot run.

279 Insufficient memory was available to set up the program name, argv array or environ array at program start-up.

378 Returning the allocated re-entrant variable data space to the operating system failed at program termination. This is likely due to a program memory overlay detected at the end of execution.

478 Returning an allocated stack segment to the operating system failed. This is likely due to a program memory overlay detected at the end of execution.

555 An invalid or corrupt handle was passed to a DCALL=SUPPLIED function.

578 Returning the initial memory allocation to the operating system failed. This is likely due to a program memory overlay detected at the end of execution.

678 Re-entrant variable initialization failed. This is either a compiler or library problem and should be reported to Dignus.

978 The stack management routines were unable to allocate further memory to expand the stack segment. ABEND 978 can be "caught" by a `SIGABND` handler only if the library has established as ESTAE (`TRAP(ON)` is true) and an alternate execution stack is provided for the signal handling function.

801 64-bit program start failed. This is typically caused by linking code compiled for 31-bit with the 64-bit Dignus runtime. When linking with the 64-bit runtime, the *–mlp64* option must be used on the **DCC** and **DCXX** compiler command lines.

3532 The SIGABRT signal was sent to a Dignus program not executing under OpenEdition. The program prints a traceback and ends with this user ABEND.

Systems/C C Library functions

The Systems/C library provides the ANSI standard functions, as well as several extensions which aide in the porting of other programs to the mainframe.

The Systems/C library is compiled using the standard Systems/C prologue and epilogue macros. The Systems/C library environment would need to be established if these functions are to be included in a Systems/C program that uses an alternate stack frame layout.

This section provides an overview of the C library functions, the return values and other common definitions and concepts.

The function return types and parameters, as well as the requisite `#include` files are described in the SYNOPSIS section. The function is then described, followed by the possible values of the global variable `errno`. Also, related functions are named in the SEE ALSO section. If the function conforms to any particular standards, that will be noted in the STANDARDS section.

The run-time library is also divided into sections for easy reference.

System Functions

System functions are those that are typically implemented by the operating system on UNIX platforms.

These are implemented in the Systems/C C run-time library as best as the host operating system allows.

Several of the system functions described below depend on IBM's OpenEdition Assembler Callable Services. If these services are not available, the functions can fail, typically setting the global variable `errno` to `EOPNOTSUPP` (operation is not supported.) For more information about OpenEdition services, see the IBM "OS/390 OpenEdition Assembler Callable Services" manual and related IBM documentation.

ACCESS(2)

NAME

access – check access permissions of an `//HFS:`-style file or pathname

SYNOPSIS

```
#include <unistd.h>

int
access(const char *path, int mode);
```

DESCRIPTION

The **access()** function checks the accessibility of the file named by *path* for the access permissions indicated by *mode*. The value of *mode* is the bitwise inclusive OR of the access permissions to be checked (`R_OK` for read permission, `W_OK` for write permission and `X_OK` for execute/search permission) or the existence test, `F_OK`. All components of the pathname path are checked for access permissions (including `F_OK`).

RETURN VALUES

If *path* cannot be found or if any of the desired access modes would not be granted, then a `-1` value is returned; otherwise a `0` value is returned.

ERRORS

Access to the file is denied if:

`[ENOTDIR]` A component of the path prefix is not a directory.

`[ENAMETOOLONG]` A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

`[ENOENT]` The named file does not exist.

`[ELOOP]` Too many symbolic links were encountered in translating the pathname.

`[EROFS]` Write access is requested for a file on a read-only file system.

[ETXTBSY] Write access is requested for a pure procedure (shared text) file presently being executed.

[EACCES] Permission bits of the file mode do not permit the requested access, or search permission is denied on a component of the path prefix. The owner of a file has permission checked with respect to the "owner" read, write, and execute mode bits, members of the file's group other than the owner have permission checked with respect to the "group" mode bits, and all others have permissions checked with respect to the "other" mode bits.

[EFAULT] *Path* points outside the process's allocated address space.

[EIO] An I/O error occurred while reading from or writing to the file system.

[ENOTSUPP] The **access()** function is not supported on this type of path name.

SEE ALSO

chmod(2), open(2), stat(2)

STANDARDS

The **access()** function call is expected to conform to IEEE Std1003.1-1990 ("POSIX") for a *path* in the HFS.

CAVEAT

access() is a potential security hole due to race conditions and should never be used. Setuid and setgid applications should restore the effective uid or gid and perform actions directly rather than use **access()** to simulate access checks for the real user of group id.

access() only operates on `//HFS:`-style files. If the **access()** function is applied to non-`//HFS:` style files, the return value will be set to -1, and `errno` will be set to `EOPNOTSUPP`.

AIO_CANCEL(2)

NAME

aio_cancel – cancel an outstanding asynchronous I/O operation

SYNOPSIS

```
#include <aio.h>

int
aio_cancel(int fildes, struct aiocb * iocb);
```

DESCRIPTION

The **aio_cancel()** function cancels the outstanding asynchronous I/O request for the file descriptor specified in *fildes*. If *iocb* is specified, only that specific asynchronous I/O request is cancelled.

Normal asynchronous notification occurs for cancelled requests. Requests complete with an error result of `ECANCELED`.

RESTRICTIONS

The **aio_cancel()** function does not cancel asynchronous I/O requests for HFS, DDN or DSN files. The **aio_cancel()** function will always return `AIO_NOTCANCELED` for file descriptors associated with HFS, DDN or DSN files.

The **aio_cancel()** function dependes on pthreads for operation, and thus requires a POSIX environment.

RETURN VALUES

The **aio_cancel()** function returns -1 to indicate an error, or one of the following:

`[AIO_CANCELED]` All outstanding requests meeting the criteria specified were cancelled.

`[AIO_NOTCANCELED]` Some requests were not cancelled, status for the requests should be checked with aio_error(2).

`[AIO_ALLDONE]` All of the requests meeting the criteria have finished.

ERRORS

An error return from **aio_cancel()** indicates:

`[EBADF]` The *fildes* argument is an invalid file descriptor.

SEE ALSO

aio_error(2), aio_read(2), aio_return(2), aio_suspend(2), aio_write(2)

STANDARDS

The **aio_cancel()** function is expected to conform to the IEEE Std 1003.1 ("POSIX.1") standard.

AIO_ERROR(2)

NAME

aio_error – retrieve error status of asynchronous I/O operation

SYNOPSIS

```
#include <aio.h>

int
aio_error(const struct aiocb *iocb);
```

DESCRIPTION

The **aio_error()** function returns the error status of the asynchronous I/O request associated with the structure pointed to by *iocb*.

RETURN VALUES

If the asynchronous I/O request has completed successfully, **aio_error()** returns 0. If the request has not yet completed, `EINPROGRESS` is returned. If the request has completed unsuccessfully the error status is returned as described in read(2), write(2), or fsync(2) is returned.

On failure, **aio_error()** returns -1 and sets `errno` to indicate the error condition.

RESTRICTIONS

The **aio_error()** function dependes on pthreads for operation, and thus requires a POSIX environment.

ERRORS

The **aio_error()** function will fail if:

`[EINVAL]` The *iocb* argument does not reference an outstanding asynchronous I/O request.

SEE ALSO

aio_cancel(2), aio_read(2), aio_return(2), aio_suspend(2), aio_write(2), fsync(2), read(2), write(2)

STANDARDS

The **aio_error()** function is expected to conform to the IEEE Std 1003.1 ("POSIX.1") standard.

AIO_READ(2)

NAME

aio_read – asynchronous read from a file

SYNOPSIS

```
#include <aio.h>

int
aio_read(struct aiocb *iocb);
```

DESCRIPTION

The **aio_read()** function allows the calling process to read *iocb*`->aio_nbytes` from the descriptor *iocb*`->aio_fildes` beginning at the offset *iocb*`->aio_offset` into the buffer pointed to by *iocb*`->aio_buf`. The call returns immediately after the read request has been enqueued to the descriptor; the read may or may not have completed at the time the call returns.

The *iocb*`->aio_lio_opcode` argument is ignored by the **aio_read()** function.

The *iocb* pointer may be subsequently used as an argument to **aio_return()** and **aio_error()** in order to determine return or error status for the enqueued operation while it is in progress.

If the request could not be enqueued (generally due to invalid arguments), then the call returns without having enqueued the request.

If the request is successfully enqueued, the value of *iocb*`->aio_offset` can be modified during the request as context, so this value must not be referenced after the request is enqueued.

RESTRICTIONS

The Asynchronous I/O Control Block structure pointed to by *iocb* and the buffer that the *iocb*`->aio_buf` member of that structure references must remain valid until the operation has completed. For this reason, use of auto (stack) variables for these objects is discouraged.

The asynchronous I/O control buffer iocb should be zeroed before the **aio_read()** call to avoid passing bogus context information to the kernel.

Modifications of the Asynchronous I/O Control Block structure or the buffer contents after the request has been enqueued, but before the request has completed, are not allowed.

If the file offset in *iocb*->aio_offset is past the offset maximum for *iocb*->aio_fildes, no I/O will occur.

The **aio_read()** function dependes on pthreads for operation, and thus requires a POSIX environment.

RETURN VALUES

The **aio_read()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable errno is set to indicate the error.

ERRORS

The **aio_read()** function will fail if:

[EAGAIN] The request was not queued because of system resource limitations.

[ENOSYS] The **aio_read()** function is not supported.

The following conditions may be synchronously detected when the **aio_read()** function call is made, or asynchronously, at any time thereafter. If they are detected at call time, **aio_read()** returns -1 and sets errno appropriately; otherwise the **aio_return()** function must be called, and will return -1, and **aio_error()** must be called to determine the actual value that would have been returned in errno.

[EBADF] The *iocb*->aio_fildes argument is invalid.

[EINVAL] The offset *iocb*->aio_offset is not valid, the priority specified by *iocb*->aio_reqprio is not a valid priority, or the number of bytes specified by *iocb*->aio_nbytes is not valid.

[EOVERFLOW] The file is a regular file, *iocb*->aio_nbytes is greater than zero, the starting offset in *iocb*->aio_offset is before the end of the file, but is at or beyond the *iocb*->aio_fildes offset maximum.

If the request is successfully enqueued, but subsequently cancelled or an error occurs, the value returned by the **aio_return()** function is per the read(2) function, and the value returned by the **aio_error()** function is either one of the error returns from the read(2) function, or one of:

[EBADF] The *iocb*->aio_fildes argument is invalid for reading.

[ECANCELLED] The request was explicitly cancelled via a call to **aio_cancel()**.

[EINVAL] The offset *iocb*->aio_offset would be invalid.

SEE ALSO

aio_cancel(2), aio_error(2), aio_return(2), aio_suspend(2), aio_write(2)

STANDARDS

The **aio_read()** function is expected to conform to the IEEE Std 1003.1 ("POSIX.1") standard.

AIO_RETURN(2)

NAME

aio_return – retrieve return status of asynchronous I/O operation

SYNOPSIS

```
#include <aio.h>

int
aio_return(struct aiocb *iocb);
```

DESCRIPTION

The **aio_return()** function returns the final status of the asynchronous I/O request associated with the structure pointed to by iocb.

The **aio_return()** function should only be called once, to obtain the final status of an asynchronous I/O operation once aio_error(2) returns something other than `EINPROGRESS`.

RETURN VALUES

If the asynchronous I/O request has completed, the status is returned as described in read(2), write(2), or fsync(2). On failure, **aio_return()** returns -1 and sets `errno` to indicate the error condition.

RESTRICTIONS

The **aio_return()** function dependes on pthreads for operation, and thus requires a POSIX environment.

ERRORS

The **aio_return()** function will fail if:

`[EINVAL]` The *iocb* argument does not reference an outstanding asynchronous I/O request.

SEE ALSO

aio_cancel(2), aio_error(2), aio_suspend(2), aio_write(2), fsync(2), read(2), write(2)

STANDARDS

The **aio_return()** function is expected to conform to the IEEE Std 1003.1 ("POSIX.1") standard.

AIO_SUSPEND(2)

NAME

aio_suspend – suspend until asynchronous I/O operations or timeout complete

SYNOPSIS

```
#include <aio.h>

int
aio_suspend(const struct aiocb * const iocbs[], int niocb,
            const struct timespec * timeout);
```

DESCRIPTION

The **aio_suspend()** function suspends the calling process until at least one of the specified asynchronous I/O requests have completed, a signal is delivered, or the timeout has passed.

The *iocbs* argument is an array of *niocb* pointers to asynchronous I/O requests. Array members containing null pointers will be silently ignored.

If *timeout* is not a null pointer, it specifies a maximum interval to suspend. If *timeout* is a null pointer, the suspend blocks indefinitely. To effect a poll, the timeout should point to a zero-value timespec structure.

RETURN VALUES

If one or more of the specified asynchronous I/O requests have completed, **aio_suspend()** returns 0. Otherwise it returns -1 and sets `errno` to indicate the error, as enumerated below.

RESTRICTIONS

The **aio_suspend()** function dependes on pthreads for operation, and thus requires a POSIX environment.

ERRORS

The **aio_suspend()** function will fail if:

[EAGAIN]	The *timeout* expired before any I/O requests completed.
[EINVAL]	At least one of the requests specified in *iocbs* is invalid.
[EINTR]	the suspend was interrupted by a signal.

SEE ALSO

aio_cancel(2), aio_error(2), aio_return(2), aio_write(2)

AIO_WRITE(2)

NAME

aio_write – asynchronous write to a file

SYNOPSIS

```
#include <aio.h>

int
aio_write(struct aiocb *iocb);
```

DESCRIPTION

The **aio_write()** function allows the calling process to write *iocb*->aio_nbytes from the buffer pointed to by *iocb*->aio_buf to the descriptor *iocb*->aio_fildes. The call returns immediately after the write request has been enqueued to the descriptor; the write may or may not have completed at the time the call returns. If the request could not be enqueued, generally due to invalid arguments, the call returns without having enqueued the request.

If O_APPEND is set for *iocb*->aio_fildes, **aio_write()** operations append to the file in the same order as the calls were made. If O_APPEND is not set for the file descriptor, the write operation will occur at the absolute position from the beginning of the file plus *iocb*->aio_offset for supported files.

The *iocb* pointer may be subsequently used as an argument to **aio_return()** and **aio_error()** in order to determine return or error status for the enqueued operation while it is in progress.

If the request is successfully enqueued, the value of *iocb*->aio_offset can be modified during the request as context, so this value must not be referenced after the request is enqueued.

RESTRICTIONS

The Asynchronous I/O Control Block structure pointed to by *iocb* and the buffer that the *iocb*->aio_buf member of that structure references must remain valid until the operation has completed. For this reason, use of auto (stack) variables for these objects is discouraged.

The asynchronous I/O control buffer *iocb* should be zeroed before the **aio_write()** function to avoid passing bogus context information.

Modifications of the Asynchronous I/O Control Block structure or the buffer contents after the request has been enqueued, but before the request has completed, are not allowed.

If the file offset in *iocb*->aio_offset is past the offset maximum for *iocb*->aio_fildes, no I/O will occur.

The **aio_write()** function dependes on pthreads for operation, and thus requires a POSIX environment.

RETURN VALUES

The **aio_write()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable errno is set to indicate the error.

ERRORS

The **aio_write()** function will fail if:

[EAGAIN] The request was not queued because of system resource limitations.

[ENOSYS] The **aio_write()** function is not supported.

The following conditions may be synchronously detected when the **aio_write()** function call is made, or asynchronously, at any time thereafter. If they are detected at call time, **aio_write()** returns -1 and sets errno appropriately; otherwise the **aio_return()** function must be called, and will return -1, and **aio_error()** must be called to determine the actual value that would have been returned in errno.

[EBADF] The *iocb*->aio_fildes argument is invalid, or is not opened for writing.

[EINVAL] The offset *iocb*->aio_offset is not valid, the priority specified by *iocb*->aio_reqprio is not a valid priority, or the number of bytes specified by *iocb*->aio_nbytes is not valid.

If the request is successfully enqueued, but subsequently canceled or an error occurs, the value returned by the **aio_return()** function is per the write(2) function, and the value returned by the **aio_error()** is either one of the error returns from the write(2) function, or one of:

[EBADF] The *iocb*->aio_fildes argument is invalid for writing.

[ECANCELED] The request was explicitly canceled via a call to **aio_cancel()**.

[EINVAL] The offset *iocb*->aio_offset would be invalid.

SEE ALSO

aio_cancel(2), aio_error(2), aio_return(2), aio_suspend(2),

STANDARDS

The **aio_write()** function is expected to conform to the IEEE Std 1003.1 ("POSIX.1") standard.

CHDIR(2)

NAME

chdir, fchdir - change current `//HFS:`-style working directory

SYNOPSIS

```
#include <unistd.h>

int
chdir(const char *path);

int
fchdir(int fd);
```

DESCRIPTION

The *path* argument points to the pathname of a directory. The **chdir()** function causes the named directory to become the current working directory, that is, the starting point for path searches of pathnames not beginning with a slash, '/'.

The *path* argument must be an `//HFS:`-style file name.

The **fchdir()** function causes the directory referenced by *fd* to become the current working directory, the starting point for path searches of pathnames not beginning with a slash, '/'.

In order for a directory to become the current directory, a process must have execute (search) access to the directory.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

chdir() will fail and the current working directory will be unchanged if one or more of the following are true:

`[ENOTDIR]` A component of the path prefix is not a directory.

[EOPNOTSUPP] The file system containing the file named by *name1* does not support directories.

[EMLINK] The link count of the file named by *name1* would exceed 32767.

[ENAMETOOLONG] A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] The named directory does not exist.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EACCES] Search permission is denied for any component of the path name.

[EFAULT] *Path* points outside the process's allocated address space.

[EIO] An I/O error occurred while reading from or writing to the file system.

fchdir() will fail and the current working directory will be unchanged if one or more of the following are true:

[EACCES] Search permission is denied for the directory referenced by the file descriptor.

[ENOTDIR] The file descriptor does not reference a directory.

[EBADF] The argument *fd* is not a valid file descriptor.

SEE ALSO

chroot(2)

STANDARDS

The **chdir()** function call is expected to conform to ISO/IEC 9945-1:1990 ("POSIX.1").

CHMOD(2)

NAME

chmod, fchmod - change mode of an `//HFS:`-style file

SYNOPSIS

```
#include <sys/stat.h>

int
chmod(const char *path, mode_t mode);

int
fchmod(int fd, mode_t mode);
```

DESCRIPTION

The file permission bits of the file named specified by *path* or referenced by the file descriptor *fd* are changed to *mode*. The **chmod()** function verifies that the process owner (user) either owns the file specified by *path* (or *fd*), or is the super-user. The **chmod()** function follows symbolic links to operate on the target of the link rather than the link itself.

A mode is created from or'd permission bit masks defined in `<sys/stat.h>`:

```
#define S_IRWXU 0000700    /* RWX mask for owner */
#define S_IRUSR 0000400    /* R for owner */
#define S_IWUSR 0000200    /* W for owner */
#define S_IXUSR 0000100    /* X for owner */

#define S_IRWXG 0000070    /* RWX mask for group */
#define S_IRGRP 0000040    /* R for group */
#define S_IWGRP 0000020    /* W for group */
#define S_IXGRP 0000010    /* X for group */

#define S_IRWXO 0000007    /* RWX mask for other */
#define S_IROTH 0000004    /* R for other */
#define S_IWOTH 0000002    /* W for other */
#define S_IXOTH 0000001    /* X for other */

#define S_ISUID 0004000    /* set user id on execution */
#define S_ISGID 0002000    /* set group id on execution */
#define S_ISVTX 0001000    /* sticky bit */
```

```
#ifndef _POSIX_SOURCE
#define S_ISTXT 0001000
#endif
```

Setting the S_ISUID bit indicates that when the file is executed, the process's effective user-id is set to the file's owner user-id, so that the process appears to be running under the user-id of the file's owner.

Settin the S_ISGID bit indicates that when the file is executed, the process's effective group-id is that of file's owner group-id.

RETURN VALUE

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

chmod() will fail and the file mode will be unchanged if:

[ENOTDIR] A component of the path prefix is not a directory.

[ENAMETOOLONG] A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] The named file does not exist.

[EACCES] Search permission is denied for a component of the path prefix.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EPERM] The effective user ID does not match the owner of the file and the effective user ID is not the super-user.

[EROFS] The named file resides on a read-only file system.

[EFAULT] *Path* points outside the process's allocated address space.

[EIO] An I/O error occurred while reading from or writing to the file system.

fchmod() will fail if:

[EBADF] The descriptor is not valid.

`[EINVAL]`	*fd* refers to a socket, not to a file.
`[EROFS]`	The file resides on a read-only file system.
`[EIO]`	An I/O error occurred while reading from or writing to the file system.

SEE ALSO

chown(2), open(2), stat(2)

STANDARDS

The **chmod()** function call is expected to conform to ISO/IEC 9945-1:1990 ("POSIX.1").

CHOWN(2)

NAME

chown, fchown, lchown – change owner and group of an `//HFS:`-style file

SYNOPSIS

```
#include <unistd.h>

int
chown(const char *path, uid_t owner, gid_t group);

int
fchown(int fd, uid_t owner, gid_t group);

int
lchown(const char *path, uid_t owner, gid_t group);
```

DESCRIPTION

The owner ID and group ID of the file named by *path* or referenced by *fd* is changed as specified by the arguments *owner* and *group*. The owner of a file may change the *group* to a group of which he or she is a member, but the change *owner* capability is restricted to the super-user.

chown() clears the set-user-id and set-group-id bits on the file to prevent accidental or mischievous creation of set-user-id and set-group-id programs if not executed by the super-user. **chown()** follows symbolic links to operate on the target of the link rather than the link itself.

lchown() is similar to **chown()** but does not follow symbolic links.

One of the owner or group id's may be left unchanged by specifying it as -1.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

chown() and **lchown()** will fail and the file will be unchanged if:

`[ENOTDIR]` A component of the path prefix is not a directory.

`[ENAMETOOLONG]` A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

`[ENOENT]` The named file does not exist.

`[EACCES]` Search permission is denied for a component of the path prefix.

`[ELOOP]` Too many symbolic links were encountered in translating the pathname.

`[EPERM]` The effective user ID is not the super-user.

`[EROFS]` The named file resides on a read-only file system.

`[EFAULT]` *Path* points outside the process's allocated address space.

`[EIO]` An I/O error occurred while reading from or writing to the file system.

fchown() will fail if:

`[EBADF]` *fd* does not refer to a valid descriptor.

`[EINVAL]` *fd* refers to a socket, not a file.

`[EPERM]` The effective user ID is not the super-user.

`[EROFS]` The named file resides on a read-only file system.

`[EIO]` An I/O error occurred while reading from or writing to the file system.

SEE ALSO

chmod(2)

STANDARDS

The **chown()** function call is expected to conform to ISO/IEC 9945-1:1990 ("POSIX.1").

CHROOT(2)

NAME

chroot - change root directory

SYSNOPSIS

```
#include <unistd.h>

int
chroot(const char *dirname);
```

DESCRIPTION

dirname is the address of the pathname of an `//HFS:`-style directory, terminated by an ASCII NUL. **chroot()** causes *dirname* to become the root directory, that is, the starting point for path searches of pathnames beginning with '/'.

In order for a directory to become the root directory a process must have execute (search) access for that directory.

It should be noted that **chroot()** has no effect on the process's current directory.

This call is restricted to the super-user.

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and `errno` is set to indicate an error.

ERRORS

chroot() will fail and the root directory will be unchanged if:

`[ENOTDIR]` A component of the path name is not a directory.

`[EPERM]` The effective user ID is not the super-user, or one or more file descriptors are open directories.

`[ENAMETOOLONG]` A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

`[ENOENT]` The named directory does not exist.

`[EACCES]` Search permission is denied for any component of the path name.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EFAULT] *dirname* points outside the process's allocated address space.

[EIO] An I/O error occurred while reading from or writing to the file system.

SEE ALSO

chdir(2)

CLOCK_GETTIME(2)

NAME

clock_gettime, clock_settime, clock_getres – get/set/calibrate date and time

SYNOPSIS

```
#include <sys/time.h>

int
clock_gettime(clockid_t clock_id, struct timespec *tp);

int
clock_settime(clockid_t clock_id, const struct timespec *tp);

int
clock_getres(clockid_t clock_id, struct timespec *tp);
```

DESCRIPTION

The **clock_gettime()** and **clock_settime()** allow the calling process to retrieve or set the value used by a clock which is specified by *clock_id*.

Only the `CLOCK_REALTIME` and `CLOCK_MONOTONIC` clocks are supported by this implementation. The *clock_id* argument can only be one of those values.

The structure pointed to by *tp* is defined in ¡sys/time.h¿ as:

```
struct timespec {
   time_t  tv_sec;         /* seconds */
   long    tv_nsec;        /* and nanoseconds */
};
```

The system TOD clock is set during the initial program load or via operator commands. The **clock_settime()** function verifies its arguments but always returns an -1 with `errno` set to `EPERM`.

The resolution (granularity) of a clock is returned by the **clock_getres()** system call. This value is placed in a (non-`NULL`) *tp.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

The following error codes may be set in `errno`:

`[EINVAL]`	The *clock_id* argument was not a valid value.
`[EFAULT]`	The **tp* argument address referenced invalid memory.
`[EPERM]`	The process is not allowed to set the time.

SEE ALSO

ctime(3)

STANDARDS

The **clock_gettime()**, **clock_settime()**, and **clock_getres()** system calls conform to IEEE Std 1003.1b-1993 ("POSIX.1").

CLOSE(2)

NAME

close - delete a descriptor

SYNOPSIS

```
#include <unistd.h>

int
close(int d);
```

DESCRIPTION

The **close()** call deletes a descriptor from the per-process object reference table. If this is the last reference to the underlying object, the object will be deactivated. For example, on the last close of a file the current seek pointer associated with the file is lost; on the last close of a socket(2) associated naming information and queued data are discarded.

When a process exits, all associated file descriptors are freed, but since there is a limit on active descriptors per processes, the **close()** function call is useful when a large quantity of file descriptors are being handled.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the global integer variable `errno` is set to indicate the error.

IMPLEMENTATION NOTES

When closing a file that has a partial written record, Systems/C will pad the record to the LRECL size and write the data. The padding byte used depends on how the file was opened. For `O_TEXT` files, the padding byte is the space character. For non-`O_TEXT` files, the padding byte is the NUL character, zero.

ERRORS

close() will fail if:

[EBADF] *d* is not an active descriptor.

[EINTR] An interrupt was received.

SEE ALSO

fcntl(2), open(2),

STANDARDS

The **close()** function call is expected to conform to IEEE Std1003.1-1990 ("POSIX"), as closely as possible given the host operating system environment.

__DCALL_ENV(2)

NAME

__dcall_env - retrieve direct-call environment pointer

SYNOPSIS

```
#include <machine/dcall.h>

void * __dcall_env(void)
```

DESCRIPTION

The **__dcall_env()** function returns the current environment pointer which can be used by subsequent

```
DCALL=SUPPLIED
```

functions.

The **__dcall_env()** function is typically used in the

```
DCALL=ALLOCATE
```

function to make the environment pointer available to the allocation function to save for later use.

__DCALL_SETRETREGVAL(2)

NAME

__dcall_setretregval - set the value of a register after invoking a DCALL routine.

SYNOPSIS

```
#include <machine/dcall.h>

void __dcall_setretregval(int reg, long val);
```

DESCRIPTION

The **__dcall_setretregval** function sets the value a register will have on return from the DCALL'd environment.

The *reg* parameter indicates the register number, and should be in the range 0 to 15. Values outside that range are ignored.

The *val* parameter indicates the value the register should have on return.

Overwritting register values that have architected uses (e.g. `R15`) is undefined and can have undesired results.

In an AMODE 64 environment, if the calling environment is AMODE 64 then *val* will represent the entire register. Otherwise if the caller was AMODE 24 or AMODE 31, then only the lower 32-bits of *val* will be set in the register *reg*.

The **__dcall_setretregval** function can be used to set a register value for returning to the calling environment, for example setting `R0` or `R1` when the DCALL'd function returns.

DDNFIND(2)

NAME

ddnfind, ddnext - determine DSN's associated with a DD name.

SYNOPSIS

```
#include <machine/syscio.h>

void *
ddnfind(char *ddn, char *dsn);

void *
ddnnext(void *token, char *dsn);
```

DESCRIPTION

The **ddnfind()** and **ddnnext()** functions are used to retrieve a DATA SET name (`DSN`) name(s) allocated to a DD name (`DD`). **ddnfind()** retrieves the first `DSN` associated with the `DD` *ddn*, and saves it in the location specified by *dsn*. **ddnfind()** returns a token, which is then passed to subsequent calls to **ddnnext()**. The Systems/C file prefix (`//DDN:`) should not be specified in the *ddn* parameter.

ddnnext() is used to retrieve subsequent `DSN`s associated with the `DD` *ddn*. **ddnnext()** accepts the *token* created from the **ddnfind()** function, and returns the token that should be used on a subsequent call to **ddnnext()**.

When the list of `DSN`s has been exhausted, **dnnext()** returns `NULL` and releases the space associated with *token*.

The storage allocated for *dsn* should be sufficiently large to contain any valid `DSN` name. The Systems/C file prefix (`//DSN:`) is not returned in *dsn*.

IMPLEMENTATION NOTES

The **ddnfind()** and **ddnnext()** functions examine the job `JFCB` control block to determine the associated `DSN`. If the `DD` is associated with a HFS file, then the returned name will appear as "`...PATH=.SPECIFIED...`".

RETURN VALUES

ddnfind() returns a pointer to the allocate token if the *ddn* is located, `NULL` if the *ddn* does not exist, or `(void *)(-1)` if there insufficient space to allocate the token. If successful, **ddnfind()** places the first DSN name in the storage addressed by *dsn*.

ddnnext() returns the *token* and places the DSN name in the storage addressed by *dsn*. At the end of the list of DSN names, **ddnnext()** returns *NULL*.

SEE ALSO

osddinfo(2)

__DYNALL(2)

NAME

dynalloc - allocate a data set

SYNOPSIS

```
#include <machine/dynit.h>

int
__dynall(__dyn_t *parms);

int
dynalloc(__dyn_t *parms);

int
__dynfre(__dyn_t *parms);

int
dynfree(__dyn_t *parms);

void
__dyninit(__dyn_t *parms);
```

DESCRIPTION

The **__dynall** function is used to dynamically allocate MVS data sets, **__dynfre** is used to unallocate an MVS data set. **dynalloc** is an alias for **__dynall** and **dynfree** is an alias for **__dynfre**.

__dynall creates the `SVC 99` parameter list based on the fields of the incoming *parms* structure and then employs the `SVC 99` facility to invoke the allocate function.

__dynfre creates the `SVC 99` parameter list based on the fields of the incoming *parms* structure and then employs the `SVC 99` facility to invoke the deallocate function.

In each case, the *parms* argument points to a *__dyn_t* structure that contains the following fields:

`char *ddname` DD name. If the string is 8 question marks, then it indicates the area where the system-generated ddname is returned, otherwise the string is truncated at 8 characters and upper-cased before being passed to the SVC 99 interface.

`char *dsname` data set name. The string has a maximum length of 1023 characters and is upper cased before being passed to the SVC 99 interface.

`int sysout` sysout dataset, set to `__DEF_CLASS` for the default SYSOUT class

`char *sysoutname` program name for SYSOUT

`char *member` member name of a PDS/PDSE

`int status` data set status, can be one of the following:

`__DISP_OLD`
`__DISP_MOD`
`__DISP_NEW`
`__DISP_SHR`

`int normdisp` data set's normal disposition, can be on of:

`__DISP_UNCATLG`
`__DISP_CATLG`
`__DISP_DELETE`
`__DISP_KEEP`

`int conddisp` data set's conditional disposition, can be one of the following:

`__DISP_UNCATLG`
`__DISP_CATLG`
`__DISP_DELETE`
`__DISP_KEEP`

`char *unit` unit name

`char *volser` a comma-separated list of volume serial numbers

`int volseq` volume sequence number

`int volcount` maximum volume count

`int label` type of tape label , can be one of the following:

`__LABEL_NL`	no label
`__LABEL_SL`	IBM standard label
`__LABEL_NSL`	non-standard label
`__LABEL_SUL`	both IBM standard and user label
`__LABEL_BLP`	bypass label processing
`__LABEL_LTM`	check and bypass leading tape mark
`__LABEL_AL`	ANSI standard label
`__LABEL_AUL`	ANSI standard and user label

`int dsorg` data set organization, can be one of the following:

`__DSORG_unknown`	unknown organization
`__DSORG_U`	unmoveable
`__DSORG_VSAM`	VSAM
`__DSORG_GS`	graphics
`__DSORG_PO`	partioned organization
`__DSORG_POU`	partioned organization unmoveable
`__DSORG_MQ`	message processing queue
`__DSORG_CQ`	direct access message queue
`__DSORG_CX`	communication line group
`__DSORG_DA`	direct access
`__DSORG_DAU`	direct access unmoveable
`__DSORG_PS`	phsyical sequential
`__DSORG_PSU`	phsyical sequential unmoveable
`__DSORG_IS`	indexed sequential (deprecated)
`__DSORG_ISU`	indexed sequential unmoveable (deprecated)

`int alcunit` unit of space allocation, one of the following:

`__CYL`	Cylinders
`__TRK`	Tracks

`int primary` primary space allocation

`int secondary` secondary space allocation

`int recfm` record format, one of, or a combination of the following,

`_M_`
`_A_`
`_S_`
`_B_`
`_D_`
`_V_`
`_F_`
`_U_`
`_FB_`
`_VB_`
`_FBS_`
`_VBS_`

`long long blksize` block size

`int lrecl` logical record length, 0x8000 indicates 'X' for BSAM and QSAM allocations.

`char *volrefds` volume serial reference

`char *dcbrefds` DSNAME for DCB reference

`char *dcbrefdd` DDNAME for DCB reference

`unsigned int flags` miscellaneous flags, a combination of the following:

`__CLOSE`	close on free
`__RELEASE`	release unused space
`__PERM`	
`__CONTIG`	request contiguous space
`__ROUND`	round allocation sizes
`__TERM`	device is a terminal
`__DUMMY_DSN`	
`__HOLDQ`	
`__WAIT`	

`char *password` data set password

`int dirblk` number of directory blocks

`int avgblk` average block length

`char *storclass` SMS storage class

`char *mgntclass` SMS management class

`char *dataclass` SMS data class

`int recorg` VSAM dataset organization , one of the following:

`__KS`	
`__ES`	
`__RR`	
`__LS`	

`int keylength` VSAM key length

`int keyoffset` VSAM key offset

`int rls` VSAM record level sharing flags, one of the following:

`__RLS_NRI`	no read integrity
`__RLS_CR`	consistent read
`__RLS_CRE`	consistent read explicit

`char *refdd` copy attributes from referenced DDNAME

`char *like` copy attributes from DSNAME

`int dsntype` Type attribute of PDS or PDSE, one of the following:

- `__DSNT_LARGE` large format, greater than 65535 trks
- `__DSNT_BASIC` basic format data set
- `__DSNT_EXTPREF` extended format preferred
- `__DSNT_EXTREQ` extended format required
- `__DSNT_HFS` HFS file system
- `__DSNT_PIPE` FIFO special pipe
- `__DSNT_PDS` PDS
- `__DSNT_LIBRARY` PDSE

`char *pathame` path name

`int pathopts` path options, one of the following:

- `__PATH_OSYNC`
- `__PATH_OCREAT`
- `__PATH_OEXCL`
- `__PATH_ONOCTTY`
- `__PATH_OTRUNC`
- `__PATH_OAPPEND`
- `__PATH_ONONBLOCK`
- `__PATH_ORDWR`
- `__PATH_ORDONLY`
- `__PATH_OWRONLY`

`int pathmode` path mode, one or a combination of the following:

- `__PATH_SISUID`
- `__PATH_SIGUID`
- `__PATH_SIRUSR`
- `__PATH_SIWUSR`
- `__PATH_SIXUSR`
- `__PATH_SIRWXU`
- `__PATH_SIRGRP`
- `__PATH_SIWGRP`
- `__PATH_SIXGRP`
- `__PATH_SIWRXG`
- `__PATH_SIROTH`

__PATH_SIWOTH

__PATH_SIXOTH

__PATH_SIWRXO

`int pathndisp` path normal disposition, can be one of the following:

__DISP_DELETE

__DISP_KEEP

`int pathcdisp` path conditional disposition , can be one of the following:

__DISP_DELETE

__DISP_KEEP

`char * __ptr31 * __ptr31 miscitems` extra text units

`struct __S99RBX * __ptr31 rbx` SVC99 RBX (request block extension) pointer

`struct __S99EMPARMS * __ptr31 emsparmlist` pointer to messages

`int infocode` SVC 99 returned info code

`int errcode` SVC 99 returned error code

The *__dyn_t* structure must be initialized before invoking **__dynall** or **__dynfre**. This is accomplised using the `__dyninit` macro, or by using the `__DYN_T_INITIALIZER` macro. Unpredictable results may occur if the structure isn't properly initialized.

The *miscitems*, *rbx* and *emsparmlist* fields can be used to pass additional information to underlying SVC99 service. For more information about these, and the underlying SVC99 service, consult the IBM "z/OS MVS Programm Authorized Assembler Services Guide" and the __svc99(2) documentation.

The **__dynfre** function deallocates a z/OS data set based on the values passed via the *parms* parameter. The only fields in the given *__dyn_t* structure used by **__dynfre** are:

```
char *ddname

char *dsname

char *member

char *pathname

char *normdisp

char *pathndisp

char *miscitems
```

all other fields are ignored.

EXAMPLES

This program dynamically allocates a file named "MYNAME.MY.DATASET", with an allocation unit of CYL, a primary quantity of 2 and a secondary quantity of 1, with a logical record length of 121, a block size of 12100 and a fixed record ASA format.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <machine/dynit.h>
#include <machine/svc99.h>


int main () {
   __dyn_t ip;

   __dyninit(&ip);

   ip.ddname = "mydd";                /* MYDD DD                  */
   ip.dsname = "MYNAME.MY.DATASET";   /* DSN='MYNAME.MY.DATASET'  */
   ip.status = __DISP_NEW;            /* DISP=(NEW,CATLG)         */
   ip.normdisp = __DISP_CATLG;
   ip.alcunit = __CYL;                /* SPACE=(CYL,(2,1)),       */
   ip.primary = 2;
   ip.secondary = 1;
   ip.dirblk = 1;
   ip.flags = __RELEASE & __CONTIG;  /*       RLSE,CONTIG)        */
   ip.dsorg = __DSORG_PO;             /* DCB=(DSORG=PO,           */
   ip.recfm = _F_ + _B_ + _A_;        /*        RECFM=FBA,        */
   ip.lrecl = 121;                    /*      LRECL=121,          */
   ip.blksize = 12100;                /*      BLKSIZE=12100)      */

   if (dynalloc(&ip) != 0) {
       int err, inf;
       err = ip.errcode;
       inf = ip.infocode;
       printf("Dynalloc failed with error code 0x%04x (%d), "
              "info code 0x%04x (%d)\n", err, inf);
   }
}
```

To deallocate a file:

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>
#include <machine/dynit.h>

int
main(void)
{
  __dyn_t ip = __DYN_T_INITIALIZER;

  ip.ddname = "mydd";
  dynfree(&ip);
}
```

RETURN VALUES

The **dynalloc()** function returns -1 if it was unable to allocate enough memory to build the parameters for the `__svc99()` function.

Otherwise, it returns the return code from the invocation of `__svc99()`.

ISSUES

The `dynalloc` and `dynfree` functions are only available on z/OS.

SEE ALSO

"z/OS MVS Programm Authorized Assembler Services Guide", __malloc31(3), __svc99(3).

DUP(2)

NAME

dup, dup2 - duplicate an existing file descriptor

SYNOPSIS

```
#include <unistd.h>

int
dup(int olddd)

int
dup2(int olddd, int newdd)
```

DESCRIPTION

dup() duplicates an existing object descriptor and returns its value to the calling process (*newd* = dup(*oldd*)). The argument *oldd* is a small non-negative integer index in the per-process descriptor table. The value must be less than the size of the table, which is returned by getdtablesize(2). The new descriptor returned by the call is the lowest numbered descriptor currently not in use by the process.

The object referenced by the descriptor does not distinguish between *oldd* and *newd* in any way. Thus if *newd* and *oldd* are duplicate references to an open file, read(2), write(2) and lseek(2) calls all move a single pointer into the file, and append mode, non-blocking I/O and synchronous I/O options are shared between the references. If a separate pointer into the file is desired, a different object reference to the file must be obtained by issuing an additional open(2) call.

In **dup2()**, the value of the new descriptor *newd* is specified. If this descriptor is already in use and *oldd* `!=` *newd*, the descriptor is first deallocated as if a close(2) call had been used. If *oldd* is not a valid descriptor, then *newd* is not closed. If *oldd* `==` *newd* and *oldd* is a valid descriptor, then **dup2()** is successful, and does nothing.

RETURN VALUES

The value `-1` is returned if an error occurs in either call. The external variable `errno` indicates the cause of the error.

ERRORS

dup() and **dup2()** fail if:

[EBADF]	*oldd* or *newd* is not a valid active descriptor.
[EMFILE]	Too many descriptors are active.
[ENOMEM]	Insufficient memory was available.

SEE ALSO

close(2), fcntl(2), getdtablesize(2), open(2)

STANDARDS

The **dup()** and **dup2()** function calls are expected to conform to IEEE Std1003.1-1990 ("POSIX"), as closely as possible given the constraints of the host operating system.

EXECVE(2)

NAME

execve - execute a file

SYNOPSIS

```
#include <unistd.h>

int
execve(const char *path, char *const argv[], char *const envp[]);
```

DESCRIPTION

execve() transforms the calling process into a new process. The new process is constructed from an ordinary `//HFS:`-style file, whose name is pointed to by *path*, called the new process file. This file is either an executable object file, or a file of data for an interpreter.

An interpreter file begins with a line of the form:

```
#! <interpreter> [<arg>]
```

When an interpreted file is is **execve()**'d, the system actually **execve**'s the specified *interpreter*. If the optional *arg* is specified, it becomes the first argument to the *interpreter*, and the name of the originally **execve()**'d file becomes the second argument; otherwise, the name of the originally **execve()**'d file becomes the first argument. The original arguments are shifted over to become the subsequent arguments. The zero'th argument is set to the specified interpreter.

The argument *argv* is a pointer to a `NULL`-terminated array of character pointers to nul-terminated character strings. These strings construct the argument list to be made available to the new process. At least one argument must be present in the array; by custom, the first element should be the name of the executed program (for example, the last component of *path*).

The argument *envp* is also a pointer to a `NULL`-terminated array of character pointers to nul-terminated strings. A pointer to this array is normally stored in the global variable `environ`. These strings pass information to the new process that is not directly an argument to the command.

`//HFS:`-style file descriptors open in the calling process image remain open in the new process image, except for those for which the close-on-exec flag is set (see close(2)

and fcntl(2)). File descriptors not associated with `//HFS:`-style files are closed as if the close-on-exec flag was set. Descriptors that remain open are unaffected by **execve()**.

Signals set to be ignored in the calling process are set to be ignored in the new process. Signals which are set to be caught in the calling process image are set to default action in the new process image. Blocked signals remain blocked regardless of changes to the signal action. The signal stack is reset to be undefined.

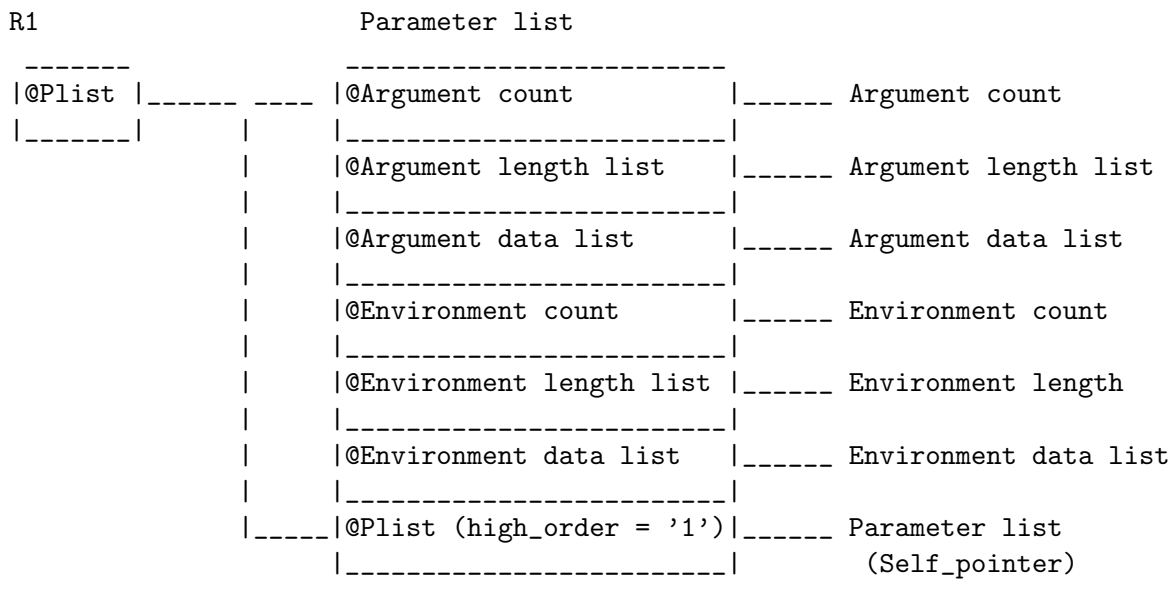
If the set-user-ID mode bit of the new process image file is set (see chmod(2)), the effective user ID of the new process image is set to the owner ID of the new process image file. If the set-group-ID mode bit of the new process image file is set, the effective group ID of the new process image is set to the group ID of the new process image file. (The effective group ID is the first element of the group list.) The real user ID, real group ID and other group IDs of the new process image remain the same as the calling process image. After any set-user-ID and set-group-ID processing, the effective user ID is recorded as the saved set-user-ID, and the effective group ID is recorded as the saved set- group-ID. These values may be used in changing the effective IDs later (see setuid(2)).

The set-ID bits are not honored if the respective file system has the `SSTFNOSUID` option enabled or if the new process file is an interpreter file.

The new process also inherits the following attributes from the calling process:

process ID	see getpid(2)
parent process ID	see getppid(2)
process group ID	see getpgrp(2)
access groups	see getgroups(2)
working directory	see chdir(2)
root directory	see chroot(2)
control terminal	
resource usages	see getrusage(2)
interval timers	
resource limits	see getrlimit(2)
file mode mask	see umask(2)
signal mask	

When a program is executed as a result of an **execve()** call, the lower-level service passes a parameter list, which is pointed to by regiter 1. The parameter list consists of the following parameter addresses, with the high-order bit set in the last value.



The Systems/C runtime recognizes this entry style and transforms the parameters into the standard:

```
main(argc, argv, envp)
int argc;
char **argv, **envp;
```

where *argc* is the number of elements in *argv* (the "arg count") and *argv* points to the array of character pointers to the arguments themselves.

For entry into Systems/C programs, the *argv* and *envp* array elements are assumed to be nul-terminated.

RETURN VALUES

As the **execve()** function overlays the current process image with a new process image the successful call has no process to return to. If **execve()** does return to the calling process an error has occurred; the return value will be -1 and the global variable `errno` is set to indicate the error.

ERRRORS

execve() will fail and return to the calling process if:

`[ENOTDIR]` A component of either path prefix is not a directory.

[ENAMETOOLONG] A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENAMETOOLONG] When invoking an interpreted script, the interpreter name exceeds MAXSHELLCMDLEN characters.

[ENOENT] The new process file does not exist.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EACCES] Search permission is denied for a component of the path prefix.

[EACCES] The new process file is not an ordinary file.

[EACCES] The new process file mode denies execute permission.

[ENOEXEC] The new process file has the appropriate access permission, but is not in the proper format to be a process image.

[ENOMEM] The new process requires more virtual memory than is allowed by the imposed maximum.

[E2BIG] The number of bytes in the new process' argument list is larger than the system-imposed limit.

[EFAULT] *Path*, *argv*, or *envp* point to an illegal address.

[EIO] An I/O error occurred while reading from the file system.

SEE ALSO

fork(2), _exit(2), execl(3), exit(3), The BPX1EXC service in the IBM publication "OpenEdition Assembler Callable Services".

_EXIT(2)

NAME

_exit - terminate the calling program

SYNOPSIS

```
#include <unistd.h>

void
_exit(int status);
```

DESCRIPTION

The **_exit()** function terminates a program with the following consequences:

- All of the descriptors open in the calling process are closed. This may entail delays, for example, waiting for output to drain.
- All allocated memory for the programs stack and heap space is released.
- For OpenEdition (POSIX) programs, if the parent OpenEdition process of the calling process has an outstanding wait(2) call or catches the `SIGCHLD` signal, it is notified of the calling process's termination and the status is set as defined by wait(2).
- For OpenEdition (POSIX) programs, the parent process-ID of all of the calling process's existing child processes are set to 1; the initialization process inherits each of these processes.
- For OpenEdition (POSIX) programs, if the termination of the process causes any process group to become orphaned (usually because the parents of all members of the group have now exited), and if any member of the orphaned group is stopped, the `SIGHUP` signal and the `SIGCONT` signal are sent to all members of the newly-orphaned process group.
- For OpenEdition (POSIX) programs, if the process is a controlling process, the `SIGHUP` signal is sent to the foreground process group of the controlling terminal, and all current access to the controlling terminal is revoked.
- For DCALL environments, the environment is destroyed and cannot be used again via DCALL=SUPPLIED.

Most C programs call the library routine exit(3), which flushes buffers, closes streams, unlinks temporary files, etc., before calling **_exit()**.

RETURN VALUES

_exit() can never return.

SEE ALSO

fork(2), wait(2), exit(3)

STANDARDS

The **_exit()** function call is expected to conform to ISO/IEC 9945-1:1990 ("POSIX.1") as much as the host system allows.

FCNTL(2)

NAME

fcntl - file control

SYNOPSIS

```
#include <fcntl.h>

int
fcntl(int fd, int cmd, ...)
```

DESCRIPTION

fcntl() provides for control over descriptors. The argument *fd* is a descriptor to be operated on by *cmd* as described below. Depending on the value of *cmd*, fcntl can take an additional third argument `int arg`.

`F_GETFL`	Get descriptor status flags, as described below (*arg* is ignored).
`F_SETFL`	Set descriptor status flags to arg.
`F_GETFD`	Get the file descriptor flags (`FD_LEAVEONCLOSE` or `FD_FREEONCLOSE`, or both) associated with the DSN/DDN file descriptor.
`F_SETFD`	Set the leave-on-close (`FD_LEAVEONCLOSE`) and/or free-on-close (`FD_FREEONCLOSE`) flags associated with the DSN/DDN file descriptor. If the `FD_LEAVEONCLOSE` bit is set in *arg*, then when the associated file is closed, the `LEAVE` option will be specified on the MVS `CLOSE` macro. If the `FD_FREEONCLOSE` bit is set in *arg*, then when the associated file is closed, the `FREE` option will be specified on the MVS `CLOSE` macro.

RETURN VALUES

Upon successful completion, the value returned depends on *cmd* as follows:

`F_GETFL`	Value of flags.
`F_GETFD`	Value of flags.
other	Value other than `-1`.

Otherwise, a value of `-1` is returned and `errno` is set to indicate the error.

ERRORS

fcntl() will fail if:

`[EBADF]` *fd* is not a valid open file descriptor.

SEE ALSO

close(2), getdtablesize(2), open(2)

FLDATA(2)

NAME

fldata - retrieve low-level file information

SYNOPSIS

```
#include <machine/syscio.h>

int
fldata(int fd, char *buf, int bufsize, fldata_t *info);
```

DESCRIPTION

The **fldata()** function examines the open file descriptor *fd* and returns information about the file. If the file descriptor was generated by a call to open(2), (or indirectly via fopen(3)), then **fldata()** returns the original name specified on the **open()** function call in *buf*, up to *bufsize* characters. The **fldata()** function does not append a `NUL` character to *buf*.

fldata() also sets various fields of the `fldata_t` structure with information from the open file. The `fldata_t` structure (shown below) is defined in `<machine/syscio.h>`.

```
typedef struct __fileData {
         /* Record formats */
        unsigned int __recfmF:1;        /* Fixed Records */
        unsigned int __recfmV:1;        /* Variable Records */
        unsigned int __recfmU:1;        /* Undefined Records */
        unsigned int __recfmS:1;        /* Spanned */
        unsigned int __recfmBlk:1;      /* Blocked data set */
        unsigned int __recfmASA:1;      /* ASA print control characters */
        unsigned int __recfmM:1;        /* Machine control character */
          /* Data Set organization */
        unsigned int __dsorgPO:1;       /* PDS */
        unsigned int __dsorgPDSmem:1;   /* PDS member specified on open */
        unsigned int __dsorgPDSdir:1;
        unsigned int __dsorgPS:1;       /* sequential file */
        unsigned int __dsorgConcat:1;
        unsigned int __dsorgMem:1;
        unsigned int __dsorgHiper:1;
        unsigned int __dsorgTemp:1;
        unsigned int __dsorgVSAM:1;
        unsigned int __dsorgHFS:1;      /* HFS file */
```

```
        unsigned int __dsorgPDSE:1;
          /* How was the file opened? */
        unsigned int __openmode:2;
        unsigned int __modeflag:4;
        unsigned int __vsamRLS:3;
        unsigned int __reserv1:8;

        __device_t __device;
        unsigned long __blksize;
        unsigned long __maxreclen;
        unsigned short __vsamtype;
        unsigned long __vsamkeylen;
        unsigned long __vsamRKP;
        char * __dsname;
        unsigned int __reserv2;
} fldata_t;
```

The fields of `fldata_t` are as follows:

`__recfmF`	Set to 1 for fixed-length records
`__recfmV`	Set to 1 for variable-length records
`__recvmU`	Set to 1 for undeifned-length records
`__recfmS`	Set to 1 for spanned records
`__recfmBlk`	Set to 1 for blocked records
`__recfmASA`	Set to 1 if the file uses ASA print-control characters
`__recfmM`	Set to 1 if the file uses machine print-control characters
`__dsorgPO`	Set to 1 for a PDS file
`__dsorgPDSmem`	Set to 1 for PDS members
`__dsorgPDSdir`	Set to 1 for PDS or PDSE directory
`__dsorgPS`	Set to 1 for sequential files
`__dsorgConcat`	Set to 1 for concatenated sequential files
`__dsorgHFS`	Set to 1 for HFS files.
`__dsorgPDSE`	Set to 1 if the file is a PDSE
`__openmode`	How the files was opened, one of `__TEXT`, `__BINARY` or `__RECORD`
`__modeflag`	How the file is altered or used, can be `__APPEND`, `__READ`, `__UPDATE`, `__WRITE`. These values can be logically OR'd together.

`__device`	The low-level "device driver" handling this file, one of `__DISK`, `__TERMINAL`, `__SOCKET` or `__HFS`
`__blksize`	Block size of the file
`__maxreclen`	Record length of the file (1-32760); or 2147483647 for an LRECL=X Variable Spanned file.
`__dsname`	For `//DDN:`-style files, this is set to a pointer to the `NUL`-terminated DSN-name associated with the file. If the `DD`-name is a concatenation, this contains the first DSN-name in the concatenation. If the name passed to open(2) was not a `//DDN:`-style name, this field will be `NULL`.

RETURN VALUES

If successful, **fldata()** returns the number of characters copied into *buf* (which may be zero.) Otherwise, **fldata()** returns -1 and sets the global variable `errno` to indicate the error.

ERRORS

fldata() will fail if:

`[EBADF]`	*fd* is not a valid descriptor.
`[EFAULT]`	Either *buf* or *info* specifies an invalid address.
`[ENOSYS]`	Couldn't determine the associated DSN name for a `//DDN:`-style name

Furthermore, for `//HFS:`-style files, **fldata()** can fail under the same conditions that fstat(2) can fail.

SEE ALSO

open(2), fstat(2), ddnfind(2), fileno(3)

ISSUES

The `__dsname` field is statically allocated in the library and should be saved between calls to **fldata**.

The `fldata_t` structure defines fields not currently supported by the Systems/C library (e.g. VSAM-related fields.) These are provided for compatibility with IBM's `fldata` function. Note that the IBM `fldata` function operates on `FILE` streams not file descriptors and has a slightly different parameter list.

FORK(2)

NAME

fork - create a new process

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

pid_t
fork(void);
```

DESCRIPTION

fork() causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process) except for the following:

- The child process has a unique process ID.
- The child process has a different parent process ID (i.e. the process ID of the parent process.)
- The child process has its own copy of the parent's descriptors. These descriptors reference the same underlying objects, so that, for instance, file pointers in file objects are shared between the child and the parent, so that an lseek(2) on a descriptor in the child process can affect a subsequent read(2) or write(2) by the parent. This descriptor copying is also used by the shell to establish standard input and output for newly created processes as well as to set up pipes.

  Any file descriptors associated with non `//HFS:`-style files are closed in the child process.
- The child process' resource utilizations are set to 0.
- All interval timers are cleared
- Any file locks previous set by the parent are not inherited by the child.
- The child has no pending signals.

RETURN VALUES

Upon successful completion, **fork()** returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of -1 is returned to the parent process, no child process is created, and the global variable `errno` is set to indicate the error.

ERRORS

fork() will fail and no child process will be created if:

`[EAGAIN]` The system-imposed limit on the total number of processes under execution would be exceeded.

`[EAGAIN]` The user is not the super user, and the "soft" resource limit on the number of per-user processes has been exhausted.

`[ENOMEM]` There is insufficient space for the new process.

SEE ALSO

execve(2), wait(2)

FSYNC(2)

NAME

fsync - synchronise changes to a file

SYNOPSIS

```
#include <unistd.h>

int
fsync(int fd);
```

DESCRIPTION

For `//HFS:`-style files, **fsync()** causes all modified data and attributes of fd to be moved to a permanent storage device. This normally results in all in-core modified copies of buffers for the associated file to be written to a disk.

fsync() should be used by programs that require a file to be in a known state, for example, in building a simple transaction facility.

Because of internal operating system buffering, for non-`//HFS:`-style files, the **fsync()** function fails with a -1 return code, and `errno` set to `EIO`.

RETURN VALUES

The **fsync()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

The **fsync()** function fails if:

`[EBADF]` *fd* is not a valid descriptor.

`[EINVAL]` *fd* refers to something that is not a regular file.

`[EIO]` An I/O error occurred while reading from or writing to the file system.

SEE ALSO

sync(2)

__GET_CPUID(2)

NAME

__get_cpuid() - return the IBM CPU identifier

SYNOPSIS

```
#include <machine/tiot.h>

int __get_cpuid(char *buff);
```

DESCRIPTION

The **__get_cpuid()** function returns the current CPU identifier as a nul-terminated string in the buffer addressed by *buff*. *Buff* must be at least 11 bytes long (10 bytes for the identifier, with a terminating zero.)

The CPU ID contains the serial number, followed by the model number.

RETURN VALUES

The **__get_cpuid()** function always returns the value 0. The CPU ID is contained in the string *buff*.

GETITIMER(2)

NAME

getitimer, setitimer – get/set value of interval timer

SYNOPSIS

```
#include <sys/time.h>
#define ITIMER_REAL      0
#define ITIMER_VIRTUAL   1
#define ITIMER_PROF      2

int
getitimer(int which, struct itimerval *value);

int
setitimer(int which, const struct itimerval *value,
         struct itimerval *ovalue);
```

DESCRIPTION

The system provides each process with three interval timers, defined in ¡sys/time.h¿. The **getitimer()** system call returns the current value for the timer specified in which in the structure at *value*. The **setitimer()** system call sets a timer to the specified *value* (returning the previous value of the timer if *ovalue* is not a null pointer).

A timer value is defined by the `itimerval` structure:

```
struct itimerval {
   struct  timeval it_interval;    /* timer interval */
   struct  timeval it_value;       /* current value */
};
```

If *it_value* is non-zero, it indicates the time to the next timer expiration. If *it_interval* is non-zero, it specifies a value to be used in reloading *it_value* when the timer expires. Setting *it_value* to 0 disables a timer, regardless of the value of *it_interval*. Setting *it_interval* to 0 causes a timer to be disabled after its next expiration (assuming *it_value* is non-zero).

Time values smaller than the resolution of the system clock are rounded up to this resolution.

The ITIMER_REAL timer decrements in real time. A SIGALRM signal is delivered when this timer expires.

The ITIMER_VIRTUAL timer decrements in process virtual time. It runs only when the process is executing. A SIGVTALRM signal is delivered when it expires.

The ITIMER_PROF timer decrements both in process virtual time and when the system is running on behalf of the process. It is designed to be used by interpreters in statistically profiling the execution of interpreted programs. Each time the ITIMER_PROF timer expires, the SIGPROF signal is delivered. Because this signal may interrupt in-progress system calls, programs using this timer must be prepared to restart interrupted system calls.

The maximum number of seconds allowed for *it_interval* and *it_value* in **setitimer()** is 100000000.

NOTES

Three macros for manipulating time values are defined in ¡sys/time.h¿. The `timerclear()` macro sets a time value to zero, `timerisset()` tests if a time value is non-zero, and `timercmp()` compares two time values.

The underlying IBM implementation uses the MVS `STIMERM` interface, if the number of concurrent `STIMERM SET` requests for the current task is exceeded, the program can abnormally end.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

The **getitimer()** and **setitimer()** system calls will fail if:

`[EFAULT]` The *value* argument specified a bad address.

`[EINVAL]` The *value* argument specified a time that was too large to be handled or was negative.

`[ENOSYS]` POSIX signals were not enabled for the program and are required for delivering the signal when the timer expires.

SEE ALSO

gettimeofday(2), select(2)

GETDTABLESIZE(2)

NAME

getdtablesize - get descriptor table size

SYNOPSIS

```
#include <unistd.h>

int
getdtablesize(void)
```

DESCRIPTION

Each process has a fixed size descriptor table, which is guaranteed to have at least 20 slots. The entries in the descriptor table are numbered with small integers starting at 0. The call **getdtablesize()** returns the size of this table.

SEE ALSO

close(2), dup(2), open(2)

GETGID(2)

NAME

getgid, getegid - get group process identification

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

gid_t
getgid(void);

gid_t
getegid(void);
```

DESCRIPTION

The **getgid()** function returns the real group ID of the calling process, **getegid()** returns the effective group ID of the calling process.

The real group ID is specified at login time.

The real group ID is the group of the user who invoked the program. As the effective group ID gives the process additional permissions during the execution of "set-group-ID" mode processes, **getgid()** is used to determine the real-user-id of the calling process.

ERRORS

As long as UNIX System Services are available, the **getgid()** and **getegid()** functions are always successful, and no return value is reserved to indicate an error.

SEE ALSO

getuid(2), setgid(2), setregid(2)

STANDARDS

The **getgid()** and **getegid()** function calls are expected to conform to ISO/IEC 9945-1:1990 ("POSIX.1"), as closely as the host system allows.

GETGROUPS(2)

NAME

getgroups - get group access list

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

int
getgroups(int gidsetlen, gid_t *gidset);
```

DESCRIPTION

getgroups() gets the current group access list of the user process and stores it in the array *gidset*. The parameter *gidsetlen* indicates the number of entries that may be placed in *gidset*. **getgroups()** returns the actual number of groups returned in *gidset*. No more than `NGROUPS_MAX` will ever be returned. If *gidsetlen* is zero, **getgroups()** returns the number of supplementary group IDs associated with the calling process without modifying the array pointed to by *gidset*.

RETURN VALUES

A successful call returns the number of groups in the group set. A value of -1 indicates that an error occurred, and the error code is stored in the global variable `errno`.

ERRORS

The possible errors for **getgroups()** are:

`[EINVAL]` The argument *gidsetlen* is smaller than the number of groups in the group set.

`[EFAULT]` The argument *gidset* specifies an invalid address.

SEE ALSO

setgroups(2)

GETLOGIN(2)

NAME

getlogin - get login name

SYNOPSIS

```
#include <unistd.h>

char *
getlogin(void);
```

DESCRIPTION

The **getlogin()** routine returns the login name of the user associated with the current session. The name is normally associated with a login step at the time a session is created, and is inherited by all processes descended from the login shell. (This is true even if some of those processes assume another user ID.)

RETURN VALUES

If a call to **getlogin()** succeeds, it returns a pointer to a NUL-terminated string in a static buffer, or NULL if the name has not been set.

ERRORS

If OpenEdition services are available, **getlogin()** should not fail. If OpenEdition services are available, and the request fails **getlogin()** will terminate the program with an abend.

STANDARDS

getlogin() conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

GETPID(2)

NAME

getpid, getppid - get parent or calling process identification

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

pid_t
getpid(void);

pid_t
getppid(void);
```

DESCRIPTION

getpid() returns the process ID of the calling process. Though the ID is guaranteed to be unique, it should NOT be used for constructing temporary file names, for security reasons; see mkstemp(3) instead.

getppid() returns the process ID of the parent of the calling process.

ERRORS

If OpenEdition services are available, the **getpid()** and **getppid()** functions are always succesful, and no return value is reserved to indicate an error.

If OpenEdition services are not available, **getpid()** and **getppid()** return zero, and the global variable `errno` is set to the value `ENOSYS`.

STANDARDS

The **getpid()** and **getppid()** function calls are expected to conform to ISO/IEC 9945-1:1990 ("POSIX.1") as closely as the host operating system allows.

GETPGRP(2)

NAME

getpgrp - get process group

SYNOPSIS

```
#include <unistd.h>

pid_t
getpgrp(void);

pid_t
getpgid(pid_t pid);
```

DESCRIPTION

The process group of the current process is returned by **getpgrp()**. The process group of the process identified by *pid* is returned by **getpgid()**. If *pid* is zero, **getpgid()** returns the process group of the current process.

Process groups are used for distribution of signals, and by terminals to arbitrate requests for their input: processes that have the same process group as the terminal are foreground and may read, while others will block with a signal if they attempt to read.

This call is thus used by programs to create process groups in implementing job control. The tcgetpgrp(3) and tcsetpgrp(3) calls are used to get/set the process group of the control terminal.

RETURN VALUES

The **getpgrp()** call always succeeds. Upon successful completion, the **getpgid()** call returns the process group of the specified process; otherwise, it returns a value of -1 and sets `errno` to indicate the error.

ERRORS

getpgrp() will succeed unless:

`[EPERM]` *pid* is not in the same session as the calling process

`[ESRCH]` there is no process whose process ID equals *pid*

SEE ALSO

getsid(2), setpgid(2)

STANDARDS

The **getpgrp()** function call is expected to conform to ISO/IEC 9945-1:1990 ("POSIX.1") as closely as the host system allows.

GETPRIORITY(2)

NAME

getpriority, setpriority - get/set program scheduling priority

SYNOPSIS

```
#include <sys/time.h>
#include <sys/resource.h>

int
getpriority(int which, int who);

int
setpriority(int which, int who, int prio);
```

DESCRIPTION

The scheduling priority of the process, process group, or user, as indicated by *which* and *who* is obtained with the **getpriority()** call and set with the **setpriority()** call. *Which* is one of `PRIO_PROCESS`, `PRIO_PGRP`, or `PRIO_USER`, and *who* is interpreted relative to *which* (a process identifier for `PRIO_PROCESS`, process group identifier for `PRIO_PGRP`, and a user ID for `PRIO_USER`). A zero value of *who* denotes the current process, process group, or user. *Prio* is a value in the range -20 to 19. The default priority is 0; lower priorities cause more favorable scheduling.

The **getpriority()** call returns the highest priority (lowest numerical value) enjoyed by any of the specified processes. The **setpriority()** call sets the priorities of all of the specified processes to the specified value. Only the super-user may lower priorities.

RETURN VALUES

Since **getpriority()** can legitimately return the value -1, it is necessary to clear the external variable `errno` prior to the call, then check it afterward to determine if a -1 is an error or a legitimate value.

The **setpriority()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

getpriority() and **setpriority()** will fail if:

[EINVAL]	*Which* was not one of PRIO_PROCESS, PRIO_PGRP, or PRIO_USER.
[EINVAL]	*Who* is not a alid process ID, group ID or user ID.
[ENOSYS]	The system does not support this function, or the installation has not enabled it.
[ESRCH]	No process was located using the which and who values specified.

In addition to the errors indicated above, **setpriority()** will fail if:

[EPERM]	A process was located, but neither its effective nor real user ID matched the effective user ID of the caller.
[EACCES]	A non super-user attempted to lower a process priority.

SEE ALSO

nice(3), fork(2)

GETPRV(2)

NAME

__getprv - return the current Pseudo Register Vector address

SYNOPSIS

```
#pragma map (__getprv,"@@GETPRV")
void *__getprv(void);
```

This function does not appear in any header file, thus, the `#pragma map` statement must be properly provided to use it.

DESCRIPTION

The **__getprv()** function returns the address of the current Pseudo Register Vector (PRV). The PRV contains global re-entrant data.

Typically **__getprv()** is used in conjunction with `#pragma prokley(...,"DCALL=ALLOCATE,PRV=0")` functions for creating stack environments that share the same global variables.

GETRUSAGE(2)

NAME

getrusage - get information about resource utilization

SYNOPSIS

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
#define   RUSAGE_SELF     0
#define   RUSAGE_CHILDREN     -1

int
getrusage(int who, struct rusage *rusage);
```

DESCRIPTION

getrusage() returns information describing the resources utilized by the current process, or all its terminated child processes. The *who* parameter is either `RUSAGE_SELF` or `RUSAGE_CHILDREN`. The buffer to which *rusage* points will be filled in with the following structure:

```
struct rusage {
        struct timeval ru_utime; /* user time used */
        struct timeval ru_stime; /* system time used */
}
```

The fields are interpreted as follows:

`ru_utime`	the total amount of time spent executing in user mode.
`ru_stime`	the total amount of time spent in the system executing on behalf of the process(es).

RETURN VALUES

The **getrusage()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **getrusage()** function will fail if:

`[EINVAL]` The *who* parameter is not a valid value. gitem[`[EFAULT]`] The address specified by the *rusage* parameter is not in a valid part of the process address space.

SEE ALSO

gettimeofday(2), wait(2)

GETSID(2)

NAME

getsid - get process session

SYNOPSIS

```
#include <unistd.h>

pid_t
getsid(pid_t pid);
```

DESCRIPTION

The session ID of the process identified by *pid* is returned by **getsid()**. If *pid* is zero, **getsid()** returns the session ID of the current process.

RETURN VALUES

Upon successful completion, the function **getsid()** returns the session ID of the specified process; otherwise, it returns a value of -1 and sets `errno` to indicate an error.

ERRORS

getsid() will succeed unless:

`[EPERM]` *pid* is not in the same session as the calling process.

`[ESRCH]` there is no process with a process ID equal to *pid*.

SEE ALSO

getpgid(2), getpgrp(2), setpgid(2), setsid(2)

GETTIMEOFDAY(2)

NAME

gettimeofday - get date and time

SYNOPSIS

```
#include <sys/time.h>

int
gettimeofday(struct timeval *tp, struct timezone *tzp);
```

DESCRIPTION

The system's notion of the current Greenwich time and the current time zone is obtained with the **gettimeofday()** call. The time is expressed in seconds and microseconds since midnight (0 hour), January 1, 1970. The resolution of the system clock is hardware dependent, and the time may be updated continuously or in "ticks". If *tp* or *tzp* is `NULL`, the associated time information will not be returned.

The structure pointed to by *tp* and *tzp* are defined in `<sys/time.h>` as:

```
struct timeval {
        long    tv_sec;         /* seconds since Jan. 1, 1970 */
        long    tv_usec;        /* and microseconds */
};


struct timezone {
        int     tz_minuteswest; /* minutes west of Greenwich */
        int     tz_dsttime;     /* type of dst correction */
};
```

The `timezone` structure indicates the local time zone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Daylight Saving time applies locally during the appropriate part of the year.

The Systems/C runtime on OS/390 and z/OS assumes the system clock is set to Greenwhich time (not local time), and uses the `CVTTZ` value to determine the time-zone offset.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

The following error codes may be set in `errno`:

`[EFAULT]` An argument address referenced invalid memory.

SEE ALSO

ctime(3)

GETUID(2)

NAME

getuid, geteuid - get user identification

SYNOPSIS

```
#include <unistd.h>
#include <sys/types.h>

uid_t
getuid(void);

uid_t
geteuid(void);
```

DESCRIPTION

The **getuid()** function returns the real user ID of the calling process. The **geteuid()** function returns the effective user ID of the calling process.

The real user ID is that of the user who has invoked the program. As the effective user ID gives the process additional permissions during execution of "set-user-ID" mode processes, **getuid()** is used to determine the real-user-id of the calling process.

ERRORS

As long as the UNIX System Services are available, the **getuid()** and **geteuid()** functions are always successful, and no return value is reserved to indicate an error.

SEE ALSO

getgid(2), setgid(2), setreuid(2), setuid(2)

STANDARDS

The **geteuid()** and **getuid()** function calls are expected to conform to ISO/IEC 9945-1:1990 ("POSIX.1"), as close as the host system allows.

GRANTPT(2)

NAME

grantpt - grant access to a slave pseudoterminal

SYNOPSIS

```
#include <stdlib.h>

int
grantpt(int filedes);
```

DESCRIPTION

The **grantpt()** function changes the ownership and mode of a slave pseudoterminal. *filedes* is a file descriptor that is the result of an open(2) of the corresponding master pseudoterminal.

Secure connections can be provided by using **grantpt()** and unlockpt(2), or by simply issuing the first open against the slave pseudoterminal from the userid or process that opened the master terminal.

RETURN VALUE

If successful, **grantpt()** returns the value 0, otherwise a -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

grantpt() will fail it:

`[EACCESS]` **grantpt()** has already been issued on this descriptor, or the slave pseudoterminal has already been opened.

`[EBADF]` *filedes* is invalid

`[EINVAL]` *filedes* is not a master pseudoterminal

`[ENOENT]` The slave pseudoterminal was not found.

SEE ALSO

ptsname(3), unlockpt(2)

IBMFD(2)

NAME

__ibmfd - return the current Pseudo Register Vector address

SYNOPSIS

```
#include <fcntl.h>

int __ibmfd(int fd);
```

DESCRIPTION

The **__ibmfd()** function returns the associated IBM BPX or SOCKET file descriptor number for the given *fd*.

The value of *fd* must come from a BPX socket-related function (e.g. **socket()** or **accept()**) or a call to **open()** specifying an HFS file.

This function can be used to map the file-descriptor number *fd* to it's underlying IBM file-descriptor number for direct calls to the lower-level IBM BPX interfaces.

Care must be taken when directly calling the low-level BPX interfaces, as the state of the file may be altered from the state managed by the Dignus runtime. For example, the Dignus runtime may consider a file descriptor to be "open", but a direct call to BPX1CLS could close the underlying IBM file-descriptor causing mysterious errors.

RETURN VALUE

If successful, **__ibmfd()** returns the value of the IBM file descriptor, otherwise a -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

__ibmfd() will fail it:

`[EBADF]` *fd* is invalid

`[EINVAL]` *fd* is not an HFS or SOCKET descriptor

SEE ALSO

open(3), socket(2), accept(2)

__ISPOSIXON(2)

NAME

__isPosixOn - determine if the OpenMVS functions are available

SYNOPSIS

```
#include <unistd.h>

int
__isPosixOn(void);
```

DESCRIPTION

The **__isPosixOn()** function returns 1 if the OpenMVS system functions are available and the program is executing in a POSIX environment, otherwise it returns 0.

__JOBNAME(2)

NAME

__jobname - return the current jobname

SYNOPSIS

```
#include <machine/tiot.h>

char *
__jobname(void);

char *
__jobname_r(char *buf);
```

DESCRIPTION

The **__jobname()** function returns the current jobname of the executing program on MVS, OS/390 and z/OS. The value returned is a pointer to a NUL-terminated string. Trailing blanks are removed from the name returned by the operating system.

__jobname() returns a pointer to a static area, care should be taken to copy this value before invoking **__jobname()** again and when using **__jobname()** in a multi-tasking environment.

__jobname_r() places the job name in the area addressed by *buf*. *buf* must be at least 9 characters in size. **__jobname_r()** returns *buf*.

SEE ALSO

__stepname(2), __procname(2), __userid(2) __querydub(2)

KILL(2)

NAME

kill - send signal to a program or process

SYNOPSIS

```
#include <sys/types.h>
#include <signal.h>

int
kill(pid_t pid, int sig);
```

DESCRIPTION

The **kill()** function sends the signal given by *sig* to *pid*, a process or a group of processes. *Sig* may be one of the valid signals, or it may be 0, in which case error checking is performed but no signal is actually sent. This can be used to check the validity of *pid*.

When running under OpenEdition, and *pid* is not the same process ID as the calling program, the `BPX1KIL` service is used to send the signal to a different process or process group. Otherwise, *pid* is ignored, and the signal is sent to the calling program.

For a process to have permission to send a signal to a process designated by *pid*, the real or effective user ID of the receiving process must match that of the sending process or the user must have appropriate privileges (such as given by a set-user-ID program or the user is the super-user). A single exception is the signal `SIGCONT`, which may always be sent to any descendant of the current process.

If *pid* is greater than zero, *sig* is sent to the process whose ID is equal to *pid*.

If *pid* is zero, *sig* is sent to all processes whose group is equal to the process group ID of the sender, and for which the process has permission.

If *pid* is -1, and the user has super-user privileges, the signal is sent to all processes excluding the process with ID 1 (usually init), and the process sending the signal. If the user is noto the super user, the signal is sent to tall processes with the same uid as the user excluding the process sending the signal. No error is returend if any process could be signaled.

If *pid* is negative, but not -1, the signal is sent to all processes whose process group ID is equal to the absolute value of *pid*.

If *sig* is `SIGABRT` and `SIGABRT` signals have not been caught via the **signal()** function, and the program is not running under OpenEdition, then a function call traceback will be generated on the `STDERR` stream, and the program will issue a user X'DCC' or 3532 `ABEND`.

RETURN VALUES

The **kill()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

kill() will fail and no signal will be sent if:

`[EINVAL]` *Sig* is not a valid signal number.

`[ESRCH]` No process can be found corresponding to that specified by *pid*.

`[ESRCH]` The process id was given as 0 but the sending process does not have a process group.

`[EPERM]` The sending process is not the super-user and its effective user id does not match the effective user-id of the receiving process. When signaling a process group, this error is returned if any members of the group could not be signaled.

SEE ALSO

getpgrp(2), getpid(2), raise(3)

STANDARDS

The **kill()** function call is expected to conform to ISO/IEC 9945-1:1990 ("POSIX.1") as closely as the host operating system allows.

LINK(2)

NAME

link - make a hard file link

SYNOPSIS

```
#include <unistd.h>

int
link(const char *name1, const char *name2);
```

DESCRIPTION

The **link()** function call atomically creates the specified directory entry (hard link) *name2* with the attributes of the underlying object pointed at by *name1*. If the link is successful, the link count of the underlying object is incremented, and *name1* and *name2* share equal access and rights to the underlying object.

If *name1* is removed, the file *name2* is not deleted and the link count of the underlying object is decremented.

Name1 must exist for the hard link to succeed and both *name1* and *name2* must be in the same file system. *name1* may not be a directory.

RETURN VALUES

The **link()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

link() will fail and no link will be created if:

`[ENOTDIR]` A component of either path prefix is not a directory.

`[ENAMETOOLONG]` A component of either pathname exceeded 255 characters, or entire length of either path name exceeded 1023 characters.

`[ENOENT]` A component of either path prefix does not exist.

[EOPNOTSUPP] The file system containing the file named by *name1* does not support links.

[EMLINK] The link count of the file named by *name1* would exceed 32767.

[EACCES] A component of either path prefix denies search permission.

[EACCES] The requested link requires writing in a directory with a mode that denies write permission.

[ELOOP] Too many symbolic links were encountered in translating one of the pathnames.

[ENOENT] The file named by *name1* does not exist.

[EEXIST] The link named by *name2* does exist.

[EPERM] The file named by *name1* is a directory.

[EXDEV] The link named by *name2* and the file named by *name1* are on different file systems.

[ENOSPC] The directory in which the entry for the new link is being placed cannot be extended because there is no space left on the file system containing the directory.

[EDQUOT] The directory in which the entry for the new link is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.

[EIO] An I/O error occurred while reading from or writing to the file system to make the directory entry.

[EROFS] The requested link requires writing in a directory on a read-only file system.

[EFAULT] One of the pathnames specified is outside the process's allocated address space.

SEE ALSO

pathconf(2), readlink(2), symlink(2), unlink(2)

STANDARDS

The **link()** function call is expected to conform to ISO/IEC 9945-1:1990 ("POSIX.1") as closely as the host operating system allows.

LIO_LISTIO(2)

NAME

lio_listio – list directed I/O

SYNOPSIS

```
#include <aio.h>

int
lio_listio(int mode, struct aiocb * const [] list, int nent,
           struct sigevent *sig);
```

DESCRIPTION

The **lio_listio()** function initiates a list of I/O requests with a single function call. The *list* argument is an array of pointers to `aiocb` structures describing each operation to perform, with *nent* elements. `NULL` elements are ignored.

The `aio_lio_opcode` field of each `aiocb` specifies the operation to be performed. The following operations are supported:

`[LIO_READ]` Read data as if by a call to aio_read(2).

`[LIO_NOP]` No operation.

`[LIO_WRITE]` Write data as if by a call to aio_write(2).

If the *mode* argument is `LIO_WAIT`, **lio_listio()** does not return until all the requested operations have been completed. If *mode* is `LIO_NOWAIT`, the requests are processed asynchronously, and the signal specified by *sig* is sent when all operations have completed. If *sig* is `NULL`, the calling process is not notified of I/O completion.

The order in which the requests are carried out is not specified; in particular, there is no guarantee that they will be executed in the order 0, 1, ..., *nent*-1.

RESTRICTIONS

The **lio_listio()** function dependes on pthreads for operation, and thus requires a POSIX environment.

RETURN VALUES

If *mode* is `LIO_WAIT`, the **lio_listio()** function returns 0 if the operations completed successfully, otherwise -1.

If *mode* is `LIO_NOWAIT`, the **lio_listio()** function 0 if the operations are successfully queued, otherwise -1.

ERRORS

The **lio_listio()** function will fail if:

`[EAGAIN]`	There are not enough resources to enqueue the requests.
`[EINVAL]`	The mode argument is neither `LIO_WAIT` nor `LIO_NOWAIT`.
`[EINTR]`	A signal interrupted the function before it could be completed.
`[EIO]`	One or more requests failed.

In addition, the **lio_listio()** function may fail for any of the reasons listed for aio_read(2) and aio_write(2).

If **lio_listio()** succeeds, or fails with an error code of `EAGAIN`, `EINTR` or `EIO`, some of the requests may have been initiated. The caller should check the error status of each `aiocb` structure individually by calling aio_error(2).

SEE ALSO

aio_error(2), aio_read(2), aio_write(2), read(2), write(2)

STANDARDS

The **lio_listio()** function is expected to conform to IEEE Std 1003.1-2001 ("POSIX.1").

LSEEK(2)

NAME

lseek - reposition read/write file offset

SYNOPSIS

```
#include <unistd.h>

off_t
lseek(int filedes, off_t offset, int whence)
```

DESCRIPTION

The **lseek()** function repositions the offset of the file descriptor *fildes* to the argument offset according to the directive *whence*. The argument *fildes* must be an open file descriptor. **lseek()** repositions the file position pointer associated with the file descriptor *fildes* as follows:

- If whence is `SEEK_SET`, the offset is set to offset bytes.
- If whence is `SEEK_CUR`, the offset is set to its current location plus offset bytes.
- If whence is `SEEK_END`, the offset is set to the size of the file plus offset bytes.

The **lseek()** function allows the file offset to be set beyond the end of the existing end-of-file of the file. If data is later written at this point, subsequent reads of the data in the gap return bytes of zeros (until data is actually written into the gap).

Some devices are incapable of seeking. The value of the pointer associated with such a device is undefined.

RETURN VALUES

Upon successful completion, **lseek()** returns the resulting offset location as measured in bytes from the beginning of the file. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

ERRORS

lseek() will fail and the file position pointer will remain unchanged if:

`[EBADF]`	*Fildes* is not an open file descriptor.
`[ESPIPE]`	*Fildes* is associated with a pipe, socket, or FIFO.
`[EINVAL]`	*Whence* is not a proper value.

SEE ALSO

dup(2), open(2)

ISSUES

This document's use of whence is incorrect English, but is maintained for historical reasons.

There are limitations to the Systems/C **lseek()** support for non-HFS files, due to implementing a byte offset file abstraction in the OS/390 and z/OS environments. `lseek(fd, 0, SEEK_CUR)` is supported for any file. This returns the internal byte count (the number of bytes read or written.) `lseek(fd, n, SEEK_CUR)` is supported if the corresponding `SEEK_SET` lseek operation is supported. That is, the value of *n* is added to the current position to determine a new offset. If lseek with the `SEEK_SET` option on the computed offset succeeds, this succeeds. `lseek(fd, 0, SEEK_SET)` succeeds on any non-HFS file for which the `NOTE` and `POINT` service is valid. `lseek(fd, n, SEEK_SET)` succeeds for any non-HFS file opened with the `O_RDONLY` mode and for which the `NOTE` and `POINT` service is valid. This will not extend the file size as the file is opened read-only. Seeking past the end of file on a read-only file will return -1 and set `errno` to `EINVAL`.

`lseek(fd, 0, SEEK_END)` is supported for non-HFS `O_RDONLY` files. This is can be an expensive operation because the entire file must be read to determine its length in bytes.

These limitations similarly affect the **fseek()** function, which uses **lseek()** in its implementation.

STANDARDS

The **lseek()** function call is expected to conform to IEEE Std1003.1-1990 ("POSIX") as closely as the host operating system allows.

MKDIR(2)

NAME

mkdir – make a directory file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

int
mkdir(const char *path, mode_t mode);
```

DESCRIPTION

The HFS directory *path* is created with the access permissions specified by *mode* and restricted by the umask(2) of the calling process. *path* must be an `//HFS:`-style file name.

The directory's owner ID is set to the process's effective user ID.

RETURN VALUES

The **mkdir()** unction returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

mkdir() will fail and no directory will be created if:

`[ENOTDIR]` A component of the path prefix is not a directory.

`[ENAMETOOLONG]` A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

`[ENOENT]` A component of the path prefix does not exist.

`[EACCES]` Search permission is denied for a component of the path prefix, or write permission is denied on the parent directory of the directory to be created.

[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EROFS]	The named file resides on a read-only file system.
[EEXIST]	The named file exists.
[ENOSPC]	The new directory cannot be created because there is no space left on the file system that will contain the directory.
[ENOSPC]	There are no free inodes on the file system on which the directory is being created.
[EDQUOT]	The new directory cannot be created because the user's quota of disk blocks on the file system that will con tain the directory has been exhausted.
[EDQUOT]	The user's quota of inodes on the file system on which the directory is being created has been exhausted.
[EIO]	An I/O error occurred while making the directory entry or allocating the inode.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[EFAULT]	Path points outside the process's allocated address space.

SEE ALSO

chmod(2), stat(2), umask(2)

STANDARDS

The **mkdir()** function call is expected to conform to IEEE Std1003.1-1990 ("POSIX") as closely as the host operating system allows.

MKFIFO(2)

NAME

mkfifo - make a fifo file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

int
mkfifo(const char *path, mode_t mode);
```

DESCRIPTION

mkfifo() creates a new fifo file with name *path*. The access permissions are specified by *mode* and restricted by the umask(2) of the calling process.

The fifo's owner ID is set to the process's effective user ID. The fifo's group ID is set to that of the parent directory in which it is created.

RETURN VALUES

The **mkfifo()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

mkfifo() will fail and no fifo will be created if:

`[ENOTSUPP]` The system does not support Unix Systems Services.

`[ENOTSUP]` The specified path is not in the HFS file system.

`[ENOTDIR]` A component of the path prefix is not a directory.

`[ENAMETOOLONG]` A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

`[ENOENT]` A component of the path prefix does not exist.

`[EACCES]` Search permission is denied for a component of the path prefix.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EROFS] The named file resides on a read-only file system.

[EEXIST] The named file exists.

[ENOSPC] The directory in which the entry for the new fifo is being placed cannot be extended because there is no space left on the file system containing the directory.

[ENOSPC] There are no free inodes on the file system on which the fifo is being created.

[EDQUOT] The directory in which the entry for the new fifo is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.

[EDQUOT] The user's quota of inodes on the file system on which the fifo is being created has been exhausted.

[EIO] An I/O error occurred while making the directory entry or allocating the inode.

[EIO] An I/O error occurred while reading from or writing to the file system.

[EFAULT] *Path* points outside the process's allocated address space.

SEE ALSO

chmod(2), mknod(2), stat(2), umask(2)

STANDARDS

The **mkfifo()** function call is expected to conform to ISO/IEC 9945-1:1990 ("POSIX.1") as closely as the host operating system allows.

MKNOD(2)

NAME

mknod - make an //HFS:-style special file node

SYNOPSIS

```
#include <unistd.h>

int
mknod(const char *path, mode_t mode, dev_t dev);
```

DESCRIPTION

The HFS filesystem node *path* is created with the file type and access permissions specified in *mode*. The access permissions are modified by the process's umask value.

If *mode* indicates a block or character special file, *dev* is a configuration dependent specification denoting a particular device on the system. Otherwise, *dev* is ignored.

mknod() requires super-user privileges.

RETURN VALUES

The **mknod()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

mknod() will fail and the file will be not created if:

`[ENOTDIR]` A component of the path prefix is not a directory.

`[ENAMETOOLONG]` A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

`[ENOENT]` A component of the path prefix does not exist.

`[EACCES]` Search permission is denied for a component of the path prefix.

`[ELOOP]` Too many symbolic links were encountered in translating the pathname.

[EPERM] The process's effective user ID is not super-user.

[EIO] An I/O error occurred while making the directory entry or allocating the inode.

[ENOSPC] The directory in which the entry for the new node is being placed cannot be extended because there is no space left on the file system containing the directory.

[ENOSPC] There are no free inodes on the file system on which the node is being created.

[EDQUOT] The directory in which the entry for the new node is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.

[EDQUOT] The user's quota of inodes on the file system on which the node is being created has been exhausted.

[EROFS] The named file resides on a read-only file system.

[EEXIST] The named file exists.

[EFAULT] *Path* points outside the process's allocated address space.

[EINVAL] Creating anything else than a character special file, regular file, FIFO or directory is not supported.

SEE ALSO

chmod(2)), mkfifo(2), stat(2), umask(2)

MMAP(2)

NAME

mmap - allocate memory, or map files or devices into memory

SYNOPSIS

```
#include <sys/types.h>
#include <sys/mman.h>

void *
mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);
```

DESCRIPTION

The **mmap()** function causes the pages starting at *addr* and continuing for at most *len* bytes to be mapped from the `//HFS:` object described by *fd*, starting at byte offset *offset*. If *len* is not a multiple of the pagesize, the mapped region may extend past the specified range, or **mmap()** may fail. Any such extension beyond the end of the mapped object will be zero-filled.

If *addr* is non-zero, it is used as a hint to the system. (As a convenience to the system, the actual address of the region may differ from the address supplied.) If *addr* is zero, an address will be selected by the system. The actual starting address of the region is returned. A successful *mmap* deletes any previous mapping in the allocated address range.

If `MAP_FIXED` is specified, a non-zero *addr* must be aligned to a page boundary, if `__MAP_MEGA` is specified *addr* a non-zero *addr* must be segment aligned. When `MAP_FIXED` is not supplied, the result will be on the nearest page boundary if possible or if `__MAP_MEGA` is specifed on the nearest segment boundary, if possible.

On systems that support it the `__MAP_64` option can be specified to request an address above-the-bar in 64-bit environents. If the *len* value is larger than 2G, or the *addr* value is larger than 64G then `__MAP_64` is implied. The `__MAP_64` option can be added to request 64-bit addresses when neither of those is true.

The protections (region accessibility) are specified in the *prot* argument by or'ing the following values:

`PROT_NONE` Pages may not be accessed.

`PROT_READ` Pages may be read.

PROT_WRITE Pages may be written.

PROT_EXEC Pages may be executed.

The *flags* parameter specifies the type of the mapped object, mapping options and wether modifications made to the mapped copy of the page are private to the process or are to be shared with other references. Sharing, mapping type and options are specified in the *flags* argument by or'ing the following values:

MAP_FIXED Do not permit the system to select a different address than the one specified. If the specified address cannot be used, **mmap()** will fail. If MAP_FIXED is specified, *addr* must be a multiple of the pagesize. Use of this option is discouraged.

MAP_PRIVATE Modifications are private.

MAP_SHARED Modifications are shared.

__MAP_MEGA Memory is mapped using segment-sized units instead of page-sized units.

__MAP_64 Use above-the-bar (64-bit) storage and support lengths larger than 0x7fffffff. When __MAP_64 is specified, MAP_SHARED is assumed. MAP_PRIVATE and MAP_MEGA may not be combined with __MAP_64.

The close(2) function does not unmap pages, see munmap(2) for further information.

RETURN VALUES

Upon successful completion, **mmap()** returns a pointer to the mapped region. Otherwise, a value of MAP_FAILED is returned and errno is set to indicate the error.

ERRORS

mmap() will fail if:

[EACCES] The flag PROT_READ was specified as part of the *prot* parameter and *fd* was not open for reading. The flags MAP_SHARED and PROT_WRITE were specified as part of the flags and *prot* parameters and *fd* was not open for writing.

[EAGAIN] The caller is not in PSW key 8.

[EBADF] *fd* is not a valid open file descriptor.

[EINVAL] `MAP_FIXED` was specified and the *addr* parameter was not page (or segment) aligned, or part of the desired address space resides out of the valid address space for a user process.

[EINVAL] *addr* was above 0x7fffffff (see ISSUES below).

[EINVAL] *len* was negative.

[EINVAL] *len* was larger than 0x7fffffff in 64-bit mode (see ISSUES below).

[EINVAL] *offset* was not page-aligned (or segment-aligned when `__MAP_MEGA` is specified.)

[EINVAL] *flags* or *prot* were invalid.

[EINVAL] An attempt to map an already mapped file with a different specification of `__MAP_MEGA`.

[EINVAL] An invalid address (greater than 0x7fffffff and less than 64G) was passed for *addr*.

[EINVAL] Both `__MAP_64` and `MAP_FIXED` were specified but the *addr* had zeros in the high-order 32-bits.

[ENODEV] *fd* refers to a non-supported file type.

[ENOMEM] `MAP_FIXED` was specified and the *addr* parameter wasn't available.

[ENOMEM] There is not enough space remaining in the address space.

[ENOMEM] There is not enough shared storage available in the entire system.

[ENOSYS] `MAP_PRIVATE` was specified, but the hardware doesn't support it.

[ENXIO] The address range is not valid for the file.

64-BIT addresses

Originally, even in 64-bit addressing mode, the z/OS **mmap** service did not allow a length greater than 2G or an address greater than the 31-bit address space. IBM APARs OA60306 and PH32235 were created to deliver the ability to handle true 64-bit lengths and 64-bit addresses.

In general, this is called "64-bit support". I can be specifically requested using the `__MAP_64` specification, or by specifying an *addr* above 64G or specifying a *len* larger than 2G.

There are several caveats to this support as outlined in the z/OS BPX4MMP system service documentation. Consult the IBM documentation "z/OS UNIX Systems Services Programming: Assembler Callable Services Reference" for more details.

ISSUES

Even in 64-bit addressing mode, the z/OS **mmap** function cannot map addresses above 0x7fffffff or specify a length larger than 0x7fffffff. The returned address will also be in the 31-bit address space.

This issue was addressed in IBM APAR OA60306 and APAR PH32235.

The `__MAP_64` flag is only valid when IBM APAR's OA60306 and PH32235 have been applied, or when running on versions of z/OS after version 2.5. Using it in other situations is undefined and may fail mysteriously. The Dignus runtime has no way to determine if its use in any runtime environment is valid.

SEE ALSO

mprotect(2), msync(2), munmap(2).

MPROTECT(2)

NAME

mprotect - control the protection of pages

SYNOPSIS

```
#include <sys/types.h>
#include <sys/mman.h>

int
mprotect(const void *addr, size_t len, int prot);
```

DESCRIPTION

The **mprotect()** system call changes the specified pages to have protection *prot*. Not all implementations will guarantee protection on a page basis; the granularity of protection changes may be as large as an entire region.

Currently these protection bits are known, which can be combined, OR'd together:

`PROT_NONE`	No permissions at all.
`PROT_READ`	The pages can be read.
`PROT_WRITE`	The pages can be written.
`PROT_EXEC`	The pages can be executed.

RETURN VALUES

The **mprotect()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

The **mprotect()** function will fail if:

`[EINVAL]`	The virtual address range specified by the *addr* and *len* arguments is not valid.
`[EACCES]`	The calling process was not allowed to change the protection to the value specified by the *prot* argument.

SEE ALSO

msync(2), munmap(2)

MSYNC(2)

NAME

msync - synchronize a mapped region

SYNOPSIS

```
#include <sys/types.h>
#include <sys/mman.h>

int
msync(void *addr, size_t len, int flags);
```

DESCRIPTION

The **msync()** system call writes any modified pages back to the filesystem and updates the file modification time. If *len* is 0, all modified pages within the region containing *addr* will be flushed; if *len* is non-zero, only those pages containing *addr* and *len*-1 succeeding locations will be examined. The flags argument may be specified as follows:

`MS_ASYNC`	Return immediately
`MS_SYNC`	Perform synchronous writes
`MS_INVALIDATE`	Invalidate all cached data

RETURN VALUES

The **msync()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

msync() will fail if:

`[EINVAL]`	*addr* is not a multiple of the hardware page size.
`[EINVAL]`	*len* is too large or negative.
`[EINVAL]`	*flags* was both `MS_ASYNC` and `MS_INVALIDATE`. Only one of these flags is allowed.
`[EIO]`	An I/O error occurred while writing to the file system.

SEE ALSO

mprotect(2), munmap(2)

MSGCTL(2)

NAME

msgctl - message control operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int
msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

DESCRIPTION

The **msgctl()** system call performs some control operations on the message queue specified by *msqid*.

Each message queue has a data structure associated with it, parts of which may be altered by **msgctl()** and parts of which determine the actions of **msgctl()**. The data structure is defined in `<sys/msg.h>` and contains (amongst others) the following members:

```
struct msqid_ds {
        struct  ipc_perm msg_perm;      /* msg queue permission bits */
        struct  msg *msg_first; /* first message in the queue */
        struct  msg *msg_last;  /* last message in the queue */
        u_long  msg_cbytes;     /* number of bytes in use on the queue */
        u_long  msg_qnum;       /* number of msgs in the queue */
        u_long  msg_qbytes;     /* max # of bytes on the queue */
        pid_t   msg_lspid;      /* pid of last msgsnd() */
        pid_t   msg_lrpid;      /* pid of last msgrcv() */
        time_t  msg_stime;      /* time of last msgsnd() */
        long    msg_pad1;
        time_t  msg_rtime;      /* time of last msgrcv() */
        long    msg_pad2;
        time_t  msg_ctime;      /* time of last msgctl() */
        long    msg_pad3;
        long    msg_pad4[4];
};
```

The `ipc_perm` structure used inside the `shmid_ds` structure is defined in `<sys/ipc.h>` and looks like this:

```
struct ipc_perm {
         ushort  cuid;   /* creator user id */
         ushort  cgid;   /* creator group id */
         ushort  uid;    /* user id */
         ushort  gid;    /* group id */
         ushort  mode;   /* r/w permission */
         ushort  seq;    /* sequence # (to generate unique msg/sem/shm id) */
         key_t   key;    /* user specified msg/sem/shm key */
 };
```

The operations to be performed by **msgctl()** is specified in *cmd* and is one of:

`IPC_STAT` Gather information about the message queue and place it in the structure pointed to by *buf*.

`IPC_SET` Set the value of the `msg_perm.uid`, `msg_perm.gid`, and `msg_qbytes` fields in the structure associated with *msqid*. The values are taken from the corresponding fields in the structure pointed to by *buf*. his operation can only be executed by the super-user, or a process that has an effective user id equal to either `msg_perm.cuid` or `msg_perm.uid` in the data structure associated with the message queue. The value of `msg_qbytes` can only be increased by the super-user. Values for `msg_qbytes` that exceed the system limit are silently truncated to that limit.

`IPC_RMID` Remove the message queue specified by *msqid* and destroy the data associated with it. Only the super-user or a process with an effective uid equal to the `msg_perm.cuid` or `msg_perm.uid` values in the data structure associated with the queue can do this.

The permission to read from or write to a message queue (see msgsnd(2) and msgrcv(2)) is determined by the `msg_perm.mode` field in the same way as is done with files (see chmod(2)), but the effective uid can match either the `msg_perm.cuid` field or the `msg_perm.uid` field, and the effective gid can match either `msg_perm.cgid` or `msg_perm.gid`.

RETURN VALUES

The **msgctl()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

The **msgctl()** function will fail if:

`[EPERM]` The *cmd* argument is equal to `IPC_SET` or `IPC_RMID` and the caller is not the super-user, nor does the effective uid match either the `msg_perm.uid` or `msg_perm.cuid` fields of the data structure associated with the message queue.

An attempt is made to increase the value of `msg_qbytes` through `IPC_SET` but the caller is not the super-user.

`[EACCES]` The command is `IPC_STAT` and the caller has no read permission for this message queue.

`[EINVAL]` The *msqid* argument is not a valid message queue identifier.

cmd is not a valid command.

`[EFAULT]` The *buf* argument specifies an invalid address.

SEE ALSO

msgget(2), msgrcv(2), msgsnd(2)

ISSUES

The underlying IBM Unix Systems Services does not support the `seq` and `key` fields of the `ipc_perm` structure, nor the `msg_first`, `msg_last` or `msg_cbytes` field of the `msqid_ds`. They are provided for compatibility and will always be zero.

MSGGET(2)

NAME

msgget - get message queue

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int
msgget(key_t key, int msgflg);
```

DESCRIPTION

The **msgget()** function returns the message queue identifier associated with *key*. A message queue identifier is a unique integer greater than zaero.

A message queue is created if either *key* is equal to `IPC_PRIVATE`, or *key* does not have a message queue identifier associated with it, and the `IPC_CREAT` bit is set in *msgflg*.

If a new message queue is created, the data structure associated with it (the *msgid_ds* structure, see msgctl(2)) is initialized as follows:

- `msg_perm.cuid` and `msg_perm.uid` are set to the effective uid of the calling process.
- `msg_perm.gid` and `msg_perm.cgid` are set to the effective gid of the calling process.
- `msg_perm.mode` is set to the lower 9 bits of *msgflg*.
- `msg_cbytes`, `msg_qnum`, `msg_lspid`, `msg_lrpid`, `msg_rtime` and `msg_stime` are set to 0.
- `msg_qbytes` is set to the system wide maximum value for the number of bytes in a queue (`MSGMNB`).
- `msg_ctime` is set to the current time.

RETURN VALUES

Upon successful completion a positive message queue identifier is returned. Otherwise, -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

[EACCES]	A message queue is already associated with *key* and the caller has no permission to access it.
[EEXIST]	Both IPC_CREAT and IPC_EXCL are set in *msgflg*, and a message queue is already associated with *key*.
[ENOSPC]	A new message queue could not be created because the system limit for the number of message queues has been reached.
[ENOENT]	IPC_CREAT was not set in *msgflg* and no message queue associated with key was found.

SEE ALSO

msgctl(2), msgrcv(2), msgsnd(2)

MSGRCV(2)

msgrcv - receive a message from a message queue

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int
msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

DESCRIPTION

The **msgrcv()** function receives a message from the message queue specified in *msqid*, and places it into the structure pointed to by *msgp*. This structure should consist of the following members:

```
long mtype;    /* message type */
char mtext[1]; /* body of message */
```

mtype is an integer greater than 0 that can be used for selecting messages, *mtext* is an array of bytes, with a size up to that of the system limit.

The value of *msgtyp* has one of the following meanings:

- The *msgtyp* argument is greater than 0. The first message of type *msgtyp* will be received.
- The *msgtyp* argument is equal to 0. The first message on the queue will be received.
- The *msgtyp* argument is less than 0. The first message of the lowest message type that is less than or equal to the absolute value of *msgtyp* will be received.

The *msgsz* argument specifies the maximum length of the requested message. If the received message has a length greater than *msgsz* it will be silently truncated if the *MSG_NOERROR* flag is set in *msgflg*, otherwise an error will be returned.

If no matching message is present on the message queue specified by *msqid*, the behavior of **msgrcv()** depends on whether the `IPC_NOWAIT` flag is set in *msgflag* or not. If `IPC_NOWAIT` is set, **msgrcv()** will immediately return a value of -1, and set `errno` to `ENOMSG`. If `IPC_NOWAIT` is not set, the calling process will be blocked until:

- A message of the requested type becomes available on the message queue.
- The message queue is removed, in which case -1 will be returned, and `errno` set to `EINVAL`.
- A signal is received and caught. -1 is returned, and `errno` set to `EINTR`.

If a message is is successfully received, the data structure associated with *msqid* is updated as follows:

- `msg_lrpid` is set to the pid of the caller.
- `msg_lrtime` is set to the current time.
- `msg_qnum` is decremented by 1.

RETURN VALUES

Upon successful completion, **msgrcv()** returns the number of bytes received into the `mtext` field of the structure pointed to by *msgp*. Otherwise, -1 is returned, and `errno` set to indicate the error.

ERRORS

The **msgrcv()** function will fail if:

`[EINVAL]`	The *msqid* argument is not a valid message queue identifier. The *msgsz* argument is less than 0.
`[E2BIG]`	A matching message was received, but its size was greater than msgsz and the `MSG_NOERROR` flag was not set in *msgflg*.
`[EACCES]`	The calling process does not have read access to the message queue.
`[EFAULT]`	The *msgp* argument points to an invalid address.
`[EIDRM]`	The message queue was removed while **msgrcv()** was waiting for a message of the requested type to become available on it.
`[EINTR]`	The system call was interrupted by the delivery of a signal.
`[ENOMSG]`	There is no message of the requested type available on the message queue, and `IPC_NOWAIT` is set in msgflg.

SEE ALSO

msgctl(2), msgget(2), msgsnd(2)

MSGSND(2)

NAME

msgsnd - send a message to a message queue

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int
msgsnd(int msqid, void *msgp, size_t msgsz, int msgflg);
```

DESCRIPTION

The **msgsnd()** function sends a message to the message queue specified in *msqid*. *msgp* points to a structure containing the message. This structure should consist of the following members:

```
long mtype;    /* message type */
char mtext[1]; /* body of message */
```

mtype is an integer greater than 0 that can be used for selecting messages (see msgrcv(2)), *mtext* is an array of bytes, with a size up to the system limit.

If the number of bytes already on the message queue plus *msgsz* is bigger than the maximum number of bytes on the message queue (`msg_qbytes`, see msgctl(2)), or the number of messages on all queues system-wide is already equal to the system limit, *msgflg* determines the action of **msgsnd()**. If *msgflg* has `IPC_NOWAIT` mask set in it, the call will return immediately. If *msgflg* does not have IPC_NOWAIT(s)et in it, the call will block until:

- The condition which caused the call to block does no longer exist. The message will be sent.
- The message queue is removed, in which case -1 will be returned, and `errno` is set to `EINVAL`.
- The caller catches a signal. The call returns with `errno` set to `EINTR`.

After a successful call, the data structure associated with the message queue is updated in the following way:

- `msg_qnum` is incremented by 1.
- `msg_lspid` is set to the pid of the calling process.
- `msg_stime` is set to the current time.

RETURN VALUES

The **msgsnd()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

msgsnd() will fail if:

`[EINVAL]`	*msqid* is not a valid message queue identifier *msgsz* is less than 0, or greater than `msg_qbytes`. *mtype* is not greater than 0.
`[EACCES]`	The calling process does not have write access to the message queue.
`[EAGAIN]`	There was no space for this message either on the queue, or in the whole system, and `IPC_NOWAIT` was set in *msgflg*.
`[EFAULT]`	*msgp* points to an invalid address.
`[EIDRM]`	The message queue was removed while **msgsnd()** was waiting for a resource to become available in order to deliver the message.
`[EINTR]`	The system call was interrupted by the delivery of a signal.

SEE ALSO

msgctl(2), msgget(2), msgrcv(2)

MUNMAP(2)

NAME

munmap - remove a mapping

SYNOPSIS

```
#include <sys/types.h>
#include <sys/mman.h>

int
munmap(void *addr, size_t len);
```

DESCRIPTION

The **munmap()** system call deletes the mapping for the specified address range, and causes further references to addresses within the range to generate invalid memory references.

RETURN VALUES

The **munmap()** returns the value 0 if successful; otherwise the value -1 is returned and the global variable **errno** is set to indicate the error.

ERRORS

munmap() will fail if:

`[EINVAL]` The *addr* parameter was not page aligned, the *len* parameter was negative, or some part of the region being unmapped is outside the valid address range for a process.

SEE ALSO

mmap(2), mprotect(2), msync(2)

NANOSLEEP(2)

NAME

nanosleep – suspend process execution for an interval measured in nanoseconds

SYNOPSIS

```
#include <time.h>

int
nanosleep(const struct timespec *rqtp, struct timespec *rmtp);
```

DESCRIPTION

The **nanosleep()** system call causes the process to sleep for the specified time. An unmasked signal will cause it to terminate the sleep early, regardless of the `SA_RESTART` value on the interrupting signal.

RETURN VALUES

If the **nanosleep()** system call returns because the requested time has elapsed, the value returned will be zero.

If the **nanosleep()** system call returns due to the delivery of a signal, the value returned will be -1, and the global variable `errno` will be set to indicate the interruption. If *rmtp* is non-`NULL`, the timespec structure it references is updated to contain the unslept amount (the request time minus the time actually slept).

ERRORS

The **nanosleep()** system call fails if:

`[EFAULT]` Either *rqtp* or *rmtp* points to memory that is not a valid part of the process address space.

`[EINTR]` The **nanosleep()** system call was interrupted by the delivery of a signal.

`[EINVAL]` The *rqtp* argument specified a nanosecond value less than zero or greater than or equal to 1000 million.

`[ENOSYS]` The *nanosleep()* system call is not supported by this implementation.

IMPLEMENTATION

The **nanosleep()** function requires the use of BPX signals to interrupt the process before the timeout occurs. If BPX signals are not enabled, the **nanosleep()** function will wait until the specified time has elapsed.

SEE ALSO

sigsuspend(2), sleep(3)

STANDARDS

The **nanosleep()** system call conforms to IEEE Std 1003.1b-1993 ("POSIX.1").

OPEN(2)

NAME

open - open or create a file for reading or writing

SYNOPSIS

```
#include <fcntl.h>

int
open(const char *path, int flags, ...)
```

DESCRIPTION

The file name specified by path is opened for reading and/or writing as specified by the argument flags and the file descriptor returned to the calling process. The flags argument may indicate the file is to be created if it does not exist (by specifying the `O_CREAT` flag). In this case open requires a third argument `mode_t mode`.

The flags specified are formed by or'ing the following values

`O_RDONLY`	open for reading only
`O_WRONLY`	open for writing only
`O_RDWR`	open for reading and writing
`O_NONBLOCK`	do not block on open
`O_APPEND`	append on each write
`O_CLOEXEC`	close the file on exec
`O_CREAT`	create file if it does not exist
`O_TRUNC`	truncate size to 0
`O_EXCL`	error if create and file exists
`O_SHLOCK`	atomically obtain a shared lock
`O_EXLOCK`	atomically obtain an exclusive lock
`_O_BINARY`	specifies that I/O is to be done in binary mode, not text translation.
`_O_TEXT`	(default) specify that I/O is to be done with text translation.

`_O_ATTR` An extra `char *` argument is found after the mode argument. This argument can be applied to non-HFS files, and specifies file attributes to use in the OS/390 DCB.

Opening a file with `O_APPEND` set causes each write on the file to be appended to the end. If `O_TRUNC` is specified and the file exists, the file is truncated to zero length. If `O_EXCL` is set with `O_CREAT` and the file already exists, **open()** returns an error. This may be used to implement a simple exclusive access locking mechanism. If the `O_NONBLOCK` flag is specified and the **open()** call would result in the process being blocked for some reason (e.g., waiting for carrier on a dialup line), **open()** returns immediately. The first time the process attempts to perform I/O on the open file it will block (not currently implemented).

For HFS files, if the `O_CLOEXEC` flag is set, then the `FD_CLOEXEC` flag will be set; otherwise it is cleared.

If `_O_BINARY` is specified, the bytes retrieved from the operating system are passed to the program without further processing.

If `_O_TEXT` is specified, on input, trailing blanks are deleted and a new-line is appended. On output, the new-line marks the end of the record, with trailing blanks appended to complete an output record.

When opening a file, a lock with flock(2) semantics can be obtained by setting `O_SHLOCK` for a shared lock, or `O_EXLOCK` for an exclusive lock. If creating a file with `O_CREAT`, the request for the lock will never fail (provided that the underlying filesystem supports locking).

If successful, **open()** returns a non-negative integer, termed a file descriptor. It returns -1 on failure. The file pointer used to mark the current position within the file is set to the beginning of the file.

The system imposes a limit on the number of file descriptors open simultaneously by one process. Getdtablesize(2) returns the current system limit.

PATH NAMES

The Systems/C **open()** function uses path name prefixes to determine how to allocate the file. A path name prefix consists of two slashes, followed by the prefix style name, followed by a colon (`//` *style*`:`). If a prefix is not specified, and the path name begins with a single slash (`/`), or the path name begins with the two characters period (`.`) and then slash (`/`), the file is treated as if the `//HFS:` prefix had been specified. Otherwise, the current default style is used.

The current default style is found in the global variable `extern char * _style`, and may be changed by assigning a new value to that variable. When a Systems/C

program is invoked from either a TSO or BATCH environment, the default value of `_style` is `//DDN:`.

When a Systems/C program is invoked via the `exec` function (i.e. under OpenEdition), the default value of `_style` is `//HFS:`.

The styles currently supported include:

`//DSN:` The specified path is a fully qualified dataset name on OS/390.

`//DDN:` The specified path is a DDN allocated via a JCL DD card, or the TSO ALLOCATE command.

`//HFS:` The specified path is a file that resides in the Hiearchical File System (HFS).

Both `//DSN:` and `//DDN:` style names may also specify PDS member names, surrounded by parentheses.

DCB ATTRIBUTES

If the `_O_ATTR` bit is set in the flags argument, and the style is not `//HFS:`, then this call to open is understood to have four arguments; the forth is a character string which describes the DCB attributes to initially use for the OS/390 OPEN service. All four arguments must be present if `_O_ATTR` is set.

These attributes are used to provide DCB during an OPEN EXIT on the OS/390 OPEN system service. Thus they can be used to provide default values when they are not present on DD cards, or can provide appropriate values when creating a new data set.

The format of that string is a comma separated list of *NAME*=*VALUE* pairs.

The following names and values are currently supported:

`abend` `abend` indicates how the runtime should handle BSAM I/O ABENDs after a successful BSAM OPEN. If `abend=abend` is specified then any ignornable I/O ABEND will become an actual ABEND to be processed by whatever ABEND processing is pertinent. If `abend=recover` is specified, any ignorable I/O ABEND will be "ignored" (although the operating system will often produce a message of some kind) and the library will report the I/O failure to the program, with `errno` set appropriately.

The default is `abend=recover`.

`blksize`	integer block size.
`blocks`	No parameter. `blocks` specifies allocations are in blocks. May be abbreviated as `blks` or `blk`.
`bufno`	integer number of buffers. `bufno` specifies the number of buffers to specify when allocating a non-HFS data set.
`cylinders`	No parameter. `cylinders` specifies allocations are in cylinders. May be abbreviated as `cyls` or `cyl`.
`directory`	integer specifying the number of directory blocks for a new PDS. If the value is omitted, directory blocks will not be specified when allocating the new PDS.
`catalog`	Indicates the file should be added to the system catalog. May be abbreviated as `catlg`.
`delete`	The file should be deleted when it is deallocated.
`keep`	The file is kept when it is deallocated.
`keylen`	integer specifying the key length value in the DCB.
`lrecl`	integer representing the logical record length, or the character 'X' indicating a record length larger than 32760 for Variable Spanned files.
`ncp`	integer representing the NCP value for BSAM I/O (number of outstanding READ/WRITE requests before a CHECK.) This is also the number of I/O buffers the library will allocate when multi-buffering I/O is allowed.
`preopen`	No parameter. `preopen` indicates that the values specified as attributes should apply to the `DCB` before opening the file, thus "overriding" any JCL specification.
`noseek`	No parameter. Indicates that the file positioning function (lseek(2)) will not be used on the returned file descriptor. When **noseek** is true, the MACRF option on a BSAM OPEN will not indicate the use of NOTE and POINT, and thus the file can read LARGE format data sets. Reading/writing of LARGE format data sets also requires the BLOCKTOKENSIZE(NOREQUIRE) in SYS1.PARMLIB. **noseek** is also required for multi-buffer support (if seeking is needed, only one I/O buffer is used.)
`primary`	integer specifying the primary allocation size. May be abbreviated as `pri`.
`recfm`	specifies the record format, either `f`, `fa`, `fb`, `fs`, `fba`, `fbs`, `fsa`, `fbsa`, `v`, `vb`, `vs`, `vbs` or `u` is supported.

`secondary`	integer specifying the secondary allocation size. May be abbreviated as `sec`
`rlse`	No parameter. Indicates any unused space for a new data set be returned (this is the default setting.)
`norlse`	No parameter. Indicates any unused space for a new data set not be returned
`tracks`	No parameter. `tracks` specifies allocations are in tracks. May be abbreviated as `trks` or `trk`
`type`	keyword specifying type of I/O. Currently, the `record` keyword is supported to indirect record I/O (e.g. `type=record`). When used with open(2), this attribute sets the `_O_RECIO` flag.
`uncatlg`	Indicates the file should removed from the system catalog. May be abbreviated as `uncatlg`.
`unit`	character string that specifies the device name.
`verbose`	No parameter. Causes allocation messages to appear in the system log. The default is not `verbose`.
`volser`	character string representing the volume serial identifier. May be abbreviated as `vol`.
`volseq`	integer specify the volume sequence number.

Invalid *NAME*=*VALUE* pairs are silently ignored.

Note that `preopen`, `blocks`, `cylinders` and `tracks` *NAME*s have no *VALUE* specified, and that the *VALUE* is optional on the `directory` *NAME*.

For example, the following code will open //DDN:MYDD for binary input with a `blksize` of 3200 and an `lrecl` of 80:

```
open("//DDN:MYDD", O_RDONLY|_O_ATTR|_O_BINARY,
     0, "blksize=3200,lrecl=80");
```

CREATING `//DSN:` FILES

Files can be created by the Systems/C runtime when the `O_CREAT` flag is specified. Many of the `O_ATTR` flags only apply when creating files.

For example, if the `O_ATTR` string specifies that a file is fixed block, but an existing file opened for input is variable blocked, the library will continue as if the file were variable blocked.

If a file doesn't exist, the initial allocation sizes and attributes may be specified in the JCL or ALLOC statement, via the O_ATTR string, or may be calculated by the Systems/C library.

Attributes are combined from these sources in the following order.

If the "preopen" attribute is not enabled, then the value from the JCL or ALLOC statement are used first. If "preopen" is specified, then the values specified in the O_ATTR string are used first. That is, if "preopen" is specified, the constructed DCB is initialized with any values specified in O_ATTR, otherwise it is not. If during OPEN processing (in the OPEN exit routine), values are not provided then the ones specified from the O_ATTR string are used. For any values still not defined, the following rules are used to calculate default values.

If RECFM is unspecified and the file is opened for output with the _O_BINARY flag, then a RECFM=FB will be used. If the file is opened for output without the _O_BINARY flag, then the device is queried. If the device is a terminal, RECFM=U will be used, otherwise RECFM=FBA will be used. If the RECFM remains unspecified for an input file, further processing stops and the OPEN will likely fail with a errno set to EIO.

If both BLKSIZE and LRECL remain unspecified (both are zero) then the library examines the RECFM to determine these values. If RECFM=U, then BLKSIZE will be the maximum blocksize for the device, and LRECL will be zero. If RECFM=F then BLKSIZE=LRECL=80 will be used. If RECFM=FB then LRECL will be the 80 and BLKSIZE will be the largest blocksize that is possible for the device. If RECFM=V then the BLKSIZE will be the maximum blocksize that is possible for the device, and LRECL will be the lessedr of BLKSIZE-4 and 1028.

If LRECL is provided, but BLKSIZE is not, then for RECFM=U files, the BLKSIZE becomes LRECL and LRECL is set to zero. For RECFM=F files, the BLKSIZE is set to be the same as the LRECL, and for RECFM=FB files, the BLKSIZE is set to be the largest multiple of the LRECL that can fit in the maximum blocksize of the device. For RECFM=V files the BLKSIZE is set to LRECL+4.

If LRECL is not provided, but BLKSIZE is, then for RECFM=U, LRECL is set to 0. For RECM=F and RECFM=FB, LRECL is set to be the BLKSIZE value. For RECFM=V files, LRECL is set to the lesser of BLKSIZE-4 and 1028.

For BSAM I/O, if the DCB has a DCBNCP value of 0 or 1 and the ncp attribute was used and it specifies a value larger than 1, then the specified ncp attribute value is used.

For example, the following statement creates a new file RECFM=FB, USER.FILE, specifying that the primary allocation is 1000 blocks, the secondary allocation is 500 blocks, the record length is 80 and the block size is 800 (note the specification of O_CREAT which causes the library to create the file if it doesn't exist):

```
open("//DSN:USER.FILE", O_WRONLY|O_CREAT|_O_ATTR|_O_BINARY,
```

```
        0,
    "recfm=fb,blksize=800,lrecl=80,blks,primary=1000,secondary=500");
```

Note that combining values via the `O_ATTR` string and JCL or other sources may create conflicting values that cause the **open()** to fail. For example, if the JCL only specifies `LRECL=133` and the program provides an attribute string that specifies `"recfm=fb,lrecl=80,blksize=8000"`, then the resulting combined attributes will be `RECFM=FB,LRECL=133,BLKSIZE=8000` which is invalid because the `BLKSIZE` is not a multiple of the `LRECL`. In such a situation, the `"preopen"` attribute can be used to indicate that the values specified in the `O_ATTR` string take precedence over the values from the JCL.

To ensure a file does not previously exist, use the `O_EXCL` flag. If `O_EXCL` is specified in combination with `O_CREAT`, and the file already exists, the **open()** function will return -1 and indicate the error in `errno`.

RECORD I/O

Typically, the C model of I/O is simply a stream of bytes; without consideration of record lengths or boundaries. The Systems/C runtime presents this abstraction to the C program, managing block and record considerations internally. Thus, for example, a read(2) request of 500 bytes will internally deblock a file, and gather 500 bytes, crossing any record boundaries as needed.

However, there are situations in mainframe programming environemnts where it is helpful to manage data in terms of records. Specifying `_O_RECIO` in the flags causes the Systems/C runtime to respect record boundaries. If `_O_RECIO` a write(2) request will only write a record-sized portion of data. Similarly, a read(2) request will only read a record-sized portion of data. After each I/O operation, the file pointer moves to the next record boundary. The "type=record" attribute can be used as well as `_O_RECIO`. The "type=record" attribute causes the `_O_RECIO` flag to be set.

The read(2), write(2), sections have more information on different semantics when `_O_RECIO` is present in flags. The fopen(3), fread(3), fwrite(3) sections have more information on different semantics when the `type=record` attribute is specified. Note that `type=record` attribute enables record I/O for fopen(3), fread(3), fwrite(3) while the `_O_RECIO` flag enables record I/O for open(2), read(2) and write(2). When `type=record` attribute is specified in open(2), it is translated to the `_O_RECIO` flag.

IMPLEMENTATION NOTES

Because fopen(3) uses open(2) to access files, the path prefixes discussed above are also valid for fopen(3) path names. Furthermore, any attributes string fopen(3) receives is similar simply passed to open(2).

O_APPEND mode is only supported on sequential files. An open of a PDS member with O_APPEND will cause a runtime abend when the file is subsequently closed.

RETURN VALUES

If successful, **open()** returns a non-negative integer, termed a file descriptor. It returns -1 on failure, and sets errno to indicate the error.

ERRORS

The named file is opened unless:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	A component of a pathname was too large for the given path prefix style, or an entire path name exceeded 1023 characters.
[ENOENT]	O_CREAT is not set and the named file does not exist.
[ENOENT]	A component of the path name that must exist does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[EACCES]	The required permissions (for reading and/or writing) are denied for the given flags.
[EACCES]	O_CREAT is specified, the file does not exist, and creation of the file was not permitted.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EISDIR]	The named file is a directory or PDS, and the arguments specify it is to be opened for writing.
[EROFS]	The named file resides on a read-only file system, and the file is to be modified.
[EMFILE]	The process has already reached its limit for open file descriptors.
[ENFILE]	The system file table is full.
[ENOMEM]	There was insufficient memory available to allocate the supporting data structures needed.
[ENXIO]	The named file is a character special or block special file, and the device associated with this special file does not exist.
[EINTR]	The **open()** operation was interrupted by a signal.

[EOPNOTSUPP] O_SHLOCK or O_EXLOCK is specified but the underlying filesystem does not support locking.

[ENOSPC] O_CREAT is specified, the file does not exist, and the directory or PDS in which the entry for the new file is being placed cannot be extended because there is no space left on the file system containing the directory, or the PDS containing the member.

[EDQUOT] O_CREAT is specified, the file does not exist, and the user's allocation quota on the file system on which the file is being created has been exhausted.

[EFTYPE] An attempt was made to open a Format-F file where the LRECL was not a multiple of the BLKSIZE, or a blocked Format-V file where the LRECL was not less than BLKSIZE-4.

[EIO] An I/O error occurred while making the directory entry or allocating the PDS directory entry for O_CREAT.

[EFAULT] Path points outside the process's allocated address space.

[EEXIST] O_CREAT and O_EXCL were specified and the file exists.

[EOPNOTSUPP] An attempt was made to open a socket (not currently implemented).

[EINVAL] An attempt was made to open a descriptor with an illegal combination of O_RDONLY, O_WRONLY, and O_RDWR.

SEE ALSO

close(2), dup(2), getdtablesize(2), lseek(2), _setmode(2),read(2), write(2)

OSDDINFO(2)

NAME

osddinfo - retrieve information about a dataset from a DD name.

SYNOPSIS

```
#include <machine/syscio.h>

int osddinfo(char *ddname, char dsname[45], char member[9],
             char *recfm_p, int *lrecl_p, int *blksize_p);
```

DESCRIPTION

The **osddinfo()** function is used to retrieve data set information based on the given *ddname*. *ddname* is a `NUL`-terminated character string specifying the DD name of the data set.

The remaining parameters are pointers to areas to contain return information. If the pointers are `NULL`, then **osddinfo()** does not store the value. Because some values may require invocation of additional operating systems services, it is best to make these `NULL` if the information is not required.

The data set name associated with the DD name *ddname* is stored in the area pointed-to by *dsname*.

If the DD name is allocated to a member of a PDS, the member name is stored in the area pointed-to by *member*. If a PDS member is not allocated to the DD name, the empty string is stored there.

The area pointed to by a non-`NULL` *recfm_p* will contain the record-format flag of the file. Possible values are defined in the ¡machine/syscio.h¿ header file and include:

`RECFM_U`	Undefined length records
`RECFM_F`	Fixed length records
`RECFM_V`	Variable length records
`RECFM_D`	Variable length ASCII records
`RECFM_T`	Track overflow
`RECFM_B`	Blocked records
`RECFM_S`	Spanned or Standard records

RECFM_A	ASA control characters are present
RECFM_M	Machine control characters are present

The values for these are the defined by the JFCB DSECT.

Note that RECFM_U is defined as RECFM_F logically OR'd with RECFM_V. So, care must be taken to test for RECFM_U before testing for RECFM_F or RECFM_V.

The area pointed to by a non-NULL *lrecl_p* contains the data sets logical record length, or 0. If the record format can be determined, and it is Variable Spanned, and the record length is defined as LRECL=X, then the special value LRECL_X is returned. LRECL_X is defined in the ¡machine/syscio.h¿ header file.

The area pointed to by a non-NULL *blksize_p* contains the data sets block size, or 0.

RETURN VALUES

The **osddinfo()** return 0 if the DD name is defined and the information can be retrieved. If **osddinfo()** cannot retrieve the information, or can't allocate sufficient memory to operate, it returns -1.

IMPLEMENTATION NOTES

The **osddinfo()** function uses the RJFJCB service to determine the dataset name and member associated with a DD name. If the allocation that created the JFCB control block did not include RECFM/LRECL/BLKSIZE statements, then **osddinfo()** uses the OBTAIN service to retrieve the information from the VTOC.

SEE ALSO

ddnfind(2), ddnnext(2)

__PASSWD(2)

NAME

__passwd - verify or change a user password

SYNOPSIS

```
#include <pwd.h>

int
__passwd(const char *username, const char *oldpass, const char *newpass);
```

DESCRIPTION

__passwd() verifies or changes the password of the user specified by *username*. *Oldpass* contains the current password and must always be present. *Newpass* optionally contains the new password, or can be `NULL`.

If *newpass* is non-`NULL`, then the old password is verified, and if it matches *newpass* becomes the new password.

If *newpass* is `NULL`, then the old password is simply verified, and the password remains unchanged.

RETURN VALUES

If successful, **__passwd()** returns a 0, otherwise -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

__passwd() The **__passwd()** function will fail if:

`[EACCES]` The password in *oldpass* is not authorized.

`[EINVAL]` The *username*, *oldpass* or *newpass* is invalid. *username*, *oldpass* and *newpass* must be 1 to 8 characters in length.

`[ENEEDAUTH]` The `BPX.DAEMON` facility is defined, but the current program is is not considered controled by a security product (e.g. `RACF`.)

[EPERM] The caller does not have permission for this operation. See the `BPX.DAEMON` class in the IBM "OpenEdition Planning" manual.

[ESRCH] The specified *username* was not found.

SEE ALSO

getpwent(2), endpwent(3)

PATHCONF(2)

NAME

pathconf, fpathconf - get configurable pathname variables for `//HFS:` files

SYNOPSIS

```
#include <unistd.h>

long
pathconf(const char *path, int name);

long
fpathconf(int fd, int name);
```

DESCRIPTION

The **pathconf()** and **fpathconf()** functions provide a method for applications to determine the current value of a configurable system limit or option variable associated with a pathname or file descriptor.

For **pathconf()**, the *path* argument is the name of an `//HFS:`-style file or directory. For **fpathconf()**, the *fd* argument is an open file descriptor that references an `//HFS:`-style file or directory. The *name* argument specifies the system variable to be queried. Symbol constants for each name value are found in the include file `<unistd.h>`.

The availalble values are as follows:

`_PC_LINK_MAX` The maximum file link count.

`_PC_MAX_CANON` The maximum number of bytes in terminal canonical input line.

`_PC_MAX_INPUT` The minimum maximum number of bytes for which space is available in a terminal input queue.

`_PC_NAME_MAX` The maximum number of bytes in a file name.

`_PC_PATH_MAX` The maximum number of bytes in a pathname.

`_PC_PIPE_BUF` The maximum number of bytes which will be written atomically to a pipe.

`_PC_CHOWN_RESTRICTED` Return 1 if appropriate privileges are required for the chown(2) system call, otherwise 0.

_PC_NO_TRUNC Return 1 if file names longer than KERN_NAME_MAX are truncated.

_PC_VDISABLE Returns the terminal character disabling value.

_PC_ACL Returns 1 if the security product supports access control lists, 0 otherwise.

_PC_ACL_ENTRIES_MAX Returns the maximum number of ACL entries that can be placed on the file.

RETURN VALUES

if the call to **pathconf()** or **fpathconf()** is not successful, -1 is returned an `errno` is set appropriately. Otherwise, if the variable is associated with functionality that does not have a limit in the system, -1 is returned and `errno` is not modified. Otherwise, the current variable value is returned.

ERRORS

If any of the following conditions occur, the **pathconf()** and **fpathconf()** functions shall return -1 and set `errno` to the corresponding value.

`[EINVAL]` The value of the *name* argument is invalid.

`[EINVAL]` The implementation does not support an association of the variable name with the associated file.

pathconf() will fail if:

[ENOTDIR] A component of the path prefix is not a directory.

[ENAMETOOLONG] A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] The named file does not exist.

[EACCES] Search permission is denied for a component of the path prefix.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EIO] An I/O error occurred while reading from or writing to the file system.

fpathconf() will fail if:

`[EBADF]` *fd* is not a valid open file descriptor.

`[EIO]` An I/O error occurred while reading from or writing to the file system.

PIPE(2)

NAME

pipe, pipe2 - create descriptor pair for interprocess communication

SYNOPSIS

```
#include <unistd.h>

int
pipe(int fildes[2]);

int
pipe2(int fildes[2], int flags);
```

The **pipe()** function creates a "pipe", which is an object allowing bidirectional data flow, and allocates a pair of file descriptors.

The **pipe2()** function allows control over the attributes of the file descriptors via the *flags* argument. Values for *flags* are constructed by a bitwise-inclusive OR of flags from the following list, defined in ¡fcntl.h¿:

`O_CLOEXEC`	Set the close-on-exec flag for the new file descriptors.
`O_NONBLOCK`	Set the non-blocking flag for the ends of the pipe.

If the *flags* argument is 0, the behavior is identical to a call to **pipe()**.

The first descriptor is used as the "read end" of the pipe, the second is the "write end", so that data written to *filedes*`[1]` appears on (i.e. can be read from) *fileds*`[0]`. This allows the output of one program to be sent to another program: the source's standard output can be set up to be the "write end" of the pipe, and the sink's standard input is set up to tbe the "read end" of the pipe. The pipe itself persists until all its associated descriptors are closed.

A pipe that has had an end closed is considered "widowed." Writing on such a pipe may cause the writing process to receive a `SIGPIPE` signal. Widowing a pipe is the only way to deliver end-of-file to a reader: after the reader consumes any buffered data, reading a widowed pipe returns a zero count.

IMPLEMENTATION NOTES

The **pipe2()** function calls the **pipe()** system call and then calls **fcntl** to set the appropriate flags.

RETURN VALUES

The **pipe()** function will fail if:

[EMFILE]	Too many descriptors are active.
[ENFILE]	The system file table is full.
[EFAULT]	The *fildes* buffer is in an invalid area of the process's address space.

SEE ALSO

read(2), write(2)

__PROCNAME(2)

NAME

__procname - return the current procedure name

SYNOPSIS

```
#include <machine/tiot.h>

char *
__procname(void);
```

DESCRIPTION

The **__procname()** function returns the current JCL procedure name of the executing program on MVS, OS/390 and z/OS. The value returned is a pointer to a NUL-terminated string. Trailing blanks are removed.

If the program was invoked directly, the returned name will be the empty string.

__procname() returns a pointer to a static area, care should be taken to copy this value before invoking **__procname()** again.

SEE ALSO

__jobname(2), __stepname(2), __userid(2)

__QUERYDUB(2)

NAME

__querydub - return the current procedure name

SYNOPSIS

```
#include <unistd.h>

int __querydub(void);
```

DESCRIPTION

The **__querydub()** function returns BPX "dub" status of the current task, indicating if the task has already been dubbed, or can possibly be dubbed.

RETURN VALUES

If successful, **__querydub()** returns one of these values:

`_QDB_DUBBED_FIRST` The task has already been dubbed. This task and this RB caused the dub.

`_QDB_DUBBED` The task has already been dubbed. Another task or another RB caused the dub.

`_QDB_DUB_MAY_FAIL` The task has not been dubbed, and any attempt to do so might fail (typically due to bad or missing authorizations.)

`_QDB_DUB_OKAY` The task has not been dubbed, and any attempt will likely succeed.

`_QDB_DUB_AS_PROCESS` The task has not beed dubbed, but its address space has. A dub of this task will result in a new process.

`_QDB_DUB_AS_THREAD` The task has not been dubbed, but its address space has. A dub of this task will result in a new thread within the process.

If not successful, **__querydub()** returns -1 and sets an error condition in `errno`.

SEE ALSO

__isPosixOn(2)

READ(2)

NAME

read - read input

SYNOPSIS

```
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>

size_t
read(int d, void *buf, size_t nbytes)

ssize_t
pread(int d, void *buf, size_t nbytes, off_t offset);
```

DESCRIPTION

read() attempts to read *nbytes* of data from the object referenced by the descriptor *d* into the buffer pointed to by *buf*. The **pread()** function perform the same function, but read from the specified position in the file without modifying the file pointer.

On objects capable of seeking, the **read()** starts at a position given by the pointer associated with *d* (see lseek(2)). Upon return from **read()**, the pointer is incremented by the number of bytes actually read.

Objects that are not capable of seeking always read from the current position. The value of the pointer associated with such an object is undefined.

Upon successful completion, **read()**, **pread()** and **readv()** return the number of bytes actually read and placed in the buffer. The system guarantees to read the number of bytes requested if the descriptor references a normal file that has that many bytes left before the end-of-file, but in no other case.

IMPLEMENTATION NOTES

If a file descriptor has been opened in `_O_TEXT` mode (the default), and references record-structured (non-`//HFS:` and non-socket) file, records will be read from the associated file, with trailing blanks removed, and a new-line character appended. If the *lrecl* of the file is 1, or the file descriptor is a socket, or the file descriptor references an HFS file, the bytes are read as if `_O_BINARY` had been specified.

If the file descriptor has been opened with _O_RECIO flag, and the file descriptor references a record-structured file, then the read operation is performed using "record I/O". In this situation, the read will read the next record in the file, returning that many bytes. If *nbytes* is smaller than the record length, the file pointer will be advanced to the start of the next record.

The **pread()** function is only supported for HFS files.

RETURN VALUES

If successful, the number of bytes actually read is returned. Upon reading end-of-file, zero is returned. Otherwise, a -1 is returned and the global variable `errno` is set to indicate the error.

When reading from a file with variable-length records using "record I/O", it is possible to encounter a zero-length record. Instead of returning zero (which would indicate end-of-file), the **read()** function will return -1 and set `errno` to `EAGAIN`. Thus, a program reading variable-length records can distinguish between end-of-file and a zero-length record by checking the value of `errno`.

ERRORS

read() will succeed unless:

`[EBADF]`	*d* is not a valid file or socket descriptor open for reading.
`[EFAULT]`	*buf* points outside the allocated address space.
`[EIO]`	An I/O error occurred while reading from the file system.
`[EINTR]`	A read from a slow device was interrupted before any data arrived by the delivery of a signal.
`[EINVAL]`	The pointer associated with *d* was negative.
`[EAGAIN]`	The file was marked for non-blocking I/O, and no data were ready to be read, or the file is a variable-length record file using record I/O and a zero-length record was encountered.
`[ENXIO]`	The file is not a supported I/O format.

The **pread()** function may also return the following errors:

`[EINVAL]`	The *offset* value was negative.

[EOVERFLOW] The file is an HFS file and an attemp was made to read beyond the maximum offset of the file.

[ESPIPE] The file descriptor is associated with a pipe, socket, or FIFO.

[ENXIO] The file does not support the operation, or the request was outside the capabilities of the device.

SEE ALSO

dup(2), fcntl(2), open(2)

STANDARDS

The **read()** function call is expected to conform to IEEE Std1003.1-1990 ("POSIX"), as closely as the host file system allows. The **readv()** and **pread()** functions are expected to conform to X/Open Portability Guide Issue 4, Version 2 ("XPG4.2").

READLINK(2)

NAME

readlink – read value of an //HFS:-style symbolic link

SYNOPSIS

```
#include <unistd.h>

int
readlink(const char *path, char *buf, int bufsiz);
```

DESCRIPTION

readlink() places the contents of the symbolic link *path* in the buffer *buf*, which has size *bufsiz*. The **readlink()** function does not append a `NUL` character to *buf*.

RETURN VALUES

The call returns the count of characters placed in the buffer if it succeeds, or a -1 if an error occurs, placing the error code in the global variable `errno`.

ERRORS

readlink() will fail if:

`[ENOTDIR]` A component of the path prefix is not a directory.

`[ENAMETOOLONG]` A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

`[ENOENT]` The named file does not exist.

`[EACCES]` Search permission is denied for a component of the path prefix.

`[ELOOP]` Too many symbolic links were encountered in translating the pathname.

`[EINVAL]` The named file is not a symbolic link.

`[EIO]` An I/O error occurred while reading from the file system.

`[EFAULT]` *Buf* extends outside the process's allocated address space.

RENAME(2)

NAME

rename - change the name of a file

SYNOPSIS

```
#include <stdio.h>

int
rename(const char *from, const char *to);
```

DESCRIPTION

rename() causes the file named *from* to be renamed as *to*. If *to* exists, it is first removed. Both *from* and *to* must be of the same type (that is, both DSN names, or both PDS members, or both HFS directories or both HFS non-directories), and must reside on the same file system. The types of *from* and *to* are determined by the Systems/C file naming conventions. See open(2) for more information regarding Systems/C file names.

IMPLEMENTATION NOTES

Only renaming of `//DSN:`-style and `//HFS:`-style names are supported in this release.

`//DSN:` style files must be entirely contained with 5 volumes, or **rename()** will fail.

RETURN VALUES

A 0 value is returned if the operation succeeds, otherwise **rename()** returns -1 and the global variable `errno` indicates the reason for the failure.

ERRORS

rename() will fail and neither of the argument files will be affected if:

`[ENAMETOOLONG]` For HFS files, a component of either name exceeded 255 characters, or the entire length of either path name exceeded 1023 characters.

[ENAMETOOLONG] For DSN files, a name was longer than 44 characters.

[ENOENT] A component of the *from* path does not exist, or a path prefix of *to* does not exist.

[EACCES] For HFS files, A component of either path prefix denies search permission.

[EACCES] For DSN files, the VTOC LOCATE macro indicates an acces violation.

[EACCES] The requested link requires writing in an HFS directory, a PDS directory or a VTOC with a mode that denies write permission.

[EPERM] For DSN files, permission was not granted to uncatalog the *from* name, or permission was not granted to catalog the *to* name.

[EPERM] For DSN files, permission was not granted to perform the RENAME operation.

[EPERM] The HFS directory containing *from* is marked sticky, and neither the containing directory nor *from* are owned by the effective user ID.

[EPERM] For HFS files, the *to* file exists, the HFS directory containing to is marked sticky, and neither the containing directory nor *to* are owned by the effective user ID.

[ELOOP] Too many symbolic links were encountered in translating either pathname for HFS files.

[ENOMEM] For DSN files, insufficient memory was available to perform the operation.

[ENOSYS] For DSN files, the *from* spans more than 5 volumes.

[ENOSYS] The rename operation involves unsupported file types.

[ENOTDIR] A component of either path prefix is not a directory for HFS files.

[ENOTDIR] *from* is an HFS directory, but *to* is not an HFS directory.

[EISDIR] *to* is an HFS directory, but *from* is not a HFS directory

[EXDEV] The link named by *to* and the file named by *from* are on different logical devices (file systems). Or, the file types of *to* and *from* do not match (i.e. to is a DSN and from is a DDN.) Note that this error code will not be returned if the implementation permits cross-device links.

[ENOSPC] The HFS directory, or VTOC, or PDS directory in which the entry for the new name is being placed cannot be extended because there is no space left.

`[EDQUOT]` The HFS directory in which the entry for the new name is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.

`[EIO]` An I/O error occurred while making or updating an HFS directory entry, or a VTOC or a PDS directory.

`[EROFS]` The requested link requires writing in an HFS directory, VTOC or PDS directory on a read-only file system.

`[EFAULT]` Either the *from* or *to* argument is `NULL`, or either *from* or *to* was a pointer outside of the allocated address space.

`[EINVAL]` *from* is an invalid name.

`[ENOTEMPTY]` *to* is a directory and is not empty.

SEE ALSO

open(2)

STANDARDS

The **rename()** function call is expected to conform to ISO/IEC 9945-1:1990 ("POSIX.1") as closely as the host operating system allows.

RMDIR(2)

NAME

rmdir – remove a directory file

SYNOPSIS

```
#include <unistd.h>

int
rmdir(const char *path);
```

DESCRIPTION

rmdir() removes an HFS directory file whose name is given by *path*. The directroy must not have any entries other that '.' and '..'.

RETURN VALUES

The **rmdir()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

The named file is removed unless:

`[ENOTDIR]` A component of the path is not a directory.

`[ENAMETOOLONG]` A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

`[ENOENT]` The named directory does not exist.

`[ELOOP]` Too many symbolic links were encountered in translating the pathname.

`[ENOTEMPTY]` The named directory contains files other than '.' and '..' in it.

`[EACCES]` Search permission is denied for a component of the path prefix.

`[EACCES]` Write permission is denied on the directory containing the link to be removed.

[EPERM] The directory containing the directory to be removed is marked sticky, and neither the containing directory nor the directory to be removed are owned by the effective user ID.

[EBUSY] The directory to be removed is the mount point for a mounted file system.

[EIO] An I/O error occurred while deleting the directory entry or deallocating the inode.

[EROFS] The directory entry to be removed resides on a read- only file system.

[EFAULT] Path points outside the process's allocated address space.

SEE ALSO

mkdir(2), unlink(2)

SCHED_YIELD(2)

NAME

sched_yield – yield processor

SYNOPSIS

```
#include <sched.h>

int
sched_yield(void);
```

DESCRIPTION

The **sched_yield()** system call forces the running process to relinquish the processor until it again becomes the head of its process list. It takes no arguments.

RETURN VALUES

The **sched_yield()** function returns the value 0 if successful; otherwise the value `-1` is returned and the global variable `errno` is set to indicate the error.

ERRORS

On failure errno will be set to the corresponding value:

`[ENOSYS]` The system is not configured to support this functionality.

STANDARDS

The **sched_yield()** system call conforms to IEEE Std 1003.1b-1993 ("POSIX.1").

SEMCTL(2)

NAME

semctl - control operations on a semaphore set

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int
semctl(int semid, int semnum, int cmd, ...);
```

DESCRIPTION

semctl() performs the operation indicated by *cmd* on the semaphore set indicated by *semid*. A fourth argument, a `union semun` *arg*, is required for certain values of *cmd*. For the commands that use the *arg* parameter, `union semun` is defined as follows:

```
union semun {
        int     val;            /* value for SETVAL */
        struct  semid_ds *buf;  /* buffer for IPC_STAT & IPC_SET */
        u_short *array;         /* array for GETALL & SETALL */
};
```

Commands are performed as follows:

`IPC_STAT` Fetch the semaphore set's `struct semid_ds`, storing it in the memory pointed to by *arg.buf*.

`IPC_SET` Changes the `sem_perm.uid`, `sem_perm.gid`, and `sem_perm.mode` members of the semaphore set's `struct semid_ds` to match those of the struct pointed to by *arg.buf*. The calling process's effective uid must match either `sem_perm.uid` or `sem_perm.cuid`, or it must have superuser privileges.

`IPC_RMID` Immediately removes the semaphore set from the system. The calling process's effective uid must equal the semaphore set's `sem_perm.uid` or `sem_perm.cuid`, or the process must have superuser privileges.

`GETVAL` Return the value of semaphore number *semnum*.

`SETVAL` Set the value of semaphore number *semnum* to *arg.val*.

`GETPID` Return the pid of the last process to perform an operation on semaphore number *semnum*.

`GETNCNT` Return the number of processes waiting for semaphore number *semnum*'s value to become greater than its current value.

`GETZCNT` Return the number of processes waiting for semaphore number *semnum*'s value to become 0.

`GETALL` Fetch the value of all of the semaphores in the set into the array pointed to by *arg.array*.

`SETALL` Set the values of all of the semaphores in the set to the values in the array pointed to by *arg.array*.

The `struct semid_ds` is defined as follows:

```
struct semid_ds {
        struct  ipc_perm sem_perm;      /* operation permission struct */
        struct  sem *sem_base;  /* pointer to first semaphore in set */
        u_short sem_nsems;      /* number of sems in set */
        time_t  sem_otime;      /* last operation time */
        long    sem_pad1;       /* SVABI/386 says I need this here */
        time_t  sem_ctime;      /* last change time */
                                /* Times measured in secs since */
                                /* 00:00:00 GMT, Jan. 1, 1970 */
        long    sem_pad2;       /* SVABI/386 says I need this here */
        long    sem_pad3[4];    /* SVABI/386 says I need this here */
};
```

The `sem_base` field is provided for compatibility with other operating systems, but on OS/390 and z/OS, this field will always be `NULL`.

RETURN VALUES

On success, when *cmd* is one of `GETVAL`, `GETPID`, `GETNCNT` or `GETZCNT`, **semctl()** returns the corresponding value; otherwise, 0 is returned. On failure, -1 is returned, and `errno` is set to indicate the error.

ERRORS

semctl() will fail if:

[EINVAL]	No semaphore set corresponds to *semid*.
[EINVAL]	*semnum* is not in the range of valid semaphores for given semaphore set.
[EPERM]	The calling process's effective uid does not match the uid of the semaphore set's owner or creator.
[EACCES]	Permission denied due to mismatch between operation and mode of semaphore set.

SEE ALSO

semget(2), semop(2)

SEMGET(2)

NAME

semget - obtain a semaphore id

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int
semget(key_t key, int nsems, int flag);
```

DESCRIPTION

Based on the values of *key* and *flag*, **semget()** returns the identifier of a newly created or previously existing set of semaphores. The *key* is analogous to a filename: it provides a handle that names an IPC object. There are three ways to specify a key:

- `IPC_PRIVATE` may be specified, in which case a new IPC object will be created.
- An integer constant may be specified. If no IPC object corresponding to *key* is specified and the `IPC_CREAT` bit is set in *flag*, a new one will be created.
- **ftok()** may be used to generate a key from a pathname. See ftok(3).

The mode of the newly created IPC object is determined by ORing the following constants into the *flag* parameter:

`SEM_R`	Read access for user.
`SEM_A`	Alter access for user.
`(SEM_R>>3)`	Read access for group.
`(SEM_A>>3)`	Alter access for group.
`(SEM_R>>6)`	Read access for other.
`(SEM_A>>6)`	Alter access for other.

If a new set of semaphores is being created, *nsems* is used to indicate the number of semaphores the set should contain. Otherwise, *nsems* may be specified as 0.

RETURN VALUES

semget() returns the id of a semaphore set if successful; otherwise, -1 is returned and `errno` is set to indicate the error.

ERRORS

semget() will fail if:

`[EACCES]`	Access permission failure.
`[EEXIST]`	`IPC_CREAT` and `IPC_EXCL` were specified, and a semaphore set corresponding to *key* already exists.
`[EINVAL]`	The number of semaphores requested exceeds the system imposed maximum per set.
`[ENOSPC]`	Insufficiently many semaphores are available.
`[ENOSPC]`	The system could not allocate a `struct semid_ds`.
`[ENOENT]`	No semaphore set was found corresponding to *key*, and `IPC_CREAT` was not specified.

SEE ALSO

semctl(2), semop(2), ftok(3)

SEMOP(2)

NAME

semop - atomic array of operations on a semaphore set

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int
semop(int semid, struct sembuf array[], unsigned nops);
```

DESCRIPTION

semop() atomically performs the array of operations indicated by *array* on the semaphore set indicated by *semid*. The length of *array* is indicated by *nops*. Each operation is encoded in a `struct sembuf`, which is defined as follows:

```
struct sembuf {
        u_short sem_num;        /* semaphore # */
        short   sem_op;         /* semaphore operation */
        short   sem_flg;        /* operation flags */
};
```

For each element in *array*, `sem_op` and `sem_flg` determine an operation to be performed on semaphore number `sem_num` in the set. The values `SEM_UNDO` and `IPC_NOWAIT` may be OR'ed into the `sem_flg` member in order to modify the behavior of the given operation.

The operation performed depends as follows on the value of `sem_op`:

- When `sem_op` is positive, the semaphore's value is incremented by `sem_op`'s value. If `SEM_UNDO` is specified, the semaphore's adjust on exit value is decremented by `sem_op`'s value. A positive value for `sem_op` generally corresponds to a process releasing a resource associated with the semaphore.
- The behavior when `sem_op` is negative depends on the current value of the semaphore:

- If the current value of the semaphore is greater than or equal to the absolute value of `sem_op`, then the value is decremented by the absolute value of `sem_op`. If `SEM_UNDO` is specified, the semaphore's adjust on exit value is incremented by the absolute value of `sem_op`.
- If the current value of the semaphore is less than `sem_op`'s value, one of the following happens:
  * If `IPC_NOWAIT` was specified, then **semop()** returns immediately with a return value of `EAGAIN`.
  * If some other process has removed the semaphore with the `IPC_RMID` option of **semctl()**, then fnnamesemop() returns immediately with a return value of `EINVAL`.
  * Otherwise, the calling process is put to sleep until the semaphore's value is greater than or equal to the absolute value of `sem_op`. When this condition becomes true, the semaphore's value is decremented by the absolute value of `sem_op`, and the semaphore's adjust on exit value is incremented by the absolute value of `sem_op`.

  A negative value for `sem_op` generally means that a process is waiting for a resource to become available.

- When `sem_op` is zero, the process waits for the semaphore's value to become zero. If it is already zero, the call to **semop()** can return immediately. Otherwise, the calling process is put to sleep until the semaphore's value becomes zero.

For each semaphore a process has in use, the operating system maintains an 'adjust on exit' value, as alluded to earlier. When a process exits, either voluntarily or involuntarily, the adjust on exit value for each semaphore is added to the semaphore's value. This can be used to insure that a resource is released if a process terminates unexpectedly.

RETURN VALUES

The **semop()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

semop() will fail if:

`[EINVAL]` No semaphore set corresponds to *semid*.

`[EACCES]` Permission denied due to mismatch between operation and mode of semaphore set.

[EAGAIN]	The semaphore's value was less than `sem_op`, and `IPC_NOWAIT` was specified.
[E2BIG]	Too many operations were specified.
[EFBIG]	*sem_num* was not in the range of valid semaphores for the set.

SEE ALSO

semctl(2), semget(2)

SETGROUPS(2)

NAME

setgroups - set group access list

SYNOPSIS

```
#include <sys/param.h>
#include <unistd.h>

int
setgroups(int ngroups, const gid_t *gidset);
```

DESCRIPTION

setgroups() sets the group access list of the current user process according to the array *gidset*. The parameter *ngroups* indicates the number of entries in the array and must be no more than `NGROUPS`, as defined in `<sys/param.h>`.

Only the super-user may set new groups.

RETURN VALUES

The **setgroups()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

The **setgroups()** call will fail if:

`[EPERM]` The caller is not the super-user.

`[EFAULT]` The address specified for *gidset* is outside the process address space.

SEE ALSO

getgroups(2)

_SETMODE(2)

NAME

_setmode - sets the file text vs. binary translation mode.

SYNOPSIS

```
#include <fcntl.h>

int
_setmode(int d, int mode)
```

DESCRIPTION

_setmode() alters the binary vs. text flag of the file descriptor *d*, setting it to the given *mode*. _setmode returns the previous mode value.

mode must be either `_O_TEXT` or `_O_BINARY`. `_O_TEXT` sets the file descriptor to text mode, `_O_BINARY` to binary mode. See open(2) for a description of these I/O translation modes.

_setmode() is typically used to change the default translation mode of `stdin` and `stdout`, but can be used on any open file. **_setmode()** should be applied before performing any input or output on the file descriptor.

RETURN VALUES

If successful, the previous mode value is returned. Otherwise, a -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

_setmode() will succeed unless:

`[EBADF]` *d* is not a valid, open file descriptor.

`[EINVAL]` *mode* is not `_O_TEXT` nor `O_BINARY`.

SEE ALSO

fcntl(2), open(2), read(2), write(2)

SETPGID(2)

NAME

setpgid, setpgrp - set process group

SYNOPSIS

```
#include <unistd.h>

int
setpgid(pid_t pid, pid_t pgrp);

int
setpgrp(pid_t pid, pid_t pgrp);
```

DESCRIPTION

setpgid() sets the process group of the specified process *pid* to the specified *pgrp*. If *pid* is zero, then the call applies to the current process.

If the invoker is not the super-user, then the affected process must have the same effective user-id as the invoker or be a descendant of the invoking process.

RETURN VALUES

The **setpgid()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

setpgid() will fail and the process group will not be altered if:

`[EACCES]` *Pid* is a valid child of the current process, but *pid* has been execve(2)'d. Access to the target process was denied.

`[EINVAL]` *pgrp* is an invalid process group.

`[EPERM]` The effective user ID of the requested process is different from that of the caller and the process is not a descendent of the calling process.

`[ESRCH]` The requested process does not exist.

SEE ALSO

getpgrp(2)

STANDARDS

The **setpgid()** function call is expected to conform to ISO/IEC 9945-1:1990 ("POSIX.1") as closely as the host system allows.

COMPATIBILITY

setpgrp() is identical to **setpgid()**, and is provided for calling convention compatibility with historical versions of BSD UNIX.

SETREGID(2)

NAME

setregid - set real and effective group ID

SYNOPSIS

```
#include <unistd.h>

int
setregid(gid_t rgid, gid_t egid);
```

DESCRIPTION

The real and effective group ID's of the current process are set to the arguments. Unprivileged users may change the real group ID to the effective group ID and vice-versa; only the super-user may make other changes.

Supplying a value of -1 for either the real or effective group ID forces the system to substitute the current ID in place of the -1 parameter.

Historically, the **setregid()** function was intended to allow swapping the real and effective group IDs in set-group-ID programs to temporarily relinquish the set-group-ID value. This function did not work correctly, and its purpose is now better served by the use of the **setegid()** function (see setuid(2)).

When setting the real and effective group IDs to the same value, the standard **setgid()** function is preferred.

RETURN VALUES

The **setregid()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

`[EAGAIN]` A RACF failure has occured.

`[EINVAL]` One of the parms is an invalid user id.

`[EPERM]` The current process is not the super-user and a change other than changing the effective group-id to the real group-id was specified.

SEE ALSO

getgid(2), setegid(2), setgid(2), setuid(2)

SETREUID(2)

NAME

setreuid - set real and effective user ID's

SYNOPSIS

```
#include <unistd.h>

int
setreuid(uid_t ruid, uid_t euid);
```

DESCRIPTION

The real and effective user IDs of the current process are set according to the arguments. If *ruid* or *euid* is -1, the current uid is filled in by the system. Unprivileged users may change the real user ID to the effective user ID and vice-versa; only the super-user may make other changes.

The **setreuid()** function has been used to swap the real and effective user IDs in set-user-ID programs to temporarily relinquish the set-user-ID value. This purpose is now better served by the use of the **seteuid()** function (see setuid(2)).

When setting the real and effective user IDs to the same value, the standard **setuid()** function is preferred.

RETURN VALUES

The **setreuid()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

`[EAGAIN]`	A RACF failure has occured.
`[EINVAL]`	One of the parms is an invalid user id.
`[EPERM]`	The current process is not the super-user and a change other than changing the effective user-id to the real user-id was specified.

SEE ALSO

getuid(2), seteuid(2), setuid(2)

SETSID(2)

NAME

setsid - create session and set process group ID

SYNOPSIS

```
#include <unistd.h>

pid_t
setsid(void);
```

DESCRIPTION

The **setsid()** function creates a new session. The calling process is the session leader of the new session, is the process group leader of a new process group and has no controlling terminal. The calling process is the only process in either the session or the process group.

RETURN VALUES

Upon successful completion, the **setsid()** function returns the value of the process group ID of the new process group, which is the same as the process ID of the calling process. If an error occurs, **setsid()** returns -1 and the global variable `errno` is set to indicate the error.

ERRORS

The **setsid()** function will fail if:

`[EPERM]` The calling process is already a process group leader, or the process group ID of a process other than the calling process matches the process ID of the calling process.

SEE ALSO

setpgid(2), tcgetpgrp(3), tcsetpgrp(3)

STANDARDS

The **setsid()** function is expected to be compliant with the ISO/IEC 9945-1:1990 ("POSIX.1") specification as closely as the host system allows.

SETUID(2)

NAME

setuid, seteuid, setgid, setegid - set user and group ID

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

int
setuid(uid_t uid);

int
seteuid(uid_t euid);

int
setgid(gid_t gid);

int
setegid(gid_t egid);
```

DESCRIPTION

The **setuid()** function sets the real and effective user IDs and the saved set-user-ID of the current process to the specified value. The **setuid()** function is permitted if the specified ID is equal to the real user ID or the effective user ID of the process, or if the effective user ID is that of the super user.

The **setgid()** function sets the real and effective group IDs and the saved set-group-ID of the current process to the specified value. The **setgid()** function is permitted if the specified ID is equal to the real group ID or the effective group ID of the process, or if the effective user ID is that of the super user.

The **seteuid()** function (**setegid()**) sets the effective user ID (group ID) of the current process. The effective user ID may be set to the value of the real user ID or the saved set-user-ID (see execve(2)); in this way, the effective user ID of a set-user-ID executable may be toggled by switching to the real user ID, then re-enabled by reverting to the set-user-ID value. Similarly, the effective group ID may be set to the value of the real group ID or the saved set-group-ID.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

The functions will fail if:

`[EAGAIN]`	A problem in RACF access occurred.
`[EINVAL]`	An invalid user (group) id was specified.
`[EPERM]`	The user is not the super user and the ID specified is not the real, effective ID, or saved ID.

SEE ALSO

getgid(2), getuid(2), setregid(2), setreuid(2)

STANDARDS

The **setuid()** and **setgid()** functions are compliant with the ISO/IEC 9945-1:1990 ("POSIX.1") specification with `_POSIX_SAVED_IDS` not defined with the permitted extensions from Appendix B.4.2.2. The **seteuid()** and **setegid()** functions are extensions based on the POSIX concept of `_POSIX_SAVED_IDS`, and have been proposed for a future revision of the standard.

SHMAT(2)

NAME

shmat, shmdt - attach or detach shared memory

SYNOPSIS

```
#include <machine/param.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

void *
shmat(int shmid, void *addr, int flag);

int
shmdt(void *addr);
```

DESCRIPTION

shmat() attaches the shared memory segment identified by *shmid* to the calling process's address space. The address where the segment is attached is determined as follows:

- If *addr* is 0, the segment is attached at an address selected by the system.
- If *addr* is nonzero and `SHM_RND` is not specified in *flag*, the segment is attached the specified address.
- If *addr* is specified and `SHM_RND` is specified, *addr* is rounded down to the nearest multiple of `SHMLBA`.

shmdt() detaches the shared memory segment at the address specified by *addr* from the calling process's address space.

RETURN VALUES

Upon success, **shmat()** returns the address where the segment is attached; otherwise, -1 is returned and `errno` is set to indicate the error.

The **shmdt()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

shmat() will fail if:

[EINVAL]	No shared memory segment was found corresponding to *shmid*.
[EINVAL]	*addr* was not an acceptable address.

SEE ALSO

shmctl(2), shmget(2)

SHMCTL(2)

NAME

shmctl - shared memory control

SYNOPSIS

```
#include <machine/param.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int
shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

DESCRIPTION

The **shmctl()** function performs the action specified by *cmd* on the shared memory segment identified by *shmid*:

`IPC_STAT` Fetch the segment's `struct shmid_ds`, storing it in the memory pointed to by *buf*.

`IPC_SET` Changes the `shm_perm.uid`, `shm_perm.gid`, and `shm_perm.mode` members of the segment's `struct shmid_ds` to match those of the struct pointed to by *buf*. The calling process's effective uid must match either `shm_perm.uid` or `shm_perm.cuid`, or it must have superuser privileges.

`IPC_RMID` Removes the segment from the system. The removal will not take effect until all processes having attached the segment have exited; however, once the `IPC_RMID` operation has taken place, no further processes will be allowed to attach the segment. For the operation to succeed, the calling process's effective uid must match `shm_perm.uid` or `shm_perm.cuid`, or the process must have superuser privileges.

The `shmid_ds` struct is defined as follows:

```
struct shmid_ds {
    struct ipc_perm shm_perm;   /* operation permission structure */
    int             shm_segsz;  /* size of segment in bytes */
```

```
        pid_t           shm_lpid;   /* process ID of last shared memory op */
        pid_t           shm_cpid;   /* process ID of creator */
        short           shm_nattch; /* number of current attaches */
        time_t          shm_atime;  /* time of last shmat() */
        time_t          shm_dtime;  /* time of last shmdt() */
        time_t          shm_ctime;  /* time of last change by shmctl() */
        void           *shm_internal; /* sysv stupidity */
    };
```

The `shm_internal` field is provided for compatibility with other functions.

RETURN VALUES

The **shmctl()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

shmctl() will fail if:

`[EINVAL]` Invalid operation, or no shared memory segment was found corresponding to *shmid*.

`[EPERM]` The calling process's effective uid does not match the uid of the shared memory segment's owner or creator.

`[EACCES]` Permission denied due to mismatch between operation and mode of shared memory segment.

SEE ALSO

shmat(2), shmdt(2), shmget(2), ftok(3)

SHMGET(2)

NAME

shmget - obtain a shared memory identifier

SYNOPSIS

```
#include <machine/param.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int
shmget(key_t key, int size, int flag);
```

DESCRIPTION

Based on the values of *key* and *flag*, **shmget()** returns the identifier of a newly created or previously existing shared memory segment. The key is analogous to a filename: it provides a handle that names an IPC object. There are three ways to specify a key:

- `IPC_PRIVATE` may be specified, in which case a new IPC object will be created.
- An integer constant may be specified. If no IPC object corresponding to `key` is specified and the `IPC_CREAT` bit is set in `flag`, a new one will be created.
- **ftok()** may be used to generate a key from a pathname. See ftok(3).

The mode of a newly created IPC object is determined by ORing the following constants into the *flag* parameter:

`SHM_R` Read access for user.

`SHM_W` Write access for user.

`(SHM_R>>3)` Read access for group.

`(SHM_W>>3)` Write access for group.

`(SHM_R>>6)` Read access for other.

`(SHM_W>>6)` Write access for other.

When creating a new shared memory segment, *size* indicates the desired size of the new segment in bytes. The size of the segment may be rounded up to a multiple convenient to the system (i.e., the page size).

RETURN VALUES

Upon successful completion, **shmget()** returns the positive integer identifier of a shared memory segment. Otherwise, -1 is returned and `errno` set to indicate the error.

ERRORS

shmget() will fail if:

`[EINVAL]` Size specified is greater than the size of the previously existing segment. Size specified is less than the system imposed minimum, or greater than the system imposed maximum.

`[ENOENT]` No shared memory segment was found matching *key*, and `IPC_CREAT` was not specified.

`[ENOSPC]` The system was unable to allocate enough memory to satisfy the request.

`[EEXIST]` `IPC_CREAT` and `IPC_EXCL` were specified, and a shared memory segment corresponding to key already exists.

SEE ALSO

shmat(2), shmctl(2), shmdt(2), ftok(3)

SIGACTION(2)

NAME

sigaction – software signal facilities

SYNOPSIS

```
#include <signal.h>

struct sigaction {
       union {
          void    (*__sa_handler)(int);
          void    (*__sa_sigaction)(int, struct __siginfo *, void *);

       } __sigaction_u;                /* signal handler */
       int     sa_flags;               /* see signal options below */
       sigset_t sa_mask;               /* signal mask to apply */
};

#define sa_handler      __sigaction_u.__sa_handler
#define sa_sigaction    __sigaction_u.__sa_sigaction

int
sigaction(int sig, const struct sigaction * restrict act,
           struct sigaction * restrict oact);

int __abendcode(void);
int __rsncode(void);
```

DESCRIPTION The system defines a set of signals that may be delivered to a process. Signal delivery resembles the occurrence of a hardware interrupt: the signal is normally blocked from further occurrence, the current process context is saved, and a new one is built. A process may specify a handler to which a signal is delivered, or specify that a signal is to be ignored. A process may also specify that a default action is to be taken by the system when a signal occurs. A signal may also be blocked, in which case its delivery is postponed until it is unblocked. The action to be taken on delivery is determined at the time of delivery. Normally, signal handlers execute on the current stack of the process. This may be changed, on a per-handler basis, so that signals are taken on a special signal stack.

Signal routines normally execute with the signal that caused their invocation blocked, but other signals may yet occur. A global signal mask defines the set of signals currently blocked from delivery to a process. The signal mask for a process is initialized from that of its parent (normally empty). It may be changed with a sigprocmask(2) call, or when a signal is delivered to the process.

When a signal condition arises for a process, the signal is added to a set of signals pending for the process. If the signal is not currently blocked by the process then it is delivered to the process. Signals may be delivered any time a process enters the operating system (e.g., during a system call, page fault or trap, or clock interrupt). If multiple signals are ready to be delivered at the same time, any signals that could be caused by traps are delivered first. Additional signals may be processed at the same time, with each appearing to interrupt the handlers for the previous signals before their first instructions. The set of pending signals is returned by the sigpending(2) system call. When a caught signal is delivered, the current state of the process is saved, a new signal mask is calculated (as described below), and the signal handler is invoked. The call to the handler is arranged so that if the sig- nal handling routine returns normally the process will resume execution in the context from before the signal's delivery. If the process wishes to resume in a different context, then it must arrange to restore the previous context itself.

When a signal is delivered to a process a new signal mask is installed for the duration of the process' signal handler (or until a sigprocmask(2) system call is made). This mask is formed by taking the union of the current signal mask set, the signal to be delivered, and the signal mask associated with the handler to be invoked.

The **sigaction()** system call assigns an action for a signal specified by *sig*. If *act* is non-zero, it specifies an action (`SIG_DFL`, `SIG_IGN`, or a handler routine) and mask to be used when delivering the specified signal. If *oact* is non-zero, the previous handling information for the signal is returned to the user.

Once a signal handler is installed, it normally remains installed until another **sigaction()** system call is made, or an execve(2) is performed. A signal-specific default action may be reset by setting *sa_handler* to `SIG_DFL`. The defaults are process termination, possibly with core dump; no action; stopping the process; or continuing the process. See the signal list below for each signal's default action. If *sa_handler* is `SIG_DFL`, the default action for the signal is to discard the signal, and if a signal is pending, the pending signal is discarded even if the signal is masked. If *sa_handler* is set to `SIG_IGN` current and pending instances of the signal are ignored and discarded.

Options may be specified by setting *sa_flags*. The meaning of the various bits is as follows:

`SA_NOCLDSTOP` If this bit is set when installing a catching function for the `SIGCHLD` signal, the `SIGCHLD` signal will be generated only when a child process exits, not when a child process stops.

`SA_NOCLDWAIT` If this bit is set when calling **sigaction()** for the `SIGCHLD` signal, the system will not create zombie processes when children of the calling process exit. If the calling process subsequently issues a wait(2) (or equivalent), it blocks until all of the calling process's child processes terminate, and then returns a value of -1 with `errno` set to `ECHILD`. The same effect of avoiding zombie cre- ation can also be achieved by setting *sa_handler* for `SIGCHLD` to `SIG_IGN`.

SA_ONSTACK If this bit is set, the system will deliver the signal to the process on a signal stack, specified with sigaltstack(2).

SA_NODEFER If this bit is set, further occurrences of the delivered signal are not masked during the execution of the handler.

SA_RESETHAND If this bit is set, the handler is reset back to SIG_DFL at the moment the signal is delivered.

SA_RESTART See paragraph below.

SA_SIGINFO If this bit is set, the handler function is assumed to be pointed to by the *sa_sigaction* member of *struct sigaction* and should match the prototype shown above or as below in EXAMPLES. This bit should not be set when assigning SIG_DFL or SIG_IGN.

If a signal is caught during some system calls, the call may be forced to terminate with the error EINTR, the call may return with a data transfer shorter than requested, or the call may be restarted. Restart of pending calls is requested by setting the SA_RESTART bit in *sa_flags*.

After a fork(2) or vfork(2) all signals, the signal mask, the signal stack, and the restart/interrupt flags are inherited by the child.

The execve(2) system call reinstates the default action for all signals which were caught and resets all signals to be caught on the user stack. Ignored signals remain ignored; the signal mask remains the same; signals that restart pending system calls continue to do so.

The following is a list of all signals with names as in the include file <signal.h>:

```
NAME            Default Action          Description
SIGHUP          terminate process       terminal line hangup
SIGINT          terminate process       interrupt program
SIGQUIT         create core image       quit program
SIGILL          create core image       illegal instruction
SIGTRAP         create core image       trace trap
SIGABRT         create core image       abort(3) call (formerly SIGIOT)
SIGEMT          create core image       emulate instruction executed
SIGFPE          create core image       floating-point exception
SIGKILL         terminate process       kill program
SIGBUS          create core image       bus error
SIGSEGV         create core image       segmentation violation
SIGSYS          create core image       non-existent system call invoked
SIGPIPE         terminate process       write on a pipe with no reader
SIGALRM         terminate process       real-time timer expired
SIGTERM         terminate process       software termination signal
SIGURG          discard signal          urgent condition present on
```

		socket
SIGSTOP	stop process	stop (cannot be caught or ignored)
SIGTSTP	stop process	stop signal generated from keyboard
SIGCONT	discard signal	continue after stop
SIGCHLD	discard signal	child status has changed
SIGTTIN	stop process	background read attempted from control terminal
SIGTTOU	stop process	background write attempted to control terminal
SIGIO	discard signal	I/O is possible on a descriptor (see fcntl(2))
SIGXCPU	terminate process	cpu time limit exceeded (see setrlimit(2))
SIGXFSZ	terminate process	file size limit exceeded (see setrlimit(2))
SIGVTALRM	terminate process	virtual time alarm (see setitimer(2))
SIGPROF	terminate process	profiling timer alarm (see setitimer(2))
SIGWINCH	discard signal	Window size change
SIGINFO	discard signal	status request from keyboard
SIGUSR1	terminate process	User defined signal 1
SIGUSR2	terminate process	User defined signal 2
SIGDANGER	terminate process	
SIGTHSTOP	terminate process	
SIGTHCONT	terminate process	
SIGTRACE	terminate process	
SIGDCE	terminate process	
SIGDUMP	terminate process	
SIGABND	terminate process	ABEND was encountered
SIGPOLL	terminate process	
SIGIOERR	terminate process	

NOTE

The *sa_mask* field specified in *act* is not allowed to block SIGKILL, SIGSTOP or SIGABND. Any attempt to do so will be silently ignored.

It is good practice to make a copy of the global variable errno and restore it before returning from the signal handler. This protects against the side effect of errno being set by functions called from inside the signal handler.

SIGABND Notes

The signal `SIGABND` can be used to establish a signal handler function to invoke when an ABEND is encountered. The Dignus runtime defines the two functions **__abendcode** and **__rsncode** that can be used to retrieve the ABEND and REASON codes from the ABEND information. These values are only valid within the signal handler.

Returning from a `SIGABND` handler restores execution at the point of the ABEND and will result in an infinite loop if the handler remains in effect. That is, the handler returns, the processor state is restored to the instruction that issued the ABEND, the ABEND occurs and the signal handler is re-entered. If the state of the `SIGABND` handler is `SIG_DFL` then the ABEND will be percolated to be processed in the normal manner.

The `SA_RESETHAND` flag can be used to set the `SIGABND` to `SIG_DFL` on entry to the signal handler; so that the handler may be executed once and then when the processor state is restored, normal ABEND handling will occur.

The Systems/C library issues an ABEND 978 when stack space has been exhausted. In order to be invoked for an ABEND 978, the signal handler must be defined to execute on the alternate signal stack; or the ABEND 978 will simply be re-issued to percolate via normal ABEND processing.

non-POSIX signal handling

For POSIX environment programs, the system defaults to using the POSIX signal handling interfaces in z/OS. In non-POSIX environments (BATCH/TSO), the system defaults to non-POSIX signal handling.

In non-POSIX signal handling, the TRAP(ON) option must be available to enable recognition of `SIGILL`, `SIGSEGV`, `SIGFPE` and `SIGABND`. When TRAP is ON, the runtime environment establishes an ESTAE exit to process these events and present the signals to the program. When TRAP is OFF, no ESTAE is established and normal system ABEND processing will occur without a signal being generated.

Also in a non-POSIX environment, the raise(3) function may be used to initiate a signal within a program, but since POSIX signal handling is not enabled, the program will not be able to receive a signal from an external process.

RETURN VALUES

The **sigaction()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

EXAMPLES

There are two possible prototypes the handler may match:

ANSI C:

```
void handler(int);
```

POSIX SA_SIGINFO:

```
void handler(int, siginfo_t *info, ucontext_t *uap);
```

The handler function should match the `SA_SIGINFO` prototype if the `SA_SIGINFO` bit is set in *sa_flags*. It then should be pointed to by the *sa_sigaction* member of *struct sigaction*. Note that you should not assign `SIG_DFL` or `SIG_IGN` this way.

If the `SA_SIGINFO` flag is not set, the handler function should match the ANSI C prototype and be pointed to by the *sa_handler* member of *struct sigaction*.

The *sig* argument is the signal number, one of the `SIG...` values from `<signal.h>`.

The *uap* argument to a POSIX `SA_SIGINFO` handler points to an instance of `ucontext_t`.

ERRORS

The sigaction() system call will fail and no new signal handler will be installed if one of the following occurs:

`[EFAULT]` Either *act* or *oact* points to memory that is not a valid part of the process address space.

`[EINVAL]` The *sig* argument is not a valid signal number.

`[EINVAL]` An attempt is made to ignore or supply a handler for `SIGKILL` or `SIGSTOP`.

`[EINVAL]` An attempt is made to ignore `SIGABND`.

SEE ALSO

kill(2), ptrace(2), sigaltstack(2), sigblock(2), sigpause(2), sigpending(2), sigprocmask(2), sigsetmask(2), sigsuspend(2), wait(2), fpsetmask(3), setjmp(3), siginterrupt(3), sigsetops(3), ucontext(3)

STANDARDS

The **sigaction()** system call is expected to conform to ISO/IEC 9945-1:1990 ("POSIX.1").

SIGPENDING(2)

NAME

sigpending – get pending signals

SYNOPSIS

```
#include <signal.h>

int
sigpending(sigset_t *set);
```

DESCRIPTION

The **sigpending()** system call returns a mask of the signals pending for delivery to the calling process in the location indicated by set. Signals may be pending because they are currently masked, or transiently before delivery (although the latter case is not normally detectable).

RETURN VALUES

The **sigpending()**— function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

The **sigpending()** system call will fail if:

`[EFAULT]`	The *set* argument specified an invalid address.
`[ENOSYS]`	The caller is not running in a POSIX environment.

SEE ALSO

sigaction(2), sigprocmask(2), sigsuspend(2), sigsetops(2)

STANDARDS

The **sigpending()** system call is expected to conform to ISO/IEC 9945-1:1990 (“POSIX.1”).

SIGPROCMASK(2)

NAME

sigprocmask – manipulate current signal mask

SYNOPSIS

```
#include <signal.h>

int
sigprocmask(int how, const sigset_t * restrict set,
    sigset_t * restrict oset);
```

DESCRIPTION

The **sigprocmask()** system call examines and/or changes the current signal mask (those signals that are blocked from delivery). Signals are blocked if they are members of the current signal mask set.

If *set* is not null, the action of **sigprocmask()** depends on the value of the *how* argument. The signal mask is changed as a function of the specified set and the current mask. The function is specified by *how* using one of the following values from ¡signal.h¿:

`SIG_BLOCK` The new mask is the union of the current mask and the specified set.

`SIG_UNBLOCK` The new mask is the intersection of the current mask and the complement of the specified set.

`SIG_SETMASK` The current mask is replaced by the specified set.

If *oset* is not null, it is set to the previous value of the signal mask. When *set* is null, the value of *how* is insignificant and the mask remains unset providing a way to examine the signal mask without modification.

The system quietly disallows `SIGKILL` or `SIGSTOP` to be blocked.

RETURN VALUES

The **sigprocmask()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

The **sigprocmask()** system call will fail and the signal mask will be unchanged if one of the following occurs:

`[EINVAL]` The *how* argument has a value other than those listed here.

SEE ALSO

kill(2), sigaction(2), sigpending(2), sigsuspend(2), fpsetmask(3), sigsetops(3)

STANDARDS

The **sigprocmask()** system call is expected to conform to ISO/IEC 9945-1:1990 ("POSIX.1").

SIGQUEUE(2)

NAME

sigqueue - queue a signal to a process.

SYNOPSIS

```
#include <signal.h>

int
sigqueue(pid_t pid, int signo, const union sigval value);
```

DESCRIPTION

The **sigqueue()** function causes the signal specified by *signo* to be sent with the value specified by *value* to the process specified by *pid*. If *signo* is zero (the null signal), error checking is performed but no signal is actually sent. The null signal can be used to check the validity of *pid*.

The conditions required for a process to have permission to queue a signal to another process are the same as for the kill(2) function. The **sigqueue()** function queues a signal to a single process specified by the *pid* argument.

The **sigqueue()** system call returns immediately. If the resources were available to queue the signal, the signal will be queued and sent to the receiving process.

If the value of *pid* causes signo to be generated for the sending process, and if *signo* is not blocked for the calling thread and if no other thread has *signo* unblocked or is waiting in a **sigwait()** system call for *signo*, either *signo* or at least the pending, unblocked signal will be delivered to the calling thread before **sigqueue()** returns.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

The **sigqueue()** system call will fail if:

[EAGAIN] No resources are available to queue the signal. The process has already queued (MAXQUEUEDSIGS) signals that are still pending at the receiver(s), or a system-wide resource limit has been exceeded.

[EINVAL] The value of the *signo* argument is an invalid or unsupported signal number.

[EPERM] The process does not have the appropriate privilege to send the signal to the receiving process.

[ESRCH] The process *pid* does not exist.

SEE ALSO

kill(2), sigaction(2), sigpending(2), sigsuspend(2), sigtimedwait(2), sigwait(2), sigwaitinfo(2), pause(3), pthread_sigmask(3), siginfo(3)

STANDARDS

The **sigqueue()** system call conforms to IEEE Std 1003.1-2004 ("POSIX.1").

SIGSUSPEND(2)

NAME

sigsuspend – atomically release blocked signals and wait for interrupt

SYNOPSIS

```
#include <signal.h>

int
sigsuspend(const sigset_t *sigmask);
```

DESCRIPTION

The **sigsuspend()** system call temporarily changes the blocked signal mask to the set to which *sigmask* points, and then waits for a signal to arrive; on return the previous set of masked signals is restored. The signal mask set is usually empty to indicate that all signals are to be unblocked for the duration of the call.

In normal usage, a signal is blocked using sigprocmask(2) to begin a critical section, variables modified on the occurrence of the signal are examined to determine that there is no work to be done, and the process pauses awaiting work by using **sigsuspend()** with the previous mask returned by sigprocmask(2).

RETURN VALUES

The **sigsuspend** function requires POSIX signal handling, if POSIX signal handling is not enabled, **sigsuspend** immediately returns -1 with `errno` set to `ENOSYS`.

Otherwise, the **sigsuspend()** system call will terminate by being interrupted, returning -1 with `errno` set to `EINTR`.

SEE ALSO

sigaction(2), sigpending(2), sigprocmask(2), sigsetops(3)

STANDARDS

The sigsuspend() system call is expected to conform to ISO/IEC 9945-1:1990 (“POSIX.1”).

FreeBSD 6.2 May 16, 1995 FreeBSD 6.2

SIGWAIT(2)

NAME

sigwait – select a set of signals

SYNOPSIS

```
#include <signal.h>

int
sigwait(const sigset_t * restrict set, int * restrict sig);
```

DESCRIPTION

The **sigwait()** system call selects a set of signals, specified by *set*. If none of the selected signals are pending, **sigwait()** waits until one or more of the selected signals has been generated. Then **sigwait()** atomically clears one of the selected signals from the set of pending signals for the process and sets the location pointed to by *sig* to the signal number that was cleared.

The signals specified by set should be blocked at the time of the call to **sigwait()**.

IMPLEMENATION NOTES

The **sigwait()** function depends on POSIX signals, if they are not enabled **sigwait()** immediately returns the value `ENOSYS`.

RETURN VALUES

If successful, **sigwait()** returns 0 and sets the location pointed to by *sig* to the cleared signal number. Otherwise, an error number is returned.

ERRORS

The **sigwait()** system call will fail if:

`[EINVAL]` The set argument specifies one or more invalid signal numbers.

`[ENOSYS]` **sigwait()** is invoked with POSIX signals disabled.

SEE ALSO

sigaction(2), sigpending(2), sigsuspend(2), pause(3), pthread_sigmask(3)

STANDARDS

The **sigwait()** function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

__SMF_RECORD(2)

NAME

__smf_record - generate an SMF record

SYNOPSIS

```
#include <unistd.h>

int
__smf_record(int type, int subtype, int length, char *record);
```

DESCRIPTION

The **__smf_record()** function generates an SMF record in the SMF data set with a type of *type* and a subtype of *subtype*. The record to write is is specified by the address *record* and is of *length* characters.

The caller must be APF-authorized, or permitted to the appropriate BPX.SMF facilty type or BPX.SMF.*type.subtype* resource.

For more information about SMF records consult the IBM document "z/OS MVS System Management Facilities (SMF)".

For more information about creating facility authorizations, consult the IBM document " z/OS UNIX System Services Planning.".

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

The **__smf_record()** system call will fail if:

`[EINVAL]`	The value of the *length* operand is incorrect.
`[EMVSERR]`	The SMF service returned a failing return code.
`[ENOMEM]`	Insufficent storage was available.
`[EPERM]`	Not sufficiently authorized or permission problems when accessing the BPX.SMF resource.

STAT(2)

NAME

stat, lstat, fstat - get //HFS: file status

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

int
stat(const char *path, struct stat *sb);

int
lstat(const char *path, struct stat *sb);

int
fstat(int fd, struct stat *sb);
```

DESCRIPTION

The **stat()** function obtains information about the file pointed to by *path*. Read, write or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable.

lstat() is similar to **stat()** except in the case where the named file is a symbolic link, in which case **lstat()** returns information about the link, while **stat()** returns information about the file the link references.

The **fstat()** function obtains the same information about an open file known by the file descriptor *fd*.

stat() and **lstat()** may only be applied to `//HFS:`-style files. **fstat()** can only be used in combination with a file descriptor associated with an `//HFS:`-style file.

The *sb* argument is a pointer to a **stat()** structure as defined by `<sys/stat.h>` (shown below) and into which information is placed concerning the file.

```
struct stat {
    dev_t     st_dev;              /* inode's device */
    ino_t     st_ino;              /* inode's number */
    mode_t    st_mode;             /* inode protection mode */
    nlink_t   st_nlink;            /* number of hard links */
```

```
    uid_t     st_uid;               /* user ID of the file's owner */
    gid_t     st_gid;               /* group ID of the file's group */
    dev_t     st_rdev;              /* device type */
#ifndef _POSIX_SOURCE
    struct timespec st_atimespec;  /* time of last access */
    struct timespec st_mtimespec;  /* time of last data modification */
    struct timespec st_ctimespec;  /* time of last file status change */
#else
    time_t    st_atime;             /* time of last access */
    long      st_atimensec;         /* nsec of last access */
    time_t    st_mtime;             /* time of last data modification */
    long      st_mtimensec;         /* nsec of last data modification */
    time_t    st_ctime;             /* time of last file status change */
    long      st_ctimensec;         /* nsec of last file status change */
#endif
    off_t     st_size;              /* file size, in bytes */
    int64_t   st_blocks;            /* blocks allocated for file */
    u_int32_t st_blksize;           /* optimal blocksize for I/O */
    u_int32_t st_flags;             /* user defined flags for file */
    u_int32_t st_gen;               /* file generation number */
};
```

The time-related fields of `struct stat` are as follows:

`st_atime` Time when file data last accessed. Changed by the mknod(2), utime(2) and read(2) system calls.

`st_mtime` Time when file data last modified. Changed by the mknod(2), utime(2) and write(2) system calls.

`st_ctime` Time when file status was last changed. Changed by the chmod(2), chown(2), link(2), mknod(2), rename(2), unlink(2), utime(2) and write(2) system calls.

If `_POSIX_SOURCE` is not defined, the time-related fields are defined as:

```
#ifndef _POSIX_SOURCE
#define st_atime st_atimespec.tv_sec
#define st_mtime st_mtimespec.tv_sec
#define st_ctime st_ctimespec.tv_sec
#endif
```

The size-related fields of `struct stat` are as follows:

`st_blksize` The optimal I/O block size for the file.

`st_blocks` The actual number of blocks allocated for the file in 512-byte units. This number may be zero.

The status information word `st_mode` has the following bits:

```
#define S_IFMT 0170000          /* type of file */
#define        S_IFIFO  0010000  /* named pipe (fifo) */
#define        S_IFCHR  0020000  /* character special */
#define        S_IFDIR  0040000  /* directory */
#define        S_IFBLK  0060000  /* block special */
#define        S_IFREG  0100000  /* regular */
#define        S_IFLNK  0120000  /* symbolic link */
#define        S_IFSOCK 0140000  /* socket */
#define        S_IFWHT  0160000  /* whiteout */
#define S_ISUID 0004000  /* set user id on execution */
#define S_ISGID 0002000  /* set group id on execution */
#define S_ISVTX 0001000  /* save swapped text even after use */
#define S_IRUSR 0000400  /* read permission, owner */
#define S_IWUSR 0000200  /* write permission, owner */
#define S_IXUSR 0000100  /* execute/search permission, owner */
```

For a list of access modes, see `<sys/stat.h>`, access(2) and chmod(2).

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

stat() and **lstat()** will fail if:

`[ENOTDIR]` A component of the path prefix is not a directory.

`[ENAMETOOLONG]` A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

`[ENOENT]` The named file does not exist.

`[EACCES]` Search permission is denied for a component of the path prefix.

`[ELOOP]` Too many symbolic links were encountered in translating the pathname.

`[EFAULT]` *sb* or *name* points to an invalid address.

`[EIO]` An I/O error occurred while reading from or writing to the file system.

fstat() will fail if:

`[EBADF]` *fd* is not a valid open file descriptor.

`[EFAULT]` *sb* points to an invalid address.

`[EIO]` An I/O error occurred while reading from or writing to the file system.

SEE ALSO

access(2), chmod(2), chown(2), utime(2), symlink(2)

STANDARDS

The **stat()** and **fstat()** function calls are expected to conform to ISO/IEC 9945-1:1990 ("POSIX.1") for `//HFS:`-style files.

__STEPNAME(2)

NAME

__stepname - return the current step name of the running program

SYNOPSIS

```
#include <machine/tiot.h>

char *
__stepname(void);
```

DESCRIPTION

The **__stepname()** function returns the current JCL step name of the executing program on MVS, OS/390 and z/OS. The value returned is a pointer to a NUL-terminated string. Trailing blanks are removed.

__stepname() returns a pointer to a static area, care should be taken to copy this value before invoking **__stepname()** again.

SEE ALSO

__jobname(2), __procname(2), __userid(2)

SYMLINK(2)

NAME

symlink – make symbolic link to a file

SYNOPSIS

```
#include <unistd.h>

int
symlink(const char *name1, const char *name2);
```

DESCRIPTION

A symbolic link *name2* is created to *name1* (*name2* is the name of the file created, *name1* is the string used in creating the symbolic link). Either name may be an `//HFS:`-style arbitrary path name; the files need not be on the same file system.

RETURN VALUES

The **symlink()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

The symbolic link succeeds unless:

`[ENOTDIR]` A component of the *name2* prefix is not a directory.

`[ENAMETOOLONG]` A component of either pathname exceeded 255 characters, or the entire length of either path name exceeded 1023 characters.

`[ENOENT]` The named file does not exist.

`[EACCES]` A component of the name2 path prefix denies search permission.

`[ELOOP]` Too many symbolic links were encountered in translating the pathname.

`[EEXIST]` *Name2* already exists.

[EIO] An I/O error occurred while making the directory entry for *name2*, or allocating the inode for *name2*, or writing out the link contents of *name2*.

[EROFS] The file *name2* would reside on a read-only file system.

[ENOSPC] The directory in which the entry for the new symbolic link is being placed cannot be extended because there is no space left on the file system containing the directory.

[ENOSPC] The new symbolic link cannot be created because there is no space left on the file system that will contain the symbolic link.

[EDQUOT] The directory in which the entry for the new symbolic link is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.

[EDQUOT] The new symbolic link cannot be created because the user's quota of disk blocks on the file system that will contain the symbolic link has been exhausted.

[EDQUOT] The user's quota of inodes on the file system on which the symbolic link is being created has been exhausted.

[EIO] An I/O error occurred while making the directory entry or allocating space.

[EFAULT] *Name1* or *name2* points outside the process's allocated address space.

SEE ALSO

link(2), lstat(2), readlink(2), unlink(2)

__SVC99(2)

NAME

__svc99 - issue SVC99/DYNALLOC macro

SYNOPSIS

```
#include <machine/svc99.h>

int
__svc99(__S99RB *request_block_ptr);
```

DESCRIPTION

The **__svc99** function is used to execute the MVS `SVC 99` or `DYNALLOC` interface. This low-level operating system interface provides for dynamically allocating or deallocating a resource, concatenating or deconcatenating data sets, or retrieving information about a data set.

The *request_block_ptr* points to a *__S99RB* structure that contains the following fields:

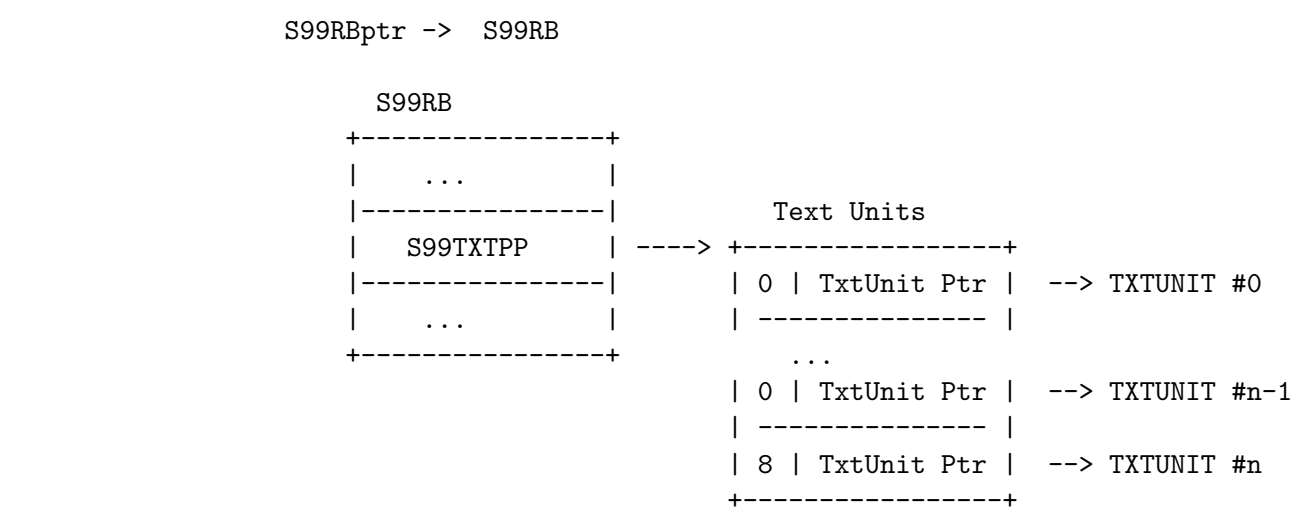
`S99RBLN`	Length (initialized to 20). output.
`S99VERB`	SVC99 action code.
`S99FLAG1`	FLAGS action code.
`S99ERROR`	Error code.
`S99INFO`	Information reason code.
`S99TXTPP`	31-bit pointer to array of text units.
`S99S99X`	31-bit pointer to request block extension (RBX).
`S99FLAGS2`	FLAGS2 field-bit pointer to request block extension.

The *request_block_ptr*, any text units and the `request block extension` must be allocated in 31-bit addressable storage. (`__malloc31()` can be employed in 64-bit programs to allocate 31-bit addressable storage.)

The *S99TXTPP* field points to an array of "Text Units" that provide further parameters and information to the `DYNALLOC` request, e.g. DDNAME, DSNAME, DSORG, etc...

A Text Unit begins with a 2-byte key that is the code for the information. The key is followed by a 2-byte field indicating the number of elements of the Text Unit (this is often simply 1.) Each element is a length-prefixed "blob" of data. The data is a 2-byte length field followed by the data bytes. This data is called the Text Unit's "parameter."

The *S99TXTPP* field of the *S99RB* structure points to an array of pointers to text units. The end of this array is marked with the high-order bit set (the VL-bit.) Thus, the basic structure is:



For more information about the `DYNALLOC` macro, "Text Units" and other functions **__svc99()** provides - refer to the "z/OS MVS Programm Authorized Assembler Services Guide" from IBM.

EXAMPLES

This program dynamically allocates a file named "MYFILE.EXAMPLE", with an allocation unit of TRACK, a primary quantity of 20 and a secondary quantity of 1, with a logical record length of 80, a block size of 80 and a fixed record format.

```
#include <stdio.h>
#include <machine/svc99.h>

int
main(void)
{
   int rc;
   __S99RB request_block;
   char *tus[10] = { /* array of text unit pointers */
```

```
    /*   TU       #      Data       Data      */
    /*  Code    Elems     Len                 */
      "\0\x02" "\0\x01" "\0\x0E" "MYFILE.EXAMPLE", /* DSN=MYFILE.EXAMPLE */
      "\0\x05" "\0\x01" "\0\x01" "\x02",           /* DISP=(,CATLG) */
      "\0\x07" "\0\0",                             /* SPACE=(TRK,.. */
      "\0\x0A" "\0\x01" "\0\x03" "\0\0\x14",       /* primary=20 */
      "\0\x0B" "\0\x01" "\0\x03" "\0\0\x01",       /* secondary=1 */
      "\0\x15" "\0\x01" "\0\x05" "SYSDA",          /* UNIT=SYSDA */
      "\0\x30" "\0\x01" "\0\x02" "\0\x50",         /* BLKSIZE=80 */
      "\0\x3C" "\0\x01" "\0\x02" "\x40\0",         /* DSORG=PS */
      "\0\x42" "\0\x01" "\0\x02" "\0\x50",         /* LRECL=80 */
      "\0\x49" "\0\x01" "\0\x01" "\x80"            /* RECFM=F */
   };

   /* The last element of the Text Units array must have */
   /* it's VL-bit set */
   tus[9] = (char *) (((unsigned int)tus[9]) | 0x80000000);

   /* Set up the SVC99 request block */
   memset(&request_block, 0, sizeof(request_block));
   request_block.S99RBLEN = 20;   /* always set to 20 */
   request_block.S99RBVERB = S99VRBAL; /* Allocation */
   request_block.S99FLAG1 |= S99NOCNV; /* Do not use an existing allocation */
   request_block.S99TXTPP = tus;

   rc = __svc99(&request_block);
   if(rc != 0) {
      printf("   SVC99 failed - Error code = %d  Information code = %d\n,
         request_block.S99ERROR, request_block.S99INFO);
   }
}
```

The following example demonstrates how to retrieve information about a file using the information retrieval function of the DYNALLOC interface. It provides a function that, given a Data Set Name displays the Data Set Organization:

```
#include <stdio.h>
#include <machine/svc99.h>

/*
*
* Interpret the DSORG value returned from SVC 99 Inquire DALRTORG
* query
**/
char *
DSORG_name(int dsorg)
{
  switch(dsorg) {
        case 0x0000: return "**UNKNOW**"; break;
        case 0x0004: return "TR"; break;        /* TCAM 3705 */
        case 0x0008: return "VSAM"; break;      /* VSAM */
        case 0x0020: return "TQ"; break;        /* TCAM message queue */
        case 0x0040: return "TX"; break;        /* TCAM line group */
        case 0x0080: return "GS"; break;        /* Graphics */
        case 0x0200: return "PO"; break;        /* Partitioned Organization */
        case 0x0300: return "POU"; break;       /* Partitioned Organization */
                                                /*  Unmovable */
        case 0x0400: return "MQ"; break;        /* Government of message */
                                                /*  transfer to or from  */
                                                /*  telecommunications   */
                                                /*  message processing queue */
        case 0x0800: return "CQ"; break;        /* Direct access message */
                                                /*  queue */
        case 0x1000: return "CX"; break;        /* Communication line group */
        case 0x2000: return "DA"; break;        /* Direct Access */
        case 0x2100: return "DAU"; break;       /* Direct Access Unmovable */
        case 0x4000: return "PS"; break;        /* Physical Sequential */
        case 0x4100: return "PSU"; break;       /* Physical Sequential */
                                                /*  Unmovable */
        case 0x8000: return "IS"; break;        /* Indexed Sequential? */
        case 0x8100: return "ISU"; break;       /* Indexed Sequential Unmovable?*/
        default: return "???"; break;
   }
}


/*
* get_DSN_org()
*       Given a DS name - get the data set organization
```

```
**/
get_DSN_org(char *dsn)
{
        int dsorg, rc;

        /* TXT UNITs */
        unsigned char TUdsname[100] = {
                0, DINDSNAM,  /* KEY - DSNAME */
                0, 1,         /* # of entries (1) */
                0, 0,         /* length of entry (set below) */
   /* remaining space used for the DSNAME (set below) */
        } ;

        unsigned char TUdsorg[] = {
                0, DINRTORG,  /* KEY - request DSORG */
                0, 1,         /* # of entries (1)  */
                0, 2,         /* parm length of 2 */
                0, 0          /* the parm bytes */
        } ;

        /* TXT units array */
        unsigned char **TextUnits[] = { TUdsname,
                        (char **)(((int)TUdsorg) | 0x80000000) /* VL-bit */
                                };

        __S99RBX request_block_extension = {
                "S99RBX",
                0x01,
                0x00,  /* No messages */
        } ;

        __S99RB request_block = {
                20,                     /* length - always 20 */
                S99VRBIN,                  /* S99VERB - Inquire function */
                0,                         /* S99FLAG1 */
                0,                         /* S99ERROR */
                0,                         /* S99INFO */
                &TextUnits,                /* S99TXTPP */
                &request_block_extension, /* S99S99X */
                0                          /* S99FLAG2 */
        } ;

        /* Set the DS NAME - up to 94 characters, blank padded */
        /*  IBM allows 44 characters here, but we have extra   */
        /*  space to allow the user to use quotes, etc.. */
        {
          int i, len;
```

```
  char *cp;
  i=0;
  len = 0;
  cp = dsn;
  while(*cp) {
        if(i<94) {
          TUdsname[i+6] = *cp;
          i++;
          len++;
        }
        cp++;
  }
  /* blank pad remaining bytes */
  for(;i<94;i++) TUdsname[i+6] = ' ';

  /* Set the length of the TU data */
  TUdsname[4] = (len >> 8);
  TUdsname[5] = len;
}

rc = __svc99(&request_block);
if(rc == 0 && request_block_extension.S99EERR == 0 &&
              request_block_extension.S99EINFO == 0) {
    /* Get the DSORG flag from the DINRTORG TextUnit */

    /* DYNALLOC doesn't fail the request if it can't */
    /* determine the DSORG, so you should check the Request Block */
    /* Extension. */
       dsorg = (TUdsorg[6] << 8) | TUdsorg[7];
       printf("get_DSN_org(\"%s\") - returned DSORG is 0x%04x (%s)\n",
                dsn, dsorg, DSORG_name(dsorg));
} else {
   printf("get_DSN_org(\"%s\") - SVC99 failed - rc is %d\n", dsn, rc);
   printf("  S99ERROR is 0x%04x\n", request_block.S99ERROR);
   if(request_block.S99ERROR == 0x0440) {
        /* DSN or Pathname not found */
      printf("     DSN not found\n");
   }
   printf("  S99INFO  is 0x%04x\n", request_block.S99INFO);
   printf("  S99EERR  is 0x%04x\n", request_block_extension.S99EERR);
   printf("  S99EINFO is 0x%04x\n", request_block_extension.S99EINFO);
}
}
```

RETURN VALUES

The **__svc99()** function returns the value 0 if successful; -1 on error, otherwise it returns the return code from the `SVC 99` invocation.

In a 64-bit environment, **__svc99()** verifies that the given request block address is in 31-bit addressable; if not it returns -1.

ISSUES

The `__svc99` function is only available on z/OS.

SEE ALSO

"z/OS MVS Programm Authorized Assembler Services Guide", __malloc31(3), __dynall(2).

SYNC(2)

NAME

sync - schedule `//HFS:` filesystem updates

SYNOPSIS

```
#include <unistd.h>

void
sync(void);
```

DESCRIPTION

The **sync()** function forces a write of dirty (modified) file system buffers in memory cache out to disk. The operating system keeps this information in memory to reduce the number of disk I/O transfers required by the system.

The function fsync(2) may be used to synchronize individual file descriptors for `//HFS:`-style files.

sync() schedules the write of file system updates, and may return before all writing is complete.

RETURN VALUES

The **sync()** function returns the value 0 if successful; otherwise the value -1 is returned.

SEE ALSO

fsync(2)

TRUNCATE(2)

NAME

truncate, ftruncate - truncate or extend a file to a specified length

SYNOPSIS

```
#include <unistd.h>

int
truncate(const char *path, off_t length);

int
ftruncate(int fd, off_t length);
```

DESCRIPTION

truncate() causes the file named by *path* or referenced by *fd* to be truncated or extended to *length* bytes in size. If the file was larger than this size, the extra data is lost. If the file was smaller than this size, it will be extended as if by writing bytes with the value zero. With **ftruncate()**, the file must be open for writing.

For non-`//HFS:`-style names, **truncate()** is implemented by opening the file and invoking **ftruncate()**.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

truncate() succeeds unless:

`[ENOTDIR]` A component of the path prefix is not a directory.

`[ENAMETOOLONG]` A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

`[ENOENT]` The named file does not exist.

[EACCES]	Search permission is denied for a component of the path prefix.
[EACCES]	The named file is not writable by the user.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EISDIR]	The named file is a directory.
[EROFS]	The named file resides on a read-only file system.
[EIO]	An I/O error occurred updating the file.
[EFAULT]	*Path* points outside the process's allocated address space.

ftruncate() succeeds unless:

[EBADF]	The *fd* is not a valid descriptor.
[EINVAL]	The *fd* references a socket, not a file.
[EINVAL]	The *fd* is not open for writing.
[EIO]	An I/O error occurred updating the file.

SEE ALSO

open(2)

ISSUES

Use of **truncate()** to extend a file is not portable.

UMASK(2)

NAME

umask – set file creation mode mask for `//HFS:`-style files

SYNOPSIS

```
#include <sys/stat.h>

mode_t
umask(mode_t numask);
```

DESCRIPTION

The **umask()** function sets the process's file mode creation mask to *numask* and returns the previous value of the mask. The 9 low-order access permission bits of *numask* are used by system calls, including open(2), mkdir(2), and mkfifo(2), to turn off corresponding bits requested in file mode for `//HFS:`-style files. (See chmod(2)). This clearing allows each user to restrict the default access to his files.

Child POSIX processes inherit the mask of the calling process.

RETURN VALUES

If OpenEdition services are available, the previous value of the file mode mask is returned. Otherwise, a value of zero is returned.

ERRORS

If OpenEdition services are not available, **umask()** returns a value of zero and sets `errno` to `ENOSYS`.

ISSUES

Unfortunately, there is no way to distinguish a return value of 0 from an intended file mask value of 0. In typical POSIX implementations, **umask()** cannot fail.

UNLINK(2)

NAME

unlink – remove `HFS:` directory entries or `//DSN:` files

SYNOPSIS

```
#include <unistd.h>

int
unlink(const char *path);
```

DESCRIPTION

For `//HFS:`-style files, the **unlink()** function removes the link named by *path* from its directory and decrements the link count of the file which was referenced by the link. If that decrement reduces the link count of the file to zero, and no process has the file open, then all resources associated with the file are reclaimed. If one or more process have the file open when the last link is removed, the link is removed, but the removal of the file is delayed until all references to it have been closed. *path* may not be a directory.

For `//DSN:`-style files, the **unlink()** function removes the entry by invoking the OS `DYNALLOC` service to allocate a the entry with a disposition of `DELETE`. **unlink()** then uses `DYNALLOC` to un-allocate the file, causing it to be deleted.

unlink() is only supported for `//HFS:` and `//DSN:`-style file names.

RETURN VALUES

The **unlink()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

The **unlink()** succeeds unless:

`[ENOTDIR]` A component of the path prefix is not a directory.

`[ENAMETOOLONG]` A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[EACCES]	Write permission is denied on the directory containing the link to be removed.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EPERM]	The named file is a directory.
[EPERM]	The directory containing the file is marked sticky, and neither the containing directory nor the file to be removed are owned by the effective user ID.
[EBUSY]	The entry to be unlinked is the mount point for a mounted file system.
[EIO]	An I/O error occurred while deleting the directory entry or deallocating the inode.
[EIO]	The `DYNALLOC` service failed for `//DSN:`-style files.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*Path* points outside the process's allocated address space.

SEE ALSO

close(2), link(2), rmdir(2)

ISSUES

The **unlink()** function only supports un-linking of `//DSN:` or `//HFS:`-style files.

UNLOCKPT(2)

NAME

unlockpt - pseudo-terminal access function

SYNOPSIS

```
#include <stdlib.h>

int
unlockpt(int filedes);
```

DESCRIPTION

The **unlockpt()** unlocks a slave pseudoterminal from its master counterpart, allowing the slave to opened. *filedes* is a file descriptor that is the result of an open(2) of the master pseudoterminal.

Secure connections can be provided by using grantpt(2) and **unlockpt()**, or by simply issuing the first open against the slave pseudoterminal from the userid or process that opened the master terminal.

RETURN VALUES

If successful, **unlockpt()** returns the value 0, otherwise a -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

unlockpt() will fail if:

`[EACCESS]` Either a grantpt(2) has not yet been issued, or **unlockpt()** has already been issued.

`[EBADF]` *filedes* is invalid, or was not opened for writing

`[EINVAL]` *filedes* is not a master pseudoterminal

SEE ALSO

grantpt(2), ptsname(3)

__USERID(2)

NAME

__userid - return the current user name

SYNOPSIS

```
#include <machine/tiot.h>

char *
__userid(void);
```

DESCRIPTION

The **__userid()** function returns the current user name of the executing program on OS/390 and z/OS. The value returned is a pointer to a NUL-terminated string. Trailing blanks are removed from the name returned by the system.

__userid() returns a pointer to a static area, care should be taken to copy this value before invoking **__userid()** again.

RETURN VALUES

If successful, **__userid()** returns a pointer to a static area that contains the current used id. If the user id is unavailable, **__userid()** returns `NULL`.

SEE ALSO

__jobname(2), __stepname(2), __procname(2)

UTIMES(2)

NAME

utimes, futimes - set //HFS: file access and modification times

SYNOPSIS

```
#include <sys/time.h>

int
utimes(const char *path, const struct timeval *times);

int
futimes(int fd, const struct timeval *times);
```

DESCRIPTION

The access and modification times of the `//HFS:` file named by *path* or referenced by *fd* are changed as specified by the argument *times*.

If *times* is `NULL`, the access and modification times are set to the current time. The caller must be the owner of the file, have permission to write the file, or be the super-user.

If *times* is non-NULL, it is assumed to point to an array of two timeval structures. The access time is set to the value of the first element, and the modification time is set to the value of the second element. The caller must be the owner of the file or be the super-user.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

utimes() will fail if:

`[EACCES]` Search permission is denied for a component of the path prefix; or the *times* argument is NULL and the effective user ID of the process does not match the owner of the file, and is not the super-user, and write access is denied.

[EFAULT] *path* or *times* points outside the process's allocated address space.

[EIO] An I/O error occurred while reading or writing the affected inode.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[ENAMETOOLONG] A component of a pathname exceeded `NAME_MAX` characters, or an entire path name exceeded `PATH_MAX` characters.

[ENOENT] The named file does not exist.

[ENOTDIR] A component of the path prefix is not a directory.

[EPERM] The *times* argument is not `NULL` and the calling process's effective user ID does not match the owner of the file and is not the super-user.

[EROFS] The file system containing the file is mounted read-only.

futimes() will fail if:

[EBADF] *fd* does not refer to a valid descriptor.

Either function will fail if:

[EACCES] The *times* argument is `NULL` and the effective user ID of the process does not match the owner of the file, and is not the super-user, and write access is denied.

[EFAULT] times points outside the process's allocated address space.

[EIO] An I/O error occurred while reading or writing the affected file information.

[EPERM] The *times* argument is not `NULL` and the calling process's effective user ID does not match the owner of the file and is not the super-user.

[EROFS] The file system containing the file is mounted read-only.

SEE ALSO

stat(2), utime(3)

VFORK(2)

NAME

vfork – spawn new process in a virtual memory efficient way

SYNOPSIS

```
#include <unistd.h>

pid_t
vfork(void);
```

DESCRIPTION

The **vfork()** system call can be used to create new processes without fully copying the address space of the old process. It is useful when the purpose of fork(2) would have been to create a new system context for an execve(2). The **vfork()** function differs from fork(2) in that the child may "borrow" the parent's memory and thread of control until a call to execve(2) or an exit (either by a call to _exit(2) or abnormally). The parent process may be suspended while the child is using its resources.

The **vfork()** system call returns 0 in the child's context and (later) the pid of the child in the parent's context.

The **vfork()** system call can normally be used just like fork(2). It does not work, however, to return while running in the child's context from the procedure that called **vfork()** since the eventual return from **vfork()** would then return to a no longer existent stack frame. The only function calls allowed in the child process are an execve(2) to load a new program image or _exit(2) to exit the child.

RETURN VALUES

Same as for fork(2).

SEE ALSO

execve(2), manref_exit2, manreffork2, manrefwait2, manrefexit3

ISSUES

The **vfork()** function has been marked as obsolete and may be removed from future standards. Portable programs should use the fork(2) function.

WAIT(2)

NAME

wait, waitpid, wait3 - wait for process termination

SYNOPSIS

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t
wait(int *status);

#include <sys/time.h>
#include <sys/resource.h>

pid_t
waitpid(pid_t wpid, int *status, int options);

pid_t
wait3(int *status, int options, struct rusage *rusage);
```

DESCRIPTION

The **wait()** function suspends execution of its calling process until *status* information is available for a terminated child process, or a signal is received. On return from a successful **wait()** call, the *status* area contains termination information about the process that exited as defined below.

For **waitpid()** the *wpid* parameter specifies the set of child processes for which to wait. If *wpid* is -1, the call waits for any child process. If *wpid* is 0, the call waits for any child process in the process group of the caller. If *wpid* is greater than zero, the call waits for the process with process id *wpid*. If *wpid* is less than -1, the call waits for any process whose process group id equals the absolute value of *wpid*.

For **waitpid()** and **wait3()**, the *status* parameter is defined below. The *options* parameter contains the bitwise OR of any of the following options. The `WNOHANG` option is used to indicate that the call should not block if there are no processes that wish to report status. If the `WUNTRACED` option is set, children of the current process that are stopped due to a SIGTTIN, SIGTTOU, SIGTSTP, or SIGSTOP signal also have their status reported.

For **wait3()**, if *rusage* is non-zero, a summary of the resources used by the terminated process and all its children is returned.

When the WNOHANG option is specified and no processes wish to report status, **wait3()** returns a process id of 0.

The **wait()** call is identical to **waitpid()** with an *options* value of zero.

The following macros may be used to test the manner of exit of the process. One of the first three macros will evaluate to a non-zero (true) value:

WIFEXITED(*status*) True if the process terminated normally by a call to _exit(2) or exit(3).

WIFSIGNALED(*status*) True if the process terminated due to receipt of a signal.

WIFSTOPPED(*status*) True if the process has not terminated, but has stopped and can be restarted. This macro can be true only if the wait call specified the WUNTRACED option or if the child process is being traced.

Depending on the values of those macros, the following macros produce the remaining status information about the child process:

WEXITSTATUS(*status*) If WIFEXITED(*status*) is true, evaluates to the low-order 8 bits of the argument passed to _exit(2) or exit(3) by the child.

WTERMSIG(*status*) If WIFSIGNALED(*status*) is true, evaluates to the number of the signal that caused the termination of the process.

WCOREDUMP(*status*) If WIFSIGNALED(*status*) is true, evaluates as true if the termination of the process was accompanied by the creation of a core file containing an image of the process when the signal was received.

WSTOPSIG(*status*) If WIFSTOPPED(*status*) is true, evaluates to the number of the signal that caused the process to stop.

NOTES

A status of 0 indicates normal termination.

If a parent process terminates without waiting for all of its child pro- cesses to terminate, the remaining child processes are assigned the par- ent process 1 ID (the init process ID).

If a signal is caught while any of the wait() calls are pending, the call may be interrupted or restarted when the signal-catching routine returns, depending on the options in effect for the signal.

RETURN VALUES

If **wait()** returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

If **wait3()**, or **waitpid()** returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. If there are no children not previously awaited, -1 is returned with errno set to `ECHILD`. Otherwise, if `WNOHANG` is specified and there are no stopped or exited children, 0 is returned. If an error is detected or a caught signal aborts the call, a value of -1 is returned and `errno` is set to indicate the error.

ERRORS

wait() will fail and return immediately if:

`[ECHILD]`	The calling process has no existing unwaited-for child processes.
`[EFAULT]`	The status or rusage arguments point to an illegal address. (May not be detected before exit of a child process.)
`[EINTR]`	The call was interrupted by a caught signal, or the signal did not have the `SA_RESTART` flag set.

STANDARDS

The **wait()** and **waitpid()** functions are defined by POSIX; **wait3()** is not specified by POSIX. The `WCOREDUMP()` macro is an extension to the POSIX interface.

SEE ALSO

_exit(2), exit(3)

WRITE(2)

NAME

write, pwrite - write output

SYNOPSIS

```
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>

size_t
write(int d, const void *buf, size_t nbytes)

ssize_t
pwrite(int d, const void *buf, size_t nbytes, off_t offset);
```

DESCRIPTION

write() attempts to write *nbytes* of data to the object referenced by the descriptor *d* from the buffer pointed to by *buf*.

On objects capable of seeking, the **write()** starts at a position given by the pointer associated with *d*, see lseek(2). Upon return from **write()**, the pointer is incremented by the number of bytes which were written.

Objects that are not capable of seeking always write from the current position. The value of the pointer associated with such an object is undefined.

When using non-blocking I/O on objects such as sockets that are subject to flow control, or when the file is opened with `_O_RECIO` flag, **write()** may write fewer bytes than requested; the return value must be noted, and the remainder of the operation should be retried when possible.

IMPLEMENTATION NOTES

When writing to objects which have been opened in `_O_TEXT` mode (the default in the Systems/C library), and the associated file is a record-structured file (non `//HFS:` and non-socket), records are padded with blanks after a new-line is encountered up to the record length. If the record if completely filled before a new-line is encountered, the record is completed and subsequent text appears on the next record, i.e. text "wraps around". Any padding bytes added are not reflected in the return value. If

the *lrecl* of the file is 1 , writes are performed as if the file had been opened with _O_BINARY specified.

If the file descriptor has been opened with _O_RECIO flag, then the write operation is performed using "record I/O". In this situation, the operation will write only record length bytes, any bytes in the buffer past the record length are discarded. If the output file is a variable-length record, then the write will generate the proper record-length specification based on the *nbytes* specified. For files with a fixed record length, if *nbytes* specifies a value smaller than the record length, the remainder of the record is filled with NUL (zero) bytes. Also note that a write of zero bytes to a variable record length file when using "record I/O" will generate a record with a zero record length. Care should be taken to ensure that is the desired result as other programs that read the file may be confused by the zero record length record. After the write operation, the file pointer will be advanced to start of the next record.

The **pwrite()** function is only supported for HFS files.

RETURN VALUES

Upon successful completion the number of bytes which were written is returned. Otherwise a -1 is returned and the global variable errno is set to indicate the error.

ERRORS

write() will fail and the file pointer will remain unchanged if:

[EAGAIN] The file was marked for non-blocking I/O, and no data could be written immediately.

[EBADF] *d* is not a valid descriptor open for writing.

[EDQUOT] The user's quota of disk blocks on the file system containing the file has been exhausted.

[EINVAL] The pointer associated with *d* was negative.

[EIO] An I/O error occurred while reading from or writing to the file system.

[ENOSPC] There is no free space remaining on the file system containing the file.

[ENXIO] The file is not a supported I/O format.

The **pwrite()** function may also return the following errors:

[EINVAL]	The *offset* value was negative.
[ESPIPE]	The file descriptor is associated with a pipe, socket, or FIFO.
[ENXIO]	The file does not support the operation, or the request was outside the capabilities of the device.

SEE ALSO

fcntl(2), lseek(2), open(2)

STANDARDS

The **write()** function call is expected to conform to IEEE Std1003.1-1990 ("POSIX"), as close as the host file system makes possible.

TCP/IP related functions

The functions described here are related to the TCP/IP implementation in the Systems/C library.

The functions described here are implemented in terms of the IBM TCP/IP implementation on OS/390. The descriptions include features which may not yet be available on that implementation (e.g. the address family `AF_UNIX` is not supported in IBM's implementation.) The description of these features are provided for completeness.

ACCEPT(2)

NAME

accept - accept a connection on a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int
accept(int s, struct sockaddr *addr, socklen_t *addrlen)
```

DESCRIPTION

The argument *s* is a socket that has been created with socket(2), bound to an address with bind(2), and is listening for connections after a listen(2). The **accept()** argument extracts the first connection request on the queue of pending connections, creates a new socket with the same properties of *s* and allocates a new file descriptor for the socket. If no pending connections are present on the queue, and the socket is not marked as non-blocking, **accept()** blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, **accept()** returns an error as described below. The accepted socket may not be used to accept more connections. The original socket *s* remains open.

The argument *addr* is a result parameter that is filled in with the address of the connecting entity, as known to the communications layer.The exact format of the *addr* parameter is determined by the domain in which the communication is occurring. The *addrlen* is a value-result parameter; it should initially contain the amount of space pointed to by *addr*; on return it will contain the actual length (in bytes) of the address returned. This call is used with connection-based socket types, currently with `SOCK_STREAM`.

It is possible to select(2) a socket for the purposes of doing an **accept()** by selecting it for read.

For certain protocols which require an explicit confirmation, such as ISO or DATAKIT, **accept()** can be thought of as merely dequeueing the next connection request and not implying confirmation. Confirmation can be implied by a normal read or write on the new file descriptor, and rejection can be implied by closing the new socket.

One can obtain user connection request data without confirming the connection by issuing a recvmsg(2) call with an *msg_iovlen* of 0 and a nonzero msg_controllen,

or by issuing a getsockopt(2) request. Similarly,one can provide user connection rejection information by issuing a sendmsg(2) call with providing only the control information, or by calling setsockopt(2).

RETURN VALUES

The call returns -1 on error. If it succeeds, it returns a non-negative integer that is a descriptor for the accepted socket.

ERRORS

The **accept()** will fail if:

[EBADF]	The descriptor is invalid.
[EINTR]	The **accept()** operation was interrupted.
[EMFILE]	The per-process descriptor table is full.
[ENFILE]	The system file table is full.
[ENOTSOCK]	The descriptor references a file, not a socket.
[EINVAL]	listen(2) has not been called on the socket descriptor.
[EFAULT]	The *addr* parameter is not in a writable part of the user address space.
[EWOULDBLOCK]	The socket is marked non-blocking and no connections are present to be accepted.

SEE ALSO

bind(2), connect(2), getpeername(2), listen(2), select(2), socket(2)

BIND(2)

NAME

bind - assign a local protocol address to a socket.

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int
bind(int s, const struct sockaddr *addr, socklen_t addrlen)
```

DESCRIPTION

bind() assigns the local protocol address to a socket. When a socket is created with socket(2) it exists in an address family space but has no protocol address assigned. **bind()** requests that *addr* be assigned to the socket.

NOTES

Binding an address in the UNIX domain creates a socket in the file system that must be deleted by the caller when it is no longer needed (using unlink(2)). UNIX domain sockets are currently unsupported in IBM's TCP/IP implementation, on which the Systems/C library is based. The documentation related to UNIX domain sockets is included for completeness.

The rules used in address binding vary between communication domains.

RETURN VALUES

If the bind is successful, a 0 value is returned. A return value of -1 indicates an error, which is further specified in the global `errno`.

ERRORS

The **bind()** call will fail if:

`[EBADF]` *s* is not a valid descriptor.

[ENOTSOCK] *s* is not a socket.

[EADDRNOTAVAIL] The specified address is not available from the local machine.

[EADDRINUSE] The specified address is already in use.

[EACCES] The requested address is protected, and the current user has inadequate permission to access it.

[EFAULT] The *addr* parameter is not in a valid part of the user address space.

The following errors are specific to binding addresses in the UNIX domain.

[ENOTDIR] A component of the path prefix is not a directory.

[ENAMETOOLONG] A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] A prefix component of the path name does not exist.

[ELOOP] Too many symbolic links were encountered in translating thepathname.

[EIO] An I/O error occurred while making the directory entry or allocating the inode.

[EROFS] The name would reside on a read-only file system.

[EISDIR] An empty pathname was specified.

SEE ALSO

connect(2), getsockname(2), listen(2), socket(2)

CONNECT(2)

NAME

connect - initiate a connection on a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int
connect(int s, const struct sockaddr *name, socklen_t namelen)
```

DESCRIPTION

The parameter *s* is a socket. If it is of type `SOCK_DGRAM`, this call specifies the peer with which the socket is to be associated; this address is that to which datagrams are to be sent, and the only address from which datagrams are to be received. If the socket is of type `SOCK_STREAM`, this call attempts to make a connection to another socket.

The other socket is specified by *name*, which is an address in the communications space of the socket. Each communications space interprets the name parameter in its own way. Generally, stream sockets may successfully **connect()** only once; datagram sockets may use **connect()** multiple times to change their association. Datagram sockets may dissolve the association by connecting to an invalid address, such as a null address.

RETURN VALUES

If the connection or binding succeeds, `0` is returned. Otherwise a `-1` is returned, and a more specific error code is stored in `errno`.

ERRORS

The **connect()** call fails if:

`[EBADF]` *s* is not a valid descriptor.

`[ENOTSOCK]` *s* is a descriptor for a file, not a socket.

[EADDRNOTAVAIL] The specified address is not available on this machine.

[EAFNOSUPPORT] Addresses in the specified address family cannot be used with this socket.

[EISCONN] The socket is already connected.

[ETIMEDOUT] connection establishment timed out without establishing a connection.

[ECONNREFUSED] The attempt to connect was forcefully rejected.

[ENETUNREACH] The network isn't reachable from this host.

[EADDRINUSE] The address is already in use.

[EFAULT] The *name* parameter specifies an area outside the process address space.

[EINPROGRESS] The socket is non-blocking and the connection cannot be completed immediately. It is possible to select(2) for completion by selecting the socket for writing.

[EALREADY] The socket is non-blocking and a previous connection attempt has not yet been completed.

The following errors are specific to connecting names in the UNIX domain. These errors may not apply in future versions of the UNIX IPC domain. Also, they are not currently support in the IBM TCP/IP implementation.

[ENOTDIR] A component of the path prefix is not a directory.

[ENAMETOOLONG] A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] The named socket does not exist.

[EACCES] Search permission is denied for a component of the path prefix.

[EACCES] Write access to the named socket is denied.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

SEE ALSO

accept(2), getpeername(2), getsockname(2), select(2), socket(2)

GETCLIENTID(2)

NAME

getclientid - get the identifier for the calling application

SYNOPSIS

```
#include <sys/socket.h>
#include <sys/types.h>

int getclientid(int domain, struct clientid *clientid);

int __getclientid(int domain, struct clientid *clientid);
```

DESCRIPTION

The **getclientid()** function call returns the identifier by which the calling application is known to the TCP/IP address space. The *clientid* can be used in the givesocket(2) and takesocket(2) calls. However, this function is supplied for use by existing programs that depend on the address space name returned.

domain is the address domain requested.

clientid as a pointer to the `struct clientid` to be filled.

The **__getclientid()** function returns the process identifier (PID) format of the `clientid` structure. This version provides improved performance and integrity over the **getclientid()** function. **__getclientid()** is only available if BPX sockets are being used.

See givesocket(2) for more information regarding the `clientid` structure.

RETURN VALUES

On success, **getclientid()** returns `0`. **getclientid()** returns `-1` on failure and sets `errno` to indicate the error:

`[EFAULT]` *clientid* points outside the caller's allocated address space.

`[ENOSYS]` **__getclientid** was invoked when the EZASMI socket interface was being used.

GETHOSTID(2)

NAME

gethostid - get unique identifier of current host

SYNOPSIS

```
#include <unistd.h>

long
gethostid(void)
```

DESCRIPTION

gethostid() returns the 32-bit identifier for the current host. Historically, this has been the unique DARPA internet address for the local machine.

RETURN VALUES

If the call fails, a value of `-1` is returned and an error code may be placed in the global location `errno`. However, `-1` is also a valid host id value.

ERRORS

The following errors may be returned by **gethostid()**:

`[ENOMEM]`	There is inadequate memory to initialize the TCP/IP system
`[EINVAL]`	The TCP/IP subsystem name is invalid
`[ENOSYS]`	The TCP/IP system is not available

SEE ALSO

gethostname(2)

ISSUES

32 bits for the unique identifier is too small. On UNIX systems, the return value `-1` is not reserved; furthermore, `-1` may be a correct return value for the host identifier. `errno` should be set to zero before the call to **gethostid()** and then examined if **gethostid()** return `-1`.

GETHOSTNAME(2)

NAME

gethostname - get name of current host

SYNOPSIS

```
#include <unistd.h>

int
gethostname(char *name, int namelen)
```

DESCRIPTION

gethostname() returns the standard host name for the current host.. The parameter *namelen* specifies the size of the *name* array. The returned name is null-terminated unless insufficient space is provided.

RETURN VALUES

If the call succeeds a value of `0` is returned. If the call fails, a value of `-1` is returned and an error code is placed in the global location `errno`.

ERRORS

The following errors may be returned by **gethostname()**:

`[EFAULT]` The *name* or *namelen* parameter gave an invalid address.

SEE ALSO

gethostid(2)

ISSUES

Host names are limited to `MAXHOSTNAMELEN` (from `<sys/param.h>`) characters, currently `256`.

GETPEERNAME(2)

NAME

getpeername - get name of connected peer

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int
getpeername(int s, struct sockaddr *name, socklen_t *namelen)
```

DESCRIPTION

getpeername() returns the name of the peer connected to socket *s*. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return it contains the actual size of the name returned (in bytes). The name is truncated if the buffer provided is too small.

RETURN VALUES

A `0` is returned if the call succeeds, `-1` if it fails.

ERRORS

The call succeeds unless:

`[EBADF]`	The argument *s* is not a valid descriptor.
`[ENOTSOCK]`	The argument *s* is a file, not a socket.
`[ENOTCONN]`	The socket is not connected.
`[ENOBUFS]`	Insufficient resources were available in the system to perform the operation.
`[EFAULT]`	The *name* parameter points to memory not in a valid part of the process address space.

SEE ALSO

accept(2), bind(2), getsockname(2), socket(2)

GETSOCKNAME(2)

NAME

getsockname - get socket name

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int
getsockname(int s, struct sockaddr *name, socklen_t *namelen)
```

DESCRIPTION

getsockname() returns the current name for the specified socket. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return it contains the actual size of the name returned (in bytes).

RETURN VALUES

A `0` is returned if the call succeeds, `-1` if it fails.

ERRORS

The call succeeds unless:

`[EBADF]`	The argument *s* is not a valid descriptor.
`[ENOTSOCK]`	The argument *s* is a file, not a socket.
`[ENOBUFS]`	Insufficient resources were available in the system to perform the operation.
`[EFAULT]`	The *name* parameter points to memory not in a valid part of the process address space.

SEE ALSO

bind(2), getpeername(2), socket(2)

ISSUES

Names bound to sockets in the UNIX domain are inaccessible; **getsockname()** returns a zero length name.

GETSOCKOPT(2)

NAME

getsockopt, setsockopt - get and set options on sockets

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int
getsockopt(int s, int level, int optname,
void *optval, socklen_t *optlen)

int
setsockopt(int s, int level, int optname,
const void *optval, socklen_t optlen)
```

DESCRIPTION

getsockopt() and **setsockopt()** manipulate the options associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost "socket" level.

When manipulating socket options the level at which the option resides and the name of the option must be specified. To manipulate options at the socket level, *level* is specified as `SOL_SOCKET`. To manipulate options at any other level the protocol number of the appropriate protocol controlling the option is supplied. For example, to indicate that an option is to be interpreted by the TCP protocol, *level* should be set to the protocol number of TCP; see getprotoent(3)

The parameters *optval* and *optlen* are used to access option values for **setsockopt()**. For **getsockopt()** they identify a buffer in which the value for the requested option(s) are to be returned. For **getsockopt()**, *optlen* is a value-result parameter, initially containing the size of the buffer pointed to by *optval*, and modified on return to indicate the actual size of the value returned. If no option value is to be supplied or returned, *optval* may be `NULL`.

optname and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The include file `<sys/socket.h>` contains definitions for socket level options, described below. Options at other protocol levels vary in format and name.

Most socket-level options utilize an `int` parameter for *optval*. For **setsockopt()**, the parameter should be non-zero to enable a boolean option, or zero if the option is to be disabled. `SO_LINGER` uses a `struct linger` parameter, defined in `<sys/socket.h>`, which specifies the desired state of the option and the linger interval (see below). `SO_SNDTIMEO` and `SO_RCVTIMEO` use a `struct timeval` parameter, defined in `<sys/time.h>`.

The following options are recognized at the socket level. Except as noted, each may be examined with **getsockopt()** and set with **setsockopt()**.

`SO_DEBUG`	enables recording of debugging information
`SO_REUSEADDR`	enables local address reuse
`SO_REUSEPORT`	enables duplicate address and port bindings
`SO_KEEPALIVE`	enables keep connections alive
`SO_DONTROUTE`	enables routing bypass for outgoing messages
`SO_LINGER`	linger on close if data present
`SO_BROADCAST`	enables permission to transmit broadcast messages
`SO_OOBINLINE`	enables reception of out-of-band data in band
`SO_SNDBUF`	set buffer size for output
`SO_RCVBUF`	set buffer size for input
`SO_SNDLOWAT`	set minimum count for output
`SO_RCVLOWAT`	set minimum count for input
`SO_SNDTIMEO`	set timeout value for output
`SO_RCVTIMEO`	set timeout value for input
`SO_TYPE`	get the type of the socket (get only)
`SO_ERROR`	get and clear error on the socket (get only)

`SO_DEBUG` enables debugging in the underlying protocol modules. `SO_REUSEADDR` indicates that the rules used in validating addresses supplied in a bind(2) call should allow reuse of local addresses. `SO_REUSEPORT` allows completely duplicate bindings by multiple processes if they all set `SO_REUSEPORT` before binding the port. This option permits multiple instances of a program to each receive UDP/IP multicast or broadcast datagrams destined for the bound port. `SO_KEEPALIVE` enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken and processes using the socket are notified via a `SIGPIPE` signal when attempting to send data. `SO_DONTROUTE` indicates that outgoing messages should bypass the standard

routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

SO_LINGER controls the action taken when unsent messages are queued on socket and a close(2) is performed. If the socket promises reliable delivery of data and SO_LINGER is set, the system will block the process on the close(2) attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period, termed the linger interval, is specified in seconds in the **setsockopt()** call when SO_LINGER is requested). If SO_LINGER is disabled and a close(2) is issued, the system will process the close in a manner that allows the process to continue as quickly as possible.

The option SO_BROADCAST requests permission to send broadcast datagrams on the socket. Broadcast was a privileged operation in earlier versions of some systems. With protocols that support out-of-band data, the SO_OOBINLINE option requests that out-of-band data be placed in the normal data input queue as received; it will then be accessible with recv(2) or read(2) calls without the MSG_OOB flag. Some protocols always behave as if this option is set. SO_SNDBUF and SO_RCVBUF are options to adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections, or may be decreased to limit the possible backlog of incoming data.

SO_SNDLOWAT is an option to set the minimum count for output operations. Most output operations process all of the data supplied by the call, delivering data to the protocol for transmission and blocking as necessary for flow control. Nonblocking output operations will process as much data as permitted subject to flow control without blocking, but will process no data if flow control does not allow the smaller of the low water mark value or the entire request to be processed. A select(2) operation testing the ability to write to a socket will return true only if the low water mark amount could be processed. The default value for SO_SNDLOWAT is set to a convenient size for network efficiency, often 1024. SO_RCVLOWAT is an option to set the minimum count for input operations. In general, receive calls will block until any (non-zero) amount of data is received, then return with the smaller of the amount available or the amount requested. The default value for SO_RCVLOWAT is 1. If SO_RCVLOWAT is set to a larger value, blocking receive calls normally wait until they have received the smaller of the low water mark value or the requested amount. Receive calls may still return less than the low water mark if an error occurs, a signal is caught, or the type of data next in the receive queue is different from that which was returned.

SO_SNDTIMEO is an option to set a timeout value for output operations. It accepts a `struct timeval` parameter with the number of seconds and microseconds used to limit waits for output operations to complete. If a send operation has blocked for this much time, it returns with a partial count or with the error EWOULDBLOCK if no data were sent. In the current implementation, this timer is restarted each time additional data are delivered to the protocol, implying that the limit applies to output portions ranging in size from the low water mark to the high water mark foroutput. SO_RCVTIMEO is an option to set a timeout value for input operations. It

accepts a `struct timeval` parameter with the number of seconds and microseconds used to limit waits for input operations to complete. In the current implementation, this timer is restarted each time additional data are received by the protocol, and thus the limit is in effect an inactivity timer. If a receive operation has been blocked for this much time without receiving additional data, it returns with a short count or with the error `EWOULDBLOCK` if no data were received.

Finally, `SO_TYPE` and `SO_ERROR` are options used only with **getsockopt()**. `SO_TYPE` returns the type of the socket, such as `SOCK_STREAM`; it is useful for servers that inherit sockets on startup. `SO_ERROR` returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

IMPLEMENTATION NOTES

Although many options are described here, only the ones available with IBM TCP/IP are actually supported. Consult the IBM TCP/IP documentation for more information.

RETURN VALUES

A `0` is returned if the call succeeds, `-1` if it fails.

ERRORS

The call succeeds unless:

`[EBADF]` The argument *s* is not a valid descriptor.

`[ENOTSOCK]` The argument *s* is a file, not a socket.

`[ENOPROTOOPT]` The option is unknown at the level indicated.

`[EFAULT]` The address pointed to by *optval* is not in a valid part of the process address space. For **getsockopt()**, this error may also be returned if *optlen* is not in a valid part of the process address space.

SEE ALSO

ioctl(2), socket(2), getprotoent(3)

GIVESOCKET(2)

NAME

givesocket - Tell TCP/IP to make the socket available

SYNOPSIS

```
#include <sys/socket.h>
int givesocket(int d, struct clientid *clientid,
int *token);
```

DESCRIPTION

givesocket() instructs TCP/IP to create a token indicating that the specified socket descriptor *d* is available to a **takesocket()** call issued by another program. The created token is returned via the token pointer, and should be used in a subsequent **takesocket()** call. Any connected stream socket can be given.

This is typically used by a master/driving program which uses **accept()** to handle incoming connections, then uses **givesocket()** to "give" the sockets to the application programs that actually handle the data. The token set by **givesocket()** is passed to the application program to use in a **takesocket()** call.

The master program passes a clientid structure to the TCP/IP system to identify the receiver of the socket.

clientid is of the form:

```
struct clientid {
   int domain;
   union  {
      char name[8];
      struct {
         int NameUpper;
         pid_t pid;
      } c_pid;
   } c_name;
   char subtaskname[8];

   struct  {
      char type;
      union  {
         char specific[19];
```

```
            struct  {
               char unused[3];
               int SockToken;
               } c_close;
         } c_func;
      } c_reserved;
};
```

`domain` is the domain of the input socket descriptor.

If the *clientid* was set by a **getclientid()** call, `c_name.name` can be set to the application program's address space name, left-justified and padded with blanks. The application program can run in the same address space as the master program, in which case this field is set to the master program's address space. Or, `c_name.name` can be set to blanks, so any OS/390 address space can take the socket.

If the *clientid* was set by a **getclientid()** call, `subtaskname` can be set to the task identifier of the taker. This, combined with a `c_name.name` value, allows only a process with this `c_name.name` and `subtaskname` to take the socket. Or, `subtaskname` can be set to blanks. If `c_name.name` has a value and `subtaskname` is blank, any task with that `c_name.name` can take the socket.

`c_reserved.type` can be set to `SO_CLOSE`, to indicate the socket should be automatically closed by **givesocket()**, and a unique socket identifying token is to be returned in `c_close.SockToken`. The `c_close.SockToken` should be passed to the taking program to be used as input to **takesocket()** instead of the socket descriptor. The now closed socket descriptor could be re-used by the time the **takesocket()** is called, so the `c_close.SockToken` should be used for **takesocket()**.

`c_close.SockToken` is a unique socket identifying token returned by givesocket to be used as input to **takesocket()**, instead of the socket descriptor when `c_reserved.type` has been set to `SO_CLOSE`.

`c_reserved` specifies binary zeros if an automatic close of a socket is not to be done by **givesocket()**.

Using `name` and `subtaskname` for givesocket/takesocket:

1. The giving program calls **getclientid()** to obtain its client ID. The giving program calls **givesocket()** to make the socket available for a **takesocket()** call. The giving program passes its client ID along with the token for the descriptor of the socket to be given to the taking program by the taking program's startup parameter list.

2. The taking program calls **takesocket()**, specifying the giving program's client ID and socket descriptor token.

3. Waiting for the taking program to take the socket, the giving program uses **select()** to test the given socket for an exception condition. When **select()** reports that an exception condition is pending, the giving program calls **close()** to free the given socket.

4. If the giving program closes the socket before a pending exception condition is indicated, the TCP connection is immediately reset, and the taking program's call to **takesocket()** is unsuccessful. Calls other than the **close()** call issued on a given socket return `-1`, with `errno` set to `EBADF`.

Note: For backward compatibility, a client ID can point to the struct client ID structure obtained when the target program calls **getclientid()**. In this case, only the target program, and no other programs in the target program's address space, can take the socket.

RETURN VALUES

On success, **givesocket()** returns `0`. On error, **givesocket()** returns `-1` and sets `errno` to the specific error.

`[EBADF]` The descriptor *d* was not a valid socket descriptor.

`[EFAULT]` The *clientid* parameter points outside the caller's allocated address space.

`[EINVAL]` The *clientid* parameter does not specify a valid client id or the domain doesn't match the domain of the input socket descriptor.

NOTES

This **givesocket()** function is different from other C libraries available on OS/390, in that it returns the token to pass to **takesocket()** as a third parameter. When porting programs from other C implementations, be sure to take this difference into account.

SEE ALSO

accept(2), close(2), getclientid(2), listen(2), select(2), takesocket(2)

IOCTL(2)

NAME

ioctl - control device

SYNOPSIS

```
#include <sys/ioctl.h>

int
ioctl(int d, unsigned long request, ...)
```

DESCRIPTION

The **ioctl()** function manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g. terminals) may be controlled with **ioctl()** requests. The argument *d* must be an open file descriptor.

The third argument to ioctl is traditionally named `char *argp`. Most uses of ioctl however, require the third argument to be a `caddr_t` or an `int`.

An ioctl request has encoded in it whether the argument is an "in" parameter or "out" parameter, and the size of the argument `argp` in bytes. Macros and defines used in specifying an ioctl request are located in the file `<sys/ioctl.h>`.

IMPLEMENTATION NOTES

The Systems/C **ioctl()** is implemented using the IBM TCP/IP ioctl interface, and thus only supports those IOCTLs that interface provides:

`FIONBIO`	Sets or clears blocking status.
`FIONREAD`	Returns the number of immediately readable bytes for the socket.
`SIOCADDRT`	Adds a specified routing table entry.
`SIOCATMARK`	Determines whether the current location in the input data is pointing to out-of-band data.
`SIOCDELRT`	Deletes a specified routine table entry.

SIOCGIFADDR	Requests the network interface address for an interface name.
SIOCGIFBRDADDR	Requests the network interface broadcast address for an interface name.
SIOCGIFCONF	Requests the network interface configuration. The configuration consists of a variable number of 32-byte arrays.
SIOCGIFDSTADDR	Requests the network interface destination address.
SIOCGIFFLAGS	Requests the network interface flags.
SIOCGIFMETRIC	Requests the network interface routing metric.
SIOCGIFNETMASK	Requests the network interface network mask.
SIOCSIFMETRIC	Sets the network interface routing metric.
SIOCSIFDSTADDR	Sets the network interface destination address.
SIOCSIFFLAGS	Sets the network interface flags.
SIOCTTLSCTL	Query or control the use of AT-TLS information for a connection.

RETURN VALUES

If an error has occurred, a value of `-1` is returned and `errno` is set to indicate the error.

ERRORS

ioctl() will fail if:

[EBADF]	*d* is not a valid descriptor.
[ENOTTY]	*d* is not associated with a character special device.
[ENOTTY]	The specified request does not apply to the kind of object that the descriptor *d* references.
[EINVAL]	Request or *argp* is not valid.
[ENOMEM]	Insufficient memory is available to satisfy the request.
[EPROTOYPE]	Socket is not TCP.

[EINVAL]	Invalid parameters passed to request.
[EPERM]	Permission denied for request.
[ENOTCONN]	Operation attempted on socket that wasn't connected.
[EPIPE]	Request was made on socket that is no longer establihed.
[EOPNOSUPP]	Request is not supported.
[EACCESS]	Access denied for request.
[EALREADY]	Request is already made or is in process.
[EPROTO]	Invalid protocol specified in request.
[EINPROGRESS]	A socket handshake is in progress.
[EWOULDBLOCK]	The socket is non-blocking and an SSL handshake is in progress.
[ENOBUFS]	The specified return area is too small.

The `errno` value can also be set according to the return value from the underlying IBM implementation. Consult the IBM "IP Communications Server" documentation for the particular ioctl() request and possible `errno` settings.

LISTEN(2)

NAME

listen - listen for connections on a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int
listen(int s, int backlog)
```

DESCRIPTION

To accept connections, a socket is first created with socket(2), a willingness to accept incoming connections and a queue limit for incoming connections are specified with **listen()**, and then the connections are accepted with accept(2). The **listen()** call applies only to sockets of type `SOCK_STREAM` or `SOCK_SEQPACKET`.

The backlog parameter defines the maximum length the queue of pending connections may grow to. If a connection request arrives with the queue full the client may receive an error with an indication of `ECONNREFUSED`, or, if the underlying protocol supports retransmission, the request may be ignored so that retries may succeed.

RETURN VALUES

A 0 return value indicates success; -1 indicates an error.

ERRORS

listen() will fail if:

`[EBADF]`	The argument *s* is not a valid descriptor.
`[ENOTSOCK]`	The argument *s* is not a socket.
`[EOPNOTSUPP]`	The socket is not of a type that supports the operation **listen()**.

SEE ALSO

accept(2), connect(2), socket(2), sysctl(3)

POLL(2)

NAME

poll – synchronous I/O multiplexing

SYNOPSIS

```
#include <poll.h>

int
poll(struct pollfd fds[], nfds_t nfds, int timeout);
```

DESCRIPTION

The **poll** function examines a set of file descriptors to see if some of them are ready for I/O. The *fds* argument is a pointer to an array of pollfd structures as defined in ¡poll.h¿ (shown below). The *nfds* argument determines the size of the *fds* array.

```
    struct pollfd {
int fd;   /* file descriptor */
short events;   /* events to look for */
short revents;  /* events returned */
    };
```

The fields of `struct pollfd` are as follows:

`fd`	File descriptor to poll. If `fd` is equal to -1 then revents is cleared (set to zero), and that pollfd is not checked.
`events`	Events to poll for. (See below.)
`revents`	Events which may occur. (See below.)

The event bitmasks in events and revents have the following bits:

`POLLIN`	Data other than high priority data may be read without blocking.
`POLLRDNORM`	Normal data may be read without blocking.

`POLLRDBAND`	Data with a non-zero priority may be read without blocking.
`POLLPRI`	High priority data may be read without blocking.
`POLLOUT`	
`POLLWRNORM`	Normal data may be written without blocking.
`POLLWRBAND`	Data with a non-zero priority may be written without blocking.
`POLLERR`	An exceptional condition has occurred on the device or socket. This flag is always checked, even if not present in the events bitmask.
`POLLHUP`	The device or socket has been disconnected. This flag is always checked, even if not present in the *events* bitmask. Note that POLLHUP and POLLOUT should never be present in the *revents* bitmask at the same time.
`POLLNVAL`	The file descriptor is not open. This flag is always checked, even if not present in the *events* bitmask.

If *timeout* is neither zero nor `INFTIM` (`-1`), it specifies a maximum interval to wait for any file descriptor to become ready, in milliseconds. If *timeout* is `INFTIM` (`-1`), the poll blocks indefinitely. If *timeout* is zero, then **poll** will return without blocking.

RETURN VALUES

The **poll** system call returns the number of descriptors that are ready for I/O, or `-1` if an error occurred. If the time limit expires, **poll** returns `0`. If **poll** returns with an error, including one due to an interrupted system call, the *fds* array will be unmodified.

COMPATIBILITY

This implementation is an emulation based on the select(2) function.

ERRORS

An error return from poll() indicates:

`[EFAULT]`	The *fds* argument points outside the process's allocated address space.

[EINTR]	A signal was delivered before the time limit expired and before any of the selected events occurred.
[EINVAL]	The specified time limit is negative.

SEE ALSO

accept(2), connect(2), kqueue(2), read(2), recv(2), select(2), send(2), write(2)

RECV(2)

NAME

recv, recvfrom, recvmsg - receive a message from a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t
recv(int s, void *buf, size_t len, int flags)

ssize_t
recvfrom(int s, void *buf, size_t len, int flags,
struct sockaddr *from, int *fromlen)

ssize_t
recvmsg(int s, struct msghdr *msg, int flags)
```

DESCRIPTION

recvfrom() and **recvmsg()** are used to receive messages from a socket, and may be used to receive data on a socket whether or not it is connection-oriented.

If *from* is non-nil, and the socket is not connection-oriented, the source address of the message is filled in. *fromlen* is a value-result parameter, initialized to the size of the buffer associated with *from*, and modified on return to indicate the actual size of the address stored there.

The **recv()** call is normally used only on a connected socket (see connect(2)) and is identical to **recvfrom()** with a nil from parameter. As it is redundant, it may not be supported in future releases.

All three routines return the length of the message on successful completion. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from (see socket(2)).

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is nonblocking (see fcntl(2)) in which case the value `-1` is returned and the external variable `errno` set to `EAGAIN`. The receive calls normally return any data available, up to the requested amount, rather than waiting for receipt of the full amount requested; this behavior is affected by the socket-level options `SO_RCVLOWAT` and `SO_RCVTIMEO` described in getsockopt(2).

The select(2) call may be used to determine when more data arrive.

The *flags* argument to a recv call is formed by or'ing one or more of the values:

`MSG_OOB`	process out-of-band data
`MSG_PEEK`	peek at incoming message
`MSG_WAITALL`	wait for full request or error

The `MSG_OOB` flag requests receipt of out-of-band data that would not be received in the normal data stream. Some protocols place expedited data at the head of the normal data queue, and thus this flag cannot be used with such protocols. The `MSG_PEEK` flag causes the receive operation to return data from the beginning of the receive queue without removing that data from the queue. Thus, a subsequent receive call will return the same data. The `MSG_WAITALL` flag requests that the operation block until the full request is satisfied. However, the call may still return less data than requested if a signal is caught, an error or disconnect occurs, or the next data to be received is of a different type than that returned.

The **recvmsg()** call uses a `msghdr` structure to minimize the number of directly supplied parameters. This structure has the following form, as defined in `<sys/socket.h>`:

```
struct msghdr {
   caddr_t msg_name;          /* optional address */
   u_int   msg_namelen;       /* size of address */
   struct  iovec *msg_iov;    /* scatter/gather */
                              /*  array */
   u_int   msg_iovlen;        /* # elements in */
                              /*  msg_iov */
   caddr_t msg_control;       /* ancillary data, */
                              /*  see below */
   u_int   msg_controllen;    /* ancillary data, */
                              /*  buffer len */
   int     msg_flags;         /* flags on */
                              /*  received message */
};
```

Here `msg_name` and `msg_namelen` specify the destination address if the socket is unconnected; `msg_name` may be given as a null pointer if no names are desired or required. `msg_iov` and `msg_iovlen` describe scatter gather locations, as discussed in read(2). `msg_control`, which has length `msg_controllen,` points to a buffer for other protocol control related messages or other miscellaneous ancillary data. The messages are of the form:

```
struct cmsghdr {
   u_int  cmsg_len;     /* data byte count, */
                        /*    including hdr */
   int    cmsg_level;   /* originating      */
                        /*    protocol      */
   int    cmsg_type;    /* protocol-specific */
                        /*    type */
                        /* followed by
   u_char cmsg_data[]; */
};
```

As an example, one could use this to learn of changes in the data-stream in XNS/SPP, or in ISO, to obtain user-connection-request data by requesting a recvmsg with no data buffer provided immediately after an **accept()** call.

Process credentials can also be passed as ancillary data for `AF_UNIX` domain sockets using a `cmsg_type` of `SCM_CREDS`. In this case, `cmsg_data` should be a structure of type `cmsgcred`, which is defined in `<sys/socket.h>` as follows:

```
struct cmsgcred {
   pid_t  cmcred_pid;          /* PID of */
                               /* sending process */
   uid_t  cmcred_uid;          /* real UID of */
                               /* sending process */
   uid_t  cmcred_euid;         /* effective UID of */
                               /* sending process */
   gid_t  cmcred_gid;          /* real GID of */
                               /*  sending process */
   short  cmcred_groups;       /* number or groups */
   gid_t  cmcred_groups[CMGROUP_MAX];  /* groups */
};
```

[Note that `AF_UNIX` domain sockets are currently not supported in the Systems/C TCP/IP library, as they are unsupported by the IBM TCP/IP implemention. This information is provided for reference.]

The kernel will fill in the credential information of the sending process and deliver it to the receiver.

The `msg_flags` field is set on return according to the message received. `MSG_EOR` indicates end-of-record; the data returned completed a record (generally used with sockets of type `SOCK_SEQPACKET`). `MSG_TRUNC` indicates that the trailing portion of a datagram was discarded because the datagram was larger than the buffer supplied. `MSG_CTRUNC` indicates that some control data were discarded due to lack of space in the buffer for ancillary data. `MSG_OOB` is returned to indicate that expedited or out-of-band data were received.

RETURN VALUES

These calls return the number of bytes received, or `-1` if an error occurred. Note that a return value of 0 indicates the connection has been closed (no bytes received.)

ERRORS

The calls fail if:

`[EBADF]`	The argument *s* is an invalid descriptor.
`[ENOTCONN]`	The socket is associated with a connection-oriented protocol and has not been connected (see connect(2) and accept(2)).
`[ENOTSOCK]`	The argument *s* does not refer to a socket.
`[EAGAIN]`	The socket is marked non-blocking, and the receive operation would block, or a receive timeout had been set, and the time-out expired before data were received.
`[EINTR]`	The receive was interrupted by delivery of a signal before any data were available.
`[EFAULT]`	The receive buffer pointer(s) point outside the process's address space.

SEE ALSO

getsockopt(2), read(2), select(2), socket(2)

SELECT(2)

NAME

select - synchronous I/O multiplexing

SYNOPSIS

```
#include <sys/types.h>
#include <sys/time.h>
#include <unistd.h>

int
select(int nfds, fd_set *readfds, fd_set *writefds,
fd_set *exceptfds, struct timeval *timeout)

FD_SET(fd, &fdset)

FD_CLR(fd, &fdset)

FD_ISSET(fd, &fdset)

FD_ZERO(&fdset)
```

DESCRIPTION

select() examines the I/O descriptor sets whose addresses are passed in *readfds*, *writefds*, and *exceptfds* to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. The only exceptional condition detectable is out-of-band data received on a socket. The first *nfds* descriptors are checked in each set; i.e., the descriptors from 0 through *nfds*-1 in the descriptor sets are examined. On return, **select()** replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. **select()** returns the total number of ready descriptors in all the sets.

The descriptor sets are stored as bit fields in arrays of integers. The following macros are provided for manipulating such descriptor sets:

`FD_ZERO(&fdset)`	initializes a descriptor set *fdset* to the null set.
`FD_SET(fd, &fdset)`	includes a particular descriptor *fd* in *fdset*.
`FD_CLR(fd, &fdset)`	removes *fd* from *fdset*.

`FD_ISSET(fd, &fdset)` — is non-zero if *fd* is a member of *fdset*, zero otherwise.

The behavior of these macros is undefined if a descriptor value is less than zero or greater than or equal to `FD_SETSIZE`, which is normally at least equal to the maxmum number of descriptors supported by the system.

If *timeout* is a non-nil pointer, it specifies a maximum interval to wait for the selection to complete. If *timeout* is a `NULL` pointer, the select blocks indefinitely. To effect a poll, the *timeout* argument should be non-`NULL`, pointing to a zero-valued `timeval` structure.

Any of *readfds*, *writefds*, and *exceptfds* may be given as `NULL` pointers if no descriptors are of interest.

RETURN VALUES

select() returns the number of ready descriptors that are contained in the descriptor sets, or `-1` if an error occurred. If the time limit expires, **select()** returns `0`. If **select()** returns with an error, including one due to an interrupted call, the descriptor sets will be unmodified.

ERRORS

An error return from **select()** indicates:

`[EBADF]` One of the descriptor sets specified an invalid descriptor.

`[EINTR]` A signal was delivered before the time limit expired and before any of the selected events occurred.

`[EINVAL]` The specified time limit is invalid. One of its components is negative or too large.

`[EINVAL]` *nfds* was invalid.

SEE ALSO

accept(2), connect(2), getdtablesize(2), gettimeofday(2), read(2), recv(2), send(2), write(2)

NOTES

The default size of `FD_SETSIZE` is currently `1024`. In order to accommodate programs which might potentially use a larger number of open files with **select()** , it is possible to increase this size by having the program define `FD_SETSIZE` before the inclusion of any header which includes `<sys/types.h>`.

If *nfds* is greater than the number of open files, **select()** is not guaranteed to examine the unused file descriptors. For historical reasons, **select()** will always examine the first `256` descriptors.

ISSUES

select() should probably return the time remaining from the original timeout, if any, by modifying the time value in place. This may be implemented in future versions of the system. Thus, it is unwise to assume that the timeout value will be unmodified by the **select()** call.

SELECTEX(2)

NAME

selectex - synchronous I/O multiplexing with extensions for message queues

SYNOPSIS

```
#include <sys/types.h>
#include <sys/time.h>
#include <unistd.h>

int
selectex(int nfds, fd_set *readfds, fd_set *writefds,
fd_set *exceptfds, struct timeval *timeout,
int *ecbptr)
```

DESCRIPTION

The **selectex()** call operates in a manner similar to select(2), except that it provides an extension to allow for using an ECB that defines an event not described by the descriptors in the *readfds*, *writefds* or *exceptfds*.

selectex() monitors activity on the file descriptors until a timeout occurs, or until the ECB is posted.

See select(2) for more information and a description of the *nfds*, *readfds*, *writefds*, *exceptfds* and *timeout* parameters and return values.

If non-`NULL`, *ecbptr* can be a pointer to a single ECB or a list of ECBs. If *ecbptr* is `NULL`, **selectex()** is equivalent to select(2).

To specify a single ECB, the high-order bit of *ecbptr* must be '0'. To specify a list of up to 59 ECBS, the high-order bit of *ecbptr* must be '1'. The high-order bit of the last pointer in the list must be '1'.

SEE ALSO

select(2)

SEND(2)

NAME

send, sendto, sendmsg - send a message from a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t
send(int s, const void *msg, size_t len, int flags)

ssize_t
sendto(int s, const void *msg, size_t len, int flags,
const struct sockaddr *to, socklen_t tolen)

ssize_t
sendmsg(int s, const struct msghdr *msg, int flags)
```

DESCRIPTION

send(), **sendto()**, and **sendmsg()** are used to transmit a message to another socket. **send()** may be used only when the socket is in a connected state, while **sendto()** and **sendmsg()** may be used at any time.

The address of the target is given by to with *tolen* specifying its size. The length of the message is given by *len*. If the message is too long to pass atomically through the underlying protocol, the error `EMSGSIZE` is returned, and the message is not transmitted.

No indication of failure to deliver is implicit in a **send()**. Locally detected errors are indicated by a return value of `-1`.

If no messages space is available at the socket to hold the message to be transmitted, then **send()** normally blocks, unless the socket has been placed in non-blocking I/O mode. The select(2) call may be used to determine when it is possible to send more data.

The flags parameter may include one or more of the following:

```
#define MSG_OOB       0x1   /* process out-of-band */
                            /*   data */
```

```
#define MSG_PEEK      0x2   /* peek at incoming */
                            /*   message */
#define MSG_DONTROUTE 0x4   /* bypass routing, */
                            /*   use direct interface */
#define MSG_EOR       0x8   /* data completes record */
#define MSG_EOF       0x100 /* data completes */
                            /* transaction */
```

The flag `MSG_OOB` is used to send "out-of-band" data on sockets that support this notion (e.g. `SOCK_STREAM`); the underlying protocol must also support "out-of-band" data. `MSG_EOR` is used to indicate a record mark for protocols which support the concept. `MSG_EOF` requests that the sender side of a socket be shut down, and that an appropriate indication be sent at the end of the specified data; this flag is only implemented for `SOCK_STREAM` sockets in the `PF_INET` protocol family, and is used to implement Transaction TCP. [Note that the Systems/C library depends on the IBM TCP/IP implementation, which may not implement this and other features.] `MSG_DONTROUTE` is usually used only by diagnostic or routing programs.

See recv(2) for a description of the `msghdr` structure.

RETURN VALUES

The call returns the number of characters sent, or `-1` if an error occurred.

ERRORS

send(), **sendto()**, and **sendmsg()** fail if:

`[EBADF]`	An invalid descriptor was specified.
`[EACCES]`	The destination address is a broadcast address, and `SO_BROADCAST` has not been set on the socket.
`[ENOTSOCK]`	The argument s is not a socket.
`[EFAULT]`	An invalid user space address was specified for a parameter.
`[EMSGSIZE]`	The socket requires that message be sent atomically, and the size of the message to be sent made this impossible.
`[EAGAIN]`	The socket is marked non-blocking and the requested operation would block.
`[ENOBUFS]`	The system was unable to allocate an internal buffer. The operation may succeed when buffers become available.

[ENOBUFS] The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but may be caused by transient congestion.

[EHOSTUNREACH] The remote host was unreachable.

ISSUES

These functions are implemented with the IBM TCP/IP interface. Not all facilities described here may be available.

Because **sendmsg()** doesn't necessarily block until the data has been transferred, it is possible to transfer an open file descriptor across an `AF_UNIX` domain socket (see recv(2)), then **close()** it before it has actu ally been sent, the result being that the receiver gets a closed file descriptor. It is left to the application to implement an acknowledgment mechanism to prevent this from happening.

SEE ALSO

getsockopt(2), recv(2), select(2), socket(2), write(2)

__SETSOCKPARM(2)

NAME

__setsockparm - define IBM TCP/IP socket function parameters

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

void
__setsockparm(int opt, ...)
```

DESCRIPTION

__setsockparm() is used to provide low-level initialization options to the underlying IBM TCP/IP implementation.

The call(s) to __setsockparm must be made before any other socket calls.

The *opt* parameter describes an option to set.

The following options are supported:

`__SP_TCPNAME`	Defines the job name to be used during socket initialization. The value following the opt parameter is a pointer to a null-terminated character string
`__SP_ADSNAME`	Defines the address name to be used during socket initialization The value following the opt parameter is a pointer to a null-terminated character string.
`__SP_SUBTASK`	Defines the subtask name. The subtask name is a field up to 8 characters which identifies the subtask. Useful for address spaces that contain multiple subtasks.

EXAMPLE

This example defines the TCP job name to be `"TCPIP"` and the address name to be `"TSO001"`:

```
char tcpname[9] = "TCPIP";
char adsname[9] = "TSO0001";

__setsockparm(__SP_TCPNAME, tcpname);
__setsockparm(__SP_ADSNAME, adsname);
```

SEE ALSO

See the *IBM Communications Server: IP Application Programming Interface Guide* for a complete description of the valid values for the TCP/IP JOB and address names.

SOCKET(2)

NAME

socket - create an endpoint for communication

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int
socket(int domain, int type, int protocol)
```

DESCRIPTION

socket() creates an endpoint for communication and returns a descriptor.

The *domain* parameter specifies a communications domain within which communication will take place; this selects the protocol family which should be used. These families are defined in the include file `<sys/socket.h>`.

The currently understood formats are

`PF_LOCAL`	(Host-internal protocols, formerly called PF_UNIX),
`PF_INET`	(ARPA Internet protocols),
`PF_ISO`	(ISO protocols),
`PF_CCITT`	(ITU-T protocols, like X.25),
`PF_NS`	(Xerox Network Systems protocols)

[Note, the Systems/C TCP/IP implementation relies on the IBM TCP/IP implementation, which only provides `PF_INET`, `PF_UNIX` and `PF_RAW`. The other communication domains are provided for reference.]

These communication domains were previously named `AF_UNIX`, `AF_INET`, `AF_ISO`, `AF_CCITT` and `AF_NS`. The older names are provided for compatibility.

The socket has the indicated *type*, which specifies the semantics of communication. Currently defined types are:

- `SOCK_STREAM`

- SOCK_DGRAM
- SOCK_RAW
- SOCK_SEQPACKET
- SOCK_RDM

[Note, the Systems/C TCP/IP implementation depends on the IBM implementation, which only provides SOCK_STREAM, SOCK_DGRAM and SOCK_RAW. Information on the other socket types is included for reference.]

A SOCK_STREAM type provides sequenced, reliable, two-way connection based byte streams. An out-of-band data transmission mechanism may be supported. A SOCK_DGRAM socket supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length). A SOCK_SEQPACKET socket may provide a sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length; a consumer may be required to read an entire packet with each read system call. This facility is protocol specific, and presently implemented only for PF_NS. SOCK_RAW sockets provide access to internal network protocols and interfaces. The types SOCK_RAW, which is available only to the superuser, and SOCK_RDM, which is planned, but not yet implemented, are not described here.

The *protocol* specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is particular to the communication domain in which communication is to take place. Some possible values for protocol are 0, IPPROTO_UDP or IPPROTO_TCP.

Sockets of type SOCK_STREAM are full-duplex byte streams, similar to pipes. A stream socket must be in a connected state before any data may be sent or received on it. A connection to another socket is created with a connect(2) call. Once connected, data may be transferred using read(2) and write(2) calls or some variant of the send(2) and recv(2) calls. (Some protocol families, such as the Internet family, support the notion of an "implied connect," which permits data to be sent piggy-backed onto a connect operation by using the sendto(2) call.) When a session has been completed a close(2) may be performed. Out-of-band data may also be transmitted as described in send(2) and received as described in recv(2).

The communications protocols used to implement a SOCK_STREAM insure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error with -1 returns and with ETIMEDOUT as the specific code in the global variable errno. The protocols optionally keep sockets "warm" by forcing transmissions roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for a extended period (e.g. 5 minutes). A SIGPIPE

signal is raised if a process sends on a broken stream; this causes naive processes, which do not handle the signal, to exit. SOCK_SEQPACKET sockets employ the same system calls as SOCK_STREAM sockets. The only difference is that read(2) calls will return only the amount of data requested, and any remaining in the arriving packet will be discarded. SOCK_DGRAM and SOCK_RAW sockets allow sending of datagrams to correspondents named in send(2) calls. Datagrams are generally received with recvfrom(2), which returns the next datagram with its return address.

The operation of sockets is controlled by socket level options. These options are defined in the file <sys/socket.h>. Setsockopt(2) and getsock opt(2) are used to set and get options, respectively.

RETURN VALUES

A -1 is returned if an error occurs, otherwise the return value is a descriptor referencing the socket.

ERRORS

The **socket()** call fails if:

[EPROTONOSUPPORT] The protocol type or the specified protocol is not supported within this domain.

[EMFILE] The per-process descriptor table is full.

[ENFILE] The system file table is full.

[EACCES] Permission to create a socket of the specified type and/or protocol is denied.

[ENOBUFS] Insufficient buffer space is available. The socket cannot be created until sufficient resources are freed.

IMPLEMENTATION NOTES

Although many options are described here, only the ones available with IBM TCP/IP are actually supported. Consult the IBM TCP/IP documentation for more information.

SEE ALSO

accept(2), bind(2), connect(2), getpeername(2), getsockname(2), getsockopt(2), ioctl(2), listen(2), read(2), recv(2), select(2), send(2), shutdown(2), socketpair(2), write(2), getprotoent(3)

SHUTDOWN(2)

NAME

shutdown - shut down part of a full-duplex connection

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int
shutdown(int s, int how)
```

DESCRIPTION

The **shutdown()** call causes all or part of a full-duplex connection on the socket associated with *s* to be shut down. If *how* is `SHUT_RD` (`0`), further receives will be disallowed. If *how* is `SHUT_WR` (`1`), further sends will be disallowed. If *how* is `SHUT_RDWR` (`2`), further sends and receives will be disallowed.

RETURN VALUES

A `0` is returned if the call succeeds, `-1` if it fails.

ERRORS

The call succeeds unless:

`[EBADF]` *s* is not a valid descriptor.

`[ENOTSOCK]` *s* is a file, not a socket.

`[ENOTCONN]` The specified socket is not connected.

SEE ALSO

connect(2), socket(2)

STANDARDS

The **shutdown()** function is expected to comply with IEEE P1003.1g ("POSIX"), when finalized.

TAKESOCKET(3)

NAME

takesocket - acquire a socket from another program

SYNOPSIS

```
#include <sys/types.h>
#include <socket.h>

int takesocket(struct clientid *clientid,int token);
```

DESCRIPTION

The **takesocket()** function acquires a socket from another program viathe token passed from the other program. Typically, the other program passes its client ID and givesocket token, and/or process id (PID), to your program through your program's startup parameter list.

clientid is a pointer to the clientid of the application from which you are taking a socket.

token is the token generated by a **givesocket()** call, which represents the socket to be taken.

If your program is using the PID to ensure integrity between **givesocket()** and **takesocket()**, before issuing the **takesocket()** call, your program should set the `c_pid.pid` field of the clientid structure to the PID of the giving program (i.e. program that issued the **givesocket()** call). This identifies the process from which the socket is to be taken. If the `c_reserved.type` field of the clientid structure was set to `SO_CLOSE` on the **givesocket()** call, `c_close.SockToken` of the clientid structure should be used as input to **takesocket()** instead of the normal socket descriptor. See givesocket(2) for a description of the clientid structure.

RETURN VALUE

takesocket() returns the new socket descriptor, or `-1` on error. If the return value is `-1`, `errno` is set to:

`[EACCES]`	The other application did not give the socket to this application.
`[EBADF]`	The *token* parameter does not specify a valid token from the other application, or the socket has already been taken.

[EFAULT] The *clientid* parameter points outside the process's allocated address space.

[EINVAL] The *clientid* parameter does not specify a valid client identifier. Either the client process cannot be found, or the client exists, but has no outstanding "given" sockets.

[EMFILE] The file descriptor table is full.

SEE ALSO

getclientid(2), givesocket(2)

Gen Library

Historically, the "gen" portion of a C library are those files which are automatically generated, or which are generated in a platform-specific manner. For the Systems/C library, the distinction isn't as meaningful as it may be on other platforms.

__ATOE(3)

NAME

__atoe, __etoa, __stratoe, __stretoa, __strnatoe, __strnetoa, __bcopy_atoe, __bcopy_etao - ASCII/EBCDIC character translation functions

SYNOPSIS

```
#include <machine/atoe.h>

unsigned int __atoe(unsigned int char);

unsigned int __etoa(unsigned int char);

void __stratoe(unsigned char * string);

void __stretoa(unsigned char * string);

void __strnatoe(unsigned char *string, int len);

void __strnetoa(unsigned char *string, int len);

void __bcopy_atoe(unsigned char *src, unsigned char *dst, int len);

void __bcopy_etoa(unsigned char *src, unsigned char *dst, int len);
```

DESCRIPTION

The **__atoe()** and **__etoa()** functions translate a value in the range 0-255 to/from ASCII and EBCDIC. The translation table employed is the same used by the Systems/C compiler and utilties, and assumes the IBM 1047 code page.

The **__stratoe()** and **__stretoa()** functions apply the translation directly to a `NUL`-terminate string.

The **__strnatoe()** and **__strnetoa()** functions apply the translation to a string; the translation stops when either the `NUL` terminating character is discovered, or the length *len* is reached.

The **__bcopy_atoe()** and **__bcopy_etoa()** functions copy *len* bytes from the *src* address to the *dst* address, translating the bytes as they are copied. If *len* is zero, no bytes are copied.

SEE ALSO

bcopy(3), strcpy(3), strncpy(3)

__TO_XX(3)

NAME

__to_b1, __to_b2, __to_b4, __to_d1, __to_d2, __to_d4, __to_h1, __to_h2, __to_h4 - floating point conversion functions

SYNOPSIS

These functions don't appear in any header file, thus, the `#pragma map` statements must be properly provided to use them.

```
#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

#pragma map(__to_b1, "@@TO@B1")
float __to_b1(unsigned int flags, void *input_p)

#pragma map(__to_b2, "@@TO@B2")
double __to_b2(unsigned int flags, void *input_p)

#pragma map(__to_b4, "@@TO@B4")
long double __to_b4(unsigned int flags, void *input_p)

#pragma map(__to_d1, "@@TO@D1")
_Decimal32 __to_d1(unsigned int flags, void *input_p)

#pragma map(__to_d2, "@@TO@D2")
_Decimal64 __to_d2(unsigned int flags, void *input_p)

#pragma map(__to_d4, "@@TO@D4")
_Decimal128 __to_d4(unsigned int flags, void *input_p)

#pragma map(__to_h1, "@@TO@H1")
float __to_h1(unsigned int flags, void *input_p)

#pragma map(__to_h2, "@@TO@H2")
double __to_h2(unsigned int flags, void *input_p)

#pragma map(__to_h4, "@@TO@H4")
long double __to_h4(unsigned int flags, void *input_p)

#ifdef __cplusplus
}
```

```
#endif /* __cplusplus */
```

DESCRIPTION

These functions convert the input floating point value addressed by *input_p*, returning an IEEE (BFP), Decimal Floating Point (DFP) or Hexadecimal Floating point (HFP) value. The **__to_b1**, **__to_b2** and **__to_b4** functions return IEEE (BFP) values of the specified return type. The **__to_d1**, **__to_d2** and **__to_d4** return Decimal Floating Point (DFP) values of the given sizes. The **__to_h1**, **__to_h2** and **__to_h4** return Hexadecimal Floating Point (HFP) values of the given sizes. The *flags* parameter provides flags indicating the type of the input value addressed by *input_p*.

These functions do not require any particular hardware architecture support.

The value of the *flags* parameter describes the input parameter and the requested rounding mode of the result. These two values are OR'd together to create the value. For the type of input parameter, one of the following values should be used:

0x000	HFP float
0x100	HFP double
0x200	HFP long double
0x500	BFP float
0x600	BFP double
0x700	BFP long double
0x800	DFP _Decimal32
0x900	DFP _Decimal64
0xA00	DFP _Decimal128

The following values should be used to indicate the rounding mode:

0x00	Round DFP values as indicated in **fe_dec_getround()**.
0x01	Round BFP values as indicated by **fegetround()**.
0x08	Round to Nearest Ties Even
0x09	Round Toward Zero
0x0A	Round Toward +Infinity

0x0B	Rount Toward -Infinity
0x0C	Round to Nearest, Ties Away from Zero
0x0D	Round to Nearest, Ties Toward from Zero
0x0E	Round Away from Zero
0x0F	Round Prepare for Shorter Precision

If the conversion specified in *conv_flag* is not valid a a value of 0.0 is returned.

SEE ALSO

An explanation of the rounding modes can be found in the z/Architecture Principles of Operations.

fenv(3)

ALARM(3)

NAME

alarm – set signal timer alarm

SYNOPSIS

```
#include <unistd.h>

unsigned int
alarm(unsigned int seconds);
```

DESCRIPTION

This interface is made obsolete by setitimer(2).

The **alarm()** function sets a timer to deliver the signal `SIGALRM` to the calling process after the specified number of seconds. If an alarm has already been set with **alarm()** but has not been delivered, another call to **alarm()** will supersede the prior call. The request **alarm(0)** voids the current alarm and the signal `SIGALRM` will not be delivered.

Due to setitimer(2) restriction the maximum number of seconds allowed is 100000000.

RETURN VALUES

The return value of **alarm()** is the amount of time left on the timer from a previous call to **alarm()**. If no alarm is currently set, the return value is 0.

SEE ALSO

setitimer(2), sigaction(2), sigpause(2), sigvec(2), signal(3), sleep(3)

ASSERT(3)

NAME

assert - expression verification macro

SYNOPSIS

```
#include <assert.h>

assert(expression)
```

DESCRIPTION

The **assert()** macro tests the given *expression* and if it is false, the calling process is terminated. A diagnostic message is written to stderr and the function abort(3) is called effectively terminating the program.

If expression is true, the **assert()** macro does nothing.

The **assert()** macro may be removed at compile time with the -DNDEBUG option, see the -D option description in the compiler documentation.

DIAGNOSTICS

The following diagnostic message is written to stderr if expression is false:

```
"assertion \"%s\" failed: file \"%s\", line %d\n", \
                    "expression", __FILE__, __LINE__
```

SEE ALSO

abort(3)

BITSTRING(3)

NAME

bit_alloc, bit_clear, bit_decl, bit_ffs, bit_nclear, bit_nset, bit_set, bitstr_size, bit_test - bit-string manipulation macros

SYMNOPSIS

```
#include <bitstring.h>

bitstr_t *
bit_alloc(int nbits);

void
bit_decl(bitstr_t *name, int nbits);

void
bit_clear(bitstr_t *name, int bit);

void
bit_ffc(bitstr_t *name, int nbits, int *value);

void
bit_ffs(bitstr_t *name, int nbits, int *value);

void
bit_nclear(bitstr_t *name, int start, int stop);

void
bit_nset(bitstr_t *name, int start, int stop);

void
bit_set(bitstr_t *name, int bit);

int
bitstr_size(int nbits);

int
bit_test(bitstr_t *name, int bit);
```

DESCRIPTION

These macros operate on strings of bits.

The macro `bit_alloc()` returns a pointer of type "`bitstr_t *`" to sufficient space to store *nbits* bits, or `NULL` if no space is available.

The macro `bit_decl()` allocates sufficient space to store *nbits* bits on the stack.

The macro `bitstr_size()` returns the number of elements of type `bitstr_t` necessary to store *nbits* bits. This is useful for copying bit strings.

The macros `bit_clear()` and `bit_set()` clear or set the zero-based numbered bit *bit*, in the bit string *name*.

The `bit_nset()` and `bit_nclear()` macros set or clear the zero-based numbered bits from *start* through *stop* in the bit string *name*.

The `bit_test()` macro evaluates to non-zero if the zero-based numbered bit *bit* of bit string *name* is set, and zero otherwise.

The `bit_ffs()` macro stores in the location referenced by *value* the zero-based number of the first bit set in the array of *nbits* bits referenced by *name*. If no bits are set, the location referenced by value is set to -1.

The macro `bit_ffc()` stores in the location referenced by *value* the zero-based number of the first bit not set in the array of *nbits* bits referenced by *name*. If all bits are set, the location referenced by *value* is set to -1.

The arguments to these macros are evaluated only once and may safely have side effects.

EXAMPLES

```
#include <limits.h>
#include <bitstring.h>

...
#define LPR_BUSY_BIT            0
#define LPR_FORMAT_BIT          1
#define LPR_DOWNLOAD_BIT        2
...
#define LPR_AVAILABLE_BIT       9
#define LPR_MAX_BITS            10

make_lpr_available()
{
    bitstr_t bit_decl(bitlist, LPR_MAX_BITS);
    ...
    bit_nclear(bitlist, 0, LPR_MAX_BITS - 1);
    ...
```

```
    if (!bit_test(bitlist, LPR_BUSY_BIT)) {
        bit_clear(bitlist, LPR_FORMAT_BIT);
        bit_clear(bitlist, LPR_DOWNLOAD_BIT);
        bit_set(bitlist, LPR_AVAILABLE_BIT);
    }
}
```

SEE ALSO

memory(3)

CLOCK(3)

NAME

clock - determine processor time used

SYNOPSIS

```
#include <time.h>

clock_t
clock(void)
```

DESCRIPTION

The **clock()** function determines the amount of processor time used since the invocation of the calling process, measured in `CLOCKS_PER_SEC`'s of a second.

RETURN VALUES

The **clock()** function returns the amount of time used unless an error occurs, in which case the return value is -1.

STANDARDS

The **clock()** function conforms to ISO/IEC 9899:1990 ("ISO C90").

CTERMID(3)

NAME

ctermid – generate terminal pathname

SYNOPSIS

```
#include <stdio.h>

char *
ctermid(char *buf);

char *
ctermid_r(char *buf);
```

DESCRIPTION

The **ctermid()** function generates a string, that, when used as a pathname, refers to the current controlling terminal of the calling process.

If *buf* is the `NULL` pointer, a pointer to a static area is returned. Otherwise, the pathname is copied into the memory referenced by *buf*. The argument *buf* is assumed to be at least `L_ctermid` (as defined in the include file `<stdio.h>`) bytes long.

The **ctermid_r()** function provides the same functionality as **ctermid()** except that if *buf* is a `NULL` pointer, `NULL` is returned.

The current implementation simply returns '`/dev/tty`' when running under OpenEdition. In any other environment, it returns the empty string.

RETURN VALUES

Upon successful completion, a non-`NULL` pointer is returned. Otherwise, a `NULL` pointer is returned and the global variable `errno` is set to indicate the error.

ERRORS

The current implementation detects no error conditions.

SEE ALSO

ttyname(3)

STANDARDS

The **ctermid()** function conforms to IEEE Std 1003.1-1988 (“POSIX.1”).

ISSUES

By default the **ctermid()** function writes all information to an internal static object. Subsequent calls to **ctermid()** will modify the same object.

DIRECTORY(3)

NAME

opendir, readdir, rewinddir, closedir, dirfd - directory operations

SYNOPSIS

```
#include <sys/types.h>
#include <dirent.h>

DIR *
opendir(const char *filename);

struct dirent *
readdir(DIR *dirp);

void
rewinddir(DIR *dirp);

int
closedir(DIR *dirp);

int
dirfd(DIR *dirp);
```

DESCRIPTION

The **opendir()** function opens the `//HFS:`-style directory named by *filename*, associates a directory stream with it and returns a pointer to be used to identify the directory stream in subsequent operations. The pointer `NULL` is returned if *filename* cannot be accessed, or if it cannot malloc(3) enough memory to hold the directory stream and related information.

The **readdir()** function returns a pointer to the next directory entry. It returns `NULL` upon reaching the end of the directory.

The **rewinddir()** function resets the position of the named directory stream to the beginning of the directory.

The **closedir()** function closes the named directory stream and frees the structure associated with the *dirp* pointer, returning 0 on success. On failure, -1 is returned and the global variable `errno` is set to indicate the error.

The **dirfd()** function returns the integer file descriptor associated with the named directory stream, see open(2).

Sample code which searches a directory for entry "`name`" is:

```
len = strlen(name);
dirp = opendir(".");
while ((dp = readdir(dirp)) != NULL)
        if (dp->d_namlen == len && !strcmp(dp->d_name, name)) {
                (void)closedir(dirp);
                return FOUND;
        }
(void)closedir(dirp);
return NOT_FOUND;
```

SEE ALSO

close(2), lseek(2), open(2), read(2)

DLOPEN(3)

NAME

dlopen, dlsym, dlfunc, dlerror, dlclose – programmatic interface to dynamic linking

SYNOPSIS

```
#include <dlfcn.h>

void *
dlopen(const char *path, int mode);

void *
dlsym(void * restrict handle, const char * restrict symbol);

dlfunc_t
dlfunc(void * restrict handle, const char * restrict symbol);

const char *
dlerror(void);

int
dlclose(void *handle);
```

DESCRIPTION

These functions provide a simple programmatic interface to the services of the Dignus shared libraries. Operations are provided to add new shared objects to a program's address space, to obtain the address bindings of symbols defined by such objects, and to remove such objects when their use is no longer required.

Contact Dignus for information regarding how to construct shared objects using the the **PLINK** utility.

The **dlopen()** function provides access to the shared object in *path*, returning a descriptor that can be used for later references to the object in calls to **dlsym()** and **dlclose()**. If *path* was not in the address space prior to the call to **dlopen()**, it is placed in the address space. If *path* has already been placed in the address space in a previous call to **dlopen()**, it is not added a second time, although a reference count of **dlopen()** operations on *path* is maintained. A `null` pointer supplied for *path* is interpreted as a reference to the main executable of the process. The *mode* argument controls the way in which external function references from the loaded object are bound to their referents. It must contain one of the following values, possibly ORed with additional flags which will be described subsequently:

`RTLD_NOW` — All external function references are bound immediately by **dlopen()**.

`RTLD_NOW` is used to ensure any undefined symbols are discovered during the call to **dlopen()**.

One of the following flags may be ORed into the mode argument:

`RTLD_GLOBAL` — Symbols from this shared object and its directed acyclic graph (DAG) of needed objects will be available for resolving undefined references from all other shared objects.

`RTLD_LOCAL` — Symbols in this shared object and its DAG of needed objects will be available for resolving undefined references only from other objects in the same DAG. This is the default, but it may be specified explicitly with this flag.

If **dlopen()** fails, it returns a `null` pointer, and sets an error condition which may be interrogated with **dlerror()**.

The **dlsym()** function returns the address binding of the symbol described in the `null`-terminated character string *symbol*, as it occurs in the shared object identified by *handle*. The symbols exported by objects added to the address space by **dlopen()** can be accessed only through calls to **dlsym()**. Such symbols do not supersede any definition of those sym bols already present in the address space when the object is loaded, nor are they available to satisfy normal dynamic linking references.

If **dlsym()** is called with the special handle `RTLD_DEFAULT`, the search for the symbol follows the algorithm used for resolving undefined symbols when objects are loaded. The objects searched are as follows, in the given order:

1. The referencing object itself (or the object from which the call to **dlsym()** is made.)
2. All objects loaded at program start-up.
3. All objects loaded via **dlopen()** with the `RTLD_GLOBAL` flag set in the mode argument.
4. All objects loaded via **dlopen()** which are in needed-object DAGs that also contain the referencing object.

If **dlsym()** is called with the special handle `RTLD_NEXT`, then the search for the symbol is limited to the shared objects which were loaded after the one issuing the call to **dlsym()**. Thus, if the function is called from the main program, all

the shared libraries are searched. If it is called from a shared library, all subsequent shared libraries are searched. `RTLD_NEXT` is useful for implementing wrappers around library functions. For example, a wrapper function getpid() could access the "real" **getpid()** with `dlsym(RTLD_NEXT, "getpid")`. (Actually, the **dlfunc()** interface, below, should be used, since **getpid()** is a function and not a data object.)

If **dlsym()** is called with the special handle `RTLD_SELF`, then the search for the symbol is limited to the shared object issuing the call to **dlsym()** and those shared objects which were loaded after it.

The **dlsym()** function returns a null pointer if the symbol cannot be found, and sets an error condition which may be queried with **dlerror()**.

The **dlerror()** function returns a `null`-terminated character string describing the last error that occurred during a call to **dlopen()**, **dladdr()**, **dlinfo()**, **dlsym()**, **dlfunc()**, or **dlclose()**. If no such error has occurred, **dlerror()** returns a `null` pointer. At each call to **dlerror()**, the error indication is reset. Thus in the case of two calls to **dlerror()**, where the second call follows the first immediately, the second call will always return a `null` pointer.

The **dlclose()** function deletes a reference to the shared object refer enced by *handle*. If the reference count drops to 0, the object is removed from the address space, and *handle* is rendered invalid. If **dlclose()** is successful, it returns a value of 0. Otherwise it returns -1, and sets an error condition that can be interrogated with **dlerror()**.

NOTES

Shared objects require special compilation and linking procedures. Contact Dignus for more information.

ERRORS

The **dlopen()**, **dlsym()**, and **dlfunc()** functions return a `null` pointer in the event of errors. The **dlclose()** function returns 0 on success, or -1 if an error occurred. Whenever an error has been detected, a message detailing it can be retrieved via a call to **dlerror()**.

SEE ALSO

PLINK in the Systems/C utilities manual.

ERR(3)

NAME

err, verr, errc, verrc, errx, verrx, warn, vwarn, warnc, vwarnc, warnx, vwarnx, err_set_exit, err_set_file - formatted error messages

SYNOPSIS

```
#include <err.h>

void
err(int eval, const char *fmt, ...);

void
err_set_exit(void (*exitf)(int));

void
err_set_file(void *vfp);

void
errc(int eval, int code, const char *fmt, ...);

void
errx(int eval, const char *fmt, ...);

void
warn(const char *fmt, ...);


void
warnc(int code, const char *fmt, ...);

void
warnx(const char *fmt, ...);

#include <stdarg.h>

void
verr(int eval, const char *fmt, va_list args);

void
verrc(int eval, int code, const char *fmt, va_list args);

void
verrx(int eval, const char *fmt, va_list args);
```

```
void
vwarn(const char *fmt, va_list args);

void
vwarnc(int code, const char *fmt, va_list args);

void
vwarnx(const char *fmt, va_list args);
```

DESCRIPTION

The **err()** and **warn()** family of functions display a formatted error message on the standard error output, or on another file specified using the **err_set_file()** function. In all cases, the last component of the program name, a colon character, and a space are output. If the *fmt* argument is not `NULL`, the printf(3) -like formatted error message is output. The output is terminated by a newline character.

The **err()**, **errc()**, **verr()**, **verrc()**, **warn()**, **warnc()**, **vwarn()**, and **vwarnc()** functions append an error message obtained from strerror(3) based on a code or the global variable `errno`, preceded by another colon and space unless the *fmt* argument is `NULL`.

In the case of the **errc()**, **verrc()**, **warnc()**, and **vwarnc()** functions, the *code* argument is used to look up the error message.

The **err()**, **verr()**, **warn()**, and **vwarn()** functions use the global variable `errno` to look up the error message.

The **errx()** and **warnx()** functions do not append an error message.

The **err()**, **verr()**, **errc()**, **verrc()**, **errx()**, and **verrx()** functions do not return, but exit with the value of the argument *eval*. It is recommended that the standard values defined in sysexits(3) be used for the value of *eval*. The **err_set_exit()** function can be used to specify a function which is called before exit(3) to perform any necessary cleanup; passing a null function pointer for *exitf* resets the hook to do nothing. The **err_set_file()** function sets the output stream used by the other functions. Its *vfp* argument must be either a pointer to an open stream (possibly already converted to void *) or a null pointer (in which case the output stream is set to standard error).

EXAMPLES

Display the current `errno` information string and exit:

```
if ((p = malloc(size)) == NULL)
        err(1, NULL);
if ((fd = open(file_name, O_RDONLY, 0)) == -1)
        err(1, "%s", file_name);
```

Display an error message and exit:

```
if (tm.tm_hour < START_TIME)
        errx(1, "too early, wait until %s", start_time_string);
```

Warn of an error:

```
if ((fd = open(raw_device, O_RDONLY, 0)) == -1)
        warnx("%s: %s: trying the block device",
            raw_device, strerror(errno));
if ((fd = open(block_device, O_RDONLY, 0)) == -1)
        err(1, "%s", block_device);
```

Warn of an error without using the global variable `errno`:

```
error = my_function();  /* returns a value from <errno.h> */
if (error != 0)
        warnc(error, "my_function");
```

SEE ALSO

exit(3), fmtmsg(3), printf(3), strerror(3), sysexits(3)

EXEC(3)

NAME

execl, execlp, execle, execv, execvp - execute a file

SYNOPSIS

```
#include <unistd.h>

extern char **environ;

int
execl(const char *path, const char *arg, ...);

int
execlp(const char *file, const char *arg, ...);

int
execle(const char *path, const char *arg, ...);

int
execv(const char *path, char *const argv[]);

int
execvp(const char *file, char *const argv[]);
```

DESCRIPTION

The **exec** family of functions replaces the current process image with a new process image. The functions described in this manual page are front-ends for the function execve(2). (See the manual page for execve(2) for detailed information about the replacement of the current process.)

The initial argument for these functions is the `//HFS:`-style pathname of a file which is to be executed.

The `const char *arg` and subsequent ellipses in the **execl()**, **execlp()**, and **execle()** functions can be thought of as `arg0`, `arg1`, ..., `argn`. Together they describe a list of one or more pointers to nul-terminated strings that represent the argument list available to the executed program. The first argument, by convention, should point to the file name associated with the file being executed. The list of arguments must be terminated by a `NULL` pointer.

The **execv()**, and **execvp()** functions provide an array of pointers to nul-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the file name associated with the file being executed. The array of pointers **must** be terminated by a `NULL` pointer.

The **execle()** function also specify the environment of the executed process by following the `NULL` pointer that terminates the list of arguments in the argument list or the pointer to the argv array with an additional argument. This additional argument is an array of pointers to nul-terminated strings and **must** be terminated by a `NULL` pointer. The other functions take the environment for the new process image from the external variable `environ` in the current process.

Some of these functions have special semantics.

The functions **execlp()** and **execvp()** will duplicate the actions of the shell in searching for an executable file if the specified file name does not contain a slash "/" character. The search path is the path specified in the environment by "`PATH`" variable. If this variable isn't specified, the default path is set according to the `_PATH_DEFPATH` definition in `<paths.h>`, which is set to "`/usr/bin:/bin`". In addition, certain errors are treated specially.

If an error is ambiguous (for simplicity, we shall consider all errors except `ENOEXEC` as being ambiguous here, although only the critical error `EACCES` is really ambiguous), then these functions will act as if they stat the file to determine whether the file exists and has suitable execute permissions. If it does, they will return immediately with the global variable `errno` restored to the value set by **execve()**. Otherwise, the search will be continued. If the search completes without performing a successful **execve()** or terminating due to an error, these functions will return with the global variable `errno` set to `EACCES` or `ENOENT` according to whether at least one file with suitable execute permissions was found.

If the header of a file isn't recognized (the attempted **execve()** returned `ENOEXEC`), these functions will execute the shell with the path of the file as its first argument. (If this attempt fails, no further searching is done.)

RETURN VALUES

If any of the **exec()** functions returns, an error will have occurred. The return value is -1, and the global variable `errno` will be set to indicate the error.

ERRORS

The **execl()**, **execle()**, **execlp()** and **execvp()** functions may fail and set `errno` for any of the errors specified for the library functions execve(2) and malloc(3).

The **execv()** function may fail and set `errno` for any of the errors specified for the library function execve(2).

SEE ALSO

execve(2)

STANDARDS

The **execl()**, **execv()**, **execle()**, **execlp()** and **execvp()** functions conform to IEEE Std 1003.1-1988 ("POSIX.1").

FMTCHECK(3)

NAME

fmtcheck - sanitizes user-supplied printf(3)-style format string

SYNOPSIS

```
#include <stdio.h>

const char *
fmtcheck(const char *fmt_suspect, const char *fmt_default);
```

DESCRIPTION

The **fmtcheck()** scans *fmt_suspect* and *fmt_default* to determine if *fmt_suspect* will consume the same argument types as *fmt_default* and to ensure that *fmt_suspect* is a valid format string.

The printf(3) family of functions cannot verify the types of arguments that they are passed at run-time. In some cases, it is useful or necessary to use a user-supplied format string with no guarantee that the format string matches the specified arguments.

The **fmtcheck()** function was designed to be used in these cases, as in:

```
printf(fmtcheck(user_format, standard_format), arg1, arg2);
```

In the check, field widths, fillers, precisions, etc. are ignored (unless the field width or precision is an asterisk '*' instead of a digit string). Also, any text other than the format specifiers is completely ignored.

RETURN VALUES

If *fmt_suspect* is a valid format and consumes the same argument types as *fmt_default*, then the **fmtcheck()** will return *fmt_suspect*. Otherwise, it will return *fmt_default*.

SEE ALSO

printf(3)

ISSUES

The **fmtcheck()** function does not understand all of the conversions that printf(3) does.

FMTMSG(3)

NAME

fmtmsg - display a detailed diagnostic message

SYNOPSIS

```
#include <fmtmsg.h>

int
fmtmsg(long classification, const char *label, int severity,
       const char *text, const char *action, const char *tag);
```

DESCRIPTION

The **fmtmsg()** function displays a detailed diagnostic message, based on the supplied arguments, to stderr and/or the system console.

The *classification* argument is the bitwise inclusive OR of zero or one of the manifest constants from each of the classification groups below. The Output classification group is an exception since both `MM_PRINT` and `MM_CONSOLE` may be specified.

Output

`MM_PRINT`	Output should take place on stderr.
`MM_CONSOLE`	Output should take place on the system console.

Source of Condition (Major)

`MM_HARD`	The source of the condition is hardware related.
`MM_SOFT`	The source of the condition is software related.
`MM_FIRM`	The source of the condition is firmware related.

Source of Condition (Minor)

`MM_APPL`	The condition was detected at the application level.
`MM_UTIL`	The condition was detected at the utility level.
`MM_OPSYS`	The condition was detected at the operating system level.

Status

MM_RECOVER	The application can recover from the condition.
MM_NRECOV	The application is unable to recover from the condition.

Alternatively, the MM_NULLMC manifest constant may be used to specify no classification.

The *label* argument indicates the source of the message. It is made up of two fields separated by a colon (':'). The first field can be up to 10 bytes, and the second field can be up to 14 bytes. The MM_NULLLBL manifest constant may be used to specify no label.

The *severity* argument identifies the importance of the condition. One of the following manifest constants should be used for this argument.

MM_HALT	The application has confronted a serious fault and is halting.
MM_ERROR	The application has detected a fault.
MM_WARNING	The application has detected an unusual condition, that could be indicative of a problem.
MM_INFO	The application is providing information about a non-error condition.
MM_NOSEV	No severity level supplied.

The *text* argument details the error condition that caused the message. There is no limit on the size of this character string. The MM_NULLTXT manifest constant may be used to specify no text.

The *action* argument details how the error-recovery process should begin. Upon output, **fmtmsg()** will prefix "TO FIX:" to the beginning of the *action* argument. The MM_NULLACT manifest constant may be used to specify no action.

The *tag* argument should reference online documentation for the message. This usually includes the *label* and a unique identifying number. An example tag is "BSD:ls:168". The MM_NULLTAG manifest constant may be used to specify no tag.

RETURN VALUES

The **fmtmsg()** function returns MM_OK upon success, MM_NOMSG to indicate output to stderr failed, MM_NOCON to indicate output to the system console failed, or MM_NOTOK to indicate output to stderr and the system console failed.

ENVIRONMENT

The `MSGVERB` (message verbosity) environment variable specifies which arguments to **fmtmsg()** will be output to stderr, and in which order. `MSGVERB` should be a colon (':') separated list of identifiers. Valid identifiers include: label, severity, text, action, and tag. If invalid identifiers are specified or incorrectly separated, the default message verbosity and ordering will be used. The default ordering is equivalent to a `MSGVERB` with a value of "label:severity:text:action:tag".

EXAMPLES

The code:

```
fmtmsg(MM_UTIL | MM_PRINT, "BSD:ls", MM_ERROR,
    "illegal option -- z", "refer to manual", "BSD:ls:001");
```

will output:

```
BSD:ls: ERROR: illegal option -- z
TO FIX: refer to manual BSD:ls:001
```

to stderr.

The same code, with `MSGVERB` set to "`text:severity:action:tag`", produces:

```
illegal option -- z: ERROR
TO FIX: refer to manual BSD:ls:001
```

STANDARDS

The **fmtmsg()** function conforms to IEEE Std 1003.1-2001 ("POSIX.1").

ISSUES

Specifying `MM_NULLMC` for the *classification* argument makes little sense, since without an output specified, **fmtmsg()** is unable to do anything useful.

In order for **fmtmsg()** to output to the system console, the effective user must have appropriate permission to write to `/dev/console`. This means that on most systems **fmtmsg()** will return `MM_NOCON` unless the effective user is root, or has other appropriate permissions.

FNMATCH(3)

NAME

fnmatch - match filename or pathname

SYNOPSIS

```
#include <fnmatch.h>

int
fnmatch(const char *pattern, const char *string, int flags);
```

DESCRIPTION

The **fnmatch()** function matches patterns according to the rules used by the shell. It checks the string specified by the *string* argument to see if it matches the pattern specified by the *pattern* argument.

The *flags* argument modifies the interpretation of *pattern* and *string*. The value of *flags* is the bitwise inclusive OR of any of the following constants, which are defined in the include file `<fnmatch.h>`.

`FNM_NOESCAPE` Normally, every occurrence of a backslash (‘) followed by a character in *pattern* is replaced by that character. This is done to negate any special meaning for the character. If the `FNM_NOESCAPE` flag is set, a backslash character is treated as an ordinary character.

`FNM_PATHNAME` Slash characters in *string* must be explicitly matched by slashes in *pattern*. If this flag is not set, then slashes are treated as regular characters.

`FNM_PERIOD` Leading periods in *string* must be explicitly matched by periods in *pattern*. If this flag is not set, then leading periods are treated as regular characters. The definition of “leading” is related to the specification of `FNM_PATHNAME`. A period is always “leading” if it is the first character in string. Additionally, if `FNM_PATHNAME` is set, a period is leading if it immediately follows a slash.

`FNM_LEADING_DIR` Ignore “`/*`” rest after successful *pattern* matching.

`FNM_CASEFOLD` Ignore case distinctions in both the *pattern* and the *string*.

RETURN VALUES

The **fnmatch()** function returns zero if string matches the pattern specified by pattern, otherwise, it returns the value `FNM_NOMATCH`.

SEE ALSO

glob(3), regex(3)

STANDARDS

The **fnmatch()** function conforms to IEEE Std 1003.2 ("POSIX.2").

ISSUES

The pattern '*' matches the empty string, even if `FNM_PATHNAME` is specified.

FTOK(3)

NAME

ftok - create IPC identifier from `//HFS:`-style path name

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t
ftok(const char *path, int id);
```

DESCRIPTION

The **ftok()** function attempts to create a unique key suitable for use with the msgget(3), semget(2) and shmget(2) functions given the `//HFS:`-style *path* of an existing file and a user-selectable *id*.

The specified *path* must specify an existing `HFS` file that is accessible to the calling process or the call will fail. Also, note that links to files will return the same key, given the same *id*.

RETURN VALUES

The **ftok()** function will return -1 if *path* does not exist, is not an `HFS` file, or if it cannot be accessed by the calling process.

ISSUES

The returned key is computed based on the device minor number and inode of the specified path in combination with the lower 8 bits of the given *id*. Thus it is quite possible for the routine to return duplicate keys.

GETCWD(3)

NAME

getcwd, getwd – get working directory pathname

SYNOPSIS

```
#include <unistd.h>

char *
getcwd(char *buf, size_t size);

char *
getwd(char *buf);
```

DESCRIPTION

The **getcwd()** function copies the absolute pathname of the current working directory into the memory referenced by *buf* and returns a pointer to *buf*. The *size* argument is the size, in bytes, of the array referenced by *buf*.

If *buf* is `NULL`, space is allocated as necessary to store the pathname. This space may later be free(3)'d.

The function **getwd()** is a compatibility routine which calls **getcwd()** with its *buf* argument and a size of `MAXPATHLEN` (as defined in the include file `<sys/param.h>`). Obviously, *buf* should be at least `MAXPATHLEN` bytes in length.

RETURN VALUES

Upon successful completion, a pointer to the pathname is returned. Otherwise a `NULL` pointer is returned and the global variable `errno` is set to indicate the error. In addition, *getwd()* copies the error message associated with errno into the memory referenced by *buf*.

ERRORS

The *getcwd()* function will fail if:

`[EACCES]` Read or search permission was denied for a component of the pathname.

[EINVAL] The size argument is zero.

[ENOENT] A component of the pathname no longer exists.

[ENOMEM] Insufficient memory is available.

[ERANGE] The size argument is greater than zero but smaller than the length of the pathname plus 1.

GETCONTEXT(3)

NAME

getcontext, setcontext – get and set user thread context

SYNOPSIS

```
#include <ucontext.h>

int
getcontext(ucontext_t *ucp);

int
setcontext(const ucontext_t *ucp);
```

DESCRIPTION

The **getcontext()** function saves the current thread's execution context in the structure pointed to by *ucp*. This saved context may then later be restored by calling **setcontext()**.

The **setcontext()** function makes a previously saved thread context the current thread context, i.e., the current context is lost and **setcontext()** does not return. Instead, execution continues in the context specified by *ucp*, which must have been previously initialized by a call to **getcontext()**, makecontext(3), or by being passed as an argument to a signal handler (see sigaction(2)).

If *ucp* was initialized by **getcontext()**, then execution continues as if the original **getcontext()** call had just returned (again).

If *ucp* was initialized by makecontext(3), execution continues with the invocation of the function specified to makecontext(3). When that function returns, *ucp*`->uc_link` determines what happens next: if *ucp*`->uc_link` is `NULL`, the process exits; otherwise, `setcontext(ucp->uc_link)` is implicitly invoked.

If *ucp* was initialized by the invocation of a signal handler, execution continues at the point the thread was interrupted by the signal.

RETURN VALUES

If successful, **getcontext()** returns zero and **setcontext()** does not return; otherwise -1 is returned.

ERRORS

No errors are defined for **getcontext()** or **setcontext()**.

IMPLEMENTATION NOTES

The **getcontext()** and **setcontext()** functions take advantage of the `EXTRACT PSW` (`EPSW`) and `RESUME PROGRAM` (`RP`) instructions. These are available in all z/Architecture and most ESA/390 environments. These functions will not operate in environments that don't provide those instructions.

SEE ALSO

sigaction(2), sigaltstack(2), makecontext(3), ucontext(3)

GETGRENT(3)

getgrent, getgrnam, getgrgid, setgroupent, setgrent, endgrent - group database operations

SYNOPSIS

```
#include <sys/types.h>
#include <grp.h>

struct group *
getgrent(void);

struct group *
getgrnam(const char *name);

struct group *
getgrgid(gid_t gid);

int
setgroupent(int stayopen);

int
setgrent(void);

void
endgrent(void);
```

DESCRIPTION

These functions operate on the group database. Each entry of the database is mapped to the structure `group` found in the include file `<grp.h>`:

```
struct group {
        char    *gr_name;       /* group name */
        char    *gr_passwd;     /* group password */
        int     gr_gid;         /* group id */
        char    **gr_mem;       /* group members */
};
```

The functions **getgrnam()** and **getgrgid()** search the group database for the given group name pointed to by *name* or the group id specifeid by *gid*, respectively, returning the first one encountered. Identical group names or group gids may result in undefined behavior.

The **getgrent()** function sequentially reads the group database and is intended for programs that wish to step through the complete list of groups.

The **setgroupent()** function opens the database, or rewinds it if it is already open. It is provided for compatibility with popular UNIX systems.

The **setgrent()** function resets the data base to the beginning so that subsequent calls to **getgrent()** start from the beginning.

The **endgrent()** function resets the data base to the beginning.

RETURN VALUES

The functions **getgrent()**, **getgrnam()**, and **getgrgid()**, return a pointer to the group entry if successful; if end-of-file is reached or an error occurs a `NULL` pointer is returned. The functions **setgroupent()** and **setgrent()** return the value 1 if successful, otherwise the value 0 is returned. The function **endgrent()** has no return value.

SEE ALSO

getpwent(3)

NOTES

The functions **getgrent()**, **getgrnam()** and **getgrgid()** leave their results in an internal static object and return a pointer to that object. Subsequent calls to the same function will modify the same object.

GETPROGNAME(3)

NAME

getprogname, setprogname - get or set the program name

SYNOPSIS

```
#include <stdlib.h>

const char *
getprogname(void);

void
setprogname(const char *progname);
```

DESCRIPTION

The **getprogname()** and **setprogname()** functions manipulate the name of the current program. They are used by error-reporting routines to produce consistent output.

The **getprogname()** function returns the name of the program. If the name has not been set yet, it will return `NULL`.

The **setprogname()** function sets the name of the program to be the last component of the *progname* argument. Since a pointer to the given string is kept as the program name, it should not be modified for the rest of the program's lifetime.

At program start-up, the Systems/C runtime attempts to determine, from the operating system, the name of the program. If the name can be determined, the name of the program is set by the start-up code that is run before **main()**; thus, running **setprogname()** is not always necessary. Programs that desire maximum portability should still call it. On some operating systems, these functions may be implemented in a portability library. Calling **setprogname()** allows the aforementioned systems to learn the program name without modifications to the start-up code.

GETPWENT(3)

NAME

getpwent, getpwnam, getpwuid, setpassent, setpwent, endpwent - password database operations

SYNOPSIS

```
#include <sys/types.h>
#include <pwd.h>

struct passwd *
getpwent(void);

struct passwd *
getpwnam(const char *login);

struct passwd *
getpwuid(uid_t uid);

int
setpassent(int stayopen);

void
setpwent(void);

void
endpwent(void);
```

DESCRIPTION

These functions operate on the OpenEdition password database. Each entry in the password database is mapped to the structure `passwd` found in the include file `<pwd.h>`:

```
struct passwd {
        char    *pw_name;       /* user name */
        char    *pw_passwd;     /* encrypted password */
        uid_t   pw_uid;         /* user uid */
        gid_t   pw_gid;         /* user gid */
        time_t  pw_change;      /* password change time */
        char    *pw_class;      /* user access class */
```

```
        char    *pw_gecos;      /* Honeywell login info */
        char    *pw_dir;        /* home directory */
        char    *pw_shell;      /* default shell */
        time_t  pw_expire;      /* account expiration */
        int     pw_fields;      /* internal: fields filled in */
};
```

The functions **getpwnam()** and **getpwuid()** search the password database for the given login name or user uid, respectively, always returning the first one encountered.

The **getpwent()** function sequentially reads the password database and is intended for programs that wish to process the complete list of users.

The **setpassent()** function is provided for compatibility with popular UNIX platforms. It causes **getpwent()** to "rewind" to the beginning of the database.

The **setpwent()** function resets the database so that the next call to **getpwent()** starts over at the beginning.

The **endpwent()** function is used to indicate the end of database access. It also resets the database so that the next call to **getpwent()** starts over at the beginning.

Because of how passwords are managed on OS/390 and z/OS, the password field of the returned structure will always point to the string `"*"`.

RETURN VALUES

The functions **getpwent()**, **getpwnam()**, and **getpwuid()**, return a valid pointer to a passwd structure on success and a `NULL` pointer if the end of the database is reached or an error occurs. The **setpassent()** function returns 0 on failure and 1 on success. The **endpwent()** and **setpwent()** functions have no return value.

SEE ALSO

getlogin(2), getgrent(3)

ISSUES

The functions **getpwent()**, **getpwnam()**, and **getpwuid()**, leave their results in an internal static object and return a pointer to that object. Subsequent calls to the same function will modify the same object.

GLOB(3)

NAME

glob, globfree - generate //HFS: pathnames matching a pattern

SYNOPSIS

```
#include <glob.h>

int
glob(const char *pattern, int flags, int (*errfunc)(const char *, int),
     glob_t *pglob);

void
globfree(glob_t *pglob);
```

DESCRIPTION

The **glob()** function is a pathname generator that implements the rules for file name pattern matching used by the shell.

The include file <glob.h> defines the structure type glob_t, which contains at least the following fields:

```
typedef struct {
        int gl_pathc;           /* count of total paths so far */
        int gl_matchc;          /* count of paths matching pattern */
        int gl_offs;            /* reserved at beginning of gl_pathv */
        int gl_flags;           /* returned flags */
        char **gl_pathv;        /* list of paths matching pattern */
} glob_t;
```

The argument *pattern* is a pointer to an //HFS:-style pathname pattern to be expanded. The **glob()** argument matches all accessible pathnames against the pattern and creates a list of the pathnames that match. In order to have access to a pathname, **glob()** requires search permission on every component of a path except the last and read permission on each directory of any filename component of pattern that contains any of the special characters '*', '?' or '['.

The **glob()** argument stores the number of matched pathnames into the *gl_pathc* field, and a pointer to a list of pointers to pathnames into the *gl_pathv* field. The first pointer after the last pathname is NULL. If the pattern does not match any pathnames, the returned number of matched paths is set to zero.

It is the caller's responsibility to create the structure pointed to by *pglob*. The **glob()** function allocates other space as needed, including the memory pointed to by *gl_pathv*.

The argument *flags* s used to modify the behavior of **glob()**. The value of *flags* is the bitwise inclusive OR of any of the following values defined in `<glob.h>`:

`GLOB_APPEND` Append pathnames generated to the ones from a previous call (or calls) to **glob()**. The value of *gl_pathc* will be the total matches found by this call and the previous call(s). The pathnames are appended to, not merged with the pathnames returned by the previous call(s). Between calls, the caller must not change the setting of the `GLOB_DOOFFS` flag, nor change the value of *gl_offs* when `GLOB_DOOFFS` is set, nor (obviously) call **globfree()** for *pglob*.

`GLOB_DOOFFS` Make use of the `gl_offs` field. If this flag is set, `gl_offs` is used to specify how many `NULL` pointers to prepend to the beginning of the *gl_pathv* field. In other words, *gl_pathv* will point to `gl_offs` `NULL` pointers, followed by *gl_pathc* pathname pointers, followed by a `NULL` pointer.

`GLOB_ERR` Causes **glob()** to return when it encounters a directory that it cannot open or read. Ordinarily, **glob()** continues to find matches.

`GLOB_MARK` Each pathname that is a directory that matches *pattern* has a slash appended.

`GLOB_NOCHECK` If *pattern* does not match any pathname, then **glob()** returns a list consisting of only *pattern*, with the number of total pathnames set to 1, and the number of matched pathnames set to 0. The effect of backslash escaping is present in the pattern returned.

`GLOB_NOESCAPE` By default, a backslash (‘) character is used to escape the following character in the pattern, avoiding any special interpretation of the character. If `GLOB_NOESCAPE` is set, backslash escaping is disabled.

`GLOB_NOSORT` By default, the pathnames are sorted in ascending order; this flag prevents that sorting (speeding up **glob()**).

The following values may also be included in *flags*, however, they are non-standard extensions to IEEE Std 103.2 (“POSIX.2”).

`GLOB_ALTDIRFUNC` The following additional fields in the `pglob` structure have been initialized with alternate functions for **glob()** to use to open, read, and close directories and to get stat information on names found in those directories.

```
void *(*gl_opendir)(const char * name);
struct dirent *(*gl_readdir)(void *);
void (*gl_closedir)(void *);
int (*gl_lstat)(const char *name, struct stat *st);
int (*gl_stat)(const char *name, struct stat *st);
```

GLOB_BRACE Pre-process the pattern string to expand 'pat,pat,...' strings like csh(1). The pattern '' is left unexpanded for historical reasons (and csh(1) does the same thing to ease typing of find(1) patterns).

GLOB_MAGCHAR Set by the **glob()** function if the pattern included globbing characters. See the description of the usage of the `gl_matchc` structure member for more details.

GLOB_NOMAGIC Is the same as `GLOB_NOCHECK` but it only appends the pattern if it does not contain any of the special characters "*", "?" or "[". `GLOB_NOMAGIC` is provided to simplify implementing the historic csh(1) globbing behavior and should probably not be used anywhere else.

GLOB_TILDE Expand patterns that start with '~' to user name home directories.

GLOB_LIMIT Limit the total number of returned pathnames to the value provided in *gl_matchc* (default `ARG_MAX`). This option should be set for programs that can be coerced into a denial of service attack via patterns that expand to a very large number of matches, such as a long string of '*/../*/..'.

If, during the search, a directory is encountered that cannot be opened or read and *errfunc* is non-`NULL`, **glob()** calls `(*errfunc)(path, errno)`. This may be unintuitive: a pattern like '*/Makefile' will try to stat(2) 'foo/Makefile' even if 'foo' is not a directory, resulting in a call to *errfunc*. The error routine can suppress this action by testing for `ENOENT` and `ENOTDIR`; however, the `GLOB_ERR` flag will still cause an immediate return when this happens.

If *errfunc* returns non-zero, **glob()** stops the scan and returns `GLOB_ABORTED` after setting *gl_pathc* and *gl_pathv* to reflect any paths already matched. This also happens if an error is encountered and `GLOB_ERR` is set in *flags*, regardless of the return value of *errfunc*, if called. If `GLOB_ERR` is not set and either *errfunc* is `NULL` or *errfunc* returns zero, the error is ignored.

The **globfree()** function frees any space associated with *pglob* from a previous call(s) to **glob()**.

RETURN VALUES

On successful completion, **glob()** returns zero. In addition the fields of *pglob* contain the values described below:

`gl_patch` contains the total number of matched pathnames so far. This includes other matches from previous invocations of **glob()** if `GLOB_APPEND` was specified.

`gl_matchc` contains the number of matched pathnames in the current invocation of **glob()**.

`gl_flags` contains a copy of the *flags* argument with the bit `GLOB_MAGCHAR` set if *pattern* contained any of the special characters "*", "?" or "[", cleared if not.

`gl_pathv` contains a pointer to a `NULL`-terminated list of matched pathnames. However, if `gl_pathc` is zero, the contents of `gl_pathv` are undefined.

If **glob()** terminates due to an error, it sets the global variable `errno` and returns one of the following non-zero constants, which are defined in the include file `<glob.h>`:

`GLOB_NOSPACE` An attempt to allocate memory failed, or if `errno` was 0 `GLOB_LIMIT` was specified in the flags and *`pglob`*`->gl_matchc` or more patterns were matched.

`GLOB_ABORTED` The scan was stopped because an error was encountered and either `GLOB_ERR` was set or `(*`*`errfunc`*`)()` returned non-zero.

`GLOB_NOMATCH` The pattern did not match a pathname and `GLOB_NOCHECK` was not set.

The arguments *`pglob`*`->gl_pathc` and *`pglob`*`->gl_pathv` are still set as specified above.

EXAMPLES

A rough equivalent of 'ls -l *.c *.h' can be obtained with the following code:

```
glob_t g;

g.gl_offs = 2;
glob("*.c", GLOB_DOOFFS, NULL, &g);
glob("*.h", GLOB_DOOFFS | GLOB_APPEND, NULL, &g);
g.gl_pathv[0] = "ls";
g.gl_pathv[1] = "-l";
execvp("ls", g.gl_pathv);
```

SEE ALSO

fnname(3), regexp(3)

STANDARDS

The **glob()** function is expected to be IEEE Std 1003.2 ("POSIX.2") compatible with the exception that the flags `GLOB_ALTDIRFUNC`, `GLOB_BRACE`, `GLOB_LIMIT`, `GLOB_MAGCHAR`, `GLOB_NOMAGIC`, and `GLOB_TILDE`, and the fields `gl_matchc` and `gl_flags` should not be used by applications striving for strict POSIX conformance.

ISSUES

Patterns longer than `MAXPATHLEN` may cause unchecked errors.

The **glob()** argument may fail and set `errno` for any of the errors specified for the library routines stat(2), closedir(3), opendir(3), readdir(3), malloc(3), and free(3).

HCREATE(3)

NAME

hcreate, hdestroy, hsearch – manage hash search table

SYNOPSIS

```
#include <search.h>

int
hcreate(size_t nel);

void
hdestroy(void);

ENTRY *
hsearch(ENTRY item, ACTION action);
```

DESCRIPTION

The **hcreate()**, **hdestroy()**, and **hsearch()** functions manage hash search tables.

The **hcreate()** function allocates sufficient space for the table, and the application should ensure it is called before **hsearch()** is used. The *nel* argument is an estimate of the maximum number of entries that the table should contain. This number may be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances.

The **hdestroy()** function disposes of the search table, and may be followed by another call to **hcreate()**. After the call to **hdestroy()**, the data can no longer be considered accessible. The **hdestroy()** function calls free(3) for each comparison key in the search table but not the data item associated with the key.

The **hsearch()** function is a hash-table search routine. It returns a pointer into a hash table indicating the location at which an entry can be found. The *item* argument is a structure of type `ENTRY` (defined in the `<search.h>` header) containing two pointers: `item.key` points to the comparison key (a `char *`), and `item.data` (a `void *`) points to any other data to be associated with that key. The comparison function used by **hsearch()** is strcmp(3). The *action* argument is a member of an enumeration type `ACTION` indicating the disposition of the entry if it cannot be found in the table. `ENTER` indicates that the item should be inserted in the table at an appropriate point. `FIND` indicates that no entry should be made. Unsuccessful resolution is indicated by the return of a `NULL` pointer.

The comparison key (passed to **hsearch()** as `item.key`) must be allocated using malloc(3) if *action* is `ENTER` and **hdestroy()** is called.

RETURN VALUES

The **hcreate()** function returns 0 if it cannot allocate sufficient space for the table; otherwise, it returns non-zero.

The **hdestroy()** function does not return a value.

The **hsearch()** function returns a `NULL` pointer if either the *action* is `FIND` and the item could not be found or the *action* is `ENTER` and the table is full.

ERRORS

The **hcreate()** and **hsearch()** functions may fail if:

`[ENOMEM]` Insufficient storage space is available.

EXAMPLES

The following example reads in strings followed by two numbers and stores them in a hash table, discarding duplicates. It then reads in strings and finds the matching entry in the hash table and prints it out.

```
#include <stdio.h>
#include <search.h>
#include <string.h>
#include <stdlib.h>

struct info {             /* This is the info stored in the table */
  int age, room;          /* other than the key. */
};

#define NUM_EMPL        5000    /* # of elements in search table. */

int
main(void)
{
  char str[BUFSIZ]; /* Space to read string */
  struct info info_space[NUM_EMPL]; /* Space to store employee info. */
  struct info *info_ptr = info_space; /* Next space in info_space. */
  ENTRY item;
```

```
  ENTRY *found_item; /* Name to look for in table. */
  char name_to_find[30];
  int i = 0;

  /* Create table; no error checking is performed. */
  (void) hcreate(NUM_EMPL);

  while (scanf("%s%d%d", str, &info_ptr->age,
               &info_ptr->room) != EOF && i++ < NUM_EMPL) {
    /* Put information in structure, and structure in item. */
    item.key = strdup(str);
    item.data = info_ptr;
    info_ptr++;
    /* Put item into table. */
    (void) hsearch(item, ENTER);
  }

  /* Access table. */
  item.key = name_to_find;
  while (scanf("%s", item.key) != EOF) {
    if ((found_item = hsearch(item, FIND)) != NULL) {
      /* If item is in the table. */
      (void)printf("found %s, age = %d, room = %d\n",
                   found_item->key,
                   ((struct info *)found_item->data)->age,
                   ((struct info *)found_item->data)->room);
    } else
      (void)printf("no such employee %s\n", name_to_find);
  }
  hdestroy();
  return 0;
}
```

SEE ALSO

bsearch(3), lsearch(3), malloc(3), strcmp(3), tsearch(3)

STANDARDS

The **hcreate()**, **hdestroy()**, and **hsearch()** functions conform to X/Open Portability Guide Issue 4.2 ("XPG4.2").

ISSUES

The interface permits the use of only one hash table at a time.

ISATTY(3)

NAME

isatty - determine if a file descriptor is associated with a terminal

SYNOPSIS

```
#include <unistd.h>

int
isatty(int fd);
```

DESCRIPTION

The **isatty()** function determines if the file descriptor *fd* refers to a valid terminal type device.

If the file descriptor is associated with a `DD` that is allocated to a terminal, or if the file descriptor is associated with an OpenEdition I/O descriptor that represents a `tty`, **isatty()** returns a non-zero value ("true").

RETURN VALUES

isatty() returns 0 if the descriptor *fd* is not associated with a terminal, non-zero otherwise.

LSEARCH(3)

NAME

lsearch, lfind - linear searching routines

SYNOPSIS

```
#include <sys/types.h>

char *
lsearch(const void *key, const void *base, size_t *nelp, size_t width,
        int (*compar)(void *, void *));

char *
lfind(const void *key, const void *base, size_t *nelp, size_t width,
        int (*compar)(void *, void *));
```

DESCRIPTION

This interface was obsolete before it was written.

The functions **lsearch()**, and **lfind()** provide basic linear searching functionality.

Base is the pointer to the beginning of an array. The argument *nelp* is the current number of elements in the array, where each element is *width* bytes long. The *compar* function is a comparison routine which is used to compare two elements. It takes two arguments which point to the *key* object and to an array member, in that order, and must return an integer less than, equivalent to, or greater than zero if the *key* object is considered, respectively, to be less than, equal to, or greater than the array member.

The **lsearch()** and **lfind()** functions return a pointer into the array referenced by *base* where *key* is located. If *key* does not exist, **lfind()** will return a `NULL` pointer and **lsearch()** will add it to the array. When an element is added to the array by **lsearch()** the location referenced by the argument *nelp* is incremented by one.

SEE ALSO

bsearch(3)

MAKECONTEXT(3)

NAME

makecontext, swapcontext – modify and exchange user thread contexts

SYNOPSIS

```
#include <ucontext.h>

void
makecontext(ucontext_t *ucp, void (*func)(void), int argc, ...);

int
swapcontext(ucontext_t *oucp, const ucontext_t *ucp);
```

DESCRIPTION

The **makecontext()** function modifies the user thread context pointed to by *ucp*, which must have previously been initialized by a call to getcontext(3) and had a stack allocated for it. The context is modified so that it will continue execution by invoking *func()* with the arguments provided. The *argc* argument must be equal to the number of additional arguments provided to **makecontext()** and also equal to the number of arguments to *func()*, or else the behavior is undefined.

The *ucp*`->uc_link` argument must be initialized before calling **makecontext()** and determines the action to take when *func()* returns: if equal to `NULL`, the process exits; otherwise, `setcontext(ucp->uc_link)` is implicitly invoked.

The **swapcontext()** function saves the current thread context in **oucp* and makes **ucp* the currently active context.

RETURN VALUES

If successful, **swapcontext()** returns zero; otherwise -1 is returned and the global variable `errno` is set appropriately.

ERRORS

The **swapcontext()** function will fail if:

`[ENOMEM]` There is not enough stack space in *ucp* to complete the operation.

SEE ALSO

setcontext(3), ucontext(3)

NICE(3)

NAME

nice - set program scheduling priority

SYNOPSIS

```
#include <unistd.h>

int
nice(int incr);
```

DESCRIPTION

This interface is obsoleted by setpriority(2).

The **nice()** function obtains the scheduling priority of the process from the system and sets it to the priority value specified in *incr*. The priority is a value in the range -20 to 19. The default priority is 0; lower priorities cause more favorable scheduling. Only the super-user may lower priorities.

Children inherit the priority of their parent processes via fork(2).

SEE ALSO

fork(2), setpriority(2)

POPEN(3)

NAME

popen, pclose - process I/O

SYNOPSIS

```
#include <stdio.h>

FILE *
popen(const char *command, const char *type);

int
pclose(FILE *stream);
```

DESCRIPTION

The **popen()** function "opens" a process by creating a pipe, forking, and invoking the shell. Since a pipe is by definition unidirectional, the type argument may specify only reading or writing, not both; the resulting stream is correspondingly read-only or write-only.

The *command* argument is a pointer to a null-terminated string containing a shell command line. This command is passed to `/bin/sh` using the `-c` flag; interpretation, if any, is performed by the shell. The *mode* argument is a pointer to a null-terminated string which must be either `"r"` for reading or `"w"` for writing.

The return value from **popen()** is a normal standard I/O stream in all respects save that it must be closed with **pclose()** rather than fclose(3). Writing to such a stream writes to the standard input of the command; the command's standard output is the same as that of the process that called **popen()**, unless this is altered by the command itself. Conversely, reading from a "popened" stream reads the command's standard output, and the command's standard input is the same as that of the process that called **popen()**.

Note that output **popen()** streams are fully buffered by default.

The **pclose()** function waits for the associated process to terminate and returns the exit status of the command as returned by waitpid(2).

RETURN VALUE

The **popen()** function returns `NULL` if the fork(2) or pipe(2) calls fail, or if it cannot allocate memory.

The **pclose()** function returns -1 if *stream* is not associated with a "popened" command, if *stream* is already "pclosed", or if waitpid(2) returns an error.

ERRORS

The **popen()** function does not reliably set `errno`.

SEE ALSO

fork(2), pipe(2), fflush(3), fclose(3), fopen(3), stdio(3), system(3)

ISSUES

Since the standard input of a command opened for reading shares its seek offset with the process that called **popen()**, if the original process has done a buffered read, the command's input position may not be as expected. Similarly, the output from a command opened for writing may become intermingled with that of the original process. The latter can be avoided by calling fflush(3) before **popen()**.

Failure to execute the shell is indistinguishable from the shell's failure to execute command, or an immediate exit of the command. The only hint is an exit status of 127.

The **popen()** argument always calls `sh`.

POSIX_SPAWN(3)

NAME

posix_spawn, posix_spawnp - spawn a process

SYNOPSIS

```
#include <spawn.h>

int
posix_spawn(pid_t *restrict pid, const char *restrict path,
         const posix_spawn_file_actions_t *file_actions,
         const posix_spawnattr_t *restrict attrp, char *const argv[restrict],
         char *const envp[restrict]);

int
posix_spawnp(pid_t *restrict pid, const char *restrict file,
         const posix_spawn_file_actions_t *file_actions,
         const posix_spawnattr_t *restrict attrp, char *const argv[restrict],
         char *const envp[restrict]);
```

DESCRIPTION

The **posix_spawn()** and **posix_spawnp()** functions create a new process (child process) from the specified process image. The new process image is constructed from a regular executable file called the new process image file.

When a C program is executed as the result of this call, it is entered as a C-language function call as follows:

```
int main(int argc, char *argv[]);
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. In addition, the variable:

```
extern char **environ;
```

points to an array of character pointers to the environment strings.

The argument *argv* is an array of character pointers to null-terminated strings. The last member of this array is a null pointer and is not counted in *argc*. These

strings constitute the argument list available to the new process image. The value in `argv[0]` should point to a filename that is associated with the process image being started by the **posix_spawn()** or **posix_spawnp()** function.

The argument *envp* is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process image. The environment array is terminated by a null pointer.

The *path* argument to **posix_spawn()** is a pathname that identifies the new process image file to execute.

The *file* parameter to **posix_spawnp()** is used to construct a pathname that identifies the new process image file. If the *file* parameter contains a slash character, the file parameter is used as the pathname for the new process image file. Otherwise, the path prefix for this file is obtained by a search of the directories passed as the environment variable "`PATH`". If this variable is not specified, the default path is set according to the `_PATH_DEFPATH` definition in ¡paths.h¿, which is set to

"`/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/sbin:/usr/local/bin`".

If *file_actions* is a null pointer, then file descriptors open in the calling process remain open in the child process, except for those whose close-on-exec flag `FD_CLOEXEC` is set (see **fcntl()**). For those file descriptors that remain open, all attributes of the corresponding open file descriptions, including file locks (see **fcntl()**), remain unchanged.

If *file_actions* is not `NULL`, then the file descriptors open in the child process are those open in the calling process as modified by the spawn file actions object pointed to by *file_actions* and the `FD_CLOEXEC` flag of each remaining open file descriptor after the spawn file actions have been processed. The effective order of processing the spawn file actions are:

1. The set of open file descriptors for the child process initially are the same set as is open for the calling process. All attributes of the corresponding open file descriptions, including file locks (see **fcntl()**), remain unchanged.

2. The signal mask, signal default actions, and the effective user and group IDs for the child process are changed as specified in the attributes object referenced by *attrp*.

3. The file actions specified by the spawn file actions object are performed in the order in which they were added to the spawn file actions object.

4. Any file descriptor that has its `FD_CLOEXEC` flag set (see **fcntl()**) is closed.

All non-posix file descriptors are closed and unavailable to the child process.

The `posix_spawnattr_t` spawn attributes object type is defined in `<spawn.h>`. It contains the attributes defined below.

If the `POSIX_SPAWN_SETPGROUP` flag is set in the spawn-flags attribute of the object referenced by *attrp*, and the spawn-pgroup attribute of the same object is non-zero, then the child's process group is as specified in the spawn-pgroup attribute of the object referenced by *attrp*.

As a special case, if the `POSIX_SPAWN_SETPGROUP` flag is set in the spawn-flags attribute of the object referenced by *attrp*, and the spawn-pgroup attribute of the same object is set to zero, then the child is in a new process group with a process group ID equal to its process ID.

If the `POSIX_SPAWN_SETPGROUP` flag is not set in the spawn-flags attribute of the object referenced by *attrp*, the new child process inherits the parent's process group.

The `POSIX_SPAWN_RESETIDS` flag in the spawn-flags attribute of the object referenced by *attrp* governs the effective user ID of the child process. If this flag is not set, the child process inherits the parent process' effective user ID. If this flag is set, the child process' effective user ID is reset to the parent's real user ID. In either case, if the set-user-ID mode bit of the new process image file is set, the effective user ID of the child process becomes that file's owner ID before the new process image begins execution.

The `POSIX_SPAWN_RESETIDS` flag in the spawn-flags attribute of the object referenced by *attrp* also governs the effective group ID of the child process. If this flag is not set, the child process inherits the parent process' effective group ID. If this flag is set, the child process' effective group ID is reset to the parent's real group ID. In either case, if the set-group-ID mode bit of the new process image file is set, the effective group ID of the child process becomes that file's group ID before the new process image begins execution.

If the `POSIX_SPAWN_SETSIGMASK` flag is set in the spawn-flags attribute of the object referenced by *attrp*, the child process initially has the signal mask specified in the spawn-sigmask attribute of the object referenced by *attrp*.

If the `POSIX_SPAWN_SETSIGDEF` flag is set in the spawn-flags attribute of the object referenced by *attrp*, the signals specified in the spawn-sigdefault attribute of the same object is set to their default actions in the child process. Signals set to the default action in the parent process is set to the default action in the child process.

Signals set to be caught by the calling process is set to the default action in the child process.

Signals set to be ignored by the calling process image is set to be ignored by the child process, unless otherwise specified by the `POSIX_SPAWN_SETSIGDEF` flag being set in the spawn-flags attribute of the object referenced by *attrp* and the signals being indicated in the spawn-sigdefault attribute of the object referenced by attrp.

If the value of the *attrp* pointer is `NULL`, then the default values are used.

All process attributes, other than those influenced by the attributes set in the object referenced by *attrp* as specified above or by the file descriptor manipulations specified in *file_actions*, appear in the new process image as though **vfork()** had been called to create a child process and then **execve()** had been called by the child process to execute the new process image.

The implementation uses **vfork()**, thus the fork handlers are not run when **posix_spawn()** or **posix_spawnp()** is called.

RETURN VALUES

Upon successful completion, **posix_spawn()** and **posix_spawnp()** return the process ID of the child process to the parent process, in the variable pointed to by a non-NULL *pid* argument, and return zero as the function return value. Otherwise, no child process is created, no value is stored into the variable pointed to by *pid*, and an error number is returned as the function return value to indicate the error. If the *pid* argument is a null pointer, the process ID of the child is not returned to the caller.

ERRORS

1. If **posix_spawn()** and **posix_spawnp()** fail for any of the reasons that would cause **vfork()** or one of the **exec** to fail, an error value is returned as described by **vfork()** and **exec**, respectively (or, if the error occurs after the calling process successfully returns, the child process exits with exit status 127).

2. If `POSIX_SPAWN_SETPGROUP` is set in the spawn-flags attribute of the object referenced by *attrp*, and **posix_spawn()** or **posix_spawnp()** fails while changing the child's process group, an error value is returned as described by **setpgid()** (or, if the error occurs after the calling process successfully returns, the child process exits with exit status 127).

3. If the *file_actions* argument is not `NULL`, and specifies any dup2 or open actions to be performed, and if *posix_spawn()* or *posix_spawnp()* fails for any of the reasons that would cause **dup2()** or **open()** to fail, an error value is returned as described by **dup2()** and **open()**, respectively (or, if the error occurs after the calling process successfully returns, the child process exits with exit status 127).

An open file action may, by itself, result in any of the errors described by **dup2()**, in addition to those described by **open()**. This implementation ignores any errors from **close()**, including trying to close a descriptor that is not open.

SEE ALSO

close(2), dup2(2), execve(2), fcntl(2), open(2), setpgid(2), vfork(2), posix_spawn_file_actions_addclose(3), posix_spawn_file_actions_adddup2(3), posix_spawn_file_actions_addopen(3), posix_spawn_file_actions_destroy(3), posix_spawn_file_actions_init(3), posix_spawnattr_destroy(3), posix_spawnattr_getflags(3), posix_spawnattr_getpgroup(3), posix_spawnattr_getsigdefault(3), posix_spawnattr_getsigmask(3), posix_spawnattr_init(3), posix_spawnattr_setflags(3), posix_spawnattr_setpgroup(3), posix_spawnattr_setsigdefault(3), posix_spawnattr_setsigmask(3)

STANDARDS

The **posix_spawn()** and **posix_spawnp()** functions conform to IEEE Std 1003.1-2001 ("POSIX.1"), except that they ignore all errors from **close()**. A future update of the Standard is expected to require that these functions not fail because a file descriptor to be closed (via **posix_spawn_file_actions_addclose()**) is not open.

The optional scheduling related functions described in the standard are not available on z/OS and not implemented.

POSIX_SPAWNATTR_GETFLAGS(3)

NAME

posix_spawnattr_getflags, posix_spawnattr_setflags - get and set the spawn-flags attribute of a spawn attributes object

SYNOPSIS

```
#include <spawn.h>

int
posix_spawnattr_getflags(const posix_spawnattr_t *restrict attr,
         short *restrict flags);

int
posix_spawnattr_setflags(posix_spawnattr_t *attr, short flags);
```

DESCRIPTION

The **posix_spawnattr_getflags()** function obtains the value of the spawn-flags attribute from the attributes object referenced by *attr*.

The **posix_spawnattr_setflags()** function sets the spawn-flags attribute in an initialized attributes object referenced by *attr*.

The spawn-flags attribute is used to indicate which process attributes are to be changed in the new process image when invoking **posix_spawn()** or **posix_spawnp()**. It is the bitwise-inclusive OR of zero or more of the following flags (see **posix_spawn()**):

`POSIX_SPAWN_RESETIDS`

`POSIX_SPAWN_SETPGROUP`

`POSIX_SPAWN_SETSIGDEF`

`POSIX_SPAWN_SETSIGMASK`

These flags are defined in `<spawn.h>`. The default value of this attribute is as if no flags were set.

RETURN VALUES

The **posix_spawnattr_getflags()** and **posix_spawnattr_setflags()** functions return zero.

SEE ALSO

posix_spawn(3), posix_spawnattr_destroy(3), posix_spawnattr_init(3), posix_spawnp(3)

STANDARDS

The **posix_spawnattr_getflags()** and **posix_spawnattr_setflags()** functions conform to IEEE Std 1003.1-2001 ("POSIX.1").

POSIX_SPAWNATTR_GETPGROUP(3)

NAME

posix_spawnattr_getpgroup, posix_spawnattr_setpgroup - get and set the spawn-pgroup attribute of a spawn attributes object

SYNOPSIS

```
#include <spawn.h>

int
posix_spawnattr_getpgroup(const posix_spawnattr_t *restrict attr,
    pid_t *restrict pgroup);

int
posix_spawnattr_setpgroup(posix_spawnattr_t *attr, pid_t pgroup);
```

DESCRIPTION

The **posix_spawnattr_getpgroup()** function obtains the value of the spawn-pgroup attribute from the attributes object referenced by *attr*.

The **posix_spawnattr_setpgroup()** function sets the spawn-pgroup attribute in an initialized attributes object referenced by *attr*.

The spawn-pgroup attribute represents the process group to be joined by the new process image in a spawn operation (if `POSIX_SPAWN_SETPGROUP` is set in the spawn-flags attribute). The default value of this attribute is zero.

RETURN VALUES

The **posix_spawnattr_getpgroup()** and **posix_spawnattr_setpgroup()** functions return zero.

SEE ALSO

posix_spawn(3), posix_spawnattr_destroy(3), posix_spawnattr_init(3), posix_spawnp(3)

STANDARDS

The **posix_spawnattr_getpgroup()** and **posix_spawnattr_setpgroup()** functions conform to IEEE Std 1003.1-2001 ("POSIX.1").

POSIX_SPAWNATTR_GETSIGDEFAULT(3)

NAME

posix_spawnattr_getsigdefault, posix_spawnattr_setsigdefault - get and set the spawn-sigdefault attribute of a spawn attributes object

SYNOPSIS

```
#include <spawn.h>

int
posix_spawnattr_getsigdefault(const posix_spawnattr_t *restrict attr,
         sigset_t *restrict sigdefault);

int
posix_spawnattr_setsigdefault(posix_spawnattr_t *attr,
         const sigset_t *restrict sigdefault);
```

DESCRIPTION

The **posix_spawnattr_getsigdefault()** function obtains the value of the spawn-sigdefault attribute from the attributes object referenced by *attr*.

The **posix_spawnattr_setsigdefault()** function sets the spawn-sigdefault attribute in an initialized attributes object referenced by *attr*.

The spawn-sigdefault attribute represents the set of signals to be forced to default signal handling in the new process image (if `POSIX_SPAWN_SETSIGDEF` is set in the spawn-flags attribute) by a spawn operation. The default value of this attribute is an empty signal set.

RETURN VALUES

The **posix_spawnattr_getsigdefault()** and **posix_spawnattr_setsigdefault()** functions return zero.

SEE ALSO

posix_spawn(3), posix_spawnattr_destroy(3), posix_spawnattr_getsigmask(3), posix_spawnattr_init(3), posix_spawnattr_setsigmask(3), posix_spawnp(3)

STANDARDS

The **posix_spawnattr_getsigdefault()** and **posix_spawnattr_setsigdefault()** functions conform to IEEE Std 1003.1-2001 ("POSIX.1").

POSIX_SPAWNATTR_GETSIGMASK(3)

NAME

posix_spawnattr_getsigmask, posix_spawnattr_setsigmask - get and set the spawn-sigmask attribute of a spawn attributes object

SYNOPSIS

```
#include <spawn.h>

int
posix_spawnattr_getsigmask(const posix_spawnattr_t *restrict attr,
         sigset_t *restrict sigmask);

int
posix_spawnattr_setsigmask(posix_spawnattr_t *attr,
         const sigset_t *restrict sigmask);
```

DESCRIPTION

The **posix_spawnattr_getsigmask()** function obtains the value of the spawn-sigmask attribute from the attributes object referenced by *attr*.

The **posix_spawnattr_setsigmask()** function sets the spawn-sigmask attribute in an initialized attributes object referenced by *attr*.

The spawn-sigmask attribute represents the signal mask in effect in the new process image of a spawn operation (if `POSIX_SPAWN_SETSIGMASK` is set in the spawn-flags attribute). The default value of this attribute is unspecified.

RETURN VALUES

The **posix_spawnattr_getsigmask()** and **posix_spawnattr_setsigmask()** functions return zero.

SEE ALSO

posix_spawn(3), posix_spawnattr_destroy(3), posix_spawnattr_getsigmask(3), posix_spawnattr_init(3), posix_spawnattr_setsigmask(3), posix_spawnp(3)

STANDARDS

The **posix_spawnattr_getsigmask()** and **posix_spawnattr_setsigmask()** functions conform to IEEE Std 1003.1-2001 ("POSIX.1").

POSIX_SPAWNATTR_INIT(3)

NAME

posix_spawnattr_init, posix_spawnattr_destroy - initialize and destroy spawn attributes object

SYNOPSIS

```
#include <spawn.h>

int
posix_spawnattr_init(posix_spawnattr_t * attr);

int
posix_spawnattr_destroy(posix_spawnattr_t * attr);
```

DESCRIPTION

The **posix_spawnattr_init()** function initializes a spawn attributes object *attr* with the default value for all of the individual attributes used by the implementation. Initializing an already initialized spawn attributes object may cause memory to be leaked.

The **posix_spawnattr_destroy()** function destroys a spawn attributes object. A destroyed attr attributes object can be reinitialized using **posix_spawnattr_init()**. The object should not be used after it has been destroyed.

A spawn attributes object is of type `posix_spawnattr_t` (defined in `<spawn.h>`) and is used to specify the inheritance of process attributes across a spawn operation.

The resulting spawn attributes object (possibly modified by setting individual attribute values), is used to modify the behavior of **posix_spawn()** or **posix_spawnp()**. After a spawn attributes object has been used to spawn a process by a call to a **posix_spawn()** or **posix_spawnp()**, any function affecting the attributes object (including destruction) will not affect any process that has been spawned in this way.

RETURN VALUES

Upon successful completion, **posix_spawnattr_init()** and **posix_spawnattr_destroy()** return zero; otherwise, an error number is returned to indicate the error.

ERRORS

The **posix_spawnattr_init()** function will fail if:

[ENOMEM] Insufficient memory exists to initialize the spawn attributes object.

SEE ALSO

posix_spawn(3), posix_spawnp(3)

STANDARDS

The **posix_spawnattr_init()** and **posix_spawnattr_destroy()** functions conform to IEEE Std 1003.1-2001 (“POSIX.1”).

POSIX_SPAWN_FILE_ACTIONS_ADDOPEN(3)

NAME

posix_spawn_file_actions_addopen, posix_spawn_file_actions_adddup2, posix_spawn_file_actions_addclose - add open, dup2 or close action to spawn file actions object

LIBRARY Standard C Library (libc, -lc)

SYNOPSIS

```
#include <spawn.h>

int
posix_spawn_file_actions_addopen(posix_spawn_file_actions_t * file_actions,
         int fildes, const char *restrict path, int oflag, mode_t mode);

int
posix_spawn_file_actions_adddup2(posix_spawn_file_actions_t * file_actions,
         int fildes, int newfildes);

int
posix_spawn_file_actions_addclose(posix_spawn_file_actions_t * file_actions,
         int fildes);
```

DESCRIPTION

These functions add an open, dup2 or close action to a spawn file actions object.

A spawn file actions object is of type `posix_spawn_file_actions_t` (defined in `<spawn.h>`) and is used to specify a series of actions to be performed by a **posix_spawn()** or **posix_spawnp()** operation in order to arrive at the set of open file descriptors for the child process given the set of open file descriptors of the parent.

A spawn file actions object, when passed to **posix_spawn()** or **posix_spawnp()**, specify how the set of open file descriptors in the calling process is transformed into a set of potentially open file descriptors for the spawned process. This transformation is as if the specified sequence of actions was performed exactly once, in the context of the spawned process (prior to execution of the new process image), in the order in which the actions were added to the object; additionally, when the new process image is executed, any file descriptor (from this new set) which has its `FD_CLOEXEC` flag set is closed (see **posix_spawn()**).

The **posix_spawn_file_actions_addopen()** function adds an open action to the object referenced by *file_actions* that causes the file named by *path* to be opened (as if

```
open(path, oflag, mode)
```

had been called, and the returned file descriptor, if not *fildes*, had been changed to *fildes*) when a new process is spawned using this file actions object. If *fildes* was already an open file descriptor, it is closed before the new file is opened.

The string described by *path* is copied by the **posix_spawn_file_actions_addopen()** function.

The **posix_spawn_file_actions_adddup2()** function adds a dup2 action to the object referenced by *file_actions* that causes the file descriptor *fildes* to be duplicated as *newfildes* (as if

```
dup2(fildes, newfildes)
```

had been called) when a new process is spawned using this file actions object, except that the `FD_CLOEXEC` flag for *newfildes* is cleared even if *fildes* is equal to *newfildes*. The difference from **dup2()** is useful for passing a particular file descriptor to a particular child process.

The **posix_spawn_file_actions_addclose()** function adds a close action to the object referenced by *file_actions* that causes the file descriptor *fildes* to be closed (as if

```
close(fildes)
```

had been called) when a new process is spawned using this file actions object.

RETURN VALUES

Upon successful completion, these functions return zero; otherwise, an error number is returned to indicate the error.

ERRORS

These functions fail if:

`[EBADF]` The value specified by *fildes* or *newfildes* is negative.

`[ENOMEM]` Insufficient memory exists to add to the spawn file actions object.

SEE ALSO

close(2), manrefdup22, manrefopen2, manrefposix_spawn3, posix_spawn_file_actions_destroy(3), manrefposix_spawn_file_actions_init3, posix_spawnp(3)

STANDARDS

The **posix_spawn_file_actions_addopen()**, **posix_spawn_file_actions_adddup2()** and **posix_spawn_file_actions_addclose()** functions conform to IEEE Std 1003.1-2001 ("POSIX.1"), with the exception of the behavior of **posix_spawn_file_actions_adddup2()** if *fildes* is equal to *newfildes* (clearing `FD_CLOEXEC`). A future update of the Standard is expected to require this behavior.

POSIX_SPAWN_FILE_ACTIONS_INIT(3)

NAME

posix_spawn_file_actions_init, posix_spawn_file_actions_destroy - initialize and destroy spawn file actions object

SYNOPSIS

```
#include <spawn.h>

int
posix_spawn_file_actions_init(posix_spawn_file_actions_t * file_actions);

int
posix_spawn_file_actions_destroy(posix_spawn_file_actions_t * file_actions);
```

DESCRIPTION

The **posix_spawn_file_actions_init()** function initialize the object referenced by *file_actions* to contain no file actions for **posix_spawn()** or **posix_spawnp()**. Initializing an already initialized spawn file actions object may cause memory to be leaked.

The **posix_spawn_file_actions_destroy()** function destroy the object referenced by *file_actions*; the object becomes, in effect, uninitialized. A destroyed spawn file actions object can be reinitialized using **posix_spawn_file_actions_init()**. The object should not be used after it has been destroyed.

RETURN VALUES

Upon successful completion, these functions return zero; otherwise, an error number is returned to indicate the error.

ERRORS

The **posix_spawn_file_actions_init()** function will fail if:

`[ENOMEM]` Insufficient memory exists to initialize the spawn file actions object.

SEE ALSO

posix_spawn(3), posix_spawn_file_actions_addclose(3), posix_spawn_file_actions_adddup2(3), posix_spawn_file_actions_addopen(3), posix_spawnp(3)

STANDARDS The **posix_spawn_file_actions_init()** and **posix_spawn_file_actions_destroy()** functions conform to IEEE Std 1003.1-2001 (“POSIX.1”).

PSELECT(3)

NAME

pselect – synchronous I/O multiplexing a la POSIX.1g

SYNOPSIS

```
#include <sys/select.h>

int pselect(int nfds,
            fd_set * restrict readfds, fd_set * restrict writefds,
            fd_set * restrict exceptfds,
            const struct timespec * restrict timeout,
            const sigset_t * restrict newsigmask);
```

DESCRIPTION

The **pselect()** function was introduced by IEEE Std 1003.1g-2000 ("POSIX.1") as a slightly stronger version of select(2). The *nfds*, *readfds*, *writefds*, and *exceptfds* arguments are all identical to the analogous arguments of **select()**. The *timeout* argument in **pselect()** points to a `const struct timespec` rather than the (modifiable) `struct timeval` used by **select()**; as in **select()**, a null pointer may be passed to indicate that **pselect()** should wait indefinitely.i Finally, *newsigmask* specifies a signal mask which is set while waiting for input. When **pselect()** returns, the original signal mask is restored.

See select(3) for a more detailed discussion of the semantics of this interface, and for macros used to manipulate the `fd_set` data type.

IMPLEMENTATION NOTES

The **pselect()** function is implemented in the C library as a wrapper around **select()**.

RETURN VALUES

The **pselect()** function returns the same values and under the same conditions as **select()**.

ERRORS

The **pselect()** function may fail for any of the reasons documented for select(3) and (if a signal mask is provided) sigprocmask(2).

SEE ALSO

poll(2), select(3), sigprocmask(2)

STANDARDS

The **pselect()** function conforms to IEEE Std 1003.1-2001 (“POSIX.1”).

PSIGNAL(3)

NAME

psignal, strsignal, sys_siglist, sys_signame - system signal messages

SYNOPSIS

```
#include <signal.h>

void
psignal(unsigned sig, const char *s);

extern const char * const sys_siglist[];
extern const char * const sys_signame[];

#include <string.h>

char *
strsignal(int sig);
```

DESCRIPTION

The **psignal()** and **strsignal()** functions locate the descriptive message string for a signal number.

The **strsignal()** function accepts a signal number argument *sig* and returns a pointer to the corresponding message string.

The **psignal()** function accepts a signal number argument *sig* and writes it to the standard error file descriptor. If the argument *s* is non-`NULL` and does not point to the null character, *s* is written to the standard error file descriptor prior to the message string, immediately followed by a colon and a space. If the signal number is not recognized, the string `"Unknown signal"` is produced.

The message strings can be accessed directly through the external array `sys_siglist`, indexed by recognized signal numbers. The external array `sys_signame` is used similarly and contains short, lower-case abbreviations for signals which are useful for recognizing signal names in user input. The defined variable `NSIG` contains a count of the strings in `sys_siglist` and `sys_signame`.

SEE ALSO

perror(3), strerror(3)

PTSNAME(3)

NAME

ptsname - get the pathname of a slave pty (pseudo-terminal)

SYNOPSIS

```
#include <stdlib.h>

char *ptsname(int filedes);
```

DESCRIPTION

ptsname() returns the name of the slave pseudo-terminal associated with a master terminal device referenced by the open *filedes*.

The minor numbers of the slave and master device will be the same.

RETURN VALUE

If successful, **ptsname()** returns the NUL-terminated name of the complete path name of the slave device, otherwise a `NULL` pointer is returned and the global variable `errno` is set to indicate the error.

ERRORS

As well as the errors described in stat(2), **ptsname()** will fail if:

`[ENOTTY]` *filedes* is not associated with a tty or does not represent a master pty.

`[ENOTTY]` The associated slave device was not present in the system, indicating a configuration error.

SEE ALSO

grantpt(2), unlockpt(2)

ISSUES

The **ptsname()** function leaves its result in an internal static object and returns a pointer to that object. Subsequent calls to **ptsname()** will modify the same object.

PAUSE(3)

NAME

pause – stop until signal

SYNOPSIS

```
#include <unistd.h>

int
pause(void);
```

DESCRIPTION

Pause() is made obsolete by sigsuspend(2).

The **pause()** function forces a process to pause until a signal is received from either the kill(2) function or an interval timer. Upon termination of a signal handler started during a **pause()**, the **pause()** call will return.

RETURN VALUES

Always returns -1.

IMPLEMENTATION NODES

The **pause()** function requires POSIX signals. If POSIX signals are not availble, **pause()** immediately returns a -1 with `errno` set to `ENOSYS`.

ERRORS

The **pause()** function always returns -1 and sets the `errno` value to:

`[EINTR]`	The call was interrupt.
`[ENOSYS]`	POSIX signals were not available.

SEE ALSO

kill(2), select(2), sigsuspend(2)

QUEUE(3)

NAME

SLIST_EMPTY, SLIST_ENTRY, SLIST_FIRST, SLIST_FOREACH, SLIST_HEAD,
SLIST_INIT, SLIST_INSERT_AFTER, SLIST_INSERT_HEAD, SLIST_NEXT,
SLIST_REMOVE_HEAD, SLIST_REMOVE, STAILQ_EMPTY, STAILQ_ENTRY,
STAILQ_FIRST, STAILQ_FOREACH, STAILQ_HEAD, STAILQ_INIT,
STAILQ_INSERT_AFTER, STAILQ_INSERT_HEAD, STAILQ_INSERT_TAIL,
STAILQ_LAST, STAILQ_NEXT, STAILQ_REMOVE_HEAD, STAILQ_REMOVE, LIST_EMPTY, LIST_ENTRY, LIST_FIRST, LIST_FOREACH, LIST_HEAD, LIST_INIT, LIST_INSERT_AFTER, LIST_INSERT_BEFORE, LIST_INSERT_HEAD, LIST_NEXT, LIST_REMOVE, TAILQ_EMPTY, TAILQ_ENTRY, TAILQ_FIRST, TAILQ_FOREACH, TAILQ_FOREACH_REVERSE, TAILQ_HEAD, TAILQ_INIT, TAILQ_INSERT_AFTER, TAILQ_INSERT_BEFORE, TAILQ_INSERT_HEAD, TAILQ_INSERT_TAIL, TAILQ_LAST, TAILQ_NEXT, TAILQ_PREV, TAILQ_REMOVE, CIRCLEQ_EMPTY, CIRCLEQ_ENTRY, CIRCLEQ_FIRST, CIRCLEQ_FOREACH, CIRCLEQ_FOREACH_REVERSE, CIRCLEQ_HEAD, CIRCLEQ_INIT, CIRCLEQ_INSERT_AFTER, CIRCLEQ_INSERT_BEFORE, CIRCLEQ_INSERT_HEAD, CIRCLEQ_INSERT_TAIL, CIRCLE_LAST, CIRCLE_NEXT, CIRCLE_PREV, CIRCLEQ_REMOVE - implementations of singly-linked lists, singly-linked tail queues, lists, tail queues, and circular queues

SYNOPSIS

```
#include <sys/queue.h>

SLIST_EMPTY(SLIST_HEAD *head);

SLIST_ENTRY(TYPE);

SLIST_FIRST(SLIST_HEAD *head);

SLIST_FOREACH(TYPE *var, SLIST_HEAD *head, SLIST_ENTRY NAME);

SLIST_HEAD(HEADNAME, TYPE);

SLIST_INIT(SLIST_HEAD *head);

SLIST_INSERT_AFTER(TYPE *listelm, TYPE *elm, SLIST_ENTRY NAME);

SLIST_INSERT_HEAD(SLIST_HEAD *head, TYPE *elm, SLIST_ENTRY NAME);

SLIST_NEXT(TYPE *elm, SLIST_ENTRY NAME);
```

```
SLIST_REMOVE_HEAD(SLIST_HEAD *head, SLIST_ENTRY NAME);

SLIST_REMOVE(SLIST_HEAD *head, TYPE *elm, TYPE, SLIST_ENTRY NAME);

STAILQ_EMPTY(STAILQ_HEAD *head);

STAILQ_ENTRY(TYPE);

STAILQ_FIRST(STAILQ_HEAD *head);

STAILQ_FOREACH(TYPE *var, STAILQ_HEAD *head, STAILQ_ENTRY NAME);

STAILQ_HEAD(HEADNAME, TYPE);

STAILQ_INIT(STAILQ_HEAD *head);

STAILQ_INSERT_AFTER(STAILQ_HEAD *head, TYPE *listelm, TYPE *elm,
                    STAILQ_ENTRY NAME);

STAILQ_INSERT_HEAD(STAILQ_HEAD *head, TYPE *elm, STAILQ_ENTRY NAME);

STAILQ_INSERT_TAIL(STAILQ_HEAD *head, TYPE *elm, STAILQ_ENTRY NAME);

STAILQ_LAST(STAILQ_HEAD *head);

STAILQ_NEXT(TYPE *elm, STAILQ_ENTRY NAME);

STAILQ_REMOVE_HEAD(STAILQ_HEAD *head, STAILQ_ENTRY NAME);

STAILQ_REMOVE(STAILQ_HEAD *head, TYPE *elm, TYPE, STAILQ_ENTRY NAME);

LIST_EMPTY(LIST_HEAD *head);

LIST_ENTRY(TYPE);

LIST_FIRST(LIST_HEAD *head);

LIST_FOREACH(TYPE *var, LIST_HEAD *head, LIST_ENTRY NAME);

LIST_HEAD(HEADNAME, TYPE);

LIST_INIT(LIST_HEAD *head);

LIST_INSERT_AFTER(TYPE *listelm, TYPE *elm, LIST_ENTRY NAME);

LIST_INSERT_BEFORE(TYPE *listelm, TYPE *elm, LIST_ENTRY NAME);
```

```
LIST_INSERT_HEAD(LIST_HEAD *head, TYPE *elm, LIST_ENTRY NAME);

LIST_NEXT(TYPE *elm, LIST_ENTRY NAME);

LIST_REMOVE(TYPE *elm, LIST_ENTRY NAME);

TAILQ_EMPTY(TAILQ_HEAD *head);

TAILQ_ENTRY(TYPE);

TAILQ_FIRST(TAILQ_HEAD *head);

TAILQ_FOREACH(TYPE *var, TAILQ_HEAD *head, TAILQ_ENTRY NAME);

TAILQ_FOREACH_REVERSE(TYPE *var, TAILQ_HEAD *head, HEADNAME,
                      TAILQ_ENTRY NAME);

TAILQ_HEAD(HEADNAME, TYPE);

TAILQ_INIT(TAILQ_HEAD *head);

TAILQ_INSERT_AFTER(TAILQ_HEAD *head, TYPE *listelm, TYPE *elm,
                   TAILQ_ENTRY NAME);

TAILQ_INSERT_BEFORE(TYPE *listelm, TYPE *elm, TAILQ_ENTRY NAME);

TAILQ_INSERT_HEAD(TAILQ_HEAD *head, TYPE *elm, TAILQ_ENTRY NAME);

TAILQ_INSERT_TAIL(TAILQ_HEAD *head, TYPE *elm, TAILQ_ENTRY NAME);

TAILQ_LAST(TAILQ_HEAD *head, HEADNAME);

TAILQ_NEXT(TYPE *elm, TAILQ_ENTRY NAME);

TAILQ_PREV(TYPE *elm, HEADNAME, TAILQ_ENTRY NAME);

TAILQ_REMOVE(TAILQ_HEAD *head, TYPE *elm, TAILQ_ENTRY NAME);

CIRCLEQ_EMPTY(CIRCLEQ_HEAD *head);

CIRCLEQ_ENTRY(TYPE);

CIRCLEQ_FIRST(CIRCLEQ_HEAD *head);

CIRCLEQ_FOREACH(TYPE *var, CIRCLEQ_HEAD *head, CIRCLEQ_ENTRY NAME);
```

```
CIRCLEQ_FOREACH_REVERSE(TYPE *var, CIRCLEQ_HEAD *head,
                        CIRCLEQ_ENTRY NAME);

CIRCLEQ_HEAD(HEADNAME, TYPE);

CIRCLEQ_INIT(CIRCLEQ_HEAD *head);

CIRCLEQ_INSERT_AFTER(CIRCLEQ_HEAD *head, TYPE *listelm, TYPE *elm,
                     CIRCLEQ_ENTRY NAME);

CIRCLEQ_INSERT_BEFORE(CIRCLEQ_HEAD *head, TYPE *listelm, TYPE *elm,
                      CIRCLEQ_ENTRY NAME);

CIRCLEQ_INSERT_HEAD(CIRCLEQ_HEAD *head, TYPE *elm, CIRCLEQ_ENTRY NAME);

CIRCLEQ_INSERT_TAIL(CIRCLEQ_HEAD *head, TYPE *elm, CIRCLEQ_ENTRY NAME);

CIRCLEQ_LAST(CIRCLEQ_HEAD *head);

CIRCLEQ_NEXT(TYPE *elm, CIRCLEQ_ENTRY NAME);

CIRCLE_PREV(TYPE *elm, CIRCLEQ_ENTRY NAME);

CIRCLEQ_REMOVE(CIRCLEQ_HEAD *head, TYPE *elm, CIRCLEQ_ENTRY NAME);
```

DESCRIPTION

These macros define and operate on five types of data structures: singly-linked lists, singly-linked tail queues, lists, tail queues, and circular queues. All five structures support the following functionality:

1. Insertion of a new entry at the head of the list.
2. Insertion of a new entry after any element in the list.
3. O(1) removal of an entry from the head of the list.
4. O(n) removal of any entry in the list.
5. Forward traversal through the list.

Singly-linked lists are the simplest of the five data structures and support only the above functionality. Singly-linked lists are ideal for applications with large datasets and few or no removals, or for implementing a LIFO queue.

Singly-linked tail queues add the following functionality:

1. Entries can be added at the end of a list.

However:

1. All list insertions must specify the head of the list.
2. Each head entry requires two pointers rather than one.
3. Code size is about 15

Singly-linked tailqs are ideal for applications with large datasets and few or no removals, or for implementing a FIFO queue.

All doubly linked types of data structures (lists, tail queues, and circle queues) additionally allow:

1. Insertion of a new entry before any element in the list.
2. O(1) removal of any entry in the list.

However:

1. Each elements requires two pointers rather than one.
2. Code size and execution time of operations (except for removal) is about twice that of the singly-linked data-structures.

Linked lists are the simplest of the doubly linked data structures and support only the above functionality over singly-linked lists.

Tail queues add the following functionality:

1. Entries can be added at the end of a list.
2. They may be traversed backwards, from tail to head.

However:

1. All list insertions and removals must specify the head of the list.
2. Each head entry requires two pointers rather than one.
3. Code size is about 15

Circular queues add the following functionality:

1. Entries can be added at the end of a list.
2. They may be traversed backwards, from tail to head.

However:

1. All list insertions and removals must specify the head of the list.
2. Each head entry requires two pointers rather than one.
3. The termination condition for traversal is more complex.
4. Code size is about 40

In the macro definitions, *TYPE* is the name of a user defined structure, that must contain a field of type `SLIST_ENTRY`, `STAILQ_ENTRY`, `LIST_ENTRY`, `TAILQ_ENTRY`, or `CIRCLEQ_ENTRY`, named *NAME*. The argument *HEADNAME* is the name of a user defined structure that must be declared using the macros `SLIST_HEAD`, `STAILQ_HEAD`, `LIST_HEAD`, `TAILQ_HEAD`, or `CIRCLEQ_HEAD`. See the examples below for further explanation of how these macros are used.

SINGLY-LINKED LISTS

A singly-linked list is headed by a structure defined by the `SLIST_HEAD` macro. This structure contains a single pointer to the first element on the list. The elements are singly linked for minimum space and pointer manipulation overhead at the expense of O(n) removal for arbitrary elements. New elements can be added to the list after an existing element or at the head of the list. An `SLIST_HEAD` structure is declared as follows:

```
SLIST_HEAD(HEADNAME, TYPE) head;
```

where *HEADNAME* is the name of the structure to be defined, and *TYPE* is the type of the elements to be linked into the list. A pointer to the head of the list can later be declared as:

```
struct HEADNAME *headp;
```

(The names head and headp are user selectable.)

The macro `SLIST_EMPTY` evaluates to true if there are no elements in the list.

The macro `SLIST_ENTRY` declares a structure that connects the elements in the list.

The macro `SLIST_FIRST` returns the first element in the list of `NULL` if the list is empty.

The macro `SLIST_FOREACH` traverses the list referenced by *head* in the forward direction, assigning each element in turn to *var*

The macro `SLIST_INIT` initializes the list referenced by *head*.

The macro SLIST_INSERT_HEAD inserts the new element *elm* at the head of the list.

The macro SLIST_INSERT_AFTER inserts the new element *elm* after the element *listelm*.

The macro SLIST_NEXT returns the next element in the list.

The macro SLIST_REMOVE_HEAD removes the element *elm* from the head of the list. For optimum efficiency, elements being removed from the head of the list should explicitly use this macro instead of the generic SLIST_REMOVE macro.

The macro SLIST_REMOVE removes the element *elm* from the list.

SINGLY-LINKED LIST EXAMPLE

```
SLIST_HEAD(slisthead, entry) head;
struct slisthead *headp;                /* Singly-linked List head. */
struct entry {
        ...
        SLIST_ENTRY(entry) entries;     /* Singly-linked List. */
        ...
} *n1, *n2, *n3, *np;

SLIST_INIT(&head);                      /* Initialize the list. */

n1 = malloc(sizeof(struct entry));      /* Insert at the head. */
SLIST_INSERT_HEAD(&head, n1, entries);

n2 = malloc(sizeof(struct entry));      /* Insert after. */
SLIST_INSERT_AFTER(n1, n2, entries);

SLIST_REMOVE(&head, n2, entry, entries);/* Deletion. */
free(n2);

n3 = SLIST_FIRST(&head);
SLIST_REMOVE_HEAD(&head, entries);      /* Deletion. */
free(n3);

                                        /* Forward traversal. */
SLIST_FOREACH(np, &head, entries)
        np-> ...

while (!SLIST_EMPTY(&head)) { /* List Deletion. */
        n1 = SLIST_FIRST(&head);
        SLIST_REMOVE_HEAD(&head, entries);
        free(n1);
}
```

SINGLY-LINKED TAIL QUEUES

A singly-linked tail queue is headed by a structure defined by the `STAILQ_HEAD` macro. This structure contains a pair of pointers, one to the first element in the tail queue and the other to the last element in the tail queue. The elements are singly linked for minimum space and pointer manipulation overhead at the expense of O(n) removal for arbitrary elements. New elements can be added to the tail queue after an existing element, at the head of the tail queue, or at the end of the tail queue. A `STAILQ_HEAD` structure is declared as follows:

```
STAILQ_HEAD(HEADNAME, TYPE) head;
```

where *HEADNAME* is the name of the structure to be defined, and *TYPE* is the type of the elements to be linked into the tail queue. A pointer to the head of the tail queue can later be declared as:

```
struct HEADNAME *headp;
```

(the names *head* and *headp* are user selectable.)

The macro `STAILQ_EMPTY` evaluates to true if there are no items on the tail queue.

The macro `STAILQ_ENTRY` declares a structure that connects the elements in the tail queue.

The macro `STAILQ_FIRST` returns the first item on the tail queue or `NULL` if the tail queue is empty.

The macro `STAILQ_FOREACH` traverses the tail queue referenced by *head* in the forward direction, assigning each element in turn to *var*.

The macro `STAILQ_INIT` initializes the tail queue referenced by *head*.

The macro `STAILQ_INSERT_HEAD` inserts the new element *elm* at the head of the tail queue.

The macro `STAILQ_INSERT_TAIL` inserts the new element *elm* at the end of the tail queue.

The macro `STAILQ_INSERT_AFTER` inserts the new element *elm* after the element *listelm*.

The macro `STAILQ_LAST` returns the last item on the tail queue. If the tail queue is empty the return value is undefined.

The macro `STAILQ_NEXT` returns the next item on the tail queue, or `NULL` this item is the last.

The macro STAILQ_REMOVE_HEAD removes the element elm from the head of the tail queue. For optimum efficiency, elements being removed from the head of the tail queue should use this macro explicitly rather than the generic STAILQ_REMOVE macro.

The macro STAILQ_REMOVE removes the element *elm* from the tail queue.

SINGLY-LINKED TAIL QUEUE EXAMPLE

```
STAILQ_HEAD(stailhead, entry) head;
struct stailhead *headp;                /* Singly-linked tail queue head. */
struct entry {
        ...
        STAILQ_ENTRY(entry) entries;    /* Tail queue. */
        ...
} *n1, *n2, *n3, *np;

STAILQ_INIT(&head);                     /* Initialize the queue. */

n1 = malloc(sizeof(struct entry));      /* Insert at the head. */
STAILQ_INSERT_HEAD(&head, n1, entries);

n1 = malloc(sizeof(struct entry));      /* Insert at the tail. */
STAILQ_INSERT_TAIL(&head, n1, entries);

n2 = malloc(sizeof(struct entry));      /* Insert after. */
STAILQ_INSERT_AFTER(&head, n1, n2, entries);

                                        /* Deletion. */
STAILQ_REMOVE(&head, n2, entry, entries);
free(n2);

                                        /* Deletion from the head */
n3 = STAILQ_FIRST(&head);
STAILQ_REMOVE_HEAD(&head, entries);
free(n3);

                                        /* Forward traversal. */
STAILQ_FOREACH(np, &head, entries)
        np-> ...
                                        /* TailQ Deletion. */
while (!STAILQ_EMPTY(&head)) {
        n1 = STAILQ_HEAD(&head);
        STAILQ_REMOVE_HEAD(&head, entries);
        free(n1);
}
```

```
                                                /* Faster TailQ Deletion. */
n1 = STAILQ_FIRST(&head);
while (n1 != NULL) {
        n2 = STAILQ_NEXT(n1, entries);
        free(n1);
        n1 = n2;
}
STAILQ_INIT(&head);
```

LISTS

A list is headed by a structure defined by the `LIST_HEAD` macro. This structure contains a single pointer to the first element on the list. The elements are doubly linked so that an arbitrary element can be removed without traversing the list. New elements can be added to the list after an existing element, before an existing element, or at the head of the list. A `LIST_HEAD` structure is declared as follows:

```
        LIST_HEAD(HEADNAME, TYPE) head;
```

where *HEADNAME* is the name of the structure to be defined, and *TYPE* is the type of the elements to be linked into the list. A pointer to the head of the list can later be declared as:

```
        struct HEADNAME *headp;
```

(The names *head* and *headp* are user selectable.)

The macro `LIST_EMPTY` valuates to true if their are no elements in the list.

The macro `LIST_ENTRY` declares a structure that connects the elements in the list.

The macro `LIST_FIRST` returns the first element in the list or `NULL` if the list is empty.

The macro `LIST_FOREACH` traverses the list referenced by *head* in the forward direction, assigning each element in turn to *var*.

The macro `LIST_INIT` initializes the list referenced by *head*.

The macro `LIST_INSERT_HEAD` inserts the new element *elm* at the head of the list.

The macro `LIST_INSERT_AFTER` inserts the new element *elm* after the element *listelm*.

The macro `LIST_INSERT_BEFORE` inserts the new element *elm* before the element *listelm*.

The macro `LIST_NEXT` returns the next element in the list, or `NULL` if this is the last.

The macro `LIST_REMOVE` removes the element *elm* from the list.

LIST EXAMPLE

```
LIST_HEAD(listhead, entry) head;
struct listhead *headp;         /* List head. */
struct entry {
        ...
        LIST_ENTRY(entry) entries;      /* List. */
        ...
} *n1, *n2, *n3, *np;

LIST_INIT(&head);                       /* Initialize the list. */

n1 = malloc(sizeof(struct entry));      /* Insert at the head. */
LIST_INSERT_HEAD(&head, n1, entries);

n2 = malloc(sizeof(struct entry));      /* Insert after. */
LIST_INSERT_AFTER(n1, n2, entries);

n3 = malloc(sizeof(struct entry));      /* Insert before. */
LIST_INSERT_BEFORE(n2, n3, entries);

LIST_REMOVE(n2, entries);               /* Deletion. */
free(n2);

                                        /* Forward traversal. */
LIST_FOREACH(np, &head, entries)
        np-> ...

while (!LIST_EMPTY(&head)) {            /* List Deletion. */
        n1 = LIST_FIRST(&head);
        LIST_REMOVE(n1, entries);
        free(n1);
}

n1 = LIST_FIRST(&head);                 /* Faster List Delete. */
while (n1 != NULL) {
        n2 = LIST_NEXT(n1, entries);
        free(n1);
        n1 = n2;
}
LIST_INIT(&head);
```

TAIL QUEUES

A tail queue is headed by a structure defined by the `TAILQ_HEAD` macro. This structure contains a pair of pointers, one to the first element in the tail queue and

the other to the last element in the tail queue. The elements are doubly linked so that an arbitrary element can be removed without traversing the tail queue. New elements can be added to the tail queue after an existing element, before an existing element, at the head of the tail queue, or at the end of the tail queue. A `TAILQ_HEAD` structure is declared as follows:

```
TAILQ_HEAD(HEADNAME, TYPE) head;
```

where *HEADNAME* is the name of the structure to be defined, and *TYPE* is the type of the elements to be linked into the tail queue. A pointer to the head of the tail queue can later be declared as:

```
struct HEADNAME *headp;
```

(The names *head* and *headp* are user selectable.)

The macro `TAILQ_EMPTY` evaluates to true if there are no items on the tail queue.

The macro `TAILQ_ENTRY` declares a structure that connects the elements in the tail queue.

The macro `TAILQ_FIRST` returns the first item on the tail queue or `NULL` if the tail queue is empty.

The macro `TAILQ_FOREACH` traverses the tail queue referenced by *head* in the forward direction, assigning each element in turn to *var*.

The macro `TAILQ_FOREACH_REVERSE` traverses the tail queue referenced by *head* in the reverse direction, assigning each element in turn to *var*.

The macro `TAILQ_INIT` initializes the tail queue referenced by *head*.

The macro `TAILQ_INSERT_HEAD` inserts the new element *elm* at the head of the tail queue.

The macro `TAILQ_INSERT_TAIL` inserts the new element *elm* at the end of the tail queue.

The macro `TAILQ_INSERT_AFTER` inserts the new element *elm* after the element *listelm*.

The macro `TAILQ_INSERT_BEFORE` inserts the new element *elm* before the element *listelm*.

The macro `TAILQ_LAST` returns the last item on the tail queue. If the tail queue is empty the return value is undefined.

The macro TAILQ_NEXT returns the next item on the tail queue, or NULL if this item is the last.

The macro TAILQ_PREV returns the previous item on the tail queue, or NULL if this item is the first.

The macro TAILQ_REMOVE removes the element *elm* from the tail queue.

TAIL QUEUE EXAMPLE

```
TAILQ_HEAD(tailhead, entry) head;
struct tailhead *headp;                 /* Tail queue head. */
struct entry {
        ...
        TAILQ_ENTRY(entry) entries;     /* Tail queue. */
        ...
} *n1, *n2, *n3, *np;

TAILQ_INIT(&head);                      /* Initialize the queue. */

n1 = malloc(sizeof(struct entry));      /* Insert at the head. */
TAILQ_INSERT_HEAD(&head, n1, entries);

n1 = malloc(sizeof(struct entry));      /* Insert at the tail. */
TAILQ_INSERT_TAIL(&head, n1, entries);

n2 = malloc(sizeof(struct entry));      /* Insert after. */
TAILQ_INSERT_AFTER(&head, n1, n2, entries);

n3 = malloc(sizeof(struct entry));      /* Insert before. */
TAILQ_INSERT_BEFORE(n2, n3, entries);

TAILQ_REMOVE(&head, n2, entries);       /* Deletion. */
free(n2);
                                        /* Forward traversal. */
TAILQ_FOREACH(np, &head, entries)
        np-> ...
                                        /* Reverse traversal. */
TAILQ_FOREACH_REVERSE(np, &head, tailhead, entries)
        np-> ...
                                        /* TailQ Deletion. */
while (!TAILQ_EMPTY(head)) {
        n1 = TAILQ_FIRST(&head);
        TAILQ_REMOVE(&head, n1, entries);
        free(n1);
}
                                        /* Faster TailQ Deletion. */
```

```
n1 = TAILQ_FIRST(&head);
while (n1 != NULL) {
        n2 = TAILQ_NEXT(n1, entries);
        free(n1);
        n1 = n2;
}
TAILQ_INIT(&head);
```

CIRCULAR QUEUES

A circular queue is headed by a structure defined by the `CIRCLEQ_HEAD` macro. This structure contains a pair of pointers, one to the first element in the circular queue and the other to the last element in the circular queue. The elements are doubly linked so that an arbitrary element can be removed without traversing the queue. New elements can be added to the queue after an existing element, before an existing element, at the head of the queue, or at the end of the queue. A `CIRCLEQ_HEAD` structure is declared as follows:

```
CIRCLEQ_HEAD(HEADNAME, TYPE) head;
```

where *HEADNAME* is the name of the structure to be defined, and *TYPE* is the type of the elements to be linked into the circular queue. A pointer to the head of the circular queue can later be declared as:

```
struct HEADNAME *headp;
```

(The names *head* and *headp* are user selectable.)

The macro `CIRCLEQ_EMPTY` evaluates to true if there are no items on the circle queue.

The macro `CIRCLEQ_ENTRY` declares a structure that connects the elements in the circular queue.

The macro `CIRCLEQ_FIRST` returns the first item on the circle queue.

The macro `CICRLEQ_FOREACH` traverses the circle queue referenced by *head* in the forward direction, assigning each element in turn to *var*.

The macro `CICRLEQ_FOREACH_REVERSE` traverses the circle queue referenced by *head* in the reverse direction, assigning each element in turn to *var*.

The macro `CIRCLEQ_INIT` initializes the circular queue referenced by *head*.

The macro `CIRCLEQ_INSERT_HEAD` inserts the new element *elm* at the head of the circular queue.

The macro CIRCLEQ_INSERT_TAIL inserts the new element *elm* at the end of the circular queue.

The macro CIRCLEQ_INSERT_AFTER inserts the new element *elm* after the element *listelm*.

The macro CIRCLEQ_INSERT_BEFORE inserts the new element *elm* before the element *listelm*.

The macro CIRCLEQ_LAST returns the last item on the circle queue.

The macro CIRCLEQ_NEXT returns the next item on the circle queue.

The macro CIRCLEQ_PREV returns the previous item on the circle queue.

The macro CIRCLEQ_REMOVE removes the element *elm* from the circular queue.

CIRCULAR QUEUE EXAMPLE

```
CIRCLEQ_HEAD(circleq, entry) head;
struct circleq *headp;                  /* Circular queue head. */
struct entry {
        ...
        CIRCLEQ_ENTRY(entry) entries;   /* Circular queue. */
        ...
    } *n1, *n2, *np;

CIRCLEQ_INIT(&head);                    /* Initialize the circular queue. */

n1 = malloc(sizeof(struct entry));      /* Insert at the head. */
CIRCLEQ_INSERT_HEAD(&head, n1, entries);

n1 = malloc(sizeof(struct entry));      /* Insert at the tail. */
CIRCLEQ_INSERT_TAIL(&head, n1, entries);

n2 = malloc(sizeof(struct entry));      /* Insert after. */
CIRCLEQ_INSERT_AFTER(&head, n1, n2, entries);

n2 = malloc(sizeof(struct entry));      /* Insert before. */
CIRCLEQ_INSERT_BEFORE(&head, n1, n2, entries);

CIRCLEQ_REMOVE(&head, n1, entries);     /* Deletion. */
free(n1);
                                        /* Forward traversal. */
CIRCLEQ_FOREACH(np, &head, entries)
        np-> ...
                                        /* Reverse traversal. */
```

```
CIRCLEQ_FOREACH_REVERSE(np, &head, entries)
        np-> ...
                                        /* CircleQ Deletion. */
while (CIRCLEQ_FIRST(&head) != (void *)&head) {
        n1 = CIRCLEQ_HEAD(&head);
        CIRCLEQ_REMOVE(&head, n1, entries);
        free(n1);
}
                                        /* Faster CircleQ Deletion. */
n1 = CIRCLEQ_FIRST(&head);
while (n1 != (void *)&head) {
        n2 = CIRCLEQ_NEXT(n1, entries);
        free(n1);
        n1 = n2;
}
CIRCLEQ_INIT(&head);
```

RAISE(3)

NAME

raise - send a signal to the current program or process

SYNOPSIS

```
#include <signal.h>

int
raise(int sig);
```

DESCRIPTION

The **raise()** function sends the signal *sig* to the current process.

When running under OpenEdition, **raise()** is the same as `kill(getpid(), sig)`.

RETURN VALUES

The **raise()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

The **raise()** function may fail and set `errno` for any of the errors specified for the library functions getpid(2) and kill(2).

STANDARDS

The **raise()** function conforms to ISO/IEC 9899:1990 ("ISO C90").

SEM_DESTROY(3)

NAME

sem_destroy – destroy an unnamed semaphore

SYNOPSIS

```
#include <semaphore.h>

int
sem_destroy(sem_t *sem);
```

DESCRIPTION

The **sem_destroy()** function destroys the unnamed semaphore pointed to by *sem*. After a successful call to **sem_destroy()**, *sem* is unusable until reinitialized by another call to sem_init(3).

RETURN VALUES

The **sem_destroy()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

The **sem_destroy()** function will fail if:

`[EINVAL]` The *sem* argument points to an invalid semaphore.

`[EBUSY]` There are currently threads blocked on the semaphore that *sem* points to.

SEE ALSO

sem_init(3)

STANDARDS

The **sem_destroy()** function conforms to ISO/IEC 9945-1:1996 (“POSIX.1”).

SEM_GETVALUE(3)

NAME

sem_getvalue – get the value of a semaphore

SYNOPSIS

```
#include <semaphore.h>

int
sem_getvalue(sem_t * restrict sem, int * restrict sval);
```

DESCRIPTION

The **sem_getvalue()** function sets the variable pointed to by *sval* to the current value of the semaphore pointed to by *sem*, as of the time that the call to **sem_getvalue()** is actually run.

RETURN VALUES

The **sem_getvalue()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

The **sem_getvalue()** function will fail if:

`[EINVAL]` The *sem* argument points to an invalid semaphore.

SEE ALSO

sem_post(3), sem_trywait(3), sem_wait(3)

STANDARDS

The **sem_getvalue()** function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

The value of the semaphore is never negative, even if there are threads blocked on the semaphore. POSIX is somewhat ambiguous in its wording with regard to what the value of the semaphore should be if there are blocked waiting threads, but this behavior is conformant, given the wording of the specification.

SEM_INIT(3)

NAME

sem_init – initialize an unnamed semaphore

SYNOPSIS

```
#include <semaphore.h>

int
sem_init(sem_t *sem, int pshared, unsigned int value);
```

DESCRIPTION

The **sem_init()** function initializes the unnamed semaphore pointed to by *sem* to have the value *value*. A non-zero value for *pshared* specifies a shared semaphore that can be used by multiple processes, which this implementation is not capable of.

Following a successful call to **sem_init()**, *sem* can be used as an argument in subsequent calls to sem_wait(3), sem_trywait(3), sem_post(3), and sem_destroy(3). The *sem* argument is no longer valid after a successful call to sem_destroy(3).

RETURN VALUES

The **sem_init()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

The **sem_init()** function will fail if:

`[EINVAL]`	The value argument exceeds `SEM_VALUE_MAX`.
`[ENOSPC]`	Memory allocation error.
`[EPERM]`	Unable to initialize a shared semaphore.

SEE ALSO

sem_destroy(3), sem_getvalue(3), sem_post(3), sem_trywait(3), sem_wait(3)

STANDARDS

The **sem_init()** function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

This implementation does not support shared semaphores, and reports this fact by setting errno to `EPERM`. This is perhaps a stretch of the intention of POSIX, but is compliant, with the caveat that **sem_init()** always reports a permissions error when an attempt to create a shared semaphore is made.

SEM_OPEN(3)

NAME

sem_open, sem_close, sem_unlink – named semaphore operations

SYNOPSIS

```
#include <semaphore.h>

sem_t *
sem_open(const char *name, int oflag, ...);

int
sem_close(sem_t *sem);

int
sem_unlink(const char *name);
```

DESCRIPTION

The **sem_open()** function creates or opens the named semaphore specified by *name*. The returned semaphore may be used in subsequent calls to sem_getvalue(3), sem_wait(3), sem_trywait(3), sem_post(3), and sem_close()(.)

The following bits may be set in the *oflag* argument:

`O_CREAT` Create the semaphore if it does not already exist.

The third argument to the call to **sem_open()** must be of type `mode_t` and specifies the mode for the semaphore. Only the `S_IWUSR`, `S_IWGRP`, and `S_IWOTH` bits are examined; it is not possible to grant only "read" permission on a semaphore. The mode is modified according to the process's file creation mask; see umask(2).

The fourth argument must be an `unsigned int` and specifies the initial value for the semaphore, and must be no greater than `SEM_VALUE_MAX`.

`O_EXCL` Create the semaphore if it does not exist. If the semaphore already exists, **sem_open()** will fail. This flag is ignored unless `O_CREAT` is also specified.

The **sem_close()** function closes a named semaphore that was opened by a call to **sem_open()**.

The **sem_unlink()** function removes the semaphore named *name*. Resources allocated to the semaphore are only deallocated when all processes that have the semaphore open close it.

RETURN VALUES

If successful, the **sem_open()** function returns the address of the opened semaphore. If the same *name* argument is given to multiple calls to **sem_open()** by the same process without an intervening call to **sem_close()**, the same address is returned each time. If the semaphore cannot be opened, **sem_open()** returns `SEM_FAILED` and the global variable `ea`errno is set to indicate the error.

The **sem_close()** and **sem_unlink()** functions return the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

The **sem_open()** function will fail if:

`[EACCESS]` The semaphore exists and the permissions specified by *oflag* at the time it was created deny access to this process.

`[EACCESS]` The semaphore does not exist, but permission to create it is denied.

`[EEXIST]` `O_CREAT` and `O_EXCL` are set but the semaphore already exists.

`[EINTR]` The call was interrupted by a signal.

`[EINVAL]` The **sem_open()** operation is not supported for the given name.

`[EINVAL]` The *value* argument is greater than `SEM_VALUE_MAX`.

`[ENAMETOOLONG]` The *name* argument is too long.

`[ENFILE]` The system limit on semaphores has been reached.

`[ENOENT]` `O_CREAT` is set but the named semaphore does not exist.

`[ENOSPC]` There is not enough space to create the semaphore.

The **sem_close()** function will fail if:

`[EINVAL]` The *sem* argument is not a valid semaphore.

The **sem_unlink()** function will fail if:

`[EACCESS]` Permission is denied to unlink the semaphore.

`[ENAMETOOLONG]` The specified name is too long.

`[ENOENT]` The named semaphore does not exist.

SEE ALSO

close(2), open(2), umask(2), unlink(2), sem_getvalue(3), sem_post(3), sem_trywait(3), sem_wait(3)

STANDARDS

The **sem_open()**, **sem_close()**, and **sem_unlink()** functions conform to ISO/IEC 9945-1:1996 ("POSIX.1").

ISSUES

This implementation places strict requirements on the value of *name*: it must begin with a slash ('/'), contain no other slash characters, and be less than 14 characters in length not including the terminating null character.

This implementation creates a file in the `/tmp` directory, which may clash with other processes using the same name.

SEM_POST(3)

NAME

sem_post – increment (unlock) a semaphore

SYNOPSIS

```
#include <semaphore.h>

int
sem_post(sem_t *sem);
```

DESCRIPTION

The **sem_post()** function increments (unlocks) the semaphore pointed to by *sem*. If there are threads blocked on the semaphore when **sem_post()** is called, then a thread that has been blocked on the semaphore will be allowed to return from **sem_wait()**.

The **sem_post()** function is signal-reentrant and may be called within signal handlers.

RETURN VALUES

The **sem_post()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

The **sem_post()** function will fail if:

`[EINVAL]` The *sem* argument points to an invalid semaphore.

SEE ALSO

sem_getvalue(3), sem_trywait(3), sem_wait(3)

STANDARDS

The **sem_post()** function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

SEM_TIMEDWAIT(3)

NAME

sem_timedwait - lock a semaphore

SYNOPSIS

```
#include <semaphore.h>
#include <time.h>

int
sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
```

DESCRIPTION

The **sem_timedwait()** function locks the semaphore referenced by *sem*, as in the sem_wait(3) function. However, if the semaphore cannot be locked without waiting for another process or thread to unlock the semaphore by performing a sem_post(3) function, this wait will be terminated when the specified timeout expires.

The timeout will expire when the absolute time specified by *abs_timeout* passes, as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time of the call.

Note that the timeout is based on the `CLOCK_REALTIME` clock.

The validity of the *abs_timeout* is not checked if the semaphore can be locked immediately.

RETURN VALUES

The **sem_timedwait()** function returns zero if the calling process successfully performed the semaphore lock operation on the semaphore designated by *sem*. If the call was unsuccessful, the state of the semaphore is unchanged, and the function returns a value of -1 and sets the global variable `errno` to indicate the error.

ERRORS

The **sem_timedwait()** function will fail if:

[EINVAL] The *sem* argument does not refer to a valid semaphore, or the process or thread would have blocked, and the *abs_timeout* parameter specified a nanoseconds field value less than zero or greater than or equal to 1000 million.

[ETIMEDOUT] The semaphore could not be locked before the specified timeout expired.

[EINTR] A signal interrupted this function.

SEE ALSO

sem_post(3), sem_trywait(3), sem_wait(3)

STANDARDS

The **sem_timedwait()** function conforms to IEEE Std 1003.1-2004 ("POSIX.1").

SEM_WAIT(3)

NAME

sem_wait, sem_trywait – decrement (lock) a semaphore

SYNOPSIS

```
#include <semaphore.h>

int
sem_wait(sem_t *sem);

int
sem_trywait(sem_t *sem);
```

DESCRIPTION

The **sem_wait()** function decrements (locks) the semaphore pointed to by *sem*, but blocks if the value of *sem* is zero, until the value is non-zero and the value can be decremented.

The **sem_trywait()** function decrements (locks) the semaphore pointed to by *sem* only if the value is non-zero. Otherwise, the semaphore is not decremented and an error is returned.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

The **sem_wait()** and **sem_trywait()** functions will fail if:

`[EINVAL]` The *sem* argument points to an invalid semaphore.

Additionally, **sem_trywait()** will fail if:

`[EAGAIN]` The semaphore value was zero, and thus could not be decremented.

SEE ALSO

sem_getvalue(3), sem_post(3)

STANDARDS

The **sem_wait()** and **sem_trywait()** functions conform to ISO/IEC 9945-1:1996 ("POSIX.1").

SIGNAL(3)

NAME

signal – simplified software signal facilities

SYNOPSIS

```
#include <signal.h>

void (*signal(int sig, void (*func)(int)))(int);
```

or in Dignus's equivalent but easier to read typedef'd version:

```
typedef void (*sig_t) (int); sig_t
signal(int sig, sig_t func);
```

DESCRIPTION

This **signal()** facility is a simplified interface to the more general sigaction(2) facility.

Signals allow the manipulation of a process from outside its domain as well as allowing the process to manipulate itself or copies of itself (children). There are two general types of signals: those that cause termination of a process and those that do not. Signals which cause termination of a program might result from an irrecoverable error or might be the result of a user at a terminal typing the 'interrupt' character. Signals are used when a process is stopped because it wishes to access its control terminal while in the background. Signals are optionally generated when a process resumes after being stopped, when the status of child processes changes, or when input is ready at the control terminal. Most signals result in the termination of the process receiving them if no action is taken; some signals instead cause the process receiving them to be stopped, or are simply discarded if the process has not requested otherwise. Except for the SIGKILL and SIGSTOP signals, the signal() function allows for a signal to be caught, to be ignored, or to generate an interrupt. These signals are defined in the file `<signal.h>`:

```
No    Name         Default Action       Description
1     SIGHUP       terminate process    terminal line hangup
2     SIGINT       terminate process    interrupt program
3     SIGQUIT      create core image    quit program
4     SIGILL       create core image    illegal instruction
5     SIGTRAP      create core image    trace trap
```

6	SIGABRT	create core image	abort program (formerly SIGIOT)
7	SIGEMT	create core image	emulate instruction executed
8	SIGFPE	create core image	floating-point exception
9	SIGKILL	terminate process	kill program
10	SIGBUS	create core image	bus error
11	SIGSEGV	create core image	segmentation violation
12	SIGSYS	create core image	non-existent system call invoked
13	SIGPIPE	terminate process	write on a pipe with no reader
14	SIGALRM	terminate process	real-time timer expired
15	SIGTERM	terminate process	software termination signal
16	SIGURG	discard signal	urgent condition present on socket
17	SIGSTOP	stop process	stop (cannot be caught or ignored)
18	SIGTSTP	stop process	stop signal generated from keyboard
19	SIGCONT	discard signal	continue after stop
20	SIGCHLD	discard signal	child status has changed
21	SIGTTIN	stop process	background read attempted from control terminal
22	SIGTTOU	stop process	background write attempted to control terminal
23	SIGIO	discard signal	I/O is possible on a descriptor (see \manref{fcntl}{2})
24	SIGXCPU	terminate process	cpu time limit exceeded (see \manref{setrlimit}{2})
25	SIGXFSZ	terminate process	file size limit exceeded (see \manref{setrlimit}{2}))
26	SIGVTALRM	terminate process	virtual time alarm (see setitimer(2) - unavailable)
27	SIGPROF	terminate process	profiling timer alarm (see setitimer(2) - unavailable)
28	SIGWINCH	discard signal	Window size change
29	SIGINFO	discard signal	status request from keyboard
30	SIGUSR1	terminate process	User defined signal 1
31	SIGUSR2	terminate process	User defined signal 2
32	SIGTHR	terminate process	thread interrupt
33	SIGDANGER	terminate process	
34	SIGTHSTOP	terminate process	thread interrupt
35	SIGTHCONT	terminate process	thread interrupt
37	SIGTRACE	terminate process	
38	SIGDCE	terminate process	
39	SIGDUMP	terminate process	
40	SIGABND	terminate process	ABEND encountered
51	SIGPOLL	terminate process	
52	SIGIOERR	terminate process	

The *sig* argument specifies which signal was received. The *func* procedure allows a user to choose the action upon receipt of a signal. To set the default action of the signal to occur as listed above, *func* should be `SIG_DFL`. A `SIG_DFL` resets the default action. To ignore the signal *func* should be `SIG_IGN`. This will cause subsequent instances of the signal to be ignored and pending instances to be discarded. If `SIG_IGN` is not used, further occurrences of the signal are automatically blocked and *func* is called.

The handled signal is unblocked when the function returns and the process continues from where it left off when the signal occurred. Unlike previous signal facilities, the handler *func()* remains installed after a signal has been delivered.

For some system calls, if a signal is caught while the call is executing and the call is prematurely terminated, the call is automatically restarted. (The handler is installed using the `SA_RESTART` flag with sigaction(2).)

When a process which has installed signal handlers forks, the child process inherits the signals. All caught signals may be reset to their default action by a call to the execve(2) function; ignored signals remain ignored.

If a process explicitly specifies `SIG_IGN` as the action for the signal SIGCHLD, the system will not create zombie processes when children of the calling process exit. As a consequence, the system will discard the exit status from the child processes. If the calling process subsequently issues a call to wait(2) or equivalent, it will block until all of the calling process's children terminate, and then return a value of -1 with `errno` set to `ECHILD`.

See sigaction(2) for a list of functions that are considered safe for use in signal handlers.

RETURN VALUES

The previous action is returned on a successful call. Otherwise, `SIG_ERR` is returned and the global variable `errno` is set to indicate the error.

ERRORS

The **signal()** function will fail and no action will take place if one of the following occur:

errlist

`[EINVAL]` The *sig* argument is not a valid signal number.

`[EINVAL]` An attempt is made to ignore or supply a handler for `SIGKILL`, `SIGSTOP` or `SIGABND`.

IMPLEMENTATION

signal() is implemented using the sigaction(2) facility.

For more information about POSIX signals and how that interacts with the Dignus runtime environment, see sigaction(2).

SEE ALSO

kill(1), kill(2), ptrace(2), sigaction(2), sigaltstack(2), sigprocmask(2), sigsuspend(2), wait(2), fpsetmask(3), setjmp(3), siginterrupt(3), tty(4)

SIGSETOPS(3)

NAME

sigemptyset, sigfillset, sigaddset, sigdelset, sigismember – manipulate signal sets

SYNOPSIS

```
#include <signal.h>

int
sigemptyset(sigset_t *set);

int
sigfillset(sigset_t *set);

int
sigaddset(sigset_t *set, int signo);

int
sigdelset(sigset_t *set, int signo);

int
sigismember(const sigset_t *set, int signo);
```

DESCRIPTION

These functions manipulate signal sets stored in a `sigset_t`. Either **sigemptyset()** or **sigfillset()** must be called for every object of type `sigset_t` before any other use of the object.

The **sigemptyset()** function initializes a signal set to be empty.

The **sigfillset()** function initializes a signal set to contain all signals.

The **sigdelset()** function deletes the specified signal *signo* from the signal set.

The **sigismember()** function returns whether a specified signal *signo* is contained in the signal set.

RETURN VALUES

The **sigismember()** function returns 1 if the signal is a member of the set, 0 otherwise. The other functions return 0 upon success. A -1 return value indicates an error occurred and the global variable `errno` is set to indicate the reason.

ERRORS

These functions could fail if one of the following occurs:

`[EINVAL]` *signo* has an invalid value.

SEE ALSO

kill(2), sigaction(2), sigpending(2), sigprocmask(2), sigsuspend(2)

STANDARDS

These functions are defined by IEEE Std 1003.1-1988 (“POSIX.1”).

SETJMP(3)

NAME

sigsetjmp, siglongjmp, setjmp, longjmp, _setjmp, _longjmp - non-local jumps

SYNOPSIS

```
#include <setjmp.h>

int
sigsetjmp(sigjmp_buf env, int savemask);

void
siglongjmp(sigjmp_buf env, int val);

int
setjmp(jmp_buf env);

void
longjmp(jmp_buf env, int val);

int
_setjmp(jmp_buf env);

void
_longjmp(jmp_buf env, int val);
```

DESCRIPTION

The **sigsetjmp()**, **setjmp()**, and **_setjmp()** functions save their calling environment in *env*. Each of these functions returns 0.

The corresponding **longjmp()** functions restore the environment saved by their most recent respective invocations of the **setjmp()** function. They then return so that program execution continues as if the corresponding invocation of the **setjmp()** call had just returned the value specified by *val*, instead of 0.

Pairs of calls may be intermixed, i.e., both **sigsetjmp()** and **siglongjmp()** and **setjmp()** and **longjmp()** combinations may be used in the same program, however, individual calls may not, e.g. the *env* argument to **setjmp()** may not be passed to **siglongjmp()**.

The **longjmp()** routines may not be called after the routine which called the **setjmp()** routines returns.

All accessible objects have values as of the time **longjmp()** routine was called, except that the values of objects of automatic storage invocation duration that do not have the `volatile` type and have been changed between the **setjmp()** invocation and **longjmp()** call are indeterminate.

The **setjmp()**/**longjmp()** pairs save and restore the signal mask while **_setjmp()**/**_longjmp()** pairs save and restore only the register set and the stack. (See sigprocmask(2).)

The **sigsetjmp()**/**siglongjmp()** function pairs save and restore the signal mask if the argument *savemask* is non-zero, otherwise only the register set and the stack are saved.

SEE ALSO

sigaction(2), sigaltstack(2), signal(3)

STANDARDS

The **setjmp()** and **longjmp()** functions conform to ISO/IEC 9899:1990 ("ISO C90"). The **sigsetjmp()** and **siglongjmp()** functions conform to IEEE Std 1003.1-1988 ("POSIX.1").

SLEEP(3)

NAME

sleep – suspend process execution for an interval measured in seconds

SYNOPSIS

```
#include <unistd.h>

unsigned int
sleep(unsigned int seconds);
```

DESCRIPTION

The **sleep()** function suspends execution of the calling process until either *seconds* seconds have elapsed or a signal is delivered to the process and its action is to invoke a signal-catching function or to terminate the process. System activity may lengthen the sleep by an indeterminate amount.

In the USS/POSIX environment, this function is implemented directly via the BPX1SLP/BPX4SLP functions, in a batch or TSO environment this function is implemented with a **select()** call with a specified timeout value. This allows for use of the SIGALRM signal in USS/POSIX environments.

RETURN VALUES

In USS/POSIX environments, if the **sleep()** function returns because the requested time has elapsed, the value returned will be zero. If the **sleep()** function returns due to the delivery of a signal, the value returned will be the unslept amount (the requested time minus the time actually slept) in seconds.

In batch or TSO environments, **sleep()** always returns 0.

STANDARDS

The **sleep()** function conforms to ISO/IEC 9945-1:1990 (“POSIX.1”).

SYSCONF(3)

NAME

sysconf - get configurable system variables

SYNOPSIS

```
#include <unistd.h>

long
sysconf(int name);
```

DESCRIPTION

This interface is defined by IEEE Std 1003.1-1988 ("POSIX.1"). This implementation does not support all of the POSIX-defined function, but a limited subset.

The **sysconf()** function provides a method for applications to determine the current value of a configurable system limit or option variable. The *name* argument specifies the system variable to be queried. Symbolic constants for each name value are found in the include file `<unistd.h>`.

The available values in this implementation are:

`_SC_CHILD_MAX`	Returns zero for the Systems/C runtime.
`_SC_CLK_TCK`	The frequency of the statistics clock in ticks per second.
`_SC_OPEN_MAX`	The maximum number of open files.

RETURN VALUES

If the call to **sysconf()** is not successful, -1 is returned and `errno` is set appropriately. Otherwise, if the variable is associated with functionality that is not supported, -1 is returned and `errno` is not modified. Otherwise, the current variable value is returned.

ERRORS

The **sysconf()** function may fail and set `errno` if the library cannot determine a value. In addition, the following error may be reported:

`[EINVAL]`	The value of the *name* argument is invalid.

SEE ALSO

getdtablesize(2)

ISSUES

The Systems/C **sysconf()** implementation is not POSIX conforming.

TCGETPGRP(3)

NAME

tcgetpgrp - get foreground process group ID

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

pid_t
tcgetpgrp(int fd);
```

DESCRIPTION

The **tcgetpgrp()** function returns the value of the process group ID of the foreground process group associated with the terminal device. If there is no foreground process group, **tcgetpgrp()** returns an invalid process ID.

ERRORS

If an error occurs, **tcgetpgrp()** returns -1 and the global variable `errno` is set to indicate the error, as follows:

`[EBADF]` The *fd* argument is not a valid file descriptor.

`[ENOTTY]` The calling process does not have a controlling termi nal or the underlying terminal device represented by *fd* is not the controlling terminal.

SEE ALSO

setpgid(2), setsid(2), tcsetpgrp(3)

STANDARDS

The **tcgetpgrp()** function is expected to be compliant with the IEEE Std 1003.1-1988 ("POSIX.1") specification.

TCSENDBREAK(3)

NAME

tcsendbreak, tcdrain, tcflush, tcflow - line control functions

SYNOPSIS

```
#include <termios.h>

int
tcdrain(int fd);

int
tcflow(int fd, int action);

int
tcflush(int fd, int action);

int
tcsendbreak(int fd, int len);
```

DESCRIPTION

The **tcdrain()** function waits until all output written to the terminal referenced by *fd* has been transmitted to the terminal.

The **tcflow()** function suspends transmission of data to or the reception of data from the terminal referenced by *fd* depending on the value of *action*. The value of action must be one of the following:

`TCOFF`	Suspend output.
`TCOON`	Restart suspended output.
`TCIOFF`	Transmit a STOP character, which is intended to cause the terminal to stop transmitting data to the system.
`TCION`	Transmit a START character, which is intended to cause the terminal to start transmitting data to the system.

The **tcflush()** function discards any data written to the terminal referenced by *fd* which has not been transmitted to the terminal, or any data received from the terminal but not yet read, depending on the value of *action*. The value of action must be one of the following:

`TCIFLUSH` Flush data received but not read.

`TCOFLUSH` Flush data written but not transmitted.

`TCIOFLUSH` Flush both data received but not read and data written but not transmitted.

The **tcsendbreak()** function transmits a continuous stream of zero-valued bits for the specified *len* to the terminal referenced by *fd*.

RETURN VALUES

Upon successful completion, all of these functions return a value of zero.

ERRORS

If any error occurs, a value of -1 is returned and the global variable `errno` is set to indicate the error, as follows:

`[EBADF]` The *fd* argument is not a valid file descriptor.

`[EINVAL]` The *action* argument is not a proper value.

`[ENOTTY]` The file associated with *fd* is not a terminal.

`[EINTR]` A signal interrupted the **tcdrain()** function.

SEE ALSO

tcsetattr(3)

STANDARDS

The **tcsendbreak()**, **tcdrain()**, **tcflush()** and **tcflow()** functions are expected to be compliant with the IEEE Std 1003.1-1988 ("POSIX.1") specification.

TCSETATTR(3)

NAME

cfgetispeed, cfsetispeed, cfgetospeed, cfsetospeed, cfsetspeed, cfmakeraw, tcgetattr, tcsetattr - manipulating the termios structure

SYNOPSIS

```
#include <termios.h>

speed_t
cfgetispeed(const struct termios *t);

int
cfsetispeed(struct termios *t, speed_t speed);

speed_t
cfgetospeed(const struct termios *t);

int
cfsetospeed(struct termios *t, speed_t speed);

int
cfsetspeed(struct termios *t, speed_t speed);

void
cfmakeraw(struct termios *t);

int
tcgetattr(int fd, struct termios *t);

int
tcsetattr(int fd, int action, const struct termios *t);
```

DESCRIPTION

The **cfmakeraw()**, **tcgetattr()** and **tcsetattr()** functions are provided for getting and setting the `termios` structure.

The **cfgetispeed()**, **cfsetispeed()**, **cfgetospeed()**, **cfsetospeed()** and **cfsetspeed()** functions are provided for getting and setting the baud rate values in the `termios` structure. The effects of the functions on the terminal as described below do not become effective, nor are all errors detected, until the **tcsetattr()**

function is called. Certain values for baud rates set in the `termios` structure and passed to **tcsetattr()** have special meanings. These are discussed in the portion of the manual page that describes the **tcsetattr()** function.

GETTING AND SETTING THE BAUD RATE

The input and output baud rates are found in the termios structure. The unsigned integer `speed_t` is typedef'd in the include file `<termios.h>`. The value of the integer corresponds directly to the baud rate being represented, however, the following symbolic values are defined.

```
#define B0      0
#define B50     50
#define B75     75
#define B110    110
#define B134    134
#define B150    150
#define B200    200
#define B300    300
#define B600    600
#define B1200   1200
#define B1800   1800
#define B2400   2400
#define B4800   4800
#define B9600   9600
#define B19200  19200
#define B38400  38400
#ifndef _POSIX_SOURCE
#define EXTA    19200
#define EXTB    38400
#endif  /*_POSIX_SOURCE */
```

The **cfgetispeed()** function returns the input baud rate in the termios structure referenced by *tp*.

The **cfsetispeed()** function sets the input baud rate in the termios structure referenced by *tp* to *speed*.

The **cfgetospeed()** function returns the output baud rate in the termios structure referenced by *tp*.

The **cfsetospeed()** function sets the output baud rate in the termios structure referenced by *tp* to *speed*.

The **cfsetspeed()** function sets both the input and output baud rate in the termios structure referenced by *tp* to *speed*.

Upon successful completion, the functions **cfsetispeed()**, **cfsetospeed()**, and **cfsetspeed()** return a value of 0. Otherwise, a value of -1 is returned and the global variable `errno` is set to indicate the error.

GETTING AND SETTING THE TERMIOS STATE

This section describes the functions that are used to control the general terminal interface. Unless otherwise noted for a specific command, these functions are restricted from use by background processes. Attempts to perform these operations shall cause the process group to be sent a SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process is allowed to perform the operation and the SIGTTOU signal is not sent.

In all the functions, although *fd* is an open file descriptor, the functions affect the underlying terminal file, not just the open file description associated with the particular file descriptor.

The **cfmakeraw()** function sets the flags stored in the termios structure to a state disabling all input and output processing, giving a "raw I/O path". It should be noted that there is no function to reverse this effect. This is because there are a variety of processing options that could be re-enabled and the correct method is for an application to snapshot the current terminal state using the function **tcgetattr()**, setting raw mode with **cfmakeraw()** and the subsequent **tcsetattr()**, and then using another **tcsetattr()** with the saved state to revert to the previous terminal state.

The **tcgetattr()** function copies the parameters associated with the terminal referenced by *fd* in the termios structure referenced by *tp*. This function is allowed from a background process, however, the terminal attributes may be subsequently changed by a foreground process.

The **tcsetattr()** function sets the parameters associated with the terminal from the termios structure referenced by *tp*. The action field is created by or'ing the following values, as specified in the include file `<termios.h>`.

`TCSANOW`	The change occurs immediately
`TCSADRAIN`	The change occurs after all output written to *fd* has been transmitted to the terminal. This value of *action* should be used when changing parameters that affect output.
`TCSAFLUSH`	The change occurs after all output written to *fd* has been transmitted to the terminal. Additionally, any input that has been received but not read is discarded.
`TCSASOFT`	If this value is or'ed into the *action* value, the values of `c_cflag`, `c_ispeed`, and `c_ospeed`fields are ignored.

The 0 baud rate is used to terminate the connection. If 0 is specified as the output speed to the function **tcsetattr()**, modem control will no longer be asserted on the terminal, disconnecting the terminal.

If zero is specified as the input speed to the function **tcsetattr()**, the input baud rate will be set to the same value as that specified by the output baud rate.

If **tcsetattr()** is unable to make any of the requested changes, it returns -1 and sets `errno`. Otherwise, it makes all of the requested changes it can. If the specified input and output baud rates differ and are a combination that is not supported, neither baud rate is changed.

Upon successful completion, the functions **tcgetattr()** and **tcsetattr()** return a value of 0. Otherwise, they return -1 and the global variable `errno` is set to indicate the error, as follows:

`[EBADF]` The *fd* argument to **tcgetattr()** or **tcsetattr()** was not a valid file descriptor.

`[EINTR]` The **tcsetattr()** function was interuppted by a signal.

`[EINVAL]` The *action* argument to the **tcsetattr()** function was not valid, or an attempt was made to change an attribute represented in the termios structure to an unsupported value.

`[ENOTTY]` The file associated with the *fd* argument to **tcgetattr()** or **tcsetattr()** is not a terminal.

SEE ALSO

tcsendbreak(3)

STANDARDS

The **cfgetispeed()**, **cfsetispeed()**, **cfgetospeed()**, **cfsetospeed()**, **tcgetattr()** and **tcsetattr()** functions are expected to be compliant with the IEEE Std 1003.1-1988 ("POSIX.1") specification. The **cfmakeraw()** and **cfsetspeed()** functions, as well as the `TCSASOFT` option to the **tcsetattr()** function are extensions to the IEEE Std 1003.1-1988 ("POSIX.1") specification.

TCSETPGRP(3)

NAME

tcsetpgrp - set foreground process group ID

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

int
tcsetpgrp(int fd, pid_t pgrp_id);
```

DESCRIPTION

If the process has a controlling terminal, the **tcsetpgrp()** function sets the foreground process group ID associated with the terminal device to *pgrp_id*. The terminal device associated with *fd* must be the controlling terminal of the calling process and the controlling terminal must be currently associated with the session of the calling process. The value of *pgrp_id* must be the same as the process group ID of a process in the same session as the calling process.

RETURN VALUES

The **tcsetpgrp()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

The **tcsetpgrp()** function will fail if:

`[EBADF]` The *fd* argument is not a valid file descriptor.

`[EINVAL]` An invalid value of *pgrp_id* was specified.

`[ENOTTY]` The calling process does not have a controlling terminal, or the file represented by *fd* is not the controlling terminal, or the controlling terminal is no longer associated with the session of the calling process.

`[EPERM]` The *pgrp_id* argument does not match the process group ID of a process in the same session as the calling process.

SEE ALSO

setpgid(2), setsid(2), tcgetpgrp(3)

STANDARDS

The **tcsetpgrp()** function is expected to be compliant with the IEEE Std 1003.1-1988 ("POSIX.1") specification.

THRD_CREATE(3)

NAME

call_once, cnd_broadcast, cnd_destroy, cnd_init, cnd_signal, cnd_timedwait, cnd_wait, mtx_destroy, mtx_init, mtx_lock, mtx_timedlock, mtx_trylock, mtx_unlock, thrd_create, thrd_current, thrd_detach, thrd_equal, thrd_exit, thrd_join, thrd_sleep, thrd_yield, tss_create, tss_delete, tss_get, tss_set - C11 threads interface

SYNOPSIS

```
#include <threads.h>

void
call_once(once_flag *flag, void (*func)(void));

int
cnd_broadcast(cnd_t *cond);

void
cnd_destroy(cnd_t *cond);

int
cnd_init(cnd_t *cond);

int
cnd_signal(cnd_t *cond);

int
cnd_timedwait(cnd_t * restrict cond, mtx_t * restrict mtx,
         const struct timespec * restrict ts);

int
cnd_wait(cnd_t *cond, mtx_t *mtx);

void
mtx_destroy(mtx_t *mtx);

int
mtx_init(mtx_t *mtx, int type);

int
mtx_lock(mtx_t *mtx);

int
mtx_timedlock(mtx_t * restrict mtx, const struct timespec * restrict ts);
```

```
int
mtx_trylock(mtx_t *mtx);

int
mtx_unlock(mtx_t *mtx);

int
thrd_create(thrd_t *thr, int (*func)(void *), void *arg);

thrd_t
thrd_current(void);

int
thrd_detach(thrd_t thr);

int
thrd_equal(thrd_t thr0, thrd_t thr1);

_Noreturn void
thrd_exit(int res);

int
thrd_join(thrd_t thr, int *res);

int
thrd_sleep(const struct timespec *duration, struct timespec *remaining);

void
thrd_yield(void);

int
tss_create(tss_t *key, void (*dtor)(void *));

void
tss_delete(tss_t key);

void *
tss_get(tss_t key);

int
tss_set(tss_t key, void *val);
```

DESCRIPTION

As of ISO/IEC 9899:2011 ("ISO C11"), the C standard includes an API for writing multithreaded applications. Since POSIX.1 already includes a threading API that is used by virtually any multithreaded application, the interface provided by the C standard can be considered superfluous.

In this implementation, the threading interface is therefore implemented as a light-weight layer on top of existing interfaces. The functions to which these routines are mapped, are listed in the following table. Please refer to the documentation of the POSIX equivalent functions for more information.

Function	POSIX equivalent
call_once()	pthread_once(3)
cnd_broadcast()	pthread_cond_broadcast(3)
cnd_destroy()	pthread_cond_destroy(3)
cnd_init()	pthread_cond_init(3)
cnd_signal()	pthread_cond_signal(3)
cnd_timedwait()	pthread_cond_timedwait(3)
cnd_wait()	pthread_cond_wait(3)
mtx_destroy()	pthread_mutex_destroy(3)
mtx_init()	pthread_mutex_init(3)
mtx_lock()	pthread_mutex_lock(3)
mtx_timedlock()	pthread_mutex_timedlock(3)
mtx_trylock()	pthread_mutex_trylock(3)
mtx_unlock()	pthread_mutex_unlock(3)
thrd_create()	pthread_create(3)
thrd_current()	pthread_self(3)
thrd_detach()	pthread_detach(3)
thrd_equal()	pthread_equal(3)
thrd_exit()	pthread_exit(3)
thrd_join()	pthread_join(3)
thrd_sleep()	nanosleep(2)

thrd_yield() pthread_yield(3)

tss_create() pthread_key_create(3)

tss_delete() pthread_key_delete(3)

tss_get() pthread_getspecific(3)

tss_set() pthread_setspecific(3)

DIFFERENCES WITH POSIX EQUIVALENTS

The **thrd_exit()** function returns an integer value to the thread calling **thrd_join()**, whereas the **pthread_exit()** function uses a pointer.

The mutex created by **mtx_init()** can be of type `mtx_plain` or `mtx_timed` to distinguish between a mutex that supports **mtx_timedlock()**. This type can be or'd with **mtx_recursive** to create a mutex that allows recursive acquisition. These properties are normally set using **pthread_mutex_init()**'s *attr* parameter.

RETURN VALUES

If successful, the **cnd_broadcast()**, **cnd_init()**, **cnd_signal()**, **cnd_timedwait()**, **cnd_wait()**, **mtx_init()**, **mtx_lock()**, **mtx_timedlock()**, **mtx_trylock()**, **mtx_unlock()**, **thrd_create()**, **thrd_detach()**, **thrd_equal()**, **thrd_join()**, **thrd_sleep()**, **tss_create()** and **tss_set()** functions return `thrd_success`. Otherwise an error code will be returned to indicate the error.

The **thrd_current()** function returns the thread ID of the calling thread.

The **tss_get()** function returns the thread-specific data value associated with the given key. If no thread-specific data value is associated with key, then the value `NULL` is returned.

ERRORS

The **cnd_init()** and **thrd_create()** functions will fail if:

`thrd_nonmem` The system has insufficient memory.

The **cnd_timedwait()** and **mtx_timedlock()** functions will fail if:

`thrd_timedout` The system time has reached or exceeded the time specified in *ts* before the operation could be completed.

The **mtx_trylock()** function will fail if:

`thrd_busy` The mutex is already locked.

In all other cases, these functions may fail by returning general error code `thrd_error`.

SEE ALSO

nanosleep(2), pthread(3)

STANDARDS

These functions are expected to conform to ISO/IEC 9899:2011 ("ISO C11").

TIME(3)

NAME

time - get time of day

SYNOPSIS

```
#include <time.h>

time_t
time(time_t *tloc)
```

DESCRIPTION

The **time()** function returns the value of time in seconds since 0 hours, 0 minutes, 0 seconds, January 1, 1970, Coordinated Universal Time.

A copy of the time value may be saved to the area indicated by the pointer tloc. If tloc is a `NULL` pointer, no value is stored.

Upon successful completion, **time()** returns the value of time. There is no error value as a value of -1 seconds would be the last second of the year 1969.

SEE ALSO

gettimeofday(2), ctime(3)

TIMES(3)

NAME

times - process times

SYNOPSIS

```
#include <sys/times.h>

clock_t
times(struct tms *tp);
```

DESCRIPTION

This interface is obsoleted by getrusage(2) and gettimeofday(2).

The **times()** function returns the value of time in `CLK_TCK`'s of a second since 0 hours, 0 minutes, 0 seconds, January 1, 1970, Coordinated Universal Time.

It also fills in the structure pointed to by *tp* with time-accounting information.

The `tms` structure is defined as follows:

```
struct tms {
        clock_t tms_utime;
        clock_t tms_stime;
        clock_t tms_cutime;
        clock_t tms_cstime;
};
```

The elements of this structure are defined as follows:

`tms_utime` The CPU time charged for the execution of user instructions.

`tms_stime` The CPU time charged for execution by the system on behalf of the process.

`tms_cutime` The sum of the `tms_utimes` and `tms_cutimes` of the child processes.

`tms_cstime` The sum of the `tms_stimes` and `tms_cstimes` of the child processes.

All times are in `CLK_TCK`'s of a second.

If an error occurs, **times()** returns the value `((clock_t)-1)`, and sets `errno` to indicate the error.

ERRORS

The **times()** function may fail and set the global variable `errno` for any of the errors specified for the library routines getrusage(2) and gettimeofday(2).

SEE ALSO

getrusage(2), gettimeofday(2), wait(2)

STANDARDS

The **times()** function conforms to IEEE Std 1003.1-1988 ("POSIX.1") as closely as the host system allows.

TIMEZONE(3)

NAME

timezone - return the timezone abbreviation

SYNOPSIS

```
char *
timezone(int zone, int dst);
```

DESCRIPTION

This interface is for compatibility only; it is impossible to reliably map `timezone`'s arguments to a time zone abbreviation. See ctime(3).

The **timezone()** function returns a pointer to a time zone abbreviation for the specified *zone* and *dst* values. *Zone* is the number of minutes west of GMT and *dst* is non-zero if daylight savings time is in effect.

SEE ALSO

ctime(3)

TPUT(3)

NAME

__tput - issue the TPUT macro

SYNOPSIS

```
#include <machine/tput.h>
void
__tput(int len, char *buffer)
```

DESCRIPTION

The **__tput()** function invokes the z/OS TPUT macro, passing the given length *len* and buffer address *buffer*.

Consult the IBM documentation for the TPUT macro for more information.

TRACEBACK(3)

NAME

__traceback - provide a function traceback

SYNOPSIS

```
#include <stdio.h>
#include <machine/trcback.h>

void
__traceback(FILE *f);

void
__tbfrom(void *stack_ptr, void message(char *, void *), void *user_data);
```

DESCRIPTION

The **__traceback()** function provides a function-level traceback of the call stack from the calling function.

__traceback() writes the traceback on the file descriptor specified by *f*. The traceback information is kept in the Systems/C pre-prologue area. The **__traceback()** function walks the stack frame backwards, printing the name found in the pre-prologue area.

The **__tbfrom()** function provides a user-controlled mechanism to access the traceback. The **__tbfrom()** function will invoke the *message* function for each line of traceback information generated. The first parameter to the *message* function will be a `NUL`-terminated character string. The 2nd parameter will be the value of *user_data* passed into **__tbfrom()**. The *stack_ptr* parameter to **__tbfrom()** is the stack pointer where the traceback should begin. Typically this is the current register 13.

Note that invoking **__traceback()** or **__tbfrom()** with a `SIGSEGV` or `SIGABND` signal handler can be dangerous as the stack may be corrupted. **__traceback()** and **__tbfrom()** require a reasonable stack to "walk-back" and retrieve the function names. A corrupt stack will cause these functions to possibly dereference invalid memory, or have other issues. Therefor, use of these functions within a corrupted stack environment may cause further `SIGSEGV` or other events.

For example, the **__traceback()** function can be implemented as:

```
#include <machine/trcback.h>
```

```
#include <stdio.h>
#include <stdlib.h>

/*
* one_line()
* Print one line to the the FILE pointer
* passed in a user-data
**/
static void
one_line(char *mess, void *user)
{
   FILE *fp;

   fp = (FILE *)user;
   fprintf(fp, "%s\n", mess);
   fflush(fp);
}

void
traceback(FILE *f)
{
        __register(13) void *r13;  /* current stack pointer */

        __tbfrom(r13, one_line, f);
}
```

SEE ALSO

funopen(3), fopen(3)

TSEARCH(3)

NAME

tsearch, tfind, tdelete, twalk – manipulate binary search trees

SYNOPSIS

```
#include <search.h>

void *
tdelete(const void *key, void **rootp,
    int (*compar) (const void *, const void *));

void *
tfind(const void *key, void **rootp,
    int (*compar) (const void *, const void *));

void *
tsearch(const void *key, void **rootp,
    int (*compar) (const void *, const void *));

void
twalk(const void *root, void (*compar) (const void *, VISIT, int));
```

DESCRIPTION

The **tdelete()**, **tfind()**, **tsearch()**, and **twalk()** functions manage binary search trees based on algorithms T and D from Knuth (6.2.2). The comparison function passed in by the user has the same style of return values as strcmp(3).

tfind() searches for the datum matched by the argument *key* in the binary tree rooted at *rootp*, returning a pointer to the datum if it is found and `NULL` if it is not.

tsearch() is identical to **tfind()** except that if no match is found, *key* is inserted into the tree and a pointer to it is returned. If *rootp* points to a `NULL` value a new binary search tree is created.

tdelete() deletes a node from the specified binary search tree and returns a pointer to the parent of the node to be deleted. It takes the same arguments as **tfind()** and **tsearch()**. If the node to be deleted is the root of the binary search tree, *rootp* will be adjusted.

twalk() walks the binary search tree rooted in *root* and calls the function *action* on each node. *Action* is called with three arguments: a pointer to the current node,

a value from the enum `typedef enum { preorder, postorder, endorder, leaf } VISIT;` specifying the traversal type, and a node level (where level zero is the root of the tree).

SEE ALSO

bsearch(3), hsearch(3), lsearch(3)

RETURN VALUES

The **tsearch()** function returns `NULL` if allocation of a new node fails (usually due to a lack of free memory).

tfind(), **tsearch()**, and **tdelete()** functions return `NULL` if *rootp* is `NULL` or the datum cannot be found.

The **twalk()** function returns no value.

TTYNAME(3)

NAME

ttyname - get name of associated terminal (tty) from file descriptor

SYNOPSIS

```
#include <unistd.h>

char *
ttyname(int fd);
```

DESCRIPTION

The **ttyname()** function operates on the system file descriptors for terminal type devices. These descriptors are not related to the standard I/O `FILE` typedef, but refer to the special device files found in `/dev` and named `/dev/tty`*xx*

The **ttyname()** function gets the related device name of a file descriptor for which **isatty()** is true, and the file descriptor is associated with an `//HFS:`-style file. **isatty()** can return true for non-`//HFS:` files when the `DCB` indicates that the file is associated with a `TSO` terminal.

RETURN VALUES

The **ttyname()** function returns the null terminated name if the device is found and **isatty()** is true, and the file descriptor is an `//HFS:`-style file; otherwise a `NULL` pointer is returned.

ISSUES

The **ttyname()** function leaves its result in an internal static object and returns a pointer to that object. Subsequent calls to **ttyname()** will modify the same object.

UCONTEXT(3)

NAME

ucontext – user thread context

SYNOPSIS

```
#include <ucontext.h>
```

DESCRIPTION

The *ucontext_t* type is a structure type suitable for holding the context for a user thread of execution. A thread's context includes its stack, saved registers, and list of blocked signals.

The *ucontext_t* structure contains at least these fields:

`ucontext_t *uc_link` context to assume when this one returns

`sigset_t uc_sigmask` signals being blocked

`stack_t uc_stack` stack area

`mcontext_t uc_mcontext` saved registers

The *uc_link* field points to the context to resume when this context's entry point function returns. If *uc_link* is equal to `NULL`, then the program exits when this context returns.

The *uc_mcontext* field is machine-dependent and should be treated as opaque by portable applications.

The following functions are defined to manipulate *ucontext_t* structures:

```
int getcontext(ucontext_t *);
int setcontext(const ucontext_t *);
void makecontext(ucontext_t *, void (*)(void), int, ...);
int swapcontext(ucontext_t *, const ucontext_t *);
```

SEE ALSO

sigaltstack(2), getcontext(3), makecontext(3)

UNAME(3)

NAME

uname - get system identification

SYNOPSIS

```
#include <sys/utsname.h>

int
uname(struct utsname *name);
```

DESCRIPTION

The **uname()** function stores nul-terminated strings of information identifying the current system into the structure referenced by *name*.

The `utsname` struction is defined in the `<sys/utsname.h>` header file, and contains the following members:

`sysname`	Name of the operating system implementation.
`nodename`	Network name of this machine.
`release`	Release level of the operating system.
`version`	Version level of the operating system.
`machine`	Machine hardware platform.

RETURN VALUES

The **unname()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

The only error value possible from **uname()** is `ENOSYS`, which can occur if OpenEdition services are not available.

STANDARDS

The **uname()** function conforms to IEEE Std 1003.1-1988 ("POSIX.1") as closely as the host system allows.

USLEEP(3)

NAME

usleep – suspend process execution for an interval measured in microseconds

SYNOPSIS

```
#include <unistd.h>

int
usleep(useconds_t microseconds);
```

DESCRIPTION

The **usleep()** function suspends execution of the calling process until either *microseconds* microseconds have elapsed or a signal is delivered to the process and its action is to invoke a signal-catching function or to terminate the process. System activity may lengthen the sleep by an indeterminate amount.

This function is implemented using nanosleep(2) by pausing for *microseconds* microseconds or until a signal occurs. Consequently, in this implementation, sleeping has no effect on the state of process timers, and there is no special handling for `SIGALRM`.

RETURN VALUES

The **usleep()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

The usleep() function will fail if:

`[EINTR]` A signal was delivered to the process and its action was to invoke a signal-catching function.

SEE ALSO

nanosleep(2), sleep(3)

UTIME(3)

NAME

utime - set //HFS:-style file times

SYNOPSIS

```
#include <sys/types.h>
#include <utime.h>

int
utime(const char *file, const struct utimbuf *timep);
```

DESCRIPTION

This interface is obsoleted by utimes(2).

The **utime()** function sets the access and modification times of the named file from the structures in the argument array *timep*.

If the times are specified (the *timep* argument is non-`NULL`) the caller must be the owner of the file or be the super-user.

If the times are not specified (the *timep* argument is NULL) the caller must be the owner of the file, have permission to write the file, or be the super-user.

ERRORS

The **utime()** function may fail and set `errno` for any of the errors specified for the library function utimes(2).

SEE ALSO

stat(2), utimes(2)

STANDARDS

The **utime()** function conforms to IEEE Std 1003.1-1988 (“POSIX.1”).

WORDEXP(3)

NAME

wordexp - perform shell-style word expansions

SYNOPSIS

```
#include <wordexp.h>

int
wordexp(const char * restrict words, wordexp_t * restrict we, int flags);

void
wordfree(wordexp_t *we);
```

DESCRIPTION

The **wordexp()** function performs shell-style word expansion on words and places the list of words into the `we_wordv` member of *we*, and the number of words into `we_wordc`.

The flags argument is the bitwise inclusive OR of any of the following constants:

`WRDE_APPEND`	Append the words to those generated by a previous call to **wordexp()**.
`WRDE_DOOFFS`	As many `NULL` pointers as are specified by the `we_offs` member of we are added to the front of `we_wordv`.
`WRDE_NOCMD`	Disallow command substitution in words. See the note in ISSUES before using this.
`WRDE_REUSE`	The we argument was passed to a previous successful call to **wordexp()** but has not been passed to **wordfree()**. The implementation may reuse the space allocated to it.
`WRDE_SHOWERR`	Do not redirect shell error messages to `/dev/null`.
`WRDE_UNDEF`	Report error on an attempt to expand an undefined shell variable.

The `wordexp_t` structure is defined in `<wordexp.h>` as:

```
typedef struct {
   size_t  we_wordc;       /* count of words matched */
   char    **we_wordv;     /* pointer to list of words */
   size_t  we_offs;        /* slots to reserve in we_wordv */
} wordexp_t;
```

The **wordfree()** function frees the memory allocated by **wordexp()**.

IMPLEMENTATION NOTES

The **wordexp()** function is implemented as a wrapper around an invocation of the POSIX shell.

RETURN VALUES

The **wordexp()** function returns zero if successful, otherwise it returns one of the following error codes:

WRDE_BADCHAR	The words argument contains one of the following unquoted characters: <newline>, '|', '&', ';', '<', '>', '(', ')', '{', '}'.
WRDE_BADVAL	An attempt was made to expand an undefined shell variable and WRDE_UNDEF is set in flags.
WRDE_CMDSUB	An attempt was made to use command substitution and WRDE_NOCMD is set in flags.
WRDE_NOSPACE	Not enough memory to store the result.
WRDE_NOSYS	Functionality not supported on this system, errno will also be set to ENOSYS enough memory to store the result.
WRDE_SYNTAX	Shell syntax error in words.

The **wordfree()** function returns no value.

ENVIRONMENT

IFS	Field separator.

EXAMPLES

Invoke the editor on all .c files in the current directory and /etc/motd (error checking omitted):

```
wordexp_t we;

wordexp("${EDITOR:-vi} *.c /etc/motd", &we, 0);
execvp(we.we_wordv[0], we.we_wordv);
```

DIAGNOSTICS

Diagnostic messages from the shell are written to the standard error output if `WRDE_SHOWERR` is set in flags.

SEE ALSO

fnmatch(3), glob(3), popen(3), system(3)

STANDARDS

The **wordexp()** and **wordfree()** functions conform to IEEE Std 1003.1-2001 ("POSIX.1").

ISSUES

Do not pass untrusted user data to **wordexp()**, regardless of whether the `WRDE_NOCMD` flag is set. The **wordexp()** function attempts to detect input that would cause commands to be executed before passing it to the shell but it does not use the same parser so it may be fooled.

The current **wordexp()** implementation does not recognize multibyte characters, since the shell (which it invokes to perform expansions) does not.

WTO(3)

NAME

__wto - issue the WTO macro

SYNOPSIS

```
#include <machine/wto.h>
void
__wto(int len, char *buffer)
```

DESCRIPTION

The **__wto()** function invokes the z/OS WTO macro, passing the given length *len* and buffer address *buffer*.

Consult the IBM documentation for the WTO macro for more information.

Locale Library

The locale library provides functions for manipulating character values in a locale specific fashion. It provides functions that define the locale, test various character values for belong to specific classes of characters and formatting various items based on the locale setting.

BTOWC(3)

NAME

btowc, wctob - convert between wide and single-byte characters

SYNOPSIS

```
#include <wchar.h>

wint_t
btowc(int c);

int
wctob(wint_t c);
```

DESCRIPTION

The **btowc()** function converts a single-byte character into a correspond- ing wide character. If the character is `EOF` or not valid in the initial shift state, **btowc()** returns `WEOF`.

The **wctob()** function converts a wide character into a corresponding sin- gle-byte character. If the wide character is `WEOF` or not able to be rep- resented as a single byte in the initial shift state, **wctob()** returns `WEOF`.

SEE ALSO

mbrtowc(3), multibyte(3), wcrtomb(3)

STANDARDS

The **btowc()** and **wctob()** functions conform to IEEE Std 1003.1-2001 (“POSIX.1”).

CTYPE(3)

NAME

isalnum, isalpha, isascii, isblank, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit, toascii tolower, toupper, - character classification macros

SYNOPSIS

```
#include <ctype.h>

int
isalnum(int c)

int
isalpha(int c)

int
isascii(int c)

int
iscntrl(int c)

int
isdigit(int c)

int
isgraph(int c)

int
islower(int c)

int
isprint(int c)

int
ispunct(int c)

int
isspace(int c)

int
isupper(int c)

int
isxdigit(int c)
```

```
int
toascii(int c)

int
tolower(int c)

int
toupper(int c)
```

DESCRIPTION

The above functions perform character tests and conversions on the integer *c*. They are available as macros, defined in the include file ¡ctype.h¿, or as true functions in the C library. See the specific manual pages for more information.

SEE ALSO

isalnum(3), isalpha(3), isascii(3), isblank(3), iscntrl(3), isdigit(3), isgraph(3), islower(3), isprint(3), ispunct(3), isspace(3), isupper(3), isxdigit(3), toascii(3), tolower(3), toupper(3)

STANDARDS

These functions, except for **isblank()**, **toupper()**, **tolower()** and **toascii()**, conform to ISO/IEC 9899:1990 ("ISO C90").

ISALNUM(3)

NAME

isalnum - alphanumeric character test

SYNOPSIS

```
#include <ctype.h>

int
isalnum(int c)
```

DESCRIPTION

The **isalnum()** function tests for any character for which isalpha(3) or isdigit(3) is true.

RETURN VALUES

The **isalnum()** function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

ctype(3), isalpha(3), isdigit(3)

STANDARDS

The **isalnum()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

ISALPHA(3)

NAME

isalpha - alphabetic character test

SYNOPSIS

```
#include <ctype.h>

int
isalpha(int c)
```

DESCRIPTION

The **isalpha()** function tests for any character for which isupper(3) or slower(3) is true.

RETURN VALUES

The **isalpha()** function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

ctype(3), islower(3), isupper(3)

STANDARDS

The **isalpha()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

ISASCII(3)

NAME

isascii - test for ASCII character

SYNOPSIS

```
#include <ctype.h>

int
isascii(int c)
```

DESCRIPTION

The **isascii()** function tests for an ASCII character, based on the EBCDIC character set. That is, isascii returns a non-zero value for any EBCDIC character which, when converted to ASCII, would have a value less than or equal to 0177.

The conversion from EBCDIC to ASCII follows the C compiler's conversion table.

SEE ALSO

ctype(3),

STANDARDS

The **isascii()** function conforms to ISO/IEC 9899:1990 ("ISO C90").

ISBLANK(3)

NAME

isblank - space or tab character test

SYNOPSIS

```
#include <ctype.h>

int
isblank(int c)
```

DESCRIPTION

The **isblank()** function tests for a space or tab character.

RETURN VALUES

The **isblank()** function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

ctype(3)

ISCNTRL(3)

NAME

iscntrl - control character test

SYNOPSIS

```
#include <ctype.h>

int
iscntrl(int c)
```

DESCRIPTION

The **iscntrl()** function tests for any control character.

RETURN VALUES

The **iscntrl()** function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

ctype(3),

STANDARDS

The **iscntrl()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

ISDIGIT(3)

NAME

isdigit - decimal-digit character test

SYNOPSIS

```
#include <ctype.h>

int
isdigit(int c)
```

DESCRIPTION

The **isdigit()** function tests for any decimal-digit character.

RETURN VALUES

The **isdigit()** function returns zero if the character tests false and returns non-zero If the character tests true.

SEE ALSO

ctype(3),

STANDARDS

The **isdigit()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

ISGRAPH(3)

NAME

isgraph - printing character test (space character exclusive)

SYNOPSIS

```
#include <ctype.h>

int
isgraph(int c)
```

DESCRIPTION

The **isgraph()** function tests for any printing character except space.

RETURN VALUES

The **isgraph()** function returns zero if the character tests false and returns non-zero If the character tests true.

SEE ALSO

ctype(3),

STANDARDS

The **isgraph()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

ISLOWER(3)

NAME

islower - lower-case character test

SYNOPSIS

```
#include <ctype.h>

int
islower(int c)
```

DESCRIPTION

The **islower()** function tests for any lower-case letters.

RETURN VALUES

The **islower()** function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

ctype(3),

STANDARDS

The **islower()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

ISPRINT(3)

NAME

isprint - printing character test (space character inclusive)

SYNOPSIS

```
#include <ctype.h>

int
isprint(int c)
```

DESCRIPTION

The **isprint()** function tests for any printing character including space (' ').

RETURN VALUES

The **isprint()** function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

ctype(3),

STANDARDS

The **isprint()** function conforms to ISO/IEC 9899:1990 ("ISO C90").

ISPUNCT(3)

NAME

ispunct - punctuation character test

SYNOPSIS

```
#include <ctype.h>

int
ispunct(int c)
```

DESCRIPTION

The **ispunct()** function tests for any printing character except for space (' ') or a character for which isalnum(3) is true.

RETURN VALUES

The **ispunct()** function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

ctype(3),

STANDARDS

The **ispunct()** function conforms to ISO/IEC 9899:1990 ("ISO C90").

ISSPACE(3)

NAME

isspace - white-space character test

SYNOPSIS

```
#include <ctype.h>

int
isspace(int c)
```

DESCRIPTION

The **isspace()** function tests for the standard white-space characters.

RETURN VALUES

The **isspace()** function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

ctype(3),

STANDARDS

The **isspace()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

ISUPPER(3)

NAME

isupper - upper-case character test

SYNOPSIS

```
#include <ctype.h>

int
isupper(int c)
```

DESCRIPTION

The **isupper()** function tests for any upper-case letter.

RETURN VALUES

The **isupper()** function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

ctype(3)

STANDARDS

The **isupper()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

ISWALNUM(3)

NAME

iswalnum, iswalpha, iswascii, iswblank, iswcntrl, iswdigit, iswgraph, iswhexnumber, iswideogram, iswlower, iswnumber, iswphonogram, iswprint, iswpunct, iswrune, iswspace, iswspecial, iswupper, iswxdigit - wide character classification utilities

SYNOPSIS

```
#include <wctype.h>
int
iswalnum(wint_t wc);

int
iswalpha(wint_t wc);

int
iswascii(wint_t wc);

int
iswblank(wint_t wc);

int
iswcntrl(wint_t wc);

int
iswdigit(wint_t wc);

int
iswgraph(wint_t wc);

int
iswhexnumber(wint_t wc);

int
iswideogram(wint_t wc);

int
iswlower(wint_t wc);

int
iswnumber(wint_t wc);

int
iswphonogram(wint_t wc);
```

```
int
iswprint(wint_t wc);

int
iswpunct(wint_t wc);

int
iswrune(wint_t wc);

int
iswspace(wint_t wc);

int
iswspecial(wint_t wc);

int
iswupper(wint_t wc);

int
iswxdigit(wint_t wc);
```

DESCRIPTION

The above functions are character classification utility functions, for use with wide character (`wchar_t` or `wint_t`). See the description for the similarly-named single byte classfunctions (e.g. isalnum(3)) for details.

RETURN VALUES

The functions return zero if the character tests false and return non-zero if the character tests true.

SEE ALSO

isalnum(3), isalpha(3), isascii(3), isblank(3), iscntrl(3), isdigit(3), isgraph(3), ishexnumber(3), isideogram(3), islower(3), isnumber(3), isphonogram(3), isprint(3), ispunct(3), isrune(3), isspace(3), isspecial(3), isupper(3), isxdigit(3), wctype(3)

STANDARDS

These functions functions conform to ISO/IEC 9899:1999 (“ISO C99”), except **iswascii()**, **iswhexnumber()**, **iswideogram()**, **iswnumber()**, **iswphono-**

gram(), **iswrune()** and **iswspecial()**, which are Systems/C extensions.

CAVEATS

The result of these functions is undefined unless the argument is `WEOF` or a valid `wchar_t` value for the current locale.

ISXDIGIT(3)

NAME

isxdigit - hexadecimal-digit character test

SYNOPSIS

```
#include <ctype.h>

int
isxdigit(int c)
```

DESCRIPTION

The **isxdigit()** function tests for any hexadecimal-digit character.

RETURN VALUES

The **isxdigit()** function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

ctype(3)

STANDARDS

The **isxdigit()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

MBLEN(3)

NAME

mblen - get number of bytes in a character

SYNOPSIS

```
#include <stdlib.h>

int
mblen(const char *mbchar, size_t nbytes);
```

DESCRIPTION

The **mblen()** function computes the length in bytes of a multibyte character *mbchar* according to the current conversion state. Up to *nbytes* bytes are examined.

A call with a null *mbchar* pointer returns nonzero if the current locale requires shift states, zero otherwise; if shift states are required, the shift state is reset to the initial state.

RETURN VALUES

If *mbchar* is `NULL`, the *mblen()* function returns nonzero if shift states are supported, zero otherwise.

Otherwise, if *mbchar* is not a null pointer, **mblen()** either returns 0 if *mbchar* represents the null wide character, or returns the number of bytes processed in *mbchar*, or returns -1 if no multibyte character could be recognized or converted. In this case, **mblen()**'s internal conversion state is undefined.

ERRORS

The **mblen()** function will fail if:

`EILSEQ`	An invalid multibyte sequence was detected.
`EINVAL`	The internal conversion state is not valid.

SEE ALSO

mbrlen(3), mbtowc(3), multibyte(3)

STANDARDS

The **mblen()** function conforms to ISO/IEC 9899:1999 (“ISO C99”).

MBRLEN(3)

NAME

mbrlen - get number of bytes in a character (restartable)

SYNOPSIS

```
#include <wchar.h>

size_t
mbrlen(const char * restrict s, size_t n, mbstate_t * restrict ps);
```

DESCRIPTION

The **mbrlen()** function inspects at most *n* bytes pointed to by *s* to determine the number of bytes needed to complete the next multibyte character.

The `mbstate_t` argument, *ps*, is used to keep track of the shift state. If it is `NULL`, **mbrlen()** uses an internal, static `mbstate_t` object, which is initialized to the initial conversion state at program startup.

It is equivalent to:

```
mbrtowc(NULL, s, n, ps);
```

Except that when *ps* is a `NULL` pointer, **mbrlen()** uses its own static, internal `mbstate_t` object to keep track of the shift state.

RETURN VALUES

The **mbrlen()** functions returns:

`0`	The next *n* or fewer bytes represent the null wide character (`L'0'`). item[`>0`] The next *n* or fewer bytes represent a valid character, **mbrlen()** returns the number of bytes used to complete the multibyte character.
`EFAULT`	*locale* was `NULL`.
(`size_t`)-2	The next *n* contribute to, but do not complete, a valid multibyte character sequence, and all *n* bytes have been processed.
(`size_t`)-1	An encoding error has occurred. The next *n* or fewer bytes do not contribute to a valid multibyte character.

EXAMPLES

A function that calculates the number of characters in a multibyte character string:

```
size_t
nchars(const char *s)
{
   size_t charlen, chars;
   mbstate_t mbs;

   chars = 0;
   memset(&mbs, 0, sizeof(mbs));
   while ((charlen = mbrlen(s, MB_CUR_MAX, &mbs)) != 0 &&
           charlen != (size_t)-1 && charlen != (size_t)-2) {
              s += charlen;
              chars++;
   }

   return (chars);
}
```

ERRORS

The **mbrlen()** function will fail if:

`EILSEQ`	An invalid multibyte sequence was detected.
`EINVAL`	The conversion state is invalid.

SEE ALSO

mblen(3), mbrtowc(3), multibyte(3)

STANDARDS

The **mbrlen()** function conforms to ISO/IEC 9899:1999 (“ISO C99”).

MBRTOWC(3)

NAME

mbrtowc - convert a character to a wide-character code (restartable)

SYNOPSIS

```
#include <wchar.h>

size_t
mbrtowc(wchar_t * restrict pwc, const char * restrict s, size_t n,
    mbstate_t * restrict ps);
```

DESCRIPTION

The **mbrtowc()** function inspects at most *n* bytes pointed to by *s* to determine the number of bytes needed to complete the next multibyte character. If a character can be completed, and *pwc* is not `NULL`, the wide character which is represented by *s* is stored in the `wchar_t` it points to.

If *s* is `NULL`, **mbrtowc()** behaves as if *pwc* was `NULL`, *s* was an empty string (`""`) and *n* was `1`.

The `mbstate_t` argument, *ps*, is used to keep track of the shift state. If it is `NULL`, **mbrtowc()** uses an internal, static `mbstate_t` object, which is initialized to the initial conversion state at program startup.

RETURN VALUES

The **mbrtowc()** function returns:

`0`	The next *n* or fewer bytes represent the null wide character (`L'0'`).
`>0`	The next *n* or fewer bytes represent a valid character, **mbrtowc()** returns the number of bytes used to complete the multibyte character.
`(size_t)-2`	The next *n* contribute to, but do not complete, a valid multibyte character sequence, and all *n* bytes have been processed.
`(size_t)-1`	An encoding error has occurred. The next *n* or fewer bytes do not contribute to a valid multibyte character.

ERRORS

The **mbrtowc()** function will fail if:

`EILSEQ`	An invalid multibyte sequence was detected.
`EINVAL`	The conversion state is invalid.

SEE ALSO

mbtowc(3), multibyte(3), setlocale(3), wcrtomb(3)

STANDARDS

The **mbrtowc()** function conforms to ISO/IEC 9899:1999 ("ISO C99").

MBSINIT(3)

NAME

mbsinit - determine conversion object status

SYNOPSIS

```
#include <wchar.h>

int
mbsinit(const mbstate_t *ps);
```

DESCRIPTION

The **mbsinit()** function determines whether the `mbstate_t` object pointed to by *ps* describes an initial conversion state.

RETURN VALUES

The **mbsinit()** function returns non-zero if *ps* is `NULL` or describes an initial conversion state, otherwise it returns zero.

STANDARDS

The **mbsinit()** function conforms to ISO/IEC 9899:1999 ("ISO C99").

MBSRTOWCS(3)

NAME

mbsrtowcs, mbsnrtowcs - convert a character string to a wide-character string (restartable)

SYNOPSIS

```
#include <wchar.h>

size_t
mbsrtowcs(wchar_t * restrict dst, const char ** restrict src, size_t len,
    mbstate_t * restrict ps);

size_t
mbsnrtowcs(wchar_t * restrict dst, const char ** restrict src,
    size_t nms, size_t len, mbstate_t * restrict ps);
```

DESCRIPTION

The **mbsrtowcs()** function converts a sequence of multibyte characters pointed to indirectly by *src* into a sequence of corresponding wide characters and stores at most *len* of them in the `wchar_t` array pointed to by *dst*, until it encounters a terminating null character (’
0’).

If *dst* is `NULL` , no characters are stored.

If *dst* is not `NULL`, the pointer pointed to by *src* is updated to point to the character after the one that conversion stopped at. If conversion stops because a null character is encountered, `*`*src* is set to `NULL`.

The `mbstate_t` argument, *ps*, is used to keep track of the shift state. If it is `NULL`, **mbsrtowcs()** uses an internal, static `mbstate_t` object, which is initialized to the initial conversion state at program startup.

The **mbsnrtowcs()** function behaves identically to **mbsrtowcs()**, except that conversion stops after reading at most *nms* bytes from the buffer pointed to by *src*.

RETURN VALUES

The **mbsrtowcs()** and **mbsnrtowcs()** functions return the number of wide characters stored in the array pointed to by *dst* if successful, otherwise it returns `(size_t)-1`.

ERRORS

The **mbsrtowcs()** and **mbsnrtowcs()** functions will fail if:

`EILSEQ`	An invalid multibyte sequence was encountered.
`EINVAL`	The conversion state is invalid.

SEE ALSO

mbrtowc(3), mbstowcs(3), multibyte(3), wcsrtombs(3)

STANDARDS

The **mbsrtowcs()** function conforms to ISO/IEC 9899:1999 (“ISO C99”).

The **mbsnrtowcs()** function is an extension to the standard.

MULTIBYTE(3)

NAME

multibyte – multibyte and wide character manipulation functions

SYNOPSIS

```
#include <limits.h>
#include <stdlib.h>
#include <wchar.h>
```

DESCRIPTION

The basic elements of some written natural languages, such as Chinese, cannot be represented uniquely with single C chars. The C standard supports two different ways of dealing with extended natural language encodings: wide characters and multibyte characters. Wide characters are an internal representation which allows each basic element to map to a single object of type `wchar_t`. Multibyte characters are used for input and output and code each basic element as a sequence of C chars. Individual basic elements may map into one or more (up to `MB_LEN_MAX`) bytes in a multibyte character.

The current locale (setlocale(3)) governs the interpretation of wide and multibyte characters. The locale category `LC_CTYPE` specifically controls this interpretation. The `wchar_t` type is wide enough to hold the largest value in the wide character representations for all locales.

Multibyte strings may contain 'shift' indicators to switch to and from particular modes within the given representation. If explicit bytes are used to signal shifting, these are not recognized as separate characters but are lumped with a neighboring character. There is always a distinguished 'initial' shift state. Some functions (e.g., mblen(3), mbtowc(3) and wctomb(3)) maintain static shift state internally, whereas others store it in an `mbstate_t` object passed by the caller. Shift states are undefined after a call to setlocale(3) with the `LC_CTYPE` or `LC_ALL` categories.

For convenience in processing, the wide character with value `0` (the null wide character) is recognized as the wide character string terminator, and the character with value `0` (the null byte) is recognized as the multibyte character string terminator. Null bytes are not permitted within multibyte characters.

The C library provides the following functions for dealing with multibyte characters:

mblen(3) get number of bytes in a character

mbrlen(3)	get number of bytes in a character (restartable)
mbrtowc(3)	convert a character to a wide-character code (restartable)
mbsrtowcs(3)	convert a character string to a wide-character string (restartable)
mbstowcs(3)	convert a character string to a wide-character string
mbtowc(3)	convert a character to a wide-character code
wcrtomb(3)	convert a wide-character code to a character (restartable)
wcstombs(3)	convert a wide-character string to a character string
wcsrtombs(3)	convert a wide-character string to a character string (restartable)
wctomb(3)	convert a wide-character code to a character

SEE ALSO

setlocale(3), stdio(3), big5(5), euc(5), gb18030(5), gb2312(5), gbk(5), mskanji(5), utf8(5)

STANDARDS

These functions conform to ISO/IEC 9899:1999 (“ISO C99”).

RUNE(3)

NAME

setrunelocale, setinvalidrune, sgetrune, sputrune - rune support for C

SYNOPSIS

```
#include <rune.h>
#include <errno.h>

int
setrunelocale(char *locale)

void
setinvalidrune(rune_t rune)

rune_t
sgetrune(const char *string, size_t n,
char const **result)

int
sputrune(rune_t rune, char *string, size_t n,
char **result)

#include <stdio.h>

long
fgetrune(FILE *stream)

int
fungetrune(rune_t rune, FILE *stream)

int
fputrune(rune_t rune, FILE *stream)
```

DESCRIPTION

The **setrunelocale()** controls the type of encoding used to represent runes as multi-byte strings as well as the properties of the runes as defined in ¡ctype.h¿. The locale argument indicates which locale to load. If the locale is successfully loaded, 0 is returned, otherwise an `errno` value is returned to indicate the type of error.

The **setinvalidrune()** function sets the value of the global value `_INVALID_RUNE` to be *rune*.

The **sgetrune()** function tries to read a single multibyte character from *string*, which is at most *n* bytes long. If **sgetrune()** is successful, the rune is returned. If *result* is not `NULL`, `*result` will point to the first byte which was not converted in *string*. If the first *n* bytes of string do not describe a full multibyte character, `_INVALID_RUNE` is returned and `*result` will point to *string*. If there is an encoding error at the start of *string*, `_INVALID_RUNE` is returned and `*result` will point to the second character of *string*.

The **sputrune()** function tries to encode *rune* as a multibyte string and store it at *string*, but no more than *n* bytes will be stored. If *result* is not `NULL`, `*result` will be set to point to the first byte in *string* following the new multibyte character. If *string* is `NULL`, `*result` will point to `(char *)0 +` *x*, where *x* is the number of bytes that would be needed to store the multibyte value. If the multibyte character would consist of more than *n* bytes and result is not `NULL`, `*result` will be set to `NULL`. In all cases, **sputrune()** will return the number of bytes which would be needed to store rune as a multibyte character.

The **fgetrune()** function operates the same as **sgetrune()** with the exception that it attempts to read enough bytes from *stream* to decode a single rune. It returns either `EOF` on end of file, `_INVALID_RUNE` on an encoding error, or the rune decoded if all went well.

The **fungetrune()** function pushes the multibyte encoding, as provided by **sputrune()**, of *rune* onto *stream* such that the next **fgetrune()** call will return *rune*. It returns `EOF` if it fails and 0 on success.

The **fputrune()** function writes the multibyte encoding of *rune*, as provided by **sputrune()**, onto *stream*. It returns `EOF` on failure and 0 on success.

RETURN VALUES

The **setrunelocale()** function returns one of the following values:

`0`	**setrunelocale()** was successful.
`EFAULT`	*locale* was `NULL`.
`ENOENT`	The locale could not be found.
`EFTYPE`	The file found was not a valid file.
`EINVAL`	The encoding indicated by the locale was unknown.

The **sgetrune()** function either returns the rune read or `_INVALID_RUNE`. The **sputrune()** function returns the number of bytes needed to store rune as a multibyte string.

SEE ALSO

mbrune(3), setlocale(3)

NOTE

The ANSI C type `wchar_t` is the same as `rune_t`. `Rune_t` was chosen to accent the purposeful choice of not basing the system with the ANSI C primitives, which were, shall we say, less aesthetic.

The **setrunelocale()** function and the other non-ANSI rune functions were inspired by Plan 9 from Bell Labs as a much more sane alternative to the ANSI multibyte and wide character support.

All of the ANSI multibyte and wide character support functions are built using the rune functions.

SETLOCALE(3)

NAME

setlocale, localeconv - natural language formatting for C

SYNOPSIS

```
#include <locale.h>

char *
setlocale(int category, const char *locale)

struct lconv *
localeconv(void)
```

DESCRIPTION

The **setlocale()** function sets the C library's notion of natural language formatting style for particular sets of routines. Each such style is called a 'locale' and is invoked using an appropriate name passed as a C string. The **localeconv()** routine returns the current locale's parameters for formatting numbers.

The **setlocale()** function recognizes several categories of routines. These are the categories and the sets of routines they select:

LC_ALL	Set the entire locale generically.
LC_COLLATE	Set a locale for string collation routines. This controls alphabetic ordering in **strcoll()** and **strxfrm()**.
LC_CTYPE	Set a locale for the ctype(3), mbrune(3), multibyte(3) and rune(3) functions. This controls recognition of upper and lower case, alphabetic or non-alphabetic characters, and so on. The real work is done by the **setrunelocale()** function.
LC_MONETARY	Set a locale for formatting monetary values; this affects the **localeconv()** function.
LC_NUMERIC	Set a locale for formatting numbers. This controls the formatting of decimal points in input and output of floating point numbers in functions such as **printf()** and **scanf()**, as well as values returned by **localeconv()**.

`LC_TIME`	Set a locale for formatting dates and times using the **strftime()** function.

Only three locales are defined by default, the empty string `""` which denotes the native environment, and the `"C"` and `"POSIX"` locales, which denote the C language environment. A *locale* argument of `NULL` causes **setlocale()** to return the current locale. By default, C programs start in the `"C"` locale. The only function in the library that sets the locale is **setlocale()**; the locale is never changed as a side effect of some other routine.

The **localeconv()** function returns a pointer to a structure which provides parameters for formatting numbers, especially currency values:

```
struct lconv {
   char  *decimal_point;
   char  *thousands_sep;
   char  *grouping;
   char  *int_curr_symbol;
   char  *currency_symbol;
   char  *mon_decimal_point;
   char  *mon_thousands_sep;
   char  *mon_grouping;
   char  *positive_sign;
   char  *negative_sign;
   char  int_frac_digits;
   char  frac_digits;
   char  p_cs_precedes;
   char  p_sep_by_space;
   char  n_cs_precedes;
   char  n_sep_by_space;
   char  p_sign_posn;
   char  n_sign_posn;
};
```

The individual fields have the following meanings:

`decimal_point`	The decimal point character, except for currency values.
`thousands_sep`	The separator between groups of digits before the decimal point, except for currency values.
`grouping`	The sizes of the groups of digits, except for currency values. This is a pointer to a vector of integers, each of size char, representing group size from low order digit groups to high order (right

to left). The list may be terminated with 0 or CHAR_MAX. If the list is terminated with 0, the last group size before the 0 is repeated to account for all the digits. If the list is terminated with CHAR_MAX, no more grouping is performed.

int_curr_symbol	The standardized international currency symbol.
currency_symbol	The local currency symbol.
mon_decimal_point	The decimal point character for currency values.
mon_thousands_sep	The separator for digit groups in currency values.
mon_grouping	Like grouping but for currency values.
positive_sign	The character used to denote nonnegative currency values, usually the empty string.
negative_sign	The character used to denote negative currency values, usually a minus sign.
int_frac_digits	The number of digits after the decimal point in an international-style currency value.
frac_digits	The number of digits after the decimal point in the local style for currency values.
p_cs_precedes	1 if the currency symbol precedes the currency value for nonnegative values, 0 if it follows.
p_sep_by_space	1 if a space is inserted between the currency symbol and the currency value for nonnegative values, 0 otherwise.
n_cs_precedes	Like p_cs_precedes but for negative values.
n_sep_by_space	Like p_sep_by_space but for negative values.
p_sign_posn	The location of the positive_sign with respect to a nonnegative quantity and the currency_symbol, coded as follows: 0 Parentheses around the entire string. 1 Before the string. 2 After the string. 3 Just before currency_symbol. 4 Just after currency_symbol.
n_sign_posn	Like p_sign_posn but for negative currency values.

Unless mentioned above, an empty string as a value for a field indicates a zero length result or a value that is not in the current locale. A `CHAR_MAX` result similarly denotes an unavailable value.

RETURN VALUES

The **setlocale()** function returns `NULL` and fails to change the locale if the given combination of category and locale makes no sense. The **localeconv()** function returns a pointer to a static object which may be altered by later calls to **setlocale()** or **localeconv()**.

SEE ALSO

colldef(1), mklocale(1), ctype(3), mbrune(3), multibyte(3), rune(3), stroll(3), strxfrm(3)

STANDARDS

The **setlocale()** and **localeconv()** functions conform to ISO/IEC 9899:1990 ("ISO C90").

ISSUES

The current implementation supports only the `"C"` and `"POSIX"` locales for all but the `LC_COLLATE`, `LC_CTYPE`, and `LC_TIME` categories.

In spite of the gnarly currency support in **localeconv()**, the standards don't include any functions for generalized currency formatting.

Use of `LC_MONETARY` could lead to misleading results until we have a real time currency conversion function. `LC_NUMERIC` and `LC_TIME` are personal choices and should not be wrapped up with the other categories.

TOASCII(3)

NAME

toascii - convert a byte to 7-bit ASCII

SYNOPSIS

```
#include <ctype.h>

int
toascii(int c)
```

DESCRIPTION

The **toascii()** function strips all but the low 7 bits from a letter, including parity or other marker bits.

RETURN VALUES

The **toascii()** function always returns a valid ASCII character.

SEE ALSO

isalnum(3), isalpha(3), isascii(3), iscntrl(3), isdigit(3), isgraph(3), slower(3), isprint(3), ispunct(3), isspace(3), isupper(3), isxdigit(3), stdio(3), tolower(3), toupper(3)

NOTE

This function makes little sense for an EBCDIC character, but is provided for compatibility.

TOLOWER(3)

NAME

tolower - upper case to lower case letter conversion

SYNOPSIS

```
#include <ctype.h>

int
tolower(int c)
```

DESCRIPTION

The **tolower()** function converts an upper-case letter to the corresponding lower-case letter.

RETURN VALUES

If the argument is an upper-case letter, the **tolower()** function returns the corresponding lower-case letter if there is one; otherwise the argument is returned unchanged.

SEE ALSO

isalnum(3), isalpha(3), isascii(3), iscntrl(3), isdigit(3), isgraph(3), slower(3), isprint(3), ispunct(3), isspace(3), isupper(3), isxdigit(3), stdio(3), toascii(3), toupper(3)

STANDARDS

The **tolower()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

TOUPPER(3)

NAME

toupper - lower case to upper case letter conversion

SYNOPSIS

```
#include <ctype.h>

int
toupper(int c)
```

DESCRIPTION

The **toupper()** function converts a lower-case letter to the corresponding upper-case letter. If the argument is a lower-case letter, the **toupper()** function returns the corresponding upper-case letter if there is one; otherwise the argument is returned unchanged.

SEE ALSO

isalnum(3), isalpha(3), isascii(3), iscntrl(3), isdigit(3), isgraph(3), slower(3), isprint(3), ispunct(3), isspace(3), isupper(3), isxdigit(3), stdio(3), toascii(3), tolower(3)

STANDARDS

The **toupper()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

TOWLOWER(3)

NAME

towlower - upper case to lower case letter conversion (wide character version)

SYNOPSIS

```
#include <wctype.h>

wint_t
towlower(wint_t wc);
```

DESCRIPTION

The **towlower()** function converts an upper-case letter to the corresponding lower-case leteter.

RETURN VALUES

If the argument is an upper-case letter, the **towlower()** function returns the corresponding lower-case letter if there is one; otherwise the arugmnet is returned unchanged.

SEE ALSO

iswlower(3), tolower(3), towupper(3), wctrans(3)

STANDARDS

The **towlower()** function conforms to ISO/IEC 9899:1999 (“ISO C99”).

TOWUPPER(3)

NAME

towupper - lower case to upper case letter conversion (wide character version)

SYNOPSIS

```
#include <wctype.h>

wint_t
towupper(wint_t wc);
```

DESCRIPTION

The **towupper()** function converts a lower-case letter to the corresponding upper-case letter.

RETURN VALUES

If the argument is a lower-case letter, the **towupper()** function returns the corresponding upper-case letter if there is one; otherwise the argument is returned unchanged.

SEE ALSO

iswupper(3), toupper(3), towlower(3), wctrans(3)

STANDARDS

The **towupper()** function conforms to ISO/IEC 9899:1999 (“ISO C99”).

WCSTOL(3)

NAME

wcstol, wcstoul, wcstoll, wcstoull, wcstoimax, wcstoumax - convert a wide character string value to a `long`, `unsigned long`, `long long`, `unsigned long long`, `intmax_t` or `uintmax_t` integer

SYNOPSIS

```
#include <wchar.h>

long
wcstol(const wchar_t * restrict nptr, wchar_t ** restrict endptr,
         int base);

unsigned long
wcstoul(const wchar_t * restrict nptr, wchar_t ** restrict endptr,
         int base);

long long
wcstoll(const wchar_t * restrict nptr, wchar_t ** restrict endptr,
         int base);

unsigned long long
wcstoull(const wchar_t * restrict nptr, wchar_t ** restrict endptr,
         int base);

#include <inttypes.h>

intmax_t
wcstoimax(const wchar_t * restrict nptr, wchar_t ** restrict endptr,
         int base);

uintmax_t
wcstoumax(const wchar_t * restrict nptr, wchar_t ** restrict endptr,
         int base);
```

DESCRIPTION

The **wcstol()**, **wcstoul()**, **wcstoll()**, **wcstoull()**, **wcstoimax()** and **wcstoumax()** functions are wide-character versions of the **strtol()**, **strtoul()**, **strtoll()**, **strtoull()**, **strtoimax()** and **strtoumax()** functions, respectively. Refer to their manual pages (for example strtol(3)) for details.

SEE ALSO

strtol(3), strtoul(3)

STANDARDS

The **wcstol()**, **wcstoul()**, **wcstoll()**, **wcstoull()**, **wcstoimax()** and **wcstoumax()** functions conform to ISO/IEC 9899:1999 ("ISO C99").

WCTRANS(3)

NAME

towctrans, wctrans - wide character mapping functions

SYNOPSIS

```
wint_t
towctrans(wint_t wc, wctrans_t desc);

wctrans_t
wctrans(const char *charclass);
```

DESCRIPTION

The **wctrans()** function returns a value of type `wctrans_t` which represents the requested wide character mapping operation and may be used as the second argument for calls to **towctrans()**.

The following character mapping names are recognized:

```
tolower      toupper
```

The **towctrans()** function transliterates the wide character *wc* according to the mapping described by *desc*.

RETURN VALUES

The **towctrans()** function returns the transliterated character if successful, otherwise it returns the character unchanged and sets `errno`.

The **wctrans()** function returns non-zero if successful, otherwise it returns zero and sets `errno`.

EXAMPLES

Reimplement **towupper()** in terms of **towctrans()** and **wctrans()**:

```
wint_t
mytowupper(wint_t wc)
{
return (towctrans(wc, wctrans("toupper")));
}
```

ERRORS

The **towctrans()** function will fail if:

`EINVAL` The supplied *desc* argument is invalid.

The **wctrans()** function will fail if:

`EINVAL` The requested mapping name is invalid.

SEE ALSO

tolower(3), toupper(3), wctype(3)

STANDARDS

The **towctrans()** and **wctrans()** functions conform to ISO/IEC 9899:1999 ("ISO C99").

WCTYPE(3)

NAME

iswctype, wctype - wide character class functions

SYNOPSIS

```
#include <wctype.h>

int
iswctype(wint_t wc, wctype_t charclass);

wctype_t
wctype(const char *property);
```

DESCRIPTION

The **wctype()** function returns a value of type `wctype_t` which represents the requested wide character class and may be used as the second argument for calls to **iswctype()**.

The following character class names are recognized:

`alnum`	`cntrl`	`ideogram`	`print`	`special`
`alpha`	`digit`	`lower`	`punct`	`upper`
`blank`	`graph`	`phonogram`	`space`	`xdigit`

The **iswctyp()** function checks whether the wide character *wc* is in the character class *charclass*.

RETURN VALUES

The **iswctype()** function returns non-zero if and only if *wc* has the property described by *charclass*, or *charclass* is zero.

The **wctype()** function returns 0 if *property* is invalid, otherwise it returns a value of type `wctype_t` that can be used in subsequent calls to **iswctype()**.

EXAMPLE

Reimplement iswalpha(3) in terms of **iswctype()** and **wctype()**:

```
int
myiswalpha(wint_t wc)
{
return (iswctype(wc, wctype("alpha")));
}
```

SEE ALSO

ctype(3)

STANDARDS

The **iswctype()** and **wctype()** functions conform to ISO/IEC 9899:1999 ("ISO C99"). The "ideogram", "phonogram" and "special" character classes are extensions.

WCWIDTH(3)

NAME

wcwidth - number of column positions of a wide-character code

SYNOPSIS

```
int
wcwidth(wchar_t wc);
```

DESCRIPTION

The **wcwidth()** function determines the number of column positions required to display the wide character *wc*.

RETURN VALUES

The **wcwidth()** function returns 0 if the *wc* argument is a null wide character (`L'\0'`), -1 if *wc* is not printable, otherwise it returns the number of column positions the character occupies.

SEE ALSO

iswprint(3), wcswidth(3)

STANDARDS

The **wcwidth()** function conforms to ISO/IEC 9899:1999 ("ISO C99").

Math library

The math library contains implementations of the transcendental functions and other support routines for manipulating floating point values.

MATH(3)

NAME

math - introduction to mathematical library functions

DESCRIPTION

These functions constitute the C math library.

Declarations for these functions may be obtained from the include file `<math.h>`.

LIST OF FUNCTIONS

Name	Appears on Page	Description
__isBFP	__isBFP(3)	determine floating-point format
acos	acos(3)	arc cosine function
acosf	acos(3)	arc cosine function
acosl	acos(3)	arc cosine function
acosh	acosh(3)	inverse hyperbolic function
acoshf	acosh(3)	inverse hyperbolic function
acoshl	acosh(3)	inverse hyperbolic function
asin	asin(3)	arc sine function
asinf	asin(3)	arc sine function
asinl	asin(3)	arc sine function
asinh	asinh(3)	inverse hyperbolic function
asinhf	asinh(3)	inverse hyperbolic function
asinhl	asinh(3)	inverse hyperbolic function
atan	atan(3)	arc tangent function of one variable
atanf	atan(3)	arc tangent function of one variable
atanl	atan(3)	arc tangent function of one variable
atanh	atanh(3)	inverse hyperbolic function
atanhf	atanh(3)	inverse hyperbolic function
atanhl	atanh(3)	inverse hyperbolic function
atan2	atan2(3)	arc tangent function of two variables
atan2f	atan2(3)	arc tangent function of two variables
atan2l	atan2(3)	arc tangent function of two variables
cbrt	sqrt(3)	cube root
cbrtf	sqrt(3)	cube root
cbrtl	sqrt(3)	cube root
ceil	ceil(3)	integer no less than
ceilf	ceil(3)	integer no less than
ceill	ceil(3)	integer no less than
copysign	copysign(3)	copy sign
copysignf	copysign(3)	copy sign
copysignl	copysign(3)	copy sign
cos	cos(3)	trigonometric function
cosf	cos(3)	trigonometric function
cosl	cos(3)	trigonometric function
cosh	cosh(3)	hyperbolic cosine function
coshf	cosh(3)	hyperbolic cosine function
coshl	cosh(3)	hyperbolic cosine function

Name	Appears on Page	Description
erf	erf(3)	error function
erff	erf(3)	error function
erfl	erf(3)	error function
erfc	erf(3)	complementary error function
erfcf	erf(3)	complementary error function
erfcl	erf(3)	complementary error function
exp	exp(3)	exponential e^x
exp2	exp(3)	exponential 2^x
exp2f	exp(3)	exponential 2^x
exp2l	exp(3)	exponential 2^x
expf	exp(3)	exponential e^x
expl	exp(3)	exponential e^x
expm1	exp(3)	$e^x - 1$
expm1f	exp(3)	$e^x - 1$
expm1f	exp(3)	$e^x - 1$
fabs	fabs(3)	absolute value
fabsf	fabs(3)	absolute value
fabsl	fabs(3)	absolute value
fdim	fdim(3)	positive difference functions
fdimf	fdim(3)	positive difference functions
fdiml	fdim(3)	positive difference functions
feenableexcept	feenableexcept(3)	floating point exception masking
fegetround	fegetround(3)	retrieve/set current IEEE floating point rounding direction
fe_dec_getround	fe_dec_getround(3)	retrieve/set current Decimal floating point rounding direction
floor	floor(3)	integer no greater than
floorf	floor(3)	integer no greater than
floorl	floor(3)	integer no greater than
fma	fma(3)	fused multiply-add
fmaf	fma(3)	fused multiply-add
fmal	fma(3)	fused multiply-add
fmax	fmax(3)	floating-point maximum and minimum functions
fmaxf	fmax(3)	floating-point maximum and minimum functions
fmaxl	fmax(3)	floating-point maximum and minimum functions
fmin	fmax(3)	floating-point maximum and minimum functions
fminf	fmax(3)	floating-point maximum and minimum functions
fminl	fmax(3)	floating-point maximum and minimum functions
fmod	fmod(3)	floating-point remainder functions
fmodf	fmod(3)	floating-point remainder functions
fmodl	fmod(3)	floating-point remainder

Name	Appears on Page	Description
frexp	frexp(3)	convert to fraction and integral components
frexpf	frexp(3)	convert to fraction and integral components
frexpl	frexp(3)	convert to fraction and integral components
hypot	hypot(3)	Euclidean distance
ilogb	ilogb(3)	extract exponent
ilogbf	ilogb(3)	extract exponent
ilogbl	ilogb(3)	extract exponent
isfinite	fpclassify(3)	classify a floating-point number
isgreater	isgreater(3)	compare two floating-point numbers
isgreaterequal	isgreater(3)	compare two floating-point numbers
isinf	fpclassify(3)	classify a floating-point number
isless	isgreater(3)	compare two floating-point numbers
islessequal	isgreater(3)	compare two floating-point numbers
islessgreater	isgreater(3)	compare two floating-point numbers
isnan	fpclassify(3)	classify a floating-point number
isnormal	fpclassify(3)	classify a floating-point number
isunordered	isgreater(3)	compare two floating-point numbers
ldexp	ldexp(3)	multiply by integral power of 2
ldexpf	ldexp(3)	multiply by integral power of 2
ldexpl	ldexp(3)	multiply by integral power of 2
lgamma	lgamma(3)	log gamma function
lgammaf	lgamma(3)	log gamma function
lgammal	lgamma(3)	log gamma function
llrint	lrint(3)	convert to integer
llrintf	lrint(3)	convert to integer
llrintl	lrint(3)	convert to integer
llround	lround(3)	convert to nearest integral value
llroundf	lround(3)	convert to nearest integral value
llroundl	lround(3)	convert to nearest integral value

Name	Appears on Page	Description
log	log(3)	natural logarithm $\ln(x)$
logf	log(3)	natural logarithm $\ln(x)$
logl	log(3)	natural logarithm $\ln(x)$
log10	log(3)	logarithm to base 10 $\log_{10}(x)$
log10f	log(3)	logarithm to base 10 $\log_{10}(x)$
log10l	log(3)	logarithm to base 10 $\log_{10}(x)$
log2	log(3)	logarithm to base 2 $\log_2(x)$
log2f	log(3)	logarithm to base 2 $\log_2(x)$
log2l	log(3)	logarithm to base 2 $\log_2(x)$
logb	ilogb(3)	extract exponent
logbf	ilogb(3)	extract exponent
logbl	ilogb(3)	extract exponent
log1p	log(3)	$\ln(1 + x)$
log1pf	log(3)	$\ln(1 + x)$
log1pl	log(3)	$\ln(1 + x)$
lrint	lrint(3)	convert to integer
lrintf	lrint(3)	convert to integer
lrintl	lrint(3)	convert to integer
lround	lround(3)	convert to nearest integral value
lroundf	lround(3)	convert to nearest integral value
lroundl	lround(3)	convert to nearest integral value
modf	modf(3)	extract signed integral and fractional values from floating-point number
modff	modf(3)	extract signed integral and fractional values from floating-point number
modfl	modf(3)	extract signed integral and fractional values from floating-point number
nan	nan(3)	quiet NaNs
nanf	nan(3)	quiet NaNs
nanl	nan(3)	quiet NaNs
nearbyint	rint(3)	round to integral value in floating-point format
nearbyintf	rint(3)	round to integral value in floating-point format
nearbyintl	rint(3)	round to integral value in floating-point format
nextafter	nextafter(3)	next representable value
nextafterf	nextafter(3)	next representable value
nextafterl	nextafter(3)	next representable value
nexttoward	nextafter(3)	next representable value
nexttowardf	nextafter(3)	next representable value
nexttowardl	nextafter(3)	next representable value

Name	Appears on Page	Description
pow	exp(3)	exponential x^y
powf	exp(3)	exponential x^y
powl	exp(3)	exponential x^y
remainder	remainder(3)	minimal residue functions
remainderf	remainder(3)	minimal residue functions
remainderl	remainder(3)	minimal residue functions
remquo	remainder(3)	minimal residue functions
remquof	remainder(3)	minimal residue functions
remquol	remainder(3)	minimal residue functions
rint	rint(3)	round to integral value in floating-point format
rintf	rint(3)	round to integral value in floating-point format
rintl	rint(3)	round to integral value in floating-point format
round	round(3)	round to nearest integral value
roundf	round(3)	round to nearest integral value
roundl	round(3)	round to nearest integral value
scalbln	scalbn(3)	adjust exponent
scalblnf	scalbn(3)	adjust exponent
scalblnl	scalbn(3)	adjust exponent
scalbn	scalbn(3)	adjust exponent
scalbnf	scalbn(3)	adjust exponent
scalbnl	scalbn(3)	adjust exponent
signbit	signbit(3)	determine whether a floating-point number's sign is negative
sin	sin(3)	trigonometric function
sinf	sin(3)	trigonometric function
sinl	sin(3)	trigonometric function
sinh	sinh(3)	hyperbolic function
sinhf	sinh(3)	hyperbolic function
sinhl	sinh(3)	hyperbolic function
sqrt	sqrt(3)	square root
sqrtf	sqrt(3)	square root
sqrtl	sqrt(3)	square root
tan	tan(3)	trigonometric function
tanf	tan(3)	trigonometric function
tanl	tan(3)	trigonometric function
tanh	tanh(3)	hyperbolic function
tanhf	tanh(3)	hyperbolic function
tanhl	tanh(3)	hyperbolic function

Name	Appears on Page	Description
tgamma	lgamma(3)	gamma function
tgammaf	lgamma(3)	gamma function
tgammal	lgamma(3)	gamma function
trunc	trunc(3)	nearest integral value with magnitude less than or equal to $\|x\|$
truncf	trunc(3)	nearest integral value with magnitude less than or equal to $\|x\|$
truncl	trunc(3)	nearest integral value with magnitude less than or equal to $\|x\|$

NOTES

These library functions support both the HFP (Hexadecimal Floating Point) format and BFP (IEEE Binary Floating Point) formats. The selection of the format is either made at compile time via compiler options, or at runtime based on the return value of the **__isBFP()** function.

SEE ALSO

An explanation of the HFP and BFP floating point formats is provided in the z/Architecture Principles of Operations, IBM publication SA22-7200.

__isBFP(3)

__FP_CAST(3)

NAME

__fp_cast - floating point cast function

SYNOPSIS

```
#include <machine/IEEE754.h>

int
__fp_cast(int mode, void *src_ptr, int src_kind,
                    void *targ_ptr, int targ_kind);
```

DESCRIPTION

The **__fp_cast()** function adjusts the size of a floating point number from the *src_kind* to the *targ_kind* sizes. **__fp_cast()** will convert the floating point value at the address specified in *src_ptr* to the target size and save the result at the address specified in *targ_ptr*.

The *mode* parameter indicates one of _FP_BFP_MODE or _FP_HFP_MODE for either BFP or HFP values.

The *src_kind* and *targ_kind* parameters indicate the size of the source and target floating point number. Each should be one of _FP_FLOAT, _FP_DOUBLE or _FP_LONG_DOUBLE.

RETURN VALUES

The **__fp_cast()** function returns 0 on success.

If any of *mode*, *src_kind* or *targ_kind* is invalid **__fp_cast()** returns -1.

SEE ALSO

__isbfp(3)

__ISBFP(3)

NAME

__isBFP, __fp_swapmode, __fp_setmode - floating point format functions

SYNOPSIS

```
#include <machine/IEE754.h>

int
__isBFP(void);

int
__fp_swapmode(int newmode);

void
__fp_setmode(int newmode);
```

DESCRIPTION

The **__isBFP()** function determines the current selection of the floating-point format. The runtime library maintains a selection state of either BFP or HFP or "undetermined." When **__isBPF()** is invoked, if the state is "undetermined", then the compile-time setting of the caller is investigated to determine the mode. **__isBFP()** returns 1 if the mode is specifically set to BFP, or the caller is determined to be BFP, 0 for HFP.

The **__fp_swapmode()** function is used to return the current library floating point state, and set a new one. It can be used to retain the current mode, set a new one, perform some operations and then restore the previous mode.

The **__fp_setmode()** function sets a particular state.

The state values are

`_FP_MODE_RESET` The mode is determined by the caller.

`_FP_HFP_MODE` The mode is HFP.

`_FP_BFP_MODE` The mode is BFP.

Many of the library functions use **__isBFP()** to determine if the operation is to proceed in BFP or HFP format.

RETURN VALUES

The **__isBFP()** function returns 1 if the current mode is `_FP_BFP_MODE` or if the current mode is `_FP_MODE_RESET` and the caller has been determined to be compiled for BFP values; otherwise it returns zero.

The **__fp_swapmode()** function returns the current mode state setting.

The current mode is thread-specific and is inherited from the parent thread when a new thread is created.

SEE ALSO

__fp_cast(3)

ACOS(3)

NAME

acos, acosf, acosl - arc cosine functions

SYNOPSIS

```
#include <math.h>

double
acos(double x);

float
acosf(float x);

long double
acosl(long double x);
```

DESCRIPTION

The **acos()**, **acosf()** and **acosl()** functions computes the principal value of the arc cosine of x. A domain error occurs for arguments not in the range $[-1, +1]$.

RETURN VALUES

The **acos()**, **acosf()** and **acosl()** functions return the arc cosine in the range $[0, \pi]$ radians. If $|x| > 1$,

The **acos()**, **acosf()** and **acosl()** functions may set the global variable `errno` to `EDOM` and a reserved operand fault may be generated.

SEE ALSO

asin(3), atan(3), atan2(3), cos(3), cosh(3), math(3), sin(3), sinh(3), tan(3), tanh(3)

STANDARDS

The **acos()** function conforms to ISO/IEC 9899:1990 ("ISO C90").

The **acosf()** and **acosl()** functions conform to ISO/IEC 9899:1999 ("ISO C99").

ACOSH(3)

NAME

acosh, acoshf, acoshl - inverse hyperbolic cosine function

SYNOPSIS

```
#include <math.h>

double
acosh(double x);

float
acoshf(float x);

long double
acoshl(long double x);
```

DESCRIPTION

The **acosh()**, **acoshf()** and **acoshl()** functions computes the inverse hyperbolic cosine of the real argument *x*. If the argument is less than 1, these functions raise an invalid exception and for BFP values return a `NaN`, for HFP alues these functions return 0.0.

RETURN VALUES

The **acosh()**, **acoshf()** and **acoshl()** functions return the inverse hyperbolic cosine of *x*.

SEE ALSO

asinh(3), atanh(3), exp(3), math(3)

STANDARDS

The **acosh()**, **acoshf()** and **acoshl()** functions conform to ISO/IEC 9899:1999 ("ISO C99").

SCALBN(3)

NAME

scalbln, scalblnf, scalblnl, scalbn, scalbnf, scalbnl – adjust exponent

SYNOPSIS

```
#include <math.h>

double
scalbln(double x, long n);

float
scalblnf(float x, long n);

long double
scalblnl(long double x, long n);

double
scalbn(double x, int n);

float
scalbnf(float x, int n);

long double
scalbnl(long double x, int n);
```

DESCRIPTION

These routines return $x * (2 * *n)$ computed by exponent manipulation.

SEE ALSO

math(3)

STANDARDS

These routines conform to ISO/IEC 9899:1999 (“ISO C99”).

ASIN(3)

NAME

asin, asinf, asinl - arc sine functions

SYNOPSIS

```
#include <math.h>

double
asin(double x);

float
asinf(float x);

long double
asinl(long double x);
```

DESCRIPTION

The **asin()**, **asinf()** and **asinl()** functions computes the principal value of the arc sine of x. A domain error occurs for arguments not in the range $[-1, +1]$.

RETURN VALUES

The **asin()**, **asinf()** and **asinl()** functions return the arc sine in the range $[-\frac{\pi}{2}, +\frac{\pi}{2}]$ radians.

SEE ALSO

acos(3), atan(3), atan2(3), cos(3), cosh(3), math(3), sin(3), sinh(3), tan(3), tanh(3)

STANDARDS

The **asin()** function conforms to ISO/IEC 9899:1990 ("ISO C90").

The **asinf()** and **asinl()** functions conform to ISO/IEC 9899:1999 ("ISO C99").

ASINH(3)

NAME

asinh, asinhf, asinhl - inverse hyperbolic sine function

SYNOPSIS

```
#include <math.h>

double
asinh(double x);

float
asinhf(float x);

long double
asinhl(long double x);
```

DESCRIPTION

The **asinh()**, **asinhf()** and **asinhl()** functions compute the inverse hyperbolic sine of the real argument *x*.

RETURN VALUES

The **asinh()**, **asinhf()** and **asinhl()** functions return the inverse hyperbolic sine of *x*.

SEE ALSO

acosh(3), atanh(3), exp(3), math(3)

STANDARDS

The **asinh()**, **asinhf()** and **asinhl()** functions conform to ISO/IEC 9899:1999 ("ISO C99").

ATAN(3)

NAME

atan, atanf, atanl - arc tangent functions of one variable

SYNOPSIS

```
#include <math.h>

double
atan(double x);

float
atanf(float x);

long double
atanl(long double x);
```

DESCRIPTION

The **atan()**, **atanf()** and **atanl()** functions compute the principal value of the arc tangent of x.

RETURN VALUES

The **atan()**, **atanf()** and **atanl()** functions return the arc tangent in the range $[-\frac{\pi}{2}, +\frac{\pi}{2}]$ radians.

SEE ALSO

acos(3), asin(3), atan2(3), cos(3), cosh(3), math(3), sin(3), sinh(3), tan(3), tanh(3)

STANDARDS

The **atan()** function conforms to ISO/IEC 9899:1990 ("ISO C90").

The **atanf()** and **atanl()** functions conform to ISO/IEC 9899:1999 ("ISO C99").

ATAN2(3)

NAME

atan2, atan2f, atan2l - arc tangent functions of two variables

SYNOPSIS

```
#include <math.h>

double
atan2(double y, double x);

float
atan2f(float y, float x);

long double
atan2l(long double y, long double x);
```

DESCRIPTION

The **atan2()**, **atan2f()** and **atan2l()** functions compute the principal value of $\arctan\left(\frac{y}{x}\right)$, using the signs of both arguments to determine the quadrant of the return value.

RETURN VALUES

The **atan2()**, **atan2f()** and **atan2l()** functions if successful, return $\arctan\left(\frac{y}{x}\right)$ in the range $[-\pi, \pi]$ radians. If both x and y are zero, the global variable `errno` is set to `EDOM`.

$$\text{atan2}(y,x) = \begin{cases} \arctan(\frac{y}{x}) & \text{if } x > 0 \\ \text{sign}(y)\left(\pi - \arctan\left(|\frac{y}{x}|\right)\right) & \text{if } x < 0 \\ 0 & \text{if } x = y = 0 \\ \text{sign}(y)\frac{\pi}{2} & \text{if } x = 0 \neq y \end{cases}$$

NOTES

The functions **atan2()**, **atan2f()** and **atan2l()** defines "if $x > 0$, $\text{atan2}(0,0) = 0$." On some systems, `atan2(0, 0)` may generate an error message. The reasons for assigning a value to `atan2(0, 0)` are these:

- Programs that test arguments to avoid computing `atan2(0, 0)` must be indifferent to its value. Programs that require it to be invalid are vulnerable to diverse reactions to that invalidity on diverse computer systems.
- The **atan2()** function is used mostly to convert from rectangular (x, y) to polar (r, θ) coordinates that must satisfy $x = r\cos\theta$ and $y = r\sin\theta$. These equations are satisfied when $(x = 0, y = 0)$ is mapped to $(r = 0, \theta = 0)$. In general, conversions to polar coordinates should be computed thus:

```
r     = hypot(x,y);  /* ... = sqrt(x*x+y*y) */
theta = atan2(y,x);
```

- The foregoing formulas need not be altered to cope in a reasonable way with signed zeros and infinities on a machine that conforms to IEEE 754. The versions of hypot(3) and **atan2()** provided for such a machine are designed to handle all cases. That is why atan2$(\pm 0, -0) = \pm\pi$ for instance. In general the formulas above are equivalent to these:

```
r = sqrt(x*x+y*y); if (r == 0) x = copysign(1,x);
```

SEE ALSO

acos(3), asin(3), atan(3), cos(3), cosh(3), math(3), sin(3), sinh(3), tan(3), tanh(3)

STANDARDS

The **atan2()** function conforms to ISO/IEC 9899:1990 ("ISO C90").

The **atan2f()** and **atan2l()** functions conform to ISO/IEC 9899:1999 ("ISO C99").

ATANH(3)

NAME

atanh, atanhf, atanhl - inverse hyperbolic tangent function

SYNOPSIS

```
#include <math.h>

double
atanh(double x);

float
atanhf(float x);

long double
atanhl(long double x);
```

DESCRIPTION

The **atanh()**, **atanhf()** and **atanhl()** functions compute the inverse hyperbolic tangent of the real argument *x*.

RETURN VALUES

The **atanh()**, **atanhf()** and **atanhl()** functions return the inverse hyperbolic tangent of *x* if successful. If the argument has the value 1.0, `HUGE_VAL` is returned, if the argument is -1.0, `-HUGE_VAL` is returned. If the absolute value of the argument is greater than 1.0, a DOMAIN error is indicated and for BFP values a `NaN` is returned, for HFP values 0.0 is returned.

SEE ALSO

acosh(3), asinh(3), exp(3), fenv(3), math(3)

STANDARDS

The **atanh()**, **atanhf()** and **atanhl()** functions conform to ISO/IEC 9899:1999 ("ISO C99").

CEIL(3)

NAME

ceil, ceilf, ceill – smallest integral value greater than or equal to x

SYNOPSIS

```
#include <math.h>

double
ceil(double x);

float
ceilf(float x);

long double
ceill(long double x);
```

DESCRIPTION

The **ceil()**, **ceilf()** and **ceill()** functions compute the smallest integral value greater than or equal to x, expressed as a floating-point number.

SEE ALSO

abs(3), fabs(3), floor(3), math(3), rint(3), round(3), trunc(3)

STANDARDS

The **ceil()** function conforms to ISO/IEC 9899:1990 (“ISO C90”). The **ceilf()** and **ceill()** functions conform to ISO/IEC 9899:1999 (“ISO C99”).

COPYSIGN(3)

NAME

copysign, copysignf, copysignl - copy sign

SYNOPSIS

```
#include <math.h>

double
copysign(double x, double y);

float
copysignf(float x, float y);

long double
copysignl(long double x, long double y);
```

DESCRIPTION

The **copysign()**, **copysignf()** and **copysignl()** functions return x with its sign changed to y's.

SEE ALSO

fabs(3), fdim(3), math(3)

STANDARDS

The **copysign()**, **copysignf()**, and **copysignl()** routines conform to ISO/IEC 9899:1999 ("ISO C99").

COS(3)

NAME

cos, cosf, cosl - cosine functions

SYNOPSIS

```
#include <math.h>

double
cos(double x);

float
cosf(double x);

long double
cosl(long double x);
```

DESCRIPTION

The **cos()**, **cosf()** and **cosl()** functions compute the cosine of x (measured in radians). A large magnitude argument may yield a result with little or no significance.

RETURN VALUES

The **cos()**, **cosf()** and **cosl()** functions return the cosine value.

SEE ALSO

acos(3), asin(3), atan(3), atan2(3), cosh(3), math(3), sin(3), sinh(3), tan(3), tanh(3)

STANDARDS

The **cos()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

The **cosf()** and **cosl()** functions conform to ISO/IEC 9899:1999 (“ISO C99”).

COSH(3)

NAME

cosh, coshf, coshl - hyperbolic cosine function

SYNOPSIS

```
#include <math.h>

double
cosh(double x);

float
coshf(float x);

long double
coshl(long double x);
```

DESCRIPTION

The **cosh()**, **coshf()** and **coshl()** functions compute the hyperbolic cosine of x.

RETURN VALUES

The **cosh()**, **coshf()** and **coshl()** functions returns the hyperbolic cosine.

SEE ALSO

acos(3), asin(3), atan(3), atan2(3), cos(3), math(3), sin(3), sinh(3), tan(3), tanh(3)

STANDARDS

The **cosh()** function conforms to ISO/IEC 9899:1990 ("ISO C90").

The **coshf()** and **coshl()** functions conform to ISO/IEC 9899:1999 ("ISO C99").

ERF(3)

NAME

erf, erff, erfl, erfc, erfcf, erfcl – error function operators

SYNOPSIS

```
#include <math.h>

double
erf(double x);

float
erff(float x);

long double
erfl(long double x);

double
erfc(double x);

float
erfcf(float x);

long double
erfcf(long double x);
```

DESCRIPTION

These functions calculate the error function of x.

The **erf()**, **erff()** and **erfl()** functions calculate the error function of x; where

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$$

The **erfc()**, **erfcf()** and **erfcfl()** functions calculate the complementary error function of x; that is **erfc()** subtracts the result of the error function `erf(x)` from 1.0. This is useful, since for large x places disappear.

SEE ALSO

math(3)

STANDARDS

The functions conform to ISO/IEC 9899:1999 (“ISO C99”).

EXP(3)

NAME

exp, expf, expl, exp2, exp2f, exp2l, expm1, expm1f, expm1l, pow, powf, powl - exponential and power functions

SYNOPSIS

```
#include <math.h>

double
exp(double x);

float
expf(float x);

long double
expl(long double x);

double
exp2(double x);

float
exp2f(float x);

long double
exp2l(long double x);

double
expm1(double x);

float
expm1f(float x);

float
expm1f(float x);

double
pow(double x, double y);

float
powf(float x, float y);

long double
powl(long double x, long double y);
```

DESCRIPTION

The **exp()**, **expf()** and **expfl()** functions compute the exponential value of the given argument *x*.

The **exp2()**, **exp2f()** and **exp2l()** functions compute the base 2 exponential value of the given argument *x*.

The **expm1()**, **expm1f()** and **expm1l()** functions compute the value `exp(x)-1.0` accurately even for tiny argument *x*.

The **pow()**, **powf()** and **powl()** functions compute the value of *x* to the exponent *y*.

ERROR (due to Roundoff etc.)

exp(), **expm1(0)**, **exp2(integer)** and **pow(integer,integer)** are exact provided they are representable. Otherwise the error in these functions is generally below one *ulp*.

RETURN VALUES

These functions will return the appropriate computation unless an error occurs or an argument is out of range. The functions **exp()**, **expm1()** and **pow()** detect if the computed value will overflow, set the global variable `errno` to `ERANGE`. The function `pow(x, y)` checks to see if $x < 0$ and y is not an integer, in the event this is true, the global variable `errno` is set to `EDOM` for HFP values or return a `NaN` for BFP values.

NOTES

The function `pow(x, 0)` returns $x^0 = 1$ for all x including $x = 0, \infty$ (not found for IBM HFP format) and `NaN` (not found in IBM HFP format).

Previous implementations of **pow()** may have defined x^0 to be undefined in some or all of these cases. Here are reasons for returning $x^0 = 1$:

- Any program that already tests whether x is zero (or infinite or `NaN`) before computing x^0 cannot care whether $0^0 = 1$ or not. Any program that depends upon 0^0 to be invalid is dubious anyway since that expression's meaning and, if invalid, its consequences vary from one computer system to another.

- Some Algebra texts (e.g. Sigler's) define $x^0 = 1$ for all x, including $x = 0$. This is compatible with the convention that accepts a_0 as the value of polynomial

$$p(x) = a_0x^0 + a_1x^1 + a_2x^2 + \cdots + a_nx^n$$

  at $x = 0$ rather than reject a_00^0 as invalid.

- Analysts will accept $0^0 = 1$ despite that x^y can approach anything or nothing as x and y approach 0 independently. The reason for setting $0^0 = 1$ anyway is this:

  > If $x(z)$ and $y(z)$ are any functions analytic (expandable in power series) in z around $z = 0$, and if there $x(0) = y(0) = 0$, then $x(z)^{y(z)} \to 1$ as $z \to 0$.

- If $0^0 = 1$, then $\infty^0 = \frac{1}{0^0} = 1$ too; and then $\texttt{NaN}^0 = 1$ too because $x^0 = 1$ for all finite and infinite x, i.e., independently of x.

SEE ALSO

fenv(3), ldexp(3), log(3), math(3)

STANDARDS

These functions conform to ISO/IEC 9899:1999 ("ISO C99").

FABS(3)

NAME

fabs, fabsf, fabsl – floating-point absolute value functions

SYNOPSIS

```
#include <math.h>

double
fabs(double x);

float
fabsf(float x);

long double
fabsl(long double x);
```

DESCRIPTION

The **fabs()**, **fabsf()** and **fabsl()** functions compute the absolute value of a floating-point number *x*.

RETURN VALUES

The **fabs()**, **fabsf()** and **fabsl()** functions return the absolute value of *x*.

SEE ALSO

abs(3), ceil(3), floor(3), math(3), rint(3)

STANDARDS

The **fabs()** function conforms to ISO/IEC 9899:1990 ("ISO C90"). The **fabsf()** and **fabsl()** functions conform to ISO/IEC 9899:1999 ("ISO C99").

FDIM(3)

NAME

fdim, fdimf, fdiml – positive difference functions

SYNOPSIS

```
#include <math.h>

double
fdim(double x, double y);

float
fdimf(float x, float y);

long double
fdiml(long double x, long double y);
```

DESCRIPTION

The **fdim()**, **fdimf()**, and **fdiml()** functions return the positive difference between *x* and *y*. That is, if *x*-*y* is positive, then *x*-*y* is returned. If either *x* or *y* is a `NaN`, then a `NaN` is returned. Otherwise, the result is +0.0.

Overflow or underflow may occur if and only if the exact result is not representable in the return type. No other exceptions are raised.

SEE ALSO

fabs(3), fmax(3), fmin(3), math(3)

STANDARDS

The **fdim()**, **fdimf()**, and **fdiml()** functions conform to ISO/IEC 9899:1999 ("ISO C99").

FEENABLEEXCEPT(3)

NAME

feenableexcept, fedisableexcept, fegetexcept – floating-point exception masking

SYNOPSIS

```
#include <fenv.h>
#pragma STDC FENV_ACCESS ON

int
feenableexcept(int excepts);

int
fedisableexcept(int excepts);

int
fegetexcept(void);
```

DESCRIPTION

The **feenableexcept()** and **fedisableexcept()** functions unmask and mask (respectively) exceptions specified in excepts. The **fegetexcept()** function returns the current exception mask. All exceptions are masked by default.

Floating-point operations that produce unmasked exceptions will trap, and a SIGFPE will be delivered to the process. By installing a signal han- dler for SIGFPE, applications can take appropriate action immediately without testing the exception flags after every operation. Note that the trap may not be immediate, but it should occur before the next floating- point instruction is executed.

For all of these functions, the possible types of exceptions include those described in fenv(3). Some architectures may define other types of floating-point exceptions.

RETURN VALUES

The **feenableexcept()**, **fedisableexcept()**, and **fegetexcept()** functions return a bitmap of the exceptions that were unmasked prior to the call.

SEE ALSO

sigaction(2), feclearexcept(3), feholdexcept(3), fenv(3), feupdateenv(3)

ISSUES

Functions in the standard library may trigger exceptions multiple times as a result of intermediate computations; however, they generally do not trigger spurious exceptions.

No interface is provided to permit exceptions to be handled in nontrivial ways. There is no uniform way for an exception handler to access information about the exception-causing instruction, or to determine whether that instruction should be reexecuted after returning from the handler.

FEGETROUND(3)

NAME

fegetround, fesetround – floating-point rounding control

SYNOPSIS

```
#include <fenv.h>
#pragma STDC FENV_ACCESS ON

int
fegetround(void);

int
fesetround(int round);
```

DESCRIPTION

The **fegetround()** function determines the current BFP floating-point rounding mode, and the **fesetround()** function sets the current BFP rounding mode to *round*. The rounding mode is one of `FE_TONEAREST`, `FE_DOWNWARD`, `FE_UPWARD`, or `FE_TOWARDZERO`, as described in fenv(3).

This is the rounding mode for BFP (IEEE) arithmetic.

RETURN VALUES

The **fegetround()** routine returns the current rounding mode. The **fesetround()** function returns 0 on success and non-zero otherwise; however, the present implementation always succeeds.

SEE ALSO

fenv(3)

STANDARDS

The **fegetround()** and **fesetround()** functions conform to ISO/IEC 9899:1999 ("ISO C99").

FE_DEC_GETROUND(3)

NAME

fe_dec_getround, fe_dec_setround – Decimal floating-point rounding control

SYNOPSIS

```
#include <fenv.h>

int
fe_dec_getround(void);

int
fe_dec_setround(int round);
```

DESCRIPTION

The **fe_dec_getround()** function determines the current decimal floating-point rounding mode, and the **fe_dec_setround()** function sets the current decimal floating point rounding mode to *round*. The rounding mode is one of `FE_DEC_TONEAREST`, `FE_DEC_TOWARDZERO`, `FE_DEC_UPWARD`, `FE_DEC_DOWNWARD` or `FE_DEC_TONEARESTFROMZERO`.

This is the rounding mode for DFP (Decimal) arithmetic.

RETURN VALUES

The **fe_dec_getround()** routine returns the current rounding mode. The **fe_dec_setround()** function returns 0 on success and non-zero otherwise; however, the present implementation always succeeds.

SEE ALSO

fenv(3)

FLOOR(3)

NAME

floor, floorf, floorl – largest integral value less than or equal to x

SYNOPSIS

```
#include <math.h>

double
floor(double x);

float
floorf(float x);

long double
floorl(long double x);
```

DESCRIPTION

The **floor()**, **floorf()** and **floorl()** functions compute the largest integral value less than or equal to x, expressed as a floating-point number.

SEE ALSO

abs(3), ceil(3), fabs(3), math(3), rint(3), round(3), trunc(3)

STANDARDS

The **floor()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

The **floorf()** and **floorl()** functions conform to ISO/IEC 9899:1999 (“ISO C99”).

FMA(3)

NAME

fma, fmaf, fmal – fused multiply-add

SYNOPSIS

```
#include <math.h>

double
fma(double x, double y, double z);

float
fmaf(float x, float y, float z);

long double
fmal(long double x, long double y, long double z);
```

DESCRIPTION

The **fma()**, **fmaf()**, and **fmal()** functions return (x * y) + z, computed with only one rounding error. Using the ordinary multiplication and addition operators, by contrast, results in two roundings: one for the intermediate product and one for the final result.

For instance, the expression 1.2e100 * 2.0e208 - 1.4e308 produces infinity due to overflow in the intermediate product, whereas fma(1.2e100, 2.0e208, -1.4e308) returns approximately 1.0e308 (for IEEE values.)

The fused multiply-add operation is often used to improve the accuracy of calculations such as dot products. It may also be used to improve performance on machines that implement it natively. The macros `FP_FAST_FMA`, `FP_FAST_FMAF` and `FP_FAST_FMAL` may be defined in ¡math.h¿ to indicate that **fma()**, **fmaf()**, and **fmal()** (respectively) have comparable or faster speed than a multiply operation followed by an add operation.

SEE ALSO

fenv(3), math(3)

STANDARDS

The **fma()**, **fmaf()**, and **fmal()** functions conform to ISO/IEC 9899:1999 ("ISO C99"). A fused multiply-add operation with virtually identical characteristics appears in IEEE draft standard 754R.

FMAX(3)

NAME

fmax, fmaxf, fmaxl, fmin, fminf, fminl – floating-point maximum and minimum functions

SYNOPSIS

```
#include <math.h>

double
fmax(double x, double y);

float
fmaxf(float x, float y);

long double
fmaxl(long double x, long double y);

double
fmin(double x, double y);

float
fminf(float x, float y);

long double
fminl(long double x, long double y);
```

DESCRIPTION

The **fmax()**, **fmaxf()**, and **fmaxl()** functions return the larger of x and y, and likewise, the **fmin()**, **fminf()**, and **fminl()** functions return the smaller of x and y. They treat +0.0 as being larger than -0.0. If one argument is a `NaN`, then the other argument is returned. If both arguments are `NaN`s, then the result is a `NaN`. These routines do not raise any floating-point exceptions.

SEE ALSO

fabs(3), fdim(3), math(3)

STANDARDS

The **fmax()**, **fmaxf()**, **fmaxl()**, **fmin()**, **fminf()**, and **fminl()** functions conform to ISO/IEC 9899:1999 ("ISO C99").

FMOD(3)

NAME

fmod, fmodf, fmodl – floating-point remainder functions

SYNOPSIS

```
#include <math.h>

double
fmod(double x, double y);

float
fmodf(float x, float y);

long double
fmodl(long double x, long double y);
```

DESCRIPTION

The **fmod()**, **fmodf()** and **fmodl()** functions compute the floating-point remainder of $\frac{x}{y}$.

RETURN VALUES

The **fmod()**, **fmodf()** and **fmodl()** functions return the value $x - i * y$ for some integer i such that, if y is non-zero, the result has the same sign as x and magnitude less than the magnitude of y. If y is zero, whether a domain error occurs or the **fmod()** function returns zero is implementation-defined and depends on the use of IEEE or HFP values.

SEE ALSO

math(3)

STANDARDS

The **fmod()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

The **fmodf()**, and **fmodl()** functions conform to ISO/IEC 9899:1999 (“ISO C99”).

FPCLASSIFY(3)

name

fpclassify, isfinite, isinf, isnan, isnormal - classify a floating-point number

SYNOPSIS

```
#include <math.h>

int
fpclassify(real-floating x);

int
isfinite(real-floating x);

int
isinf(real-floating x);

int
isnan(real-floating x);

int
isnormal(real-floating x);
```

DESCRIPTION

The **fpclassify()** macro takes an argument of x and returns one of the following manifest constants.

`FP_INFINITE`	Indicates that x is an infinite number.
`FP_NAN`	Indicates that x is not a number (NaN).
`FP_NORMAL`	Indicates that x is a normalized number.
`FP_SUBNORMAL`	Indicates that x is a denormalized number.
`FP_ZERO`	Indicates that x is zero (0 or -0).

The **isfinite()** macro returns a non-zero value if and only if its argument has a finite (zero, subnormal, or normal) value. The **isinf()**, **isnan()**, and **isnormal()** macros return non-zero if and only if x is an infinity, `NaN`, or a non-zero normalized number, respectively. Note that HFP values do not support infinity or `NaN`.

SEE ALSO

isgreater(3), math(3), signbit(3)

STANDARDS

The **fpclassify()**, **isfinite()**, **isinf()**, **isnan()**, and **isnormal()** macros conform to ISO/IEC 9899:1999 ("ISO C99").

FREXP(3)

NAME

frexp, frexpf, frexpl – convert floating-point number to fractional and integral components

SYNOPSIS

```
#include <math.h>

double
frexp(double value, int *exp);

float
frexpf(float value, int *exp);

long double
frexpl(long double value, int *exp);
```

DESCRIPTION

The **frexp()**, **frexpf()** and **frexpl()** functions break a floating-point number into a normalized fraction and an integral power of 2. They store the integer in the `int` object pointed to by *exp*.

RETURN VALUES

These functions return the value x, such that x is a `double` with magnitude in the interval [1/2, 1) or zero, and *value* equals x times 2 raised to the power ***exp**. If *value* is zero, both parts of the result are zero.

SEE ALSO

ldexp(3) math(3), modf(3)

STANDARDS

The **frexp()**, **frexpf()** and **frexpl()** functions conform to ISO/IEC 9899:1999 ("ISO C99").

HYPOT(3)

NAME

hypot, hypotf, hypotl - euclidean distance functions

SYNOPSIS

```
#include <math.h>

double
hypot(double x, double y);

float
hypotf(float x, float y);

long double
hypotl(long double x, long double y);
```

DESCRIPTION

The **hypot()**, **hypotf()** and **hypotl()** functions compute $\sqrt{x^2 + y^2}$ in such a way that underflow will not happen, and overflow occurs only if the final result deserves it.

hypot(∞, v) = **hypot**$(v, \infty) = +\infty$ for all v, including `NaN`. (∞ and `NAN` are not found in the IBM HFP format).

SEE ALSO

math(3), sqrt(3)

STANDARDS

The **hypot()**, **hypotf() hypotl()** functions conform to ISO/IEC 9899:1999 ("ISO C99").

ILOGB(3)

NAME

ilogb, ilogbf, ilogbl, logb, logbf, logbl – extract exponent

SYNOPSIS

```
#include <math.h>

int
ilogb(double x);

int
ilogbf(float x);

int
ilogbl(long double x);

double
logb(double x);

float
logbf(float x);

long double
logbl(long double x);
```

DESCRIPTION

ilogb(), **ilogbf()** and **ilogbl()** return x's exponent in integer format. For BFP (IEEE) values ilogb(+-$infinity$) returns `INT_MAX`, ilogb(+-`NaN`) returns `FP_ILOGBNAN` . ilogb(0) returns `FP_ILOGB0`.

logb(), **logbf()**, and **logbl()** return x's exponent in floating-point format with the same precision as x. For BFP (IEEE) values logb(+-infinity) returns +infinity. If x is +/-0, **logb()**, **logbf()**, and **logbl()** return `-HUGE_VAL`, `-HUGE_VALF` and `-HUGE_VALL` respectively.

BFP values have a radix of 2 and HFP values have a radix of 16; the return exponent values are in the radix of the format (`FLT_RADIX` defined in `<float.h>`).

SEE ALSO

frexp(3), math(3), scalbn(3)

STANDARDS

The **ilogb()**, **ilogbf()**, **ilogbl()**, **logb()**, **logbf()**, and **logbl()** routines conform to ISO/IEC 9899:1999 ("ISO C99").

ISGREATER(3)

NAME

isgreater, isgreaterequal, isless, islessequal, islessgreater, isunordered – compare two floating-point numbers

SYNOPSIS

```
#include <math.h>

int
isgreater(real-floating x, real-floating y);

int
isgreaterequal(real-floating x, real-floating y);

int
isless(real-floating x, real-floating y);

int
islessequal(real-floating x, real-floating y);

int
islessgreater(real-floating x, real-floating y);

int
isunordered(real-floating x, real-floating y);
```

DESCRIPTION

Each of the macros **isgreater()**, **isgreaterequal()**, **isless()**, **islessequal()**, and **islessgreater()** take arguments x and y and return a non-zero value if and only if its nominal relation on x and y is true. These macros always return zero if either argument is not a number (`NaN`), but unlike the corresponding C operators, they never raise a floating point exception.

The **isunordered()** macro takes arguments x and y and returns non-zero if and only if neither x nor y are `NaN`s. For any pair of floating-point values, one of the relationships (less, greater, equal, unordered) holds.

Note that HFP floating-point values do not support `NaN`, therefor **isunordered()** is always false for HFP values.

SEE ALSO

fpclassify(3), math(3), signbit(3)

STANDARDS

The **isgreater()**, **isgreaterequal()**, **isless()**, **islessequal()**, **islessgreater()**, and **isunordered()** macros conform to ISO/IEC 9899:1999 ("ISO C99").

LDEXP(3)

NAME

ldexp, ldexpf, ldexpl - multiply floating-point number by integral power of 2

SYNOPSIS

```
#include <math.h>

double
ldexp(double x, int exp);

float
ldexpf(float x, int exp);

long double
ldexpl(long double x, int exp);
```

DESCRIPTION

The **ldexp()**, **ldexpf()**, and **ldexpl()** functions multiply a floating-point number by an integral power of 2.

RETURN VALUES

These functions return the value of *x* times 2 raied to the power *exp*.

SEE ALSO

frexp(3), math(3), modf(3)

STANDARDS

The **ldexp()**, **ldexpf()** and **ldexpl()** functions conform to ISO/IEC 9899:1999 ("ISO C99").

LGAMMA(3)

NAME

lgamma, lgamma_r, lgammaf, lgammaf_r, lgammal, lgammal_r, tgamma, tgammaf, tgammal - log gamma functions, gamma functions

SYNOPSIS

```
#include <math.h>

extern int signgam;

double
lgamma(double x);

double
lgamma_r(double x, int *signgamp);

float
lgammaf(float x);

float
lgammaf_r(float x, int *signgamp);

double
tgamma(double x);

float
tgammaf(float x);

long double
tgammal(long double x);
```

DESCRIPTION

`lgamma(`x`)`, `lgammaf(`x`)`, and `lgammal(`x`)` functions return $\ln|\Gamma(x)|$.

The external integer `signgam` returns the sign of $\Gamma(x)$.

`lgamma_r(`x`,`*signgamp*`)`, `lgammaf_r(`x`,`*signgamp*`)`, and `lgammal_r(`x`,`*signgamp*`)` provide the same functionality as `lgamma(`x`)`, `lgammaf)`x`)` and `lgammal(`x`)`but the caller must provide an integer to store the sign of $\Gamma(x)$.

The `tgamma(`x`)`, `tgammaf(`x`)`, and `tgammal(`x`)` functions return $\Gamma(x)$ with no effect on `signgam`.

IDIOSYNCRASIES

Do not use the expression "`g = signgam*exp(lgamma(x))`" to compute $g = \Gamma(x)$. Instead use a program like this (in C):

```
lg = lgamma(x); g = signgam*exp(lg);
```

Only after **lgamma()** has returned can `signgam` be correct.

For arguments in its range **tgamma()** is preferred, as for positive arguments it is accurate to within one unit in the last place. Exponentiation of **lgamma()** will lose up to 10 significant bits.

RETURN VALUES

tgamma(), **tgammaf()**, **tgammal()**, **lgamma()**, **lgammaf()**, **lgammal()**, **lgamma_r()**, **lgammaf_r()**, **lgammal_r()**, return appropriate values unless an argument is out of range. Overflow will occur for sufficiently large positive values, and non-positive integers. For large non-integer negative values, **tgamma()**, **tgammaf()** and **tgammal()** will underflow.

SEE ALSO

math(3)

STANDARDS

The **lgamma()**, **lgammaf()**, **lgammal()**, **tgamma()**, **tgammaf()** and **tgammal()** functions are expected to conform to ISO/IEC 9899:1999 ("ISO C99").

LOG(3)

NAME

log, logf, logl, log10, log10f, log10l, log1p, log1pf, log1pl, log2, log2f, log2l - logarithm functions

SYNOPSIS

```
#include <math.h>

double
log(double x);

float
logf(float x);

long double
logl(long double x);

double
log10(double x);

float
log10f(float x);

long double
log10l(long double x);

double
log1p(double x);

float
log1pf(float x);

long double
log1pl(long double x);

double
log2(double x);

float
log2f(float x);

long double
log2l(long double x);
```

DESCRIPTION

The **log()**, **logf()** and **logl()** functions compute the natural logarithm of *x*.

The **log10()**, **log10f()** and **log10l()** functions compute the logarithm base 10 of *x*.

The **log1p()**, **log1pf()** and **log1pl()** functions compute the natural logarithm of $1+x$. Computing the natural logarithm as log1p(x) is more accurate than computing it as log(1 + x) when *x* is close to zero.

The **log2()**, **log2f()** and **log2l()** functions compute the logarithm base 2 of *x*.

RETURN VALUES

These functions return the requested logarithm; the logarithm of 1 is +0. An attempt to take the logarithm of +-0 results in a divide-by-zero exception, and `-HUGE_VAL` is returned. Otherwise, attempting to take the logarithm of a negative number results in an invalid exception and a return value of NaN for BFP values and 0 for HFP values.

SEE ALSO

exp(3), ilogb(3), math(3), pow(3)

STANDARDS

The **log()**, **logf()**, **logl()**, **log10()**, **log10f()**, **log10l()**, **log1p()**, **log1pf()**, **log1pl()**, **log2()**, **log2f()** and **log2l()** functions conform to ISO/IEC 9899:1999 ("ISO C99").

LRINT(3)

NAME

llrint, llrintf, llrintl, lrint, lrintf, lrintl - convert to integer

SYNOPSIS

```
#include <math.h>

long long
llrint(double x);

long long
llrintf(float x);

long long
llrintl(long double x);

long
lrint(double x);

long
lrintf(float x);

long
lrintl(long double x);
```

DESCRIPTION

The **lrint()** function returns the integer nearest to its argument x according to the current rounding mode. If the rounded result is too large to be represented as a long value, an invalid exception is raised and the return value is undefined. Otherwise, if x is not an integer, **lrint()** raises an inexact exception. When the rounded result is representable as a long, the expression

```
lrint(x)
```

is equivalent to

```
(long)rint(x)
```

(although the former may be more efficient).

The **llrint()**, **llrintf()**, **llrintl()**, **lrintf()**, and **lrintl()** functions differ from **lrint()** only in their input and output types.

SEE ALSO

lround(3), math(3), rint(3), round(3)

STANDARDS

These functions conform to ISO/IEC 9899:1999 ("ISO C99").

LROUND(3)

NAME

llround, llroundf, llroundl, lround, lroundf, lroundl - convert to nearest integral value

SYNOPSIS

```
#include <math.h>

long long
llround(double x);

long long
llroundf(float x);

long long
llroundl(long double x);

long
lround(double x);

long
lroundf(float x);

long
lroundl(long double x);
```

DESCRIPTION

The **lround()** function returns the integer nearest to its argument x, rounding away from zero in halfway cases. If the rounded result is too large to be represented as a long value, an invalid exception is raised and the return value is undefined. Otherwise, if x is not an integer, **lround()** may raise an inexact exception. When the rounded result is representable as a long, the expression

```
lround(x)
```

is equivalent to

```
(long)round(x)
```

(although the former may be more efficient).

The **llround()**, **llroundf()**, **llroundl()**, **lroundf()** and **lroundl()** functions differ from **lround()** only in their input and output types.

SEE ALSO

lrint(3), math(3), rint(3), round(3)

STANDARDS

The **llround()**, **llroundf()**, **llroundl()**, **lround()**, **lroundf()**, and **lroundl()** functions conform to ISO/IEC 9899:1999 ("ISO C99").

MODF(3)

NAME

modf, modff, modfl - extract signed integral and fractional values from floating-point number

SYNOPSIS

```
#include <math.h>

double
modf(double value, double *iptr);

float
modff(float value, float *iptr);

long double
modfl(long double value, long double *iptr);
```

DESCRIPTION

The **modf()**, **modff()**, and **modfl()** functions break the argument value into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part as a floating point number in the object pointed to by *iptr*.

RETURN VALUES

These functions return the signed fractional part of *value*.

SEE ALSO

frexp(3), ldexp(3), math(3)

STANDARDS

The **modf()**, **modff()**, and **modfl()** functions conform to ISO/IEC 9899:1999 ("ISO C99").

NAN(3)

NAME

nan, nanf, nanl - quiet NaNs

SYNOPSIS

```
#include <math.h>

double
nan(const char *s);

float
nanf(const char *s);

long double
nanl(const char *s);
```

DESCRIPTION

The `NAN` macro expands to a quiet NaN (Not A Number). Similarly, each of the **nan()**, **nanf()** and **nanl()** functions generate a quiet `NaN` value without raising an invalid exception. The argument *s* should point to either an empty string or a hexadecimal representation of a non-negative 32-bit integer (e.g., "0x1234".) In the latter case, the integer is encoded in some free bits in the representation of the `NaN`, which sometimes store machine-specific information about why a particular `NaN` was generated. There are 22 such bits available for float variables, 51 bits for double variables, and at least 51 bits for a long double. If *s* is improperly formatted or represents an integer that is too large, then the particular encoding of the quiet `NaN` that is returned is indeterminate.

Only BFP floating-point supports `NaN` values. When HFP is enabled, these functions return 0.0.

COMPATIBILITY

Calling these functions with a non-empty string isn't portable. Another implementation may translate the string into a different `NaN` encoding, and furthermore, the meaning of a given `NaN` encoding varies across machine architectures and implementations.

SEE ALSO

fenv(3), isnan(3), math(3), strtod(3)

STANDARDS

The **nan()**, **nanf()**, and **nanl()** functions and the `NAN` macro conform to ISO/IEC 9899:1999 ("ISO C99").

NEXTAFTER(3)

NAME

nextafter, nextafterf, nextafterl, nexttoward, nexttowardf, nexttowardl - next representable value

SYNOPSIS

```
#include <math.h>

double
nextafter(double x, double y);

float
nextafterf(float x, float y);

long double
nextafterl(long double x, long double y);

double
nexttoward(double x, long double y);

float
nexttowardf(float x, long double y);

long double
nexttowardl(long double x, long double y);
```

DESCRIPTION

These functions return the next machine representable number from x in direction y. The returned value may not be normalized, for either HFP or BFP.

SEE ALSO

math(3)

STANDARDS

The **nextafter()**, **nextafterf()**, **nextafterl()**, **nexttoward()**, **nexttowardf()**, and **nexttowardl()** routines conform to ISO/IEC 9899:1999 ("ISO C99").

REMAINDER(3)

NAME

remainder, remainderf, remainderl, remquo, remquof, remquol - minimal residue functions

SYNOPSIS

```
#include <math.h>

double
remainder(double x, double y);

float
remainderf(float x, float y);

long double
remainderl(long double x, long double y);

double
remquo(double x, double y, int *quo);

float
remquof(float x, float y, int *quo);

long double
remquol(long double x, long double y, int *quo);
```

DESCRIPTION

remainder(), **remainderf()**, **remainderl()**, **remquo()**, **remquof()**, and **remquol()** return the remainder r := x - n*y where n is the integer nearest the exact value of x/y; moreover if —n - x/y— = 1/2 then n is even. Consequently the remainder is computed exactly and —r— ¡= —y—/2. But attempting to take the remainder when *y* is 0 or *x* is +-infinity is an invalid operation that produces a `NaN`.

The **remquo()**, **remquof()**, and **remquol()** functions also store the last k bits of n in the location pointed to by *quo*, provided that n exists. The number of bits k is platform-specific, but is guaranteed to be at least 3.

SEE ALSO

fmod(3), math(3)

STANDARDS

The **remainder()**, **remainderf()**, **remainderl()**, **remquo()**, **remquof()**, and **remquol()** routines conform to ISO/IEC 9899:1999 ("ISO C99").

RINT(3)

NAME

nearbyint, nearbyintf, nearbyintl, rint, rintf, rintl - round to integral value in floating-point format

SYNOPSIS

```
#include <math.h>

double
nearbyint(double x);

float
nearbyintf(float x);

long double
nearbyintl(long double x);

double
rint(double x);

float
rintf(float x);

long double
rintl(long double x);
```

DESCRIPTION

The **rint()**, **rintf()**, and **rintl()** functions return the integral value nearest to *x*. For BFP values, the rounding is performed according to the prevailing rounding mode. For HFP values, the rounding mode is always round-toward-zero. These functions raise an inexact exception when the original argument is not an exact integer.

The **nearbyint()**, **nearbyintf()**, and **nearbyintl()** functions perform the same operation, except that they do not raise an inexact exception.

SEE ALSO

abs(3), ceil(3), fabs(3), fenv(3), floor(3), lrint(3), lround(3), math(3), round(3)

STANDARDS

These functions conform to ISO/IEC 9899:1999 (“ISO C99”).

ROUND(3)

NAME

round, roundf, roundl - round to nearest integral value

SYNOPSIS

```
#include <math.h>

double
round(double x);

float
roundf(float x);

long double
roundl(long double x);
```

DESCRIPTION

The **round()**, **roundf()**, and **roundl()** functions return the nearest integral value to x; if x lies halfway between two integral values, then these functions return the integral value with the larger absolute value (i.e., they round away from zero).

SEE ALSO

ceil(3), floor(3), lrint(3), lround(3), math(3), rint(3), trunc(3)

STANDARDS

These functions conform to ISO/IEC 9899:1999 (“ISO C99”).

SIGNBIT(3)

NAME

signbit - determine whether a floating-point number's sign is negative

SYNOPSIS

```
#include <math.h>

int
signbit(real-floating x);
```

DESCRIPTION

The **signbit()** macro takes an argument of *x* and returns non-zero if the value of its sign is negative, otherwise 0.

SEE ALSO

fpclassify(3), math(3)

STANDARDS

The **signbit()** macro conforms to ISO/IEC 9899:1999 ("ISO C99").

SIN(3)

NAME

sin, sinf, sinl - sine functions

SYNOPSIS

```
#include <math.h>

double
sin(double x);

float
sinf(float x);

long double
sinl(long double x);
```

DESCRIPTION

The **sin()**, **sinf()** and **sinl()** functions compute the sine of x (measured in radians). A large magnitude argument may yield a result with little or no significance.

RETURN VALUES

The **sin()**, **sinf()** and **sinl()** functions return the sine value.

SEE ALSO

acos(3), asin(3), atan(3), atan2(3), cos(3), cosh(3), math(3), sinh(3), tan(3), tanh(3)

STANDARDS

The **sin()** function conforms to ISO/IEC 9899:1990 ("ISO C90").

The **sinf()** and **sinl()** functions conform to ISO/IEC 9899:1999 ("ISO C99").

SINH(3)

NAME

sinh, sinhf, sinhl - hyperbolic sine function

SYNOPSIS

```
#include <math.h>

double
sinh(double x);

float
sinhf(float x);

long double
sinhl(long double x);
```

DESCRIPTION

The **sinh()**, **sinhf()** and **sinhl()** functions computes the hyperbolic sine of x.

RETURN VALUES

The **sinh()**, **sinhf()** and **sinhl()** functions return the hyperbolic sine value.

SEE ALSO

acos(3), asin(3), atan(3), atan2(3), cos(3), cosh(3), math(3), sin(3), tan(3), tanh(3)

STANDARDS

The **sinh()** function conforms to ISO/IEC 9899:1990 ("ISO C90").

The **sinhf()** and **sinhl()** functions conform to ISO/IEC 9899:1999 ("ISO C99").

SQRT(3)

NAME

cbrt, cbrtf, sqrt, sqrtf, sqrtl – cube root and square root functions

SYNOPSIS

```
#include <math.h>

double
cbrt(double x);

float
cbrtf(float x);

double
sqrt(double x);

float
sqrtf(float x);

long double
sqrtl(long double x);
```

DESCRIPTION

The **cbrt()**, **cbrtf()** and **cbrtl()** functions compute the cube root of x.

The **sqrt()**, **sqrtf()** and **sqrtl()** functions compute the non-negative square root of x.

RETURN VALUES

The **cbrt()**, **cbrtf()** and **cbrtl()** functions return the requested cube root. The **sqrt()**, **sqrtf()** and **sqrtl()** functions return the requested square root unless an error occurs. An attempt to take the square root of a negative x causes an error; in this event, the global variable `errno` is set to `EDOM` and 0.0 is returned, or for BFP values a `NaN` is returned.

SEE ALSO

fenv(3), math(3)

STANDARDS

The **sqrt()** function conforms to ISO/IEC 9899:1990 ("ISO C90").

The **cbrt()**, **cbrtf()**, **cbrtl()**, **sqrtf()** and **sqrtl()** functions conforms to ISO/IEC 9899:1999 ("ISO C99").

TAN(3)

NAME

tan, tanf, tanl - tangent functions

SYNOPSIS

```
#include <math.h>

double
tan(double x);

float
tanf(float x);

long double
tanl(long double x);
```

DESCRIPTION

The **tan()**, **tanf()** and **tanl()** functions compute the tangent of x (measured in radians). A large magnitude argument may yield a result with little or no significance.

RETURN VALUES

The **tan()**, **tanf()** and **tanl()** functions return the tangent value.

SEE ALSO

acos(3), asin(3), atan(3), atan2(3), cos(3), cosh(3), math(3), sin(3), sinh(3), tanh(3)

STANDARDS

The **tan()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

The **tanf()** and **tanl()** functions conform to ISO/IEC 9899:1999 (“ISO C99”).

STANDARDS

TANH(3)

NAME

tanh, tanhf, tanhl - hyperbolic tangent function

SYNOPSIS

```
#include <math.h>

double
tanh(double x)

float
tanhf(float x)

long double
tanhl(long double x)
```

DESCRIPTION

The **tanh()**, **tanhf()** and **tanhl()** functions compute the hyperbolic tangent of *x*.

RETURN VALUES

The **tanh()**, **tanhf()** and **tanhl()** functions return the hyperbolic tangent value.

SEE ALSO

acos(3), asin(3), atan(3), atan2(3), cos(3), cosh(3), math(3), sin(3), sinh(3), tan(3)

STANDARDS

The **tanh()** function conforms to ISO/IEC 9899:1990 ("ISO C90").

The **tanhf()** and **tanhl()** functions conform to ISO/IEC 9899:1999 ("ISO C99").

TRUNC(3)

NAME

trunc, truncf, truncl - nearest integral value with magnitude less than or equal to $|x|$

SYNOPSIS

```
#include <math.h>

double
trunc(double x);

float
truncf(float x);

long double
truncl(long double x);
```

DESCRIPTION

The **trunc()**, **truncf()** and **truncl()** functions return the nearest integral value with magnitude less than or equal to $|x|$. They are equivalent to **rint()**, **rintf()**, and **rintl()**, respectively, in the `FE_TOWARDZERO` rounding mode.

SEE ALSO

ceil(3), fesetround(3), floor(3), math(3), nextafter(3), rint(3), round(3)

STANDARDS

The **trunc()**, **truncf()**, and **truncl()** functions conform to ISO/IEC 9899:1999 ("ISO C99").

Standard I/O Library

The ANSI C standard provides for a set of input and output functions known as the Standard I/O Library.

STDIO(3)

NAME

stdio - standard input/output library functions

SYNOPSIS

```
#include <stdio.h>
FILE *stdin;
FILE *stdout;
FILE *stderr;
```

DESCRIPTION

The standard I/O library provides a simple and efficient buffered stream I/O interface. Input and output is mapped into logical data streams and the physical I/O characteristics are concealed. The functions and macros are listed below; more information is available from the individual man pages.

A stream is associated with an external file (which may be a physical device) by opening a file, which may involve creating a new file. Creating an existing file causes its former contents to be discarded. If a file can support positioning requests (such as a disk file, as opposed to a terminal) then a file position indicator associated with the stream is positioned at the start of the file (byte zero), unless the file is opened with append mode. If append mode is used, the position indicator will be placed the end-of-file. The position indicator is maintained by subsequent reads, writes and positioning requests. All input occurs as if the characters were read by successive calls to the fgetc(3) function; all output takes place as if all characters were read by successive calls to the fputc(3) function.

A file is disassociated from a stream by closing the file. Output streams are flushed (any unwritten buffer contents are transferred to the host environment) before the stream is disassociated from the file. The value of a pointer to a FILE object is indeterminate after a file is closed (garbage).

A file may be subsequently reopened, by the same or another program execution, and its contents reclaimed or modified (if it can be repositioned at the start). If the main function returns to its original caller, or the exit(3) function is called, all open files are closed (hence all output streams are flushed) before program termination. Other methods of program termination such as abort(3) do not bother to close files properly.

This implementation makes a distinction between “text” and “binary” streams. Records written in non-binary mode are padded after a new-line is written to fill

the record. If no new-line is written before the record length is exhausted, the record will be "split". That is, when the record length is reached, the record will be written and a new one started.

At program startup, three streams are predefined and need not be opened explicitly:

- standard input (for reading conventional input)
- standard output (for writing conventional output) and
- standard error (for writing diagnostic output).

These streams are abbreviated `stdin`, `stdout` and `stderr`. Initially, the standard error stream is unbuffered, the standard input and output streams are fully buffered if and only if the streams do not refer to an interactive or "terminal" device, as determined by the isatty(3) function. In fact, all freshly-opened streams that refer to terminal device default to line buffering, and pending output to such streams is written automatically whenever an such an input stream is read. Note that this applies only to "true reads"; if the read request can be satisfied by existing buffered data, no automatic flush will occur. In these cases, or when a large amount of computation is done after printing part of a line on an output terminal, it is necessary to fflush(3) the standard output before going off and computing so that the output will appear.

Alternatively, these defaults may be modified via the setvbuf(3) function.

The SYNOPSIS sections of the following information indicate which include files are to be used, what the compiler declaration for the function looks like and which external variables are of interest.

The following are defined as macros; these names may not be re-used without first removing their current definitions with `#undef`: `BUFSIZ`, `EOF`, `FILENAME_MAX`, `FOPEN_MAX`, `L_cuserid`, `L_ctermid`, `L_tmpnam`, `NULL`, `SEEK_END`, `SEEK_SET`, `SEEK_CUR`, `TMP_MAX`, `clearerr`, `feof`, `ferror`, `fileno`, `reopen`, `fwopen`, `getc`, `getchar`, `putc`, `putchar`, `stderr`, `stdin`, `stdout`. Function versions of the macro functions **feof()**, **ferror()**, **clearerr()**, **fileno()**, **getc()**, **getchar()**, **putc()**, and **putchar()** exist and will be used if the macros definitions are explicitly removed.

SEE ALSO

close(2), open(2), read(2), write(2)

ISSUES

The standard buffered functions do not interact well with certain other library and system functions, especially abort(3).

STANDARDS

The stdio library conforms to ISO/IEC 9899:1990 (“ISO C90”).

LIST OF FUNCTIONS

Function	Description
clearerr	check and reset stream status
fclose	close a stream
fdopen	stream open functions
feof	check and reset stream status
ferror	check and reset stream status
fflush	flush a stream
fgetc	get next character or word from input stream
fgetln	get a line from a stream
fgetpos	reposition a stream
fgets	get a line from a stream
fileno	check and reset stream status
fopen	stream open functions
fprintf	formatted output conversion
fpurge	flush a stream
fputc	output a character or word to a stream
fputs	output a line to a stream
fread	binary stream input/output
freopen	stream open functions
fropen	open a stream
fscanf	input format conversion
fseek	reposition a stream
fsetpos	reposition a stream
ftell	reposition a stream
funopen	open a stream

fwopen	open a stream
fwrite	binary stream input/output
getc	get next character or word from input stream
getchar	get next character or word from input stream
getdelim	get a line from a stream
getline	get a line from a stream
gets	get a line from a stream
getw	get next character or word from input stream
mkstemp	create unique temporary file
mktemp	create unique temporary file
perror	system error messages
printf	formatted output conversion
putc	output a character or word to a stream
putchar	output a character or word to a stream
puts	output a line to a stream
putw	output a character or word to a stream
remove	remove directory entry
rewind	reposition a stream
scanf	input format conversion
setbuf	stream buffering operations
setbuffer	stream buffering operations
setlinebuf	stream buffering operations
setvbuf	stream buffering operations
snprintf	formatted output conversion
sprintf	formatted output conversion
sscanf	input format conversion
strerror	system error messages
sys_errlist	system error messages
sys_nerr	system error messages

tempnam	temporary file routines
tmpfile	temporary file routines
tmpnam	temporary file routines
ungetc	un-get character from input stream
vfprintf	formatted output conversion
vfscanf	input format conversion
vprintf	formatted output conversion
vscanf	input format conversion
vsnprintf	formatted output conversion
vsprintf	formatted output conversion
vsscanf	input format conversion

FCLOSE(3)

NAME

fclose - close a stream

SYNOPSIS

```
#include <stdio.h>

int
fclose(FILE *stream)
```

DESCRIPTION

The **fclose()** function dissociates the named stream from its underlying file or set of functions. If the stream was being used for output, any buffered data is written first, using fflush(3).

RETURN VALUES

Upon successful completion 0 is returned. Otherwise, `EOF` is returned and the global variable `errno` is set to indicate the error. In either case no further access to the stream is possible.

ERRORS

`[EBADF]` The argument stream is not an open stream.

The **fclose()** function may also fail and set `errno` for any of the errors specified for the routines close(2) or fflush(3).

SEE ALSO

close(2), fflush(3), fopen(3), setbuf(3)

STANDARDS

The **fclose()** function conforms to ISO/IEC 9899:1990 ("ISO C90").

FERROR(3)

NAME

clearerr, feof, ferror, fileno - check and reset stream status

SYNOPSIS

```
#include <stdio.h>

void
clearerr(FILE *stream)

int
feof(FILE *stream)

int
ferror(FILE *stream)

int
fileno(FILE *stream)
```

DESCRIPTION

The function **clearerr()** clears the end-of-file and error indicators for the stream pointed to by stream.

The function **feof()** tests the end-of-file indicator for the stream pointed to by stream, returning non-zero if it is set. The end-of-file indicator can only be cleared by the function **clearerr()**.

The function **ferror()** tests the error indicator for the stream pointed to by stream, returning non-zero if it is set. The error indicator can only be reset by the **clearerr()** function.

The function **fileno()** examines the argument stream and returns its integer descriptor.

ERRORS

These functions should not fail and do not set the external variable `errno`.

SEE ALSO

open(2), stdio(3)

STANDARDS

The functions **clearerr()**, **feof()**, and **ferror()** conform to ISO/IEC 9899:1990 ("ISO C").

FFLUSH(3)

NAME

fflush, fpurge - flush a stream

SYNOPSIS

```
#include <stdio.h>

int
fflush(FILE *stream)

int
fpurge(FILE *stream)
```

DESCRIPTION

The function **fflush()** forces a write of all buffered data for the given output or update stream via the stream's underlying write function. The open status of the stream is unaffected.

If the *stream* argument is `NULL`, **fflush()** flushes all open output streams.

The function **fpurge()** erases any input or output buffered in the given stream. For output streams this discards any unwritten output. For input streams this discards any input read from the underlying object but not yet obtained via getc(3). This includes any text pushed back via ungetc.

RETURN VALUES

Upon successful completion 0 is returned. Otherwise, `EOF` is returned and the global variable `errno` is set to indicate the error.

ERRORS

`[EBADF]` *Stream* is not an open stream, or, in the case of **fflush()**, not a stream open for writing.

The function **fflush()** may also fail and set `errno` for any of the errors specified for the routine write(2).

SEE ALSO

write(2), fclose(3), fopen(3), setbuf(3)

STANDARDS

The **fflush()** function conforms to ISO/IEC 9899:1990 ("ISO C90").

FGETLN(3)

NAME

fgetln - get a line from a stream

SYNOPSIS

```
#include <stdio.h>

char *
fgetln(FILE *stream, size_t *len)
```

DESCRIPTION

The **fgetln()** function returns a pointer to the next line from the stream referenced by *stream*. This line is not a C string as it does not end with a terminating `NUL` character. The length of the line, including the final newline, is stored in the memory location to which *len* points. (Note, however, that if the line is the last in a file that does not end in a newline, the returned text will not contain a newline.)

RETURN VALUES

Upon successful completion a pointer is returned; this pointer becomes invalid after the next I/O operation on *stream* (whether successful or not) or as soon as the stream is closed. Otherwise, `NULL` is returned. The **fgetln()** function does not distinguish between end-of-file and error; the routines feof(3) and ferror(3) must be used to determine which occurred. If an error occurs, the global variable `errno` is set to indicate the error. The end-of-file condition is remembered, even on a terminal, and all subsequent attempts to read will return `NULL` until the condition is cleared with clearerr(3).

The text to which the returned pointer points may be modified provided that no changes are made beyond the returned size. These changes are lost as soon as the pointer becomes invalid.

ERRORS

`[EBADF]` The argument *stream* is not a stream open for reading.

The **fgetln()** function may also fail and set `errno` for any of the errors specified for the routines fflush(3), mallow(3), read(2), stat(2), or realloc(3).

SEE ALSO

ferror(3), fgets(3), fopen(3), putc(3)

FGETWLN(3)

NAME

fgetwln - get a line of wide characters from a stream

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

wchar_t *
fgetwln(FILE * restrict stream, size_t * restrict len)
```

DESCRIPTION

The **fgetwln()** function returns a pointer to the next line from the stream referenced by *stream*. This line is not a standard wide character string as it does not end with a terminating null wide character. The length of the line, including the final newline, is stored in the memory location to which *len* points. (Note, however, that if the line is the last in a file that does not end in a newline, the returned text will not contain a newline.)

RETURN VALUES

Upon successful completion a pointer is returned; this pointer becomes invalid after the next I/O operation on *stream* (whether successful or not) or as soon as the stream is closed. Otherwise, `NULL` is returned. The **fgetwln()** function does not distinguish between end-of-file and error; the routines feof(3) and ferror(3) must be used to determine which occurred. If an error occurs, the global variable `errno` is set to indicate the error. The end-of-file condition is remembered, even on a terminal, and all subsequent attempts to read will return `NULL` until the condition is cleared with clearerr(3).

The text to which the returned pointer points may be modified, provided that no changes are made beyond the returned size. These changes are lost as soon as the pointer becomes invalid.

ERRORS

`[EBADF]` The argument *stream* is not a stream open for reading.

The **fgetwln()** function may also fail and set `errno` for any of the errors specified for the routines mbrtowc(3), realloc(3), or read(2).

SEE ALSO

ferror(3), fgetln(3), fgetws(3), fopen(3)

GETLINE(3)

NAME

getdelim, getline - get a line from a stream

SYNOPSIS

```
#define _WITH_GETLINE

#include <stdio.h>

ssize_t
getdelim(char ** restrict linep, size_t * restrict linecapp,
 int delimiter,  FILE * restrict stream);

ssize_t
getline(char ** restrict linep, size_t * restrict linecapp,
  FILE * restrict stream);
```

DESCRIPTION

The **getdelim()** function reads a line from *stream*, delimited by the character *delimiter*. The **getline()** function is equivalent to **getdelim()** with the newline character as the delimiter. The delimiter character is included as part of the line, unless the end of the file is reached.

The caller may provide a pointer to a malloc'd buffer for the line in **linep*, and the capacity of that buffer in **linecapp*. These functions expand the buffer as needed, as if via **realloc()**. If *linep* points to a `NULL` pointer, a new buffer will be allocated. In either case, **linep* and **linecapp* will be updated accordingly.

RETURN VALUES

The **getdelim()** and **getline()** functions return the number of characters stored in the buffer, excluding the terminating `NUL` character. The value -1 is returned if an error occurs, or if end-of-file is reached.

EXAMPLES

The following code fragment reads lines from a file and writes them to standard output. The **fwrite()** function is used in case the line contains embedded `NUL` characters.

```
char *line = NULL;
size_t linecap = 0;
ssize_t linelen;
while ((linelen = getline(&line, &linecap, fp)) > 0)
fwrite(line, linelen, 1, stdout);
free(line);
```

COMPATIBILITY

Many application writers used the name `getline` before the **getline()** function was introduced in IEEE Std 1003.1 ("POSIX.1"), so a prototype is not provided by default in order to avoid compatibility problems. Applications that wish to use the **getline()** function described herein should either request a strict IEEE Std 1003.1-2008 ("POSIX.1") environment by defining the macro `_POSIX_C_SOURCE` to the value 200809 or greater, or by defining the macro `_WITH_GETLINE`, prior to the inclusion of ¡stdio.h¿. For compatibility with GNU libc, defining either `_BSD_SOURCE` or `_GNU_SOURCE` prior to the inclusion of ¡stdio.h¿ will also make **getline()** available.

ERRORS

These functions may fail if:

`[EINVAL]` Either linep or linecapp is `NULL`.

`[EOVERFLOW]` No delimiter was found in the first `SSIZE_MAX` characters.

These functions may also fail due to any of the errors specified for **fgets()** and **malloc()**.

SEE ALSO

fgetln(3), fgets(3), malloc(3)

STANDARDS

The **getdelim()** and **getline()** functions conform to IEEE Std 1003.1-2008 ("POSIX.1").

ISSUES

There are no wide character versions of **getdelim()** or **getline()**.

FGETS(3)

NAME

fgets, gets - get a line from a stream

SYNOPSIS

```
#include <stdio.h>

char *
fgets(char *str, int size, FILE *stream)

char *
gets(char *str)
```

DESCRIPTION

The **fgets()** function reads at most one less than the number of characters specified by size from the given stream and stores them in the string str. Reading stops when a newline character is found, at end-of-file or error. The newline, if any, is retained. If any characters are read and there is no error, a `'\0'` character is appended to end the string.

The **gets()** function is equivalent to **fgets()** with an infinite size and a stream of `stdin`, except that the newline character (if any) is not stored in the string. It is the caller's responsibility to ensure that the input line, if any, is sufficiently short to fit in the string.

RETURN VALUES

Upon successful completion, **fgets()** and **gets()** return a pointer to the string. If end-of-file occurs before any characters are read, they return `NULL` and the buffer content is unchanged. If an error occurs, they return `NULL` and the buffer content is indeterminate. The **fgets()** and **gets()** functions do not distinguish between end-of-file and error, and callers must use feof(3) and ferror(3) to determine which occurred.

ERRORS

`[EBADF]` The given stream is not a readable stream.

The function **fgets()** may also fail and set `errno` for any of the errors specified for the routines fflush(3), fstat(2), read(2), or malloc(3).

The function **gets()** may also fail and set `errno` for any of the errors specified for the routine getchar(3).

SEE ALSO

feof(3), ferror(3), fgetln(3)

STANDARDS

The functions **fgets()** and **gets()** conform to ISO/IEC 9899:1990 ("ISO C90").

ISSUES

Since it is usually impossible to ensure that the next input line is less than some arbitrary length, and because overflowing the input buffer is almost invariably a security violation, programs should NEVER use **gets()**.

The **gets()** function exists purely to conform to ISO/IEC 9899:1990 ("ISO C90").

FGETWS(3)

NAME

fgetws - get a line of wide characters from a stream

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

wchar_t *
fgetws(wchar_t * restrict ws, int n, FILE * restrict fp)
```

DESCRIPTION

The **fgetws()** function reads at most one less than the number of characters specified by *n* from the given *fp* and stores them in the wide character string *ws*. Reading stops when a newline character is found, at end-of-file or error. The newline, if any, is retained. If any characters are read and there is no error, a `'\0'` character is appended to end the string.

RETURN VALUES

Upon successful completion, **fgetws()** returns *ws*. If end-of-file occurs before any characters are read, **fgetws()** returns `NULL` and the buffer contents remain unchanged. If an error occurs, **fgetws()** returns `NULL` and the buffer contents are indeterminate. The **fgetws()** function does not distinguish between end-of-file and error, and callers must use feof(3) and ferror(3) to determine which occurred.

ERRORS

The **fgetws()** function will fail if:

`[EBADF]` The given *fp* argument is not a readable stream.

`[EILSEQ]` The data obtained from the input stream does not form a valid multibyte character.

The function **fgetws()** may also fail and set `errno` for any of the errors specified for the routines fflush(3), fstat(2), read(2), or malloc(3).

SEE ALSO

feof(3), ferror(3), fgets(3)

STANDARDS

The **fgetws()** function conforms to IEEE Std 1003.1-2001 (“POSIX.1”).

FOPEN(3)

NAME

fopen, fdopen, freopen - stream open functions

SYNOPSIS

```
#include <stdio.h>

FILE *
fopen(char *path, char *mode)

FILE *
fdopen(int fildes, char *mode)

FILE *
freopen(char *path, char *mode, FILE *stream)
```

DESCRIPTION

The **fopen()** function opens the file whose name is the string pointed to by *path* and associates a stream with it.

The argument mode points to a string beginning with one of the following sequences (Additional characters may follow these sequences.):

"`r`" Open text file for reading. The stream is positioned at the beginning of the file.

"`r+`" Open for reading and writing. The stream is positioned at the beginning of the file.

"`w`" Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.

"`w+`" Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.

"`a`" Open for writing. The file is created if it does not exist. The stream is positioned at the end of the file.

"`a+`" Open for reading and writing. The file is created if it does not exist. The stream is positioned at the end of the file.

The mode string can also include the letter "`b`" either as a third character or as a character between the characters in any of the two-character strings described above. The "`b`" indicates that I/O should be performed in binary mode, instead of the default text mode.

If a comma is found after the mode specification, the remaining text is taken to be DCB attributes which will be passed to the open(2) function. See open(2) for a description of these attributes.

Reads and writes may be intermixed on read/write streams in any order, and do not require an intermediate seek as in other versions of studio. This is not portable to other systems. However; ANSI C requires that a file positioning function intervene between output and input, unless an input operation encounters end-of-file.

The **fdopen()** function associates a stream with the existing file descriptor, fildes. The mode of the stream must be compatible with the mode of the file descriptor.

The **freopen()** function opens the file whose name is the string pointed to by *path* and associates the stream pointed to by *stream* with it. The original stream (if it exists) is closed. The *mode* argument is used just as in the **fopen()** function. The primary use of the **freopen()** function is to change the file associated with a standard text stream (stderr, stdin, or stdout).

RECORD I/O

If the `type=record` attribute is specified after the mode specification; the lower-level file descriptor will be processed in "record I/O" mode. In this mode, the file is set to non-buffering to directly pass write requests to the lower level write functions. Any read processing should reset the file buffer to a buffer size sufficient to handle the expected record length of the file. See the fread(3) and fwrite(3) sections for more information regarding record I/O.

RETURN VALUES

Upon successful completion **fopen()**, **fdopen()** and **freopen()** return a FILE pointer. Otherwise, `NULL` is returned and the global variable `errno` is set to indicate the error.

EXAMPLE

The following opens the file specified by the character pointer `output_file_name`, for text output. Note that it uses the DCB attributes string to specify that the file format should be FIXED BLOCKED, with a block size of 8000 and a logical record length of 80:

```
char *output_file_name;
FILE *output_file;
       ...
output_file = fopen(output_file_name,
        "w,recfm=fb,blksize=8000,lrecl=80");
```

ERRORS

[EINVAL] The mode provided to **fopen()**, **fdopen()**, or **freopen()** was invalid.

The **fopen()**, **fdopen()** and **freopen()** functions may also fail and set `errno` for any of the errors specified for the routine malloc(3).

The **fopen()** function may also fail and set `errno` for any of the errors specified for the routine open(2).

The **fdopen()** function may also fail and set `errno` for any of the errors specified for the routine fcntl(2).

The **freopen()** function may also fail and set `errno` for any of the errors specified for the routines open(2), fclose(3) and fflush(3).

SEE ALSO

open(2), fclose(3), fseek(3), funopen(3)

ISSUES

fopen() is based on **open()**. Any restrictions mentioned on the open(3) description apply to **fopen()**.

STANDARDS

The **fopen()** and **freopen()** functions conform to ISO/IEC 9899:1990 ("ISO C90"). The **fdopen()** function conforms to IEEE Std1003.1-1988 ("POSIX").

FPUTS(3)

NAME

fputs, puts - output a line to a stream

SYNOPSIS

```
#include <stdio.h>

int
fputs(const char *str, FILE *stream)

int
puts(const char *str)
```

DESCRIPTION

The function **fputs()** writes the string pointed to by *str* to the stream pointed to by *stream*.

The function **puts()** writes the string *str*, and a terminating newline character, to the stream `stdout`.

RETURN VALUES

The **fputs()** function returns 0 on success and `EOF` on error; **puts()** returns a nonnegative integer on success and `EOF` on error.

ERRORS

`[EBADF]` The stream supplied is not a writable stream.

The functions **fputs()** and **puts()** may also fail and set `errno` for any of the errors specified for the routines write(2).

SEE ALSO

ferror(3), putc(3), stdio(3)

STANDARDS

The functions **fputs()** and **puts()** conform to ISO/IEC 9899:1990 (“ISO C90”).

FPUTWS(3)

NAME

fputws - output a line of wide characters to a stream

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

int
fputws(const wchar_t * restrict ws, FILE * restrict fp)
```

DESCRIPTION

The **fputws()** function writes the wide character string pointed to by *ws* to the stream pointed to by *fp*.

RETURN VALUES

The **fputws()** function returns 0 on success and -1 on error.

ERRORS

The **fputws()** function will fail if:

`[EBADF]` The *fp* argument supplied is not a writable stream.

The **fputws()** function may also fail and set `errno` for any of the errors specified for the routine write(2).

SEE ALSO

ferror(3), fputs(3), putwc(3), stdio(3)

STANDARDS

The **fputws()** function conforms to IEEE Std 1003.1-2001 (“POSIX.1”).

FREAD(3)

NAME

fread, fwrite - binary stream input/output

SYNOPSIS

```
#include <stdio.h>

size_t
fread(void *ptr, size_t size, size_t nmemb,
FILE *stream)

size_t
fwrite(const void *ptr, size_t size, size_t nmemb,
FILE *stream)
```

DESCRIPTION

The function **fread()** reads *nmemb* objects, each *size* bytes long, from the stream pointed to by *stream*, storing them at the location given by *ptr*.

The function **fwrite()** writes *nmemb* objects, each size bytes long, to the stream pointed to by *stream*, obtaining them from the location given by *ptr*.

RECORD I/O

If the stream was opened with the attribute of `type=record`, then the lower-level read/write operation is performed in "record I/O" mode. Furthermore, the stream is set to be unbuffered.

For **fwrite()**, the unbuffered stream causes the specified number of bytes (`size` * `nmemb`) to be directly passed to the **write()** operation and written as one record. A subsequent **fwrite()** will advance to the next output record.

For **fread()**, the unbuffered stream will cause a read of 1 byte from each record in the input file. Thus, for **fread()**, the buffer should be reset using setvbuf(3), to a size that is appropriate for the expected maximum input record length. Then, the lower-level read operation will fill this buffer with a single record, and that buffer will used to satisfy the **fread()** request. When record-I/O is employed, every call to **fread()** forces a refresh of the input buffer by invoking read(2) to read the next record.

For example, to read a variable-length file `MYFILE` a record a time; where the maximum record length in `MYFILE` is 8000 bytes:

```
char record[8000];
FILE *f;
int num_read, rec_len;

/* open the binary file in record-I/O mode */
f = fopen("MYFILE", "rb,type=record");

        /* set the input buffering to be record-sized */
        setvbuf(f, NULL, _IOFBF, 8000);

/* read a record */
num_read = fread(record, 1, 8000, f);
```

Note that the *size* value in this example is set to `1`, and the number of elements (`nmemb`) is set to `8000`, allowing **fread()** to return the number of bytes read on the record.

RETURN VALUES

The functions **fread()** and **fwrite()** advance the file position indicator for the stream by at least the number of bytes read or written. They return the number of objects read or written. If an error occurs, or the end-of-file is reached, the return value is a short object count (or zero).

The function **fread()** does not distinguish between end-of-file and error. Callers must use feof(3) and ferror(3) to determine which occurred. The function **fwrite()** returns a value less than *nmemb* only if a write error has occurred, or the the amount of data is larger than the maximum record length when using "record I/O".

When reading from a variable-length record using "record I/O" (`type=record`), it is possible to read a zero-length record. In this case, there is no data read; which is typically the situation end-of-file. In this event, the lower-level **read()** will set the error condition, with `errno` set to `EAGAIN`. Thus, programs using **fread()** reading from variable-length input with `type=record` need to be aware that ferror()(3) will be true for zero-length input records. The error should be cleared with clearerr()(3) to proceed to the next record. This approach allows the program to distinguish between reading a zero-length record and reaching end-of-file.

Writing to variable-length record files cannot produce a zero-length record. If it is desired to produce zero-length records in the resulting output file, then the write()(2) function with `type=record` should be used.

SEE ALSO

read(2), write(2)

STANDARDS

The functions **fread()** and **fwrite()** conform to ISO/IEC 9899:1990 ("ISO C90").

FSEEK(3)

NAME

fgetpos, fseek, fseeko, fsetpos, ftell, ftello, rewind - reposition a stream

SYNOPSIS

```
#include <stdio.h>

int
fseek(FILE *stream, long offset, int whence)

long
ftell(FILE *stream)

void
rewind(FILE *stream)

int
fgetpos(FILE *stream, fpos_t *pos)

int
fsetpos(FILE *stream, fpos_t *pos)

int
fseeko(FILE *stream, off_t offset, int whence);

off_t
ftello(FILE *stream);
```

DESCRIPTION

The **fseek()** function sets the file position indicator for the stream pointed to by *stream*. The new position, measured in bytes, is obtained by adding offset bytes to the position specified by *whence*. If *whence* is set to `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`, the offset is relative to the start of the file, the current position indicator, or end-of-file, respectively. A successful call to the **fseek()** function clears the end-of-file indicator for the stream and undoes any effects of the ungetc(3) function on the same stream.

The **ftell()** function obtains the current value of the file position indicator for the stream pointed to by stream.

The **rewind()** function sets the file position indicator for the stream pointed to by stream to the beginning of the file. It is equivalent to:

```
(void)fseek(stream, 0L, SEEK_SET)
```

except that the error indicator for the stream is also cleared (see clearerr(3)).

The **fseeko()** function is identical to **fseek()**, except it takes an `off_t` argument instead of a `long`. Likewise, the **ftello()** function is identical to **ftell()**, except it returns an `off_t`.

The **fgetpos()** and **fsetpos()** functions are alternate interfaces equivalent to **ftell()** and **fseek()** (with whence set to `SEEK_SET`), setting and storing the current value of the file offset into or from the object referenced by *pos*. The `fpos_t` object may be a complex object and these routines may be the only way to reposition a text stream in a portable fashion.

RETURN VALUES

The **rewind()** function returns no value.

The **fgetpos()**, **fseek()**, **fseeko()** and **fsetpos()** functions return the value 0 if successfu; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

Upon successful completion, **ftell()** and **ftello()** return the current offset. Otherwise, -1 is returned and the global variable *errno* is set to indicate the error.

ISSUES

These functions depend on lseek(2). Any restrictions provided in the lseek(2) description apply to these functions as well.

ERRORS

`[EBADF]` The stream specified is not a seekable stream.

`[EINVAL]` The whence argument to **fseek()** was not `SEEK_SET`, `SEEK_END`, or `SEEK_CUR`.

The function **fgetpos()**, **fseek()**, **fseeko()**, **fsetpos()**, **ftell()** and **ftello()** may also fail and set `errno` for any of the errors specified for the routines flush(3), fstat(2), lseek(2), and malloc(3).

SEE ALSO

lseek(2)

STANDARDS

The **fgetpos()**, **fsetpos()**, **fseek()**, **ftell()**, and **rewind()** functions conform to ISO 9899: 1990 ("ISO C90").

The **fseeko()** and **ftello()** functions conform to Version 2 of the Single UNIX Specification ("SUSv2").

FUNOPEN(3)

NAME

funopen, fropen, fwopen - open a stream

SYNOPSIS

```
#include <stdio.h>

FILE *
funopen(const void *cookie,
int (*readfn)(void *, char *, int),
int (*writefn)(void *, const char *, int),
fpos_t (*seekfn)(void *, fpos_t, int),
int (*closefn)(void *))

FILE *
fropen(void *cookie,
int (*readfn)(void *, char *, int))

FILE *
fwopen(void *cookie,
int (*writefn)(void *, const char *, int))
```

DESCRIPTION

The **funopen()** function associates a stream with up to four "I/O functions". Either *readfn* or *writefn* must be specified; the others can be given as an appropriately-typed `NULL` pointer. These I/O functions will be used to read, write, seek and close the new stream.

In general, omitting a function means that any attempt to perform the associated operation on the resulting stream will fail. If the close function is omitted, closing the stream will flush any buffered output and then succeed.

The calling conventions of *readfn*, *writefn*, *seekfn* and *closefn* must match those, respectively, of read(2), write(2), seek(2), and close(2) with the single exception that they are passed the cookie argument specified to **funopen()** in place of the traditional file descriptor argument.

Read and write I/O functions are allowed to change the underlying buffer on fully buffered or line buffered streams by calling setvbuf(3). They are also not required to completely fill or empty the buffer. They are not, however, allowed to change streams from unbuffered to buffered or to change the state of the line buffering flag.

They must also be prepared o have read or write calls occur on buffers other than the one most recently specified.

All user I/O functions can report an error by returning -1. Additionally, all of the functions should set the external variable errno appropriately if an error occurs.

An error on *closefn*() does not keep the stream open.

As a convenience, the include file `<stdio.h>` defines the macros **frozen()** and **fwopen()** as calls to **funopen()** with only a read or write function specified.

RETURN VALUES

Upon successful completion, **funopen()** returns a FILE pointer. Otherwise, `NULL` is returned and the global variable `errno` is set to indicate the error.

ERRORS

`[EINVAL]` The **funopen()** function was called without either a read or write function. The **funopen()** function may also fail and set `errno` for any of the errors specified for the routine malloc(3).

SEE ALSO

fcntl(2), open(2), fclose(3), fopen(3), fseek(3), setbuf(3)

FWIDE(3)

NAME

fwide - get/set orientation of a stream

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

int
fwide(FILE *stream, int mode)
```

DESCRIPTION

The **fwide()** function determines the orientation of the stream pointed at by *stream*.

If the orientation of *stream* has already been determined, **fwide()** leaves it unchanged. Otherwise, **fwide()** sets the orientation of *stream* according to *mode*.

If *mode* is less than zero, *stream* is set to byte-oriented. If it is greater than zero, *stream* is set to wide-oriented. Otherwise, *mode* is zero, and *stream* is unchanged.

RETURN VALUES

The **fwide()** function returns a value according to orientation after the call of **fwide()**; a value less than zero if byte-oriented, a value greater than zero if wide-oriented, and zero if the stream has no orientation.

SEE ALSO

ferror(3), fgetc(3), fgetwc(3), fopen(3), fputc(3), fputwc(3), freopen(3), stdio(3)

STANDARDS

The **fwide()** function conforms to ISO/IEC 9899:1999 (“ISO C99”).

GETC(3)

NAME

fgetc, getc, getchar, getw - get next character or word from input stream

SYNOPSIS

```
#include <stdio.h>

int
fgetc(FILE *stream)

int
getc(FILE *stream)

int
getchar()

int
getw(FILE *stream)
```

DESCRIPTION

The **fgetc()** function obtains the next input character (if present) from the stream pointed at by stream, or the next character pushed back on the stream via ungetc(3).

The **getc()** function acts essentially identically to **fgetc()**, but is a macro that expands in-line.

The **getchar()** function is equivalent to: getc with the argument `stdin`.

The **getw()** function obtains the next int (if present) from the stream pointed at by stream.

RETURN VALUES

If successful, these routines return the next requested object from the stream. If the stream is at end-of-file or a read error occurs, the routines return `EOF`. The routines feof(3) and ferror(3) must be used to distinguish between end-of-file and error. If an error occurs, the global variable `errno` is set to indicate the error. The end-of-file condition is remembered, even on a terminal, and all subsequent attempts to read will return `EOF` until the condition is cleared with clearerr(3).

SEE ALSO

ferror(3), fopen(3), fread(3), putc(3), ungetc(3)

STANDARDS

The **fgetc()**, **getc()** and **getchar()** functions conform to ISO/IEC 9899:1990 ("ISO C90").

ISSUES

Since `EOF` is a valid integer value, feof(3) and ferror(3) must be used to check for failure after calling **getw()**. The size and byte order of an int varies from one machine to another, and **getw()** is not recommended for portable applications.

GETWC(3)

NAME

fgetwc, getwc, getwchar - get next wide character from input stream

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

wint_t
fgetwc(FILE *stream)

wint_t
getwc(FILE *stream)

wint_t
getwchar()
```

DESCRIPTION

The **fgetwc()** function obtains the next input wide character (if present) from the stream pointed at by *stream*, or the next character pushed back on the stream via ungetwc(3).

The **getwc()** function acts essentially identically to **fgetwc()**.

The **getwchar()** function is equivalent to **getwc()** with the argument `stdin`.

RETURN VALUES

If successful, these routines return the next wide character from the *stream*. If the stream is at end-of-file or a read error occurs, the routines return `WEOF`. The routines feof(3) and ferror(3) must be used to distinguish between end-of-file and error. If an error occurs, the global variable `errno` is set to indicate the error. The end-of-file condition is remembered, even on a terminal, and all subsequent attempts to read will return `WEOF` until the condition is cleared with clearerr(3).

SEE ALSO

ferror(3), fopen(3), fread(3), getc(3), putwc(3), stdio(3), ungetwc(3)

STANDARDS

The **fgetwc()**, **getwc()** and **getwchar()** functions conform to ISO/IEC 9899:1999 ("ISO C99").

MKTEMP(3)

NAME

mktemp - make temporary file name (unique)

SYNOPSIS

```
#include <unistd.h>

char *
mktemp(char *template);

int
mkstemp(char *template);

int
mkstemps(char *template, int suffixlen);

char *
mkdtemp(char *template);
```

DESCRIPTION

The **mktemp()** function takes the given file name template and overwrites a portion of it to create a file name. This file name is guaranteed not to exist at the time of function invocation and is suitable for use by the application. The template may be any `//HFS:`-style file name with some number of '`X`s' appended to it, for example `/tmp/temp.XXXXXX`. The trailing '`X`s' are replaced with a unique alphanumeric combination. The number of unique file names **mktemp()** can return depends on the number of '`X`s' provided; six '`X`s' will result in **mktemp()** selecting one of 56800235584 (62 ** 6) possible temporary file names.

The **mkstemp()** function makes the same replacement to the template and creates the template file, mode `0600`, returning a file descriptor opened for reading and writing. This avoids the race between testing for a file's existence and opening it for use.

The **mkstemps()** function acts the same as **mkstemp()**, except it permits a suffix to exist in the template. The template should be of the form `/tmp/tmpXXXXXXsuffix`. **mkstemps()** is told the length of the suffix string.

The **mkdtemp()** function makes the same replacement to the template as in **mktemp(3)** and creates the template directory, mode `0700`.

RETURN VALUES

The **mktemp()** and **mkdtemp()** functions return a pointer to the template on success and `NULL` on failure. The **mkstemp()** and **mkstemps()** functions return -1 if no suitable file could be created. If either call fails an error code is placed in the global variable `errno`.

ERRORS

The **mkstemp()**, **mkstemps()** and **mkdtemp()** functions may set `errno` to one of the following values:

`[ENOTDIR]` The pathname portion of the template is not an existing directory.

The **mkstemp()**, **mkstemps()** and **mkdtemp()** functions may also set `errno` to any value specified by the stat(2) function.

The **mkstemp()** and **mkstemps()** functions may also set `errno` to any value specified by the open(2) function.

The **mkdtemp()** function may also set `errno` to any value specified by the mkdir(2) function.

NOTES

The **mktemp()**, **mkstemp()**, **mkstemps()** and **mkdtemp()** functions only return `//HFS:`-style names, and thus require OpenEdition services.

A common problem that results in abnormal termination is that the programmer passes in a read-only string to **mktemp()**, **mkstemp()**, **mkstemps()** or **mkdtemp()**. This is common with programs that were developed before ISO/IEC 9899:1990 ("ISO C90") compilers were common. For example, calling **mkstemp()** with an argument of `"/tmp/tempfile.XXXXXX"` may result in abnormal termination due to **mkstemp()** attempting to modify the string constant that was given.

ISSUES

This family of functions produces filenames which can be guessed, though the risk is minimized when large numbers of '`X`s' are used to increase the number of possible temporary filenames. This makes the race in **mktemp()**, between testing for a file's existence (in the **mktemp()** function call) and opening it for use (later in the user application) particularly dangerous from a security perspective. Whenever

it is possible, **mkstemp()** should be used instead, since it does not have the race condition. If **mkstemp()** cannot be used, the filename created by **mktemp()** should be created using the `O_EXCL` flag to open(2) and the return status of the call should be tested for failure. This will ensure that the program does not continue blindly in the event that an attacker has already created the file with the intention of manipulating or reading its contents.

SEE ALSO

chmod(2), getpid(2), mkdir(2), open(2), stat(2)

PRINTF(3)

NAME

printf, fprintf, sprintf, snprintf, asprintf, vprintf, vfprintf, vsprintf, vsnprintf, vasprintf - formatted output conversion

SYNOPSIS

```
#include <stdio.h>

int
printf(const char *format, ...)

int
fprintf(FILE *stream, const char *format, ...)

int
sprintf(char *str, const char *format, ...)

int
snprintf(char *str, size_t size, const char *format,
...)

int
asprintf(char **ret, const char *format, ...)

#include <stdarg.h>

int
vprintf(const char *format, va_list ap)

int
vfprintf(FILE *stream, const char *format, va_list ap)

int
vsprintf(char *str, char *format, va_list ap)

int
vsnprintf(char *str, size_t size, const char *format,
va_list ap)

int
vasprintf(char **ret, const char *format, va_list ap)
```

DESCRIPTION

The **printf()** family of functions produces output according to a format as described below. The **printf()** and **vprintf()** functions write output to stdout, the standard output stream; **fprintf()** and **vfprintf()** write output to the given output *stream*; **sprintf()**, **snprintf()**, **vsprintf()**, and **vsnprintf()** write to the character string *str*; and **asprintf()** and **vasprintf()** dynamically allocate a new string with malloc(3) / realloc(3).

These functions write the output under the control of a format string that specifies how subsequent arguments (or arguments accessed via the variable-length argument facilities of stdarg(3)) are converted for output.

These functions return the number of characters printed (not including the trailing `'\0'` used to end output to strings).

asprintf() and **vasprintf()** return a pointer to a buffer sufficiently large to hold the string in the ret argument; This pointer should be passed to free(3) to release the allocated storage when it is no longer needed. If sufficient space cannot be allocated, **asprintf()** and **vasprintf()** will return -1 and set ret to be a `NULL` pointer.

snprintf() and **vsnprintf()** will write at most size-1 of the characters printed into the output string (the size'th character then gets the terminating `'\0'`); if the return value is greater than or equal to the size argument, the string was too short and some of the printed characters were discarded.

The **sprintf()** and **vsprintf()** functions effectively assume an infinite size.

The format string is composed of zero or more directives: ordinary characters (not `%`), which are copied unchanged to the output stream and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character `%`. The arguments must correspond properly (after type promotion) with the conversion specifier. After the `%`, the following appear in sequence:

- Zero or more of the following flags:
  - A `#` character specifying that the value should be converted to an "alternate form." For `c`, `d`, `i`, `n`, `p`, `s`, and `u`, conversions, this option has no effect. For `o` conversions, the precision of the number is increased to force the first character of the output string to a zero (except if a zero value is printed with an explicit precision of zero). For `x` and `X` conversions, a non-zero result has the string '`0x`' (or '`0X`' for X conversions) pretended to it. For `e`, `E`, `f`, `g`, and `G`, conversions, the result will always contain a decimal point, even if no digits follow it (normally, a decimal point appears in the results of those conversions only if a digit follows). For `g` and `G` conversions, trailing zeros are not removed from the result as they would otherwise be.

  - A zero '0' character specifying zero padding. For all conversions except `n`, the converted value is padded on the left with zeros rather than blanks. If a precision is given with a numeric conversion (`d`, `i`, `o`, `u`, `i`, `x`, and `X`), the '0' flag is ignored.
  - A negative field width flag '-' indicates the converted value is to be left adjusted on the field boundary. Except for `n` conversions, the converted value is padded on the right with blanks, rather than on the left with blanks or zeros. A '-' overrides a '0' if both are given.
  - A space, specifying that a blank should be left before a positive number produced by a signed conversion (`d`, `e`, `E`, `f`, `g`, `G`, or `i`).
  - A '+' character specifying that a sign always be placed before a number produced by a signed conversion. A '+' overrides a space if both are used.

- An optional decimal digit string specifying a minimum field width. If the converted value has fewer characters than the field width, it will be padded with spaces on the left (or right, if the left-adjustment flag has been given) to fill out the field width.

- An optional precision, in the form of a period '.' followed by an optional digit string. If the digit string is omitted, the precision is taken as zero. This gives the minimum number of digits to appear for `d`, `i`, `o`, `u`, `x`, and `X` conversions, the number of digits to appear after the decimal-point for `e`, `E`, and `f` conversions, the maximum number of significant digits for `g` and `G` conversions, or the maximum number of characters to be printed from a string for `s` conversions.

- The optional character `h`, specifying that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion corresponds to a `short int` or `unsigned short int` argument, or that a following `n` conversion corresponds to a pointer to a `short int` argument.

- The optional character `l` (ell) specifying that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion applies to a pointer to a `long int` or `unsigned long int` argument, or that a following `n` conversion corresponds to a pointer to a `long int` argument.

- The optional characters `ll` (ell ell), specifying that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion corresponds to a `long long` or `unsigned long long` argument, or that a following `n` conversion corresponds to a pointer to a `long long` argument. The deprecated character `q` specifies the same behavior. Programs should use `ll` instead.

- The character `L` specifying that a following `e`, `E`, `f`, `g`, or `G` conversion corresponds to a `long double` argument.

- A character that specifies the type of conversion to be applied.

A field width or precision, or both, may be indicated by an asterisk '`*`' instead of a digit string. In this case, an `int` argument supplies the field width or precision. A negative field width is treated as a left adjustment flag followed by a positive field width; a negative precision is treated as though it were missing.

The conversion specifiers and their meanings are:

`diouxX` The `int` (or appropriate variant) argument is converted to signed decimal (`d` and `i`), unsigned octal (`o`), unsigned decimal (`u`), or unsigned hexadecimal (`x` and `X`) notation. The letters `abcdef` are used for `x` conversions; the letters `ABCDEF` are used for `X` conversions. The precision, if any, gives the minimum number of digits that must appear; if the converted value requires fewer digits, it is padded on the left with zeros.

`eE` The `double` argument is rounded and converted in the style [-]*d.ddd*e+-*dd* where there is one digit before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero, no decimal-point character appears. An `E` conversion uses the letter `E` (rather than `e`) to introduce the exponent. The exponent always contains at least two digits; if the value is zero, the exponent is 00.

`fF` The `double` argument is rounded and converted to decimal notation in the style [-]*ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is explicitly zero, no decimal-point character appears. If a decimal point appears, at least one digit appears before it.

`gG` The `double` argument is converted in style f or e (or `E` for `G` conversions). The precision specifies the number of significant digits. If the precision is missing, 6 digits are given; if the precision is zero, it is treated as 1. Style `e` is used if the exponent from its conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional part of the result; a decimal point appears only if it is followed by at least one digit.

`aA` The `double` argument is printed in style [-]*h.hhh*p+-*d*, where there is one hexadecimal digit before the hexadecimal point and the number after is equal to the precision specification for the argument; when the precision is missing, enough digits are produced to convey the argument's exact double-precision floating-point representation.

`D` The `_Decimal` argument is printed. After the conversion specifier, an argument of (*size*,*prec*) must appear specifying the size and precision.

`c` The `int` argument is converted to an `unsigned char`, and the resulting character is written.

`s` The `char *` argument is expected to be a pointer to an array of character type (pointer to a string). Characters from the array are written up to (but not including) a terminating NUL character; if a precision is specified, no more than the number specified are written. If a precision

is given, no null character need be present. If the precision is not specified or is greater than the size of the array, the array must contain a terminating NUL character.

`p` The `void *` pointer argument is printed in hexadecimal (as if by '`%#x`' or '`%#lx`').

`n` The number of characters written so far is stored into the integer indicated by the `int *` (or variant) pointer argument. No argument is converted.

`%` A '`%`' is written. No argument is converted. The complete conversion specification is '`%%`'.

In no case does a non-existent or small field width cause truncation of a field. If the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

EXAMPLES

To print a date and time in the form 'Sunday, July 3, 10:02', where *weekday* and *month* are pointers to strings:

```
#include <stdio.h>
fprintf(stdout, "%s, %s %d, %.2d:%.2d\n",
        weekday, month, day, hour, min);
```

To print PI to five decimal places:

```
#include <math.h>
#include <stdio.h>
fprintf(stdout, "pi = %.5f\n", 4 * atan(1.0));
```

To allocate a 128 byte string and print into it:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
char *newfmt(const char *fmt, ...)
{
   char *p;
   va_list ap;
   if ((p = malloc(128)) == NULL)
```

```
        return (NULL);
    va_start(ap, fmt);
    (void) vsnprintf(p, 128, fmt, ap);
    va_end(ap);
    return (p);
}
```

SEE ALSO

scanf(3)

STANDARDS

The **fprintf()**, **printf()**, **sprintf()**, **vprintf()**, **vfprintf()**, and **vsprintf()** functions conform to ISO/IEC 9899:1990 ("ISO C90").

ISSUES

The effect of padding the `%p` format with zeros (either by the '`0`' flag or by specifying a precision), and the benign effect (i.e., none) of the '`#`' flag on `%n` and `%p` conversions, as well as other nonsensical combinations such as `%Ld`, are not standard such combinations should be avoided.

Because **sprintf()** and **vsprintf()** assume an infinitely long string, callers must be careful not to overflow the actual space; this is often hard to assure. For safety, programmers should use the **snprintf()** interface instead. Unfortunately, this interface is not portable.

PUTC(3)

NAME

fputc, putc, putchar, putw - output a character or word to a stream

SYNOPSIS

```
#include <stdio.h>

int
fputc(int c, FILE *stream)

int
putc(int c, FILE *stream)

int
putchar(int c)

int
putw(int w, FILE *stream)
```

DESCRIPTION

The **fputc()** function writes the character *c* (converted to an `unsigned char`) to the output stream pointed to by *stream*.

putc() acts essentially identically to **fputc()**, but is a macro that expands in-line. It may evaluate *stream* more than once, so arguments given to **putc()** should not be expressions with potential side effects.

putchar() is identical to **putc()** with an output stream of stdout.

The **putw()** function writes the specified int to the named output stream.

RETURN VALUES

The functions, **fputc()**, **putc()** and **putchar()** return the character written. If an error occurs, the value `EOF` is returned. The **putw()** function returns 0 on success; `EOF` is returned if a write error occurs, or if an attempt is made to write a read-only stream.

SEE ALSO

ferror(3), fopen(3), getc(3), stdio(3)

STANDARDS

The functions **fputc()**, **putc()**, and **putchar()**, conform to ISO/IEC 9899:1990 ("ISO C").

CAUTIONS

The size and byte order of an `int` varies from one machine to another, and **putw()** is not recommended for portable applications.

PUTWC(3)

NAME

fputwc, putwc, putwchar - output a wide character to a stream

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

wint_t
fputwc(wchar_t wc, FILE *stream)

wint_t
putwc(wchar_t wc, FILE *stream)

wint_t
putwchar(wchar_t wc)
```

DESCRIPTION

The **fputwc()** function writes the wide character *wc* to the output stream pointed to by *stream*.

The **putwc()** function acts essentially identically to **fputwc()**.

The **putwchar()** function is identical to **putwc()** with an output stream of `stdout`.

RETURN VALUES

The **fputwc()**, **putwc()**, and **putwchar()** functions return the wide character written. If an error occurs, the value `WEOF` is returned.

SEE ALSO

ferror(3), fopen(3), getwc(3), putc(3), stdio(3)

STANDARDS

The **fputwc()**, **putwc()**, and **putwchar()** functions conform to ISO/IEC 9899:1999 ("ISO C99").

REMOVE(3)

NAME

remove - remove file system entry

SYNOPSIS

```
#include <stdio.h>

int
remove(const char *path)
```

DESCRIPTION

The **remove()** function is an alias for the unlink(2) system call. It deletes the file referenced by path.

RETURN VALUES

Upon successful completion, **remove()** returns 0. Otherwise, -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

The **remove()** function may fail and set `errno` for any of the errors specified for the routine unlink(2).

SEE ALSO

unlink(2)

STANDARDS

The **remove()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

SCANF(3)

NAME

scanf, fscanf, sscanf, vscanf, vsscanf, vfscanf - input format conversion

SYNOPSIS

```
#include <stdio.h>

int
scanf(const char *format, ...)

int
fscanf(FILE *stream, const char *format, ...)

int
sscanf(const char *str, const char *format, ...)

#include <stdarg.h>

int
vscanf(const char *format, va_list ap)

int
vsscanf(const char *str, const char *format,
va_list ap)

int
vfscanf(FILE *stream, const char *format, va_list ap)
```

DESCRIPTION

The **scanf()** family of functions scans input according to a format as described below. This format may contain conversion specifiers the results from such conversions, if any, are stored through the pointer arguments. The **scanf()** function reads input from the standard input stream stdin, **fscanf()** reads input from the stream pointer stream, and **sscanf()** reads its input from the character string pointed to by str. The **vfscanf()** function is analogous to vfprintf(3) and reads input from the stream pointer stream using a variable argument list of pointers (see stdarg(3)). The **vscanf()** function scans a variable argument list from the standard input and the **vsscanf()** function scans it from a string; these are analogous to the **vprintf()** and **vsprintf()** functions respectively. Each successive pointer argument must correspond properly with each successive conversion specifier (but see 'suppression'

below). All conversions are introduced by the % (percent sign) character. The format string may also contain other characters. White space (such as blanks, tabs, or newlines) in the format string match any amount of white space, including none, in the input. Everything else matches only itself. Scanning stops when an input character does not match such a format character. Scanning also stops when an input conversion cannot be made (see below).

CONVERSIONS

Following the % character introducing a conversion there may be a number of flag characters, as follows:

`*` Suppresses assignment. The conversion that follows occurs as usual, but no pointer is used; the result of the conversion is simply discarded.

`hh` Indicates that the conversion will be one of `dioux` or `n` and the next pointer is a pointer to a `char` (rather than `int`).

`h` Indicates that the conversion will be one of `dioux` or `n` and the next pointer is a pointer to a `short int` (rather than `int`).

`l` Indicates either that the conversion will be one of `dioux` or `n` and the next pointer is a pointer to a `long int` (rather than `int`), or that the conversion will be one of `efg` and the next pointer is a pointer to `double` (rather than `float`).

`ll` Indicates that the conversion will be one of `dioux` or `n` and the next pointer is a pointer to a `long long int` (rather than `int`).

`L` Indicates that the conversion will be `efg` and the next pointer is a pointer to `long double`. (This type is not implemented; the L flag is currently ignored.)

`j` Indicates that the conversion will be one of `dioux` or `n` and the next pointer is a pointer to a `intmax_t` (rather than `int`).

`t` Indicates that the conversion will be one of `dioux` or `n` and the next pointer is a pointer to a `ptrdiff_t` (rather than `int`).

`z` Indicates that the conversion will be one of `dioux` or `n` and the next pointer is a pointer to a `size_t` (rather than `int`).

`q` Indicates that the conversion will be one of `dioux` or `n` and the next pointer is a pointer to a `long long int` (rather than `int`). This use is deprecated, and will be removed in a future release. The standard defines `ll` for this purpose.

In addition to these flags, there may be an optional maximum field width, expressed as a decimal integer, between the `%` and the conversion. If no width is given, a default of "infinity" is used (with one exception, below); otherwise at most this many characters are scanned in processing the conversion. Before conversion begins, most conversions skip white space; this white space is not counted against the field width.

The following conversions are available:

`%` Matches a literal '`%`'. That is, '`%%`' in the format string matches a single input '`%`' character. No conversion is done, and assignment does not occur.

`d` Matches an optionally signed decimal integer; the next pointer must be a pointer to `int`.

`i` Matches an optionally signed integer; the next pointer must be a pointer to `int`. The integer is read in base 16 if it begins with '`0x`' or '`0X`', in base 8 if it begins with '`0`', and in base 10 otherwise. Only characters that correspond to the base are used.

`o` Matches an octal integer; the next pointer must be a pointer to `unsigned int`.

`u` Matches an optionally signed decimal integer; the next pointer must be a pointer to `unsigned int`.

`x` Matches an optionally signed hexadecimal integer; the next pointer must be a pointer to `unsigned int`.

`e, E, f, F, g, G` Matches a floating-point number in the style of strtod(3). The next pointer must be a pointer to `float` (unless `l` or `L` is specified.)

`s` Matches a sequence of non-white-space characters; the next pointer must be a pointer to `char`, and the array must be large enough to accept all the sequence and the terminating NUL character. The input string stops at white space or at the maximum field width, whichever occurs first.

`c` Matches a sequence of width count characters (default 1); the next pointer must be a pointer to `char`, and there must be enough room for all the characters (no terminating NUL is added). The usual skip of leading white space is suppressed. To skip white space first, use an explicit space in the format.

`[` Matches a non-empty sequence of characters from the specified set of accepted characters; the next pointer must be a pointer to `char`, and there must be enough room for all the characters in the string, plus a terminating NUL character. The usual skip of leading white space is suppressed. The string is to be made up of characters in (or not in) a particular set; the set is defined by the characters between the open bracket `[` character and a close bracket `]` character. The set excludes those characters if the

first character after the open bracket is a circumflex ^. To include a close bracket in the set, make it the first character after the open bracket or the circumflex; any other position will end the set. The hyphen character - is also special; when placed between two other characters, it adds all intervening characters to the set. To include a hyphen, make it the last character before the final close bracket. For instance, '`[^]0-9-]`' means the set 'everything except close bracket, zero through nine, and hyphen'. The string ends with the appearance of a character not in the (or, with a circumflex, in) set or when the field width runs out.

`p` Matches a pointer value (as printed by '`%p`' in printf(3)); the next pointer must be a pointer to `void`.

`n` Nothing is expected; instead, the number of characters consumed thus far from the input is stored through the next pointer, which must be a pointer to `int`. This is not a conversion, although it can be suppressed with the * flag.

For backwards compatibility, a "conversion" of '`%\0`' causes an immediate return of `EOF`.

RETURN VALUES

These functions return the number of input items assigned, which can be fewer than provided for, or even zero, in the event of a matching failure. Zero indicates that, while there was input available, no conversions were assigned; typically this is due to an invalid input character, such as an alphabetic character for a '`%d`' conversion. The value `EOF` is returned if an input failure occurs before any conversion such as an end-of-file occurs. If an error or end-of-file occurs after conversion has begun, the number of conversions which were successfully completed is returned.

SEE ALSO

getc(3), printf(3), strtod(3), strtol(3), strtoul(3)

STANDARDS

The functions **fscanf()**, **scanf()**, and **sscanf()** conform to ISO/IEC 9899:1990 ("ISO C").

ISSUES

Numerical strings are truncated to 512 characters; for example, `%f` and `%d` are implicitly `%512f` and `%512d`.

SETBUF(3)

NAME

setbuf, setbuffer, setlinebuf, setvbuf - stream buffering operations

SYNOPSIS

```
#include <stdio.h>

void
setbuf(FILE *stream, char *buf)

void
setbuffer(FILE *stream, char *buf, size_t size)

int
setlinebuf(FILE *stream)

int
setvbuf(FILE *stream, char *buf, int mode, size_t size)
```

DESCRIPTION

The three types of buffering available are unbuffered, block buffered, and line buffered. When an output stream is unbuffered, information appears on the destination file or terminal as soon as written. When it is block buffered many characters are saved up and written as a block. When it is line buffered characters are saved up until a newline is output or input is read from any stream attached to a terminal device (typically stdin). The function fflush(3) may be used to force the block out early. (See fclose(3).)

Normally all files are block buffered. When the first I/O operation occurs on a file, malloc(3) is called, and an optimally-sized buffer is obtained. If a stream refers to a terminal (as stdout normally does) it is line buffered. The standard error stream stderr is always unbuffered.

The **setvbuf()** function may be used to alter the buffering behavior of a stream. The mode parameter must be one of the following three macros:

`_IONBF`	unbuffered
`_IOLBF`	line buffered
`_IOFBF`	fully buffered

The *size* parameter may be given as zero to obtain deferred optimal-size buffer allocation as usual. If it is not zero, then except for unbuffered files, the *buf* argument should point to a buffer at least *size* bytes long; this buffer will be used instead of the current buffer. If the *size* argument is not zero but *buf* is `NULL`, a buffer of the given size will be allocated immediately, and released on close. This is an extension to ANSI C; portable code should use a size of 0 with any `NULL` buffer.

The **setvbuf()** function may be used at any time, but may have peculiar side effects (such as discarding input or flushing output) if the stream is "active." Portable applications should call it only once on any given stream, and before any I/O is performed.

The other three calls are, in effect, simply aliases for calls to **setvbuf()**. Except for the lack of a return value, the **setbuf()** function is exactly equivalent to the call

```
setvbuf(stream, buf,
        buf ? _IOFBF : _IONBF, BUFSIZ);
```

The **setbuffer()** function is the same, except that the size of the buffer is up to the caller, rather than being determined by the default BUFSIZ. The **setlinebuf()** function is exactly equivalent to the call:

```
setvbuf(stream, (char *)NULL, _IOLBF, 0);
```

RETURN VALUES

The **setvbuf()** function returns 0 on success, or `EOF` if the request cannot be honored (note that the stream is still functional in this case).

The **setlinebuf()** function returns what the equivalent **setvbuf()** would have returned.

SEE ALSO

fclose(3), fopen(3), fread(3), malloc(3), printf(3), puts(3)

STANDARDS

The **setbuf()** and **setvbuf()** functions conform to ISO/IEC 9899:1990 ("ISO C90").

TMPFILE(3)

NAME

tempnam, tmpfile, tmpnam – //HFS:-style temporary file routines

SYNOPSIS

```
#include <stdio.h>

FILE *
tmpfile(void);

char *
tmpnam(char *str);

char *
tempnam(const char *tmpdir, const char *prefix);
```

DESCRIPTION

The **tmpfile()** function returns a pointer to a stream associated with a file descriptor returned by the routine mkstemp(3). The created file is unlinked before **tmpfile()** returns, causing the file to be automatically deleted when the last reference to it is closed. The file is opened with the access value `'w+'`. The file is created in the directory determined by the environment variable `TMPDIR` if set. The default location if `TMPDIR` is not set is `/tmp`.

The **tmpnam()** function returns a pointer to a file name, in the `P_tmpdir` directory, which did not reference an existing file at some indeterminate point in the past. `P_tmpdir` is defined in the include file `<stdio.h>`. If the argument str is non-`NULL`, the file name is copied to the buffer it references. Otherwise, the file name is copied to a static buffer. In either case, **tmpnam()** returns a pointer to the file name.

The buffer referenced by *str* is expected to be at least `L_tmpnam` bytes in length. `L_tmpnam` is defined in the include file `<stdio.h>`.

The **tempnam()** function is similar to **tmpnam()**, but provides the ability to specify the directory which will contain the temporary file and the file name prefix.

The environment variable `TMPDIR` (if set), the argument *tmpdir* (if non-`NULL`), the directory `P_tmpdir` and the directory `/tmp` are tried, in the listed order, as directories in which to store the temporary file.

The argument *prefix*, if non-`NULL`, is used to specify a file name prefix, which will be the first part of the created file name. **tempnam()** allocates memory in which

to store the file name; the returned pointer may be used as a subsequent argument to free(3).

RETURN VALUES

The **tmpfile()** function returns a pointer to an open file system on success, and a `NULL` pointer on error.

The **tmpnam()** and **tempfile()** functions return a pointer to a file name on success, and a `NULL` pointer on error.

ERRORS

The **tmpfile()** function may fail and set the global variable `errno` for any of the errors specified for the library functions fdopen(3) or mkstemp(3).

The **tmpnam()** function may fail and set `errno` for any of the errors specified for the library function mktemp(3).

The **tempnam()** function may fail and set `errno` for any of the errors specified for the library functions malloc(3) or mktemp(3).

SEE ALSO

mkstemp(3), mktemp(3)

STANDARDS

ISSUES

These interfaces are provided for System V and ANSI compatibility only. They only produce //HFS:-style file names, and thus require OpenEdition services. The mkstemp(3) interface is strongly preferred.

There are four important problems with these interfaces (as well as with the historic mktemp(3) interface). First, there is an obvious race between file name selection and file creation and deletion. Second, most historic implementations provide only a limited number of possible temporary file names (usually 26) before file names will start being recycled. Third, the System V implementations of these functions (and of mktemp(3)) uses the access(2) function to determine whether or not the temporary file may be created. This has obvious ramifications for setuid or setgid programs, complicating the portable use of these interfaces in such programs. Finally, there is no specification of the permissions with which the temporary files are created.

This implementation does not have these flaws, but portable software cannot depend on that. In particular, the **tmpfile()** interface should not be used in software expected to be used on other systems if there is any possibility that the user does not wish the temporary file to be publicly readable and writable.

UNGETC(3)

NAME

ungetc - un-get character from input stream

SYNOPSIS

```
#include <stdio.h>

int
ungetc(int c, FILE *stream)
```

DESCRIPTION

The **ungetc()** function pushes the character *c* (converted to an `unsigned char`) back onto the input stream pointed to by *stream*. The pushed-backed characters will be returned by subsequent reads on the stream (in reverse order). A successful intervening call, using the same stream, to one of the file positioning functions (fseek(3), fsetpos(3), or rewind(3)) will discard the pushed back characters.

One character of push-back is guaranteed, but as long as there is sufficient memory, an effectively infinite amount of pushback is allowed.

If a character is successfully pushed-back, the end-of-file indicator for the stream is cleared.

RETURN VALUES

The **ungetc()** function returns the character pushed-back after the conversion, or `EOF` if the operation fails. If the value of the argument c character equals `EOF`, the operation will fail and the stream will remain unchanged.

SEE ALSO

fseek(3), getc(3), setvbuf(3)

STANDARDS

The **ungetc()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

UNGETWC(3)

NAME

ungetwc - un-get wide character from input stream

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

wint_t
ungetwc(wint_t wc, FILE *stream)
```

DESCRIPTION

The **ungetwc()** function pushes the wide character *wc* (converted to an `wchar_t`) back onto the input stream pointed to by *stream*. The pushed-backed wide characters will be returned by subsequent reads on the stream (in reverse order). A successful intervening call, using the same stream, to one of the file positioning functions fseek(3), fsetpos(3), or rewind(3) will discard the pushed back wide characters.

One wide character of push-back is guaranteed, but as long as there is sufficient memory, an effectively infinite amount of pushback is allowed.

If a character is successfully pushed-back, the end-of-file indicator for the stream is cleared.

RETURN VALUES

The **ungetwc()** function returns the wide character pushed-back after the conversion, or `WEOF` if the operation fails. If the value of the argument *wc* character equals `WEOF`, the operation will fail and the stream will remain unchanged.

SEE ALSO

fseek(3), getwc(3)

STANDARDS

The **ungetwc()** function conforms to ISO/IEC 9899:1999 (“ISO C99”).

WPRINTF(3)

NAME

wprintf, fwprintf, swprintf, vwprintf, vfwprintf, vswprintf - formatted wide character output conversion

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

int
fwprintf(FILE * restrict stream, const wchar_t * restrict format, ...)

int
swprintf(wchar_t * restrict ws, size_t n,
        const wchar_t * restrict format, ...)

int
wprintf(const wchar_t * restrict format, ...)

#include <stdarg.h>

int
vfwprintf(FILE * restrict stream,
        const wchar_t * restrict, va_list ap)

int
vswprintf(wchar_t * restrict ws, size_t n,
        const wchar_t *restrict format, va_list ap)

int
vwprintf(const wchar_t * restrict format, va_list ap)
```

DESCRIPTION

The **wprintf()** family of functions produces output according to a *format* as described below. The **wprintf()** and **vwprintf()** functions write output to `stdout`, the standard output stream; **fwprintf()** and **vfwprintf()** write output to the given output *stream*; **swprintf()** and **vswprintf()** write to the wide character string *ws*.

These functions write the output under the control of a *format* string that specifies how subsequent arguments (or arguments accessed via the variable-length argument facilities of stdarg(3)) are converted for output.

These functions return the number of characters printed (not including the trailing '\0' used to end output to strings).

The **swprintf()** and **vswprintf()** functions will fail if n or more wide characters were requested to be written,

The format string is composed of zero or more directives: ordinary characters (not %), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the % character. The arguments must correspond properly (after type promotion) with the conversion specifier. After the %, the following appear in sequence:

- An optional field, consisting of a decimal digit string followed by a $, specifying the next argument to access. If this field is not provided, the argument following the last argument accessed will be used. Arguments are numbered starting at 1. If unaccessed arguments in the format string are interspersed with ones that are accessed the results will be indeterminate.
- Zero or more of the following flags:

'#' The value should be converted to an "alternate form". For c, d, i, n, p, s, and u conversions, this option has no effect. For o conversions, the precision of the number is increased to force the first character of the output string to a zero (except if a zero value is printed with an explicit precision of zero). For x and X conversions, a non-zero result has the string '0x' (or '0X' for X conversions) prepended to it. For a, A, e, E, f, F, g, and G conversions, the result will always contain a decimal point, even if no digits follow it (normally, a decimal point appears in the results of those conversions only if a digit follows). For g and G conversions, trailing zeros are not removed from the result as they would otherwise be.

'0' (zero) Zero padding. For all conversions except n, the converted value is padded on the left with zeros rather than blanks. If a precision is given with a numeric conversion (d, i, o, u, i, x, and X), the 0 flag is ignored.

'-' A negative field width flag; the converted value is to be left adjusted on the field boundary. Except for n conversions, the converted value is padded on the right with blanks, rather than on the left with blanks or zeros. A - overrides a 0 if both are given.

' ' (space) A blank should be left before a positive number produced by a signed conversion (a, A, d, e, E, f, F, g, G, or i).

'+' A sign must always be placed before a number produced by a signed conversion. A + overrides a space if both are used.

‘,’ Decimal conversions (`d`, `u`, or `i`) or the integral portion of a floating point conversion (`f` or `F`) should be grouped and separated by thousands using the non-monetary separator returned by localeconv(3).

- An optional decimal digit string specifying a minimum field width. If the converted value has fewer characters than the field width, it will be padded with spaces on the left (or right, if the left-adjustment flag has been given) to fill out the field width.

- An optional precision, in the form of a period `.` followed by an optional digit string. If the digit string is omitted, the precision is taken as zero. This gives the minimum number of digits to appear for `d`, `i`, `o`, `u`, `x`, and `X` conversions, the number of digits to appear after the decimal-point for `a`, `A`, `e`, `E`, `f`, and `F` conversions, the maximum number of significant digits for `g` and `G` conversions, or the maximum number of characters to be printed from a string for `s` conversions.

- An optional length modifier, that specifies the size of the argument. The following length modifiers are valid for the `d`, `i`, `n`, `o`, `u`, `x`, or `X` conversion:

Modifier	`d`, `i`	`o`, `u`, `x`, `X`	`n`
`hh`	`signed char`	`unsigned char`	`signed char *`
`h`	`short`	`unsigned short`	`short *`
`l` (ell)	`long`	`unsigned long`	`long *`
`ll` (ell ell)	`long long`	`unsigned long long`	`long long *`
`j`	`intmax_t`	`uintmax_t`	`intmax_t *`
`t`	`ptrdiff_t`	(see note)	`ptrdiff_t *`
`z`	(see note)	`size_t`	(see note)
`q` *(deprecated)*	`quad_t`	`u_quad_t`	`quad_t *`

  Note: the `t` modifier, when applied to a `o`, `u`, `x`, or `X` conversion, indicates that the argument is of an unsigned type equivalent in size to a `ptrdiff_t`. The `z` modifier, when applied to a `d` or `i` conversion, indicates that the argument is of a signed type equivalent in size to a `size_t`. Similarly, when applied to an `n` conversion, it indicates that the argument is a pointer to a signed type equivalent in size to a `size_t`.

  The following length modifier is valid for the `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion:

Modifier	`a`, `A`, `e`, `E`, `f`, `F`, `g`, `G`
`L`	`long double`

  The following length modifier is valid for the `c` or `s` conversion:

Modifier	`c`	`s`
`l` (ell)	`wint_t`	`wchar_t *`

- A character that specifies the type of conversion to be applied.

A field width or precision, or both, may be indicated by an asterisk ‘`*`’ or an asterisk followed by one or more decimal digits and a ‘`$`’ instead of a digit string. In this case, an `int` argument supplies the field width or precision. A negative field width

is treated as a left adjustment flag followed by a positive field width; a negative precision is treated as though it were missing. If a single format directive mixes positional (nn$) and non-positional arguments, the results are undefined.

The conversion specifiers and their meanings are:

`diouxX` The `int` (or appropriate variant) argument is converted to signed decimal (`d` and `i`), unsigned octal (`o`), unsigned decimal (`u`), or unsigned hexadecimal (`x` and `X`) notation. The letters `abcdef` are used for `x` conversions; the letters `ABCDEF` are used for `X` conversions. The precision, if any, gives the minimum number of digits that must appear; if the converted value requires fewer digits, it is padded on the left with zeros.

`eE` The `double` argument is rounded and converted in the style [-]*d.ddd*e+-*dd* where there is one digit before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero, no decimal-point character appears. An `E` conversion uses the letter '`E`' (rather than '`e`') to introduce the exponent. The exponent always contains at least two digits; if the value is zero, the exponent is 00.

For `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversions, positive and negative infinity are represented as `inf` and `-inf` respectively when using the lowercase conversion character, and `INF` and `-INF` respectively when using the uppercase conversion character. Similarly, NaN is represented as `nan` when using the lowercase conversion, and `NAN` when using the uppercase conversion.

`fF` The `double` argument is rounded and converted to decimal notation in the style [-]*ddd.ddd* where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is explicitly zero, no decimal-point character appears. If a decimal point appears, at least one digit appears before it.

`gG` The `double` argument is converted in style `f` or `e` (or `F` or `E` for `G` conversions). The precision specifies the number of significant digits. If the precision is missing, 6 digits are given; if the precision is zero, it is treated as 1. Style `e` is used if the exponent from its conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional part of the result; a decimal point appears only if it is followed by at least one digit.

`aA` The `double` argument is converted to hexadecimal notation in the style [-]0x*h.hhhp*[+-]d where the number of digits after the hexadecimal-point character is equal to the precision specification. If the precision is missing, it is taken as enough to exactly represent the floating-point number; if the precision is explicitly zero, no hexadecimal-point character appears. This is an exact conversion of the mantissa+exponent internal

floating point representation; the [-]0x*h.hhh* portion represents exactly the mantissa; only denormalized mantissas have a zero value to the left of the hexadecimal point. The `p` is a literal character '`p`'; the exponent is preceded by a positive or negative sign and is represented in decimal, using only enough characters to represent the exponent. The `A` conversion uses the prefix "`0X`" (rather than "`0x`"), the letters "`ABCDEF`" (rather than "`abcdef`") to represent the hex digits, and the letter '`P`' (rather than '`p`') to separate the mantissa and exponent.

`D` The `_Decimal` argument is printed. After the conversion specifier, an argument of (*size*`,`*prec*) must appear specifying the size and precision.

`C` Treated as `c` with the `l` (ell) modifier.

`c` The `int` argument is converted to an `unsigned char`, then to a `wchar_t` as if by btowc(3), and the resulting character is written.

If the `l` (ell) modifier is used, the `wint_t` argument is converted to a `wchar_t` and written.

`S` Treated as `s` with the `l` (ell) modifier.

`s` The `char *` argument is expected to be a pointer to an array of character type (pointer to a string) containing a multibyte sequence. Characters from the array are converted to wide characters and written up to (but not including) a terminating `NUL` character; if a precision is specified, no more than the number specified are written. If a precision is given, no null character need be present; if the precision is not specified, or is greater than the size of the array, the array must contain a terminating `NUL` character.

If the `l` (ell) modifier is used, the `wchar_t *` argument is expected to be a pointer to an array of wide characters (pointer to a wide string). Each wide character in the string is written. Wide characters from the array are written up to (but not including) a terminating wide `NUL` character; if a precision is specified, no more than the number specified are written (including shift sequences). If a precision is given, no null character need be present; if the precision is not specified, or is greater than the number of characters in the string, the array must contain a terminating wide `NUL` character.

`p` The `void *` pointer argument is printed in hexadecimal (as if by '`%#x`' or '`%#lx`').

`n` The number of characters written so far is stored into the integer indicated by the `int *` (or variant) pointer argument. No argument is converted.

`%` A '`%`' is written. No argument is converted. The complete conversion specification is '`%%`'.

The decimal point character is defined in the program's locale (category `LC_NUMERIC`).

In no case does a non-existent or small field width cause truncation of a numeric field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

SECURITY CONSIDERATIONS

Refer to printf(3).

SEE ALSO

btowc(3), fputws(3), printf(3), putwc(3), setlocale(3), wcsrtombs(3), wscanf(3)

STANDARDS

The **wprintf()**, **fwprintf()**, **swprintf()**, **vwprintf()**, **vfwprintf()** and **vswprintf()** functions conform to ISO/IEC 9899:1999 ("ISO C99").

WSCANF(3)

NAME

wscanf, fwscanf, swscanf, vwscanf, vswscanf, vfwscanf - wide character input format conversion

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

int
wscanf(const wchar_t * restrict format, ...)

int
fwscanf(FILE * restrict stream, const wchar_t * restrict format,
        ...)

int
swscanf(const wchar_t * restrict str,
        const wchar_t * restrict format, ...)

#include <stdarg.h>

int
vwscanf(const wchar_t * restrict format, va_list ap)

int
vswscanf(const wchar_t * restrict str,
        const wchar_t * restrict format, va_list ap)

int
vfwscanf(FILE * restrict stream, const wchar_t * restrict format,
        va_list ap)
```

DESCRIPTION

The **wscanf()** family of functions scans input according to a *format* as described below. This format may contain *conversion specifiers*; the results from such conversions, if any, are stored through the *pointer* arguments. The **wscanf()** function reads input from the standard input stream `stdin`, **fwscanf()** reads input from the stream pointer *stream*, and **swscanf()** reads its input from the wide character string pointed to by *str*. The **vfwscanf()** function is analogous to vfwprintf(3)

and reads input from the stream pointer *stream* using a variable argument list of pointers (see stdarg(3)). The **vwscanf()** function scans a variable argument list from the standard input and the **vswscanf()** function scans it from a wide character string; these are analogous to the **vwprintf()** and **vswprintf()** functions respectively. Each successive *pointer* argument must correspond properly with each successive conversion specifier (but see the `*` conversion below). All conversions are introduced by the `%` (percent sign) character. The *format* string may also contain other characters. White space (such as blanks, tabs, or newlines) in the *format* string match any amount of white space, including none, in the input. Everything else matches only itself. Scanning stops when an input character does not match such a format character. Scanning also stops when an input conversion cannot be made (see below).

CONVERSIONS

Following the `%` character introducing a conversion there may be a number of *flag* characters, as follows:

`*`	Suppresses assignment. The conversion that follows occurs as usual, but no pointer is used; the result of the conversion is simply discarded.
`hh`	Indicates that the conversion will be one of `dioux` or `n` and the next pointer is a pointer to a `char` (rather than `int`).
`h`	Indicates that the conversion will be one of `dioux` or `n` and the next pointer is a pointer to a `short int` (rather than `int`).
`l` (ell)	Indicates that the conversion will be one of `dioux` or `n` and the next pointer is a pointer to a `long int` (rather than `int`), that the conversion will be one of `a`, `e`, `f`, or `g` and the next pointer is a pointer to `double` (rather than `float`), or that the conversion will be one of `c` or `s` and the next pointer is a pointer to an array of `wchar_t` (rather than `char`).
`ll` (ell ell)	Indicates that the conversion will be one of `dioux` or `n` and the next pointer is a pointer to a `long long int` (rather than `int`).
`L`	Indicates that the conversion will be one of `a`, `e`, `f`, or `g` and the next pointer is a pointer to `long double`.
`j`	Indicates that the conversion will be one of `dioux` or `n` and the next pointer is a pointer to a `intmax_t` (rather than `int`).
`t`	Indicates that the conversion will be one of `dioux` or `n` and the next pointer is a pointer to a `ptrdiff_t` (rather than `int`).
`z`	Indicates that the conversion will be one of `dioux` or `n` and the next pointer is a pointer to a `size_t` (rather than `int`).

`q` (deprecated) Indicates that the conversion will be one of `dioux` or `n` and the next pointer is a pointer to a `long long int` (rather than `int`).

In addition to these flags, there may be an optional maximum field width, expressed as a decimal integer, between the `%` and the conversion. If no width is given, a default of "infinity" is used (with one exception, below); otherwise at most this many characters are scanned in processing the conversion. Before conversion begins, most conversions skip white space; this white space is not counted against the field width.

The following conversions are available:

`%` Matches a literal '`%`'. That is, "`%%`" in the format string matches a single input '`%`' character. No conversion is done, and assignment does not occur.

`d` Matches an optionally signed decimal integer; the next pointer must be a pointer to `int`.

`i` Matches an optionally signed integer; the next pointer must be a pointer to `int`. The integer is read in base 16 if it begins with '`0x`' or '`0X`', in base 8 if it begins with '`0`', and in base 10 otherwise. Only characters that correspond to the base are used.

`o` Matches an octal integer; the next pointer must be a pointer to `unsigned int`.

`u` Matches an optionally signed decimal integer; the next pointer must be a pointer to `unsigned int`.

`x`, `X` Matches an optionally signed hexadecimal integer; the next pointer must be a pointer to `unsigned int`.

`a`, `A`, `e`, `E`, `f`, `F`, `g`, `G` Matches a floating-point number in the style of wcstod(3). The next pointer must be a pointer to `float` (unless `l` or `L` is specified.)

`s` Matches a sequence of non-white-space wide characters; the next pointer must be a pointer to `char`, and the array must be large enough to accept the multibyte representation of all the sequence and the terminating `NUL` character. The input string stops at white space or at the maximum field width, whichever occurs first.

If an `l` qualifier is present, the next pointer must be a pointer to `wchar_t`, into which the input will be placed.

`S` The same as `ls`.

`c` Matches a sequence of *width* count wide characters (default 1); the next pointer must be a pointer to `char`, and there must be enough room for the multibyte representation of all the characters (no terminating `NUL` is added). The usual skip of leading white space is suppressed. To skip white space first, use an explicit space in the format.

If an `l` qualifier is present, the next pointer must be a pointer to `wchar_t`, into which the input will be placed.

`C` The same as `lc`.

`[` Matches a nonempty sequence of characters from the specified set of accepted characters; the next pointer must be a pointer to `char`, and there must be enough room for the multibyte representation of all the characters in the string, plus a terminating `NUL` character. The usual skip of leading white space is suppressed. The string is to be made up of characters in (or not in) a particular set; the set is defined by the characters between the open bracket `[` character and a close bracket `]` character. The set *excludes* those characters if the first character after the open bracket is a circumflex `^`. To include a close bracket in the set, make it the first character after the open bracket or the circumflex; any other position will end the set. To include a hyphen in the set, make it the last character before the final close bracket; some implementations of **wscanf()** use "`A-Z`" to represent the range of characters between '`A`' and '`Z`'. The string ends with the appearance of a character not in the (or, with a circumflex, in) set or when the field width runs out.

If an `l` qualifier is present, the next pointer must be a pointer to `wchar_t`, into which the input will be placed.

`p` Matches a pointer value (as printed by '`%p`' in wprintf(3)); the next pointer must be a pointer to `void`.

`n` Nothing is expected; instead, the number of characters consumed thus far from the input is stored through the next pointer, which must be a pointer to `int`. This is *not* a conversion, although it can be suppressed with the `*` flag.

The decimal point character is defined in the program's locale (category `LC_NUMERIC`).

For backwards compatibility, a "conversion" of '`%\0`' causes an immediate return of `EOF`.

RETURN VALUES

These functions return the number of input items assigned, which can be fewer than provided for, or even zero, in the event of a matching failure. Zero indicates that, while there was input available, no conversions were assigned; typically this is due to an invalid input character, such as an alphabetic character for a '`%d`' conversion. The value `EOF` is returned if an input failure occurs before any conversion such as an end-of-file occurs. If an error or end-of-file occurs after conversion has begun, the number of conversions which were successfully completed is returned.

SEE ALSO

fgetwc(3), scanf(3), wcrtomb(3), wcstod(3), wcstol(3), wcstoul(3), wprintf(3)

STANDARDS

The **fwscanf()**, **wscanf()**, **swscanf()**, **vfwscanf()**, **vwscanf()** and **vswscanf()** functions conform to ISO/IEC 9899:1999 ("ISO C99").

BUGS

In addition to the bugs documented in scanf(3), **wscanf()** does not support the "`A-Z`" notation for specifying character ranges with the character class conversion ('`%[`').

The Standard Library

Also present in the ANSI C definition are miscellaneous functions considered to be part of the "standard" library. This section also describes some Systems/C extensions which are similar to some of the ANSI standard functions.

__FREE24(3)

NAME

__free24 - free memory allocated with __malloc24

SYNOPSIS

```
#include <stdlib.h>

void
__free24(void * ptr)
```

DESCRIPTION

The **__free24()** function causes the space pointed to by *ptr* to be deallocated, that is, made available for further allocation. If *ptr* is a null pointer, no action occurs. Otherwise, if the argument does not match a pointer earlier returned by the **__malloc24()** function, or if the space has been deallocated by a call to **__free24()**, then general havoc may occur.

RETURN VALUES

The **__free24()** function returns no value.

SEE ALSO

__malloc24(3), malloc(3), free(3), __malloc31(3), __free31(3)

__FREE31(3)

NAME

__free31 - free memory allocated with __malloc31

SYNOPSIS

```
#include <stdlib.h>

void
__free31(void * ptr)
```

DESCRIPTION

The **__free31()** function causes the space pointed to by *ptr* to be deallocated, that is, made available for further allocation. If *ptr* is a null pointer, no action occurs. Otherwise, if the argument does not match a pointer earlier returned by the **__malloc31()** function, or if the space has been deallocated by a call to **__free31()**, then general havoc may occur.

RETURN VALUES

The **__free31()** function returns no value.

SEE ALSO

__malloc31(3), malloc(3), free(3), __malloc24(3), __free24(3)

__MALLOC24(3)

NAME

__malloc24 - allocate memory which is guaranteed to be 24-bit addressable

SYNOPSIS

```
#include <stdlib.h>

void *
__malloc24(size_t size)
```

DESCRIPTION

The **__malloc24()** function allocates uninitialized space for an object whose size is specified by *size*. The **__malloc24()** function maintains lists of free blocks, allocating space from the appropriate list when possible.

Any space returned by the **__malloc24()** function is guaranteed to reside below the 16-megabyte "line", and thus be addressable as a 24-bit address.

Memory space allocated with **__malloc24()** must be returned to the list of available space via the **__free24()** function.

RETURN VALUES

The **__malloc24()** function returns a pointer to the allocated space if successful; otherwise a null pointer is returned.

SEE ALSO

__free24(3), malloc(3), free(3), __malloc31(3), __free31(3), memory(3)

__MALLOC31(3)

NAME

__malloc31 - allocate memory which is guaranteed to be 31-bit addressable

SYNOPSIS

```
#include <stdlib.h>

void *
__malloc31(size_t size)
```

DESCRIPTION

The **__malloc31()** function allocates uninitialized space for an object whose size is specified by *size*. The **__malloc31()** function maintains lists of free blocks, allocating space from the appropriate list when possible.

Any space returned by the **__malloc31()** function is guaranteed to reside below the 2-gigabytes "bar", and thus be addressable with a 31-bit pointer.

Memory space allocated with **__malloc31()** must be returned to the list of available space via the **__free31()** function.

RETURN VALUES

The **__malloc31()** function returns a pointer to the allocated space if successful; otherwise a null pointer is returned.

SEE ALSO

__free31(3), malloc(3), free(3), __malloc24(3), __free24(3), memory(3)

ABORT(3)

NAME

abort - cause abnormal program termination

SYNOPSIS

```
#include <stdlib.h>

void
abort(void)
```

DESCRIPTION

The **abort()** function causes abnormal program termination to occur, unless the signal `SIGABRT` is being caught and the signal handler does not return.

Any open streams are flushed and closed.

RETURN VALUES

The **abort()** function never returns.

SEE ALSO

signal(2), exit(3)

ABS(3)

NAME

abs - integer absolute value function

SYNOPSIS

```
#include <stdlib.h>

int
abs(int j)
```

DESCRIPTION

The **abs()** function computes the absolute value of the integer *j*.

RETURN VALUES

The **abs()** function returns the absolute value.

SEE ALSO

cabs(3), floor(3), imaxabs(3), hypot(3), labs(3), math(3)

STANDARDS

The **abs()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

ISSUES

The absolute value of the most negative integer remains negative.

ARC4RANDOM(3)

NAME

arc4random, arc4random_stir, arc4random_addrandom - arc4 random number generator

SYNOPSIS

```
#include <stdlib.h>

u_int32_t
arc4random(void);

void
arc4random_stir(void);

void
arc4random_addrandom(unsigned char *dat, int datlen);
```

DESCRIPTION

The **arc4random()** function uses the key stream generator employed by the arc4 cipher, which uses 8*8 8 bit S-Boxes. The S-Boxes can be in about (2**1700) states. The **arc4random()** function returns pseudo-random numbers in the range of 0 to (2**31)-1, and therefore has twice the range of `RAND_MAX`.

The **arc4random_stir()** function attemps to reads data from a random device generator or other random values and use that data to permute the S-Boxes via **arc4random_addrandom()**.

There is no need to **call arc4random_stir()** before using **arc4random()**, since **arc4random()** automatically initializes itself.

EXAMPLES

The following produces a drop-in replacement for the traditional **rand()** and **random()** functions using **arc4random()**:

```
#define foo4random() (arc4random() % ((unsigned)RAND_MAX + 1))
```

SEE ALSO

rand(3), random(3)

HISTORY

`RC4` was designed by RSA Data Security, Inc. It was posted anonymously to USENET and was confirmed to be equivalent by several sources who had access to the original cipher. Since `RC4` used to be a trade secret, the cipher is now referred to as `ARC4`.

ATEXIT(3)

NAME

atexit - register a function to be called on exit

SYNOPSIS

```
#include <stdlib.h>

int
atexit(void (*function)(void))
```

DESCRIPTION

The **atexit()** function registers the given *function* to be called at program exit, whether via exit(3) or via return from the program's **main()**. Functions so registered are called in reverse order; no arguments are passed. At least 32 functions can always be registered, and more are allowed as long as sufficient memory can be allocated.

RETURN VALUES

The **atexit()** function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

`[ENOMEM]` No memory was available to add the function to the list. The existing list of functions is unmodified.

SEE ALSO

exit(3)

STANDARDS

The **atexit()** function conforms to ISO/IEC 9899:1990 ("ISO C90").

ATOF(3)

NAME

atof - convert string to double

SYNOPSIS

```
#include <stdlib.h>

double
atof(const char *nptr)
```

DESCRIPTION

The **atof()** function converts the initial portion of the string pointed to by *nptr* to `double` representation.

It is equivalent to:

```
strtod(nptr, (char **)NULL);
```

SEE ALSO

atoi(3), atol(3), strtod(3), strtol(3), strtoul(3)

STANDARDS

The **atof()** function conforms to ISO/IEC 9899:1990 ("ISO C90").

ISSUES

This manual page represents intent instead of actual practice. While it is intended that **atof()** be implemented using strtod(3), this has not yet happened.

In the current system, **atof()** translates a string in the following form to a `double`: a string of leading white space, possibly followed by a sign ("`+`" or "`-`"), followed by a digit string which may contain one decimal point ("`.`"), that may be followed by either of the exponent flags ("`E`" or "`e`"), and lastly, followed by a signed or unsigned integer.

ATOI(3)

NAME

atoi - convert string to integer

SYNOPSIS

```
#include <stdlib.h>

int
atoi(const char *nptr)
```

DESCRIPTION

The **atoi()** function converts the initial portion of the string pointed to by *nptr* to integer representation.

It is equivalent to:

```
(int)strtol(nptr, (char **)NULL, 10);
```

SEE ALSO

atof(3), atol(3), strtod(3), strtol(3), strtoul(3)

STANDARDS

The **atoi()** function conforms to ISO/IEC 9899:1990 ("ISO C90").

ATOL(3)

NAME

atol - convert string to long integer

SYNOPSIS

```
#include <stdlib.h>

long
atol(const char *nptr)

long long
atoll(const char *nptr);
```

DESCRIPTION

The **atol()** function converts the initial portion of the string pointed to by *nptr* to long integer representation.

It is equivalent to:

```
strtol(nptr, (char **)NULL, 10);
```

The **atoll()** function converts the initial portion of the string pointed to by *nptr* to long long integer representation.

It is equivalent to:

```
strtoll(nptr, (char **)NULL, 10);
```

ERRORS

The functions **atol()** and **atoll()** need not affect the value of `errno` on an error.

SEE ALSO

atof(3), atoi(3), strtod(3), strtol(3), strtoul(3)

STANDARDS

The **atol()** function conforms to ISO/IEC 9899:1990 ("ISO C90"). The **atoll()** function conforms to ISO/IEC 9899:1999 ("ISO C99").

BSEARCH(3)

NAME

bsearch - binary search of a sorted table

SYNOPSIS

```
#include <stdlib.h>

void *
bsearch(const void *key, const void *base,
size_t nmemb, size_t size,
int (*compar) (const void *, const void *))
```

DESCRIPTION

The **bsearch()** function searches an array of *nmemb* objects, the initial member of which is pointed to by *base*, for a member that matches the object pointed to by *key*. The size of each member of the array is specified by *size*.

The contents of the array should be in ascending sorted order according to the comparison function referenced by *compar*. The *compar* routine is expected to have two arguments which point to the key object and to an array member, in that order, and should return an integer less than, equal to, or greater than zero if the key object is found, respectively, to be less than, to match, or be greater than the array member.

RETURN VALUES

The **bsearch()** function returns a pointer to a matching member of the array, or a null pointer if no match is found. If two members compare as equal, which member is matched is unspecified.

SEE ALSO

lsearch(3), qsort(3)

STANDARDS

The **bsearch()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

CALLOC(3)

NAME

calloc - allocate clean memory (zero initialized space)

SYNOPSIS

```
#include <stdlib.h>

void *
calloc(size_t nmemb, size_t size)
```

DESCRIPTION

The **calloc()** function allocates space for an array of *nmemb* objects, each of whose size is *size*. The space is initialized to all bits zero.

RETURN VALUES

The **calloc()** function returns a pointer to the allocated space if successful; otherwise a null pointer is returned.

SEE ALSO

malloc(3), realloc(3), free(3),

STANDARDS

The **calloc()** function conforms to ISO/IEC 9899:1990 ("ISO C90").

DIV(3)

NAME

div - return quotient and remainder from division

SYNOPSIS

```
#include <stdlib.h>

div_t
div(int num, int denom)
```

DESCRIPTION

The **div()** function computes the value *num*/*denum* and returns the quotient and remainder in a structure named `div_t` that contains two `int` integer members named `quot` and `rem`. Unlike the implementation defined semantics of normal integer division, **div()** provides precise semantics on the `quot` and `rem` values.

The sign of `quot` is the same as the algebraic quotient.

If the division is inexact, the magnitude of the resulting quotient is the largest integer less than the magnitude of the algebraic quotient.

If the result cannot be represented, the values of `quot` and `rem` are undefined.

If the result can be represented, then `quot * `*denom*` + rem` will equal *num*.

SEE ALSO

imaxdiv(3), ldiv(3), lldiv(3)

STANDARDS

The **div()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

ENVIRON(7)

NAME

environ - user environment

SYNOPSIS

```
extern char **environ;
```

DESCRIPTION

An array of strings called the **environment** is made available when a program begins. By convention, these strings have the form `"name=value"`. When a Systems/C program is initiated via execve(2), these strings are taken from the POSIX environment.

The *environ* variable is the anchor for the getenv(3) family of functions, and should not be otherwised used in programs.

SEE ALSO

execve(2), execle(3), getenv(3), setenv(3), setlocale(3), system(3)

EXIT(3)

NAME

exit - perform normal program termination

SYNOPSIS

```
#include <stdlib.h>

void
exit(int status)
```

DESCRIPTION

exit() terminates a process.

Before termination it performs the following functions in the order listed:

1. Call the functions registered with the atexit(3) function, in the reverse order of their registration.

2. Flush all open output streams.

3. Close all open streams.

4. Unlink all files created with the tmpfile(3) function.

Passing arbitrary values back to the environment as *status* is considered bad style. Instead, use the values as described in sysexits(3).

RETURN VALUES

The **exit()** function never returns.

SEE ALSO

_exit(2), atexit(3), intro(3), sysexits(3)

STANDARDS

The **exit()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

FREE(3)

NAME

free - free memory allocated with malloc, calloc or realloc

SYNOPSIS

```
#include <stdlib.h>

void
free(void *ptr)
```

DESCRIPTION

The **free()** function causes the space pointed to by *ptr* to be deallocated, that is, made available for further allocation. If *ptr* is a null pointer, no action occurs. Otherwise, if the argument does not match a pointer earlier returned by the calloc, malloc, or realloc function, or if the space has been deallocated by a call to **free()** or realloc, then general havoc may occur.

RETURN VALUES

The **free()** function returns no value.

SEE ALSO

calloc(3), malloc(3), realloc(3)

STANDARDS

The **free()** function conforms to ISO/IEC 9899:1990 ("ISO C90").

GETENV(3)

NAME

getenv, putenv, setenv, unsetenv - environment variable functions

SYNOPSIS

```
#include <stdlib.h>

char *
getenv(const char *name)

int
setenv(const char *name, const char *value,
int overwrite)

int
putenv(const char *string)

int
unsetenv(const char *name)
```

DESCRIPTION

These functions set, unset and fetch environment variables from the host environment list. For compatibility with differing environment conventions, the given arguments name and value may be appended and prepended, respectively, with an equal sign "=".

The **getenv()** function obtains the current value of the environment variable, *name*. If the variable name is not in the current environment, a null pointer is returned.

The **setenv()** function inserts or resets the environment variable name in the current environment list. If the variable name does not exist in the list, it is inserted with the given value. If the variable does exist, the argument *overwrite* is tested; if *overwrite* is zero, the variable is not reset, otherwise it is reset to the given value.

The **putenv()** function takes an argument of the form "name=value" and is equivalent to:

```
setenv(name, value, 1);
```

The **unsetenv()** function deletes all instances of the variable name pointed to by *name* from the list.

RETURN VALUES

The functions **setenv()**, **putenv()** and **unsetenv()** return zero if successful. Otherwise the global variable `errno` is set to indicate the error and a -1 is returned.

ERRORS

`[EINVAL]` The function **setenv()** or **unsetenv()** failed because the name is a NULL pointer, points to an empty string, or points to a string containing an "=" character.

`[ENOMEM]` The function **setenv()** or **putenv()** failed because they were unable to allocate memory for the environment.

SEE ALSO

environ(7)

STANDARDS

The **getenv()** function conforms to ISO/IEC 9899:1990 ("ISO C90").

GETOPT(3)

NAME

getopt - get option character from command line argument list

SYNOPSIS

```
#include <stdlib.h>

extern char *optarg;
extern int optind;
extern int optopt;
extern int opterr;
extern int optreset;

int
getopt(int argc, char * const *argv,
const char *optstring)
```

DESCRIPTION

The **getopt()** function incrementally parses a command line argument list *argv* and returns the next known option character. An option character is known if it has been specified in the string of accepted option characters, optstring.

The option string *optstring* may contain the following elements: individual characters, and characters followed by a colon to indicate an option argument is to follow. For example, an option string `"x"` recognizes an option `"-x"`, and an option string `"x:"` recognizes an option and argument `"-x argument"`. Leading white space in a following argument does not affect the operation of **getopt()**.

On return from **getopt()**, `optarg` points to an option argument, if it is anticipated, and the variable `optind` contains the index to the next *argv* argument for a subsequent call to **getopt()**. The variable `optopt` saves the last known option character returned by **getopt()**.

The variable `opterr` and `optind` are both initialized to 1. The `optind` variable may be set to another value before a set of calls to **getopt()** in order to skip over more or less *argv* entries.

In order to use **getopt()** to evaluate multiple sets of arguments or to evaluate a single set of arguments multiple times, the variable `optreset` must be set to 1 before the second and each additional set of calls to **getopt()**, and the variable `optind` must be reinitialized.

The **getopt()** function returns -1 when the argument list is exhausted, or '?' if a non-recognized option is encountered. The interpretation of options in the argument list may be canceled by the option "`--`" (double dash) which causes **getopt()** to signal the end of argument processing and return `-1`. When all options have been processed (i.e., up to the first non-option argument), **getopt()** returns `-1`.

DIAGNOSTICS

If the **getopt()** function encounters a character not found in the string `optarg` or detects a missing option argument it writes an error message to the stderr and returns '`?`'. Setting `opterr` to a zero will disable these error messages. If *optstring* has a leading '`:`' then a missing option argument causes a '`:`' to be returned in addition to suppressing any error messages.

Option arguments are allowed to begin with "`-`"; this is reasonable but reduces the amount of error checking possible.

EXTENSIONS

The `optreset` variable was added to make it possible to call the **getopt()** function multiple times. This is an extension to the IEEE Std1003.2 ("POSIX.2") specification.

EXAMPLE

```
extern char *optarg;
extern int optind;
int bflag, ch, fd;

bflag = 0;
while ((ch = getopt(argc, argv, "bf:")) != -1) {
   switch(ch) {
   case 'b':
      bflag = 1;
      break;
   case 'f':
      if ((fd = open(optarg, O_RDONLY, 0)) < 0) {
         (void)fprintf(stderr,
                       "myname: %s: %s\n",
                       optarg,
                       strerror(errno));
         exit(1);
      }
      break;
```

```
   case '?':
      default:
      usage();
   }
}
argc -= optind;
argv += optind;
```

ISSUES

The **getopt()** function was once specified to return `EOF` instead of -1. This was changed by IEEE Std1003.2-1992 ("POSIX.2") to decouple **getopt()** from `<stdio.h>`.

A single dash "`-`" may be specified as a character in *optstring*, however it should never have an argument associated with it. This allows **getopt()** to be used with programs that expect "`-`" as an option flag. This practice is wrong, and should not be used in any current development. It is provided for backward compatibility only. By default, a single dash causes **getopt()** to return -1. This is, we believe, compatible with System V.

It is also possible to handle digits as option letters. This allows **getopt()** to be used with programs that expect a number ("-3") as an option. This practice is wrong, and should not be used in any current development. It is provided for backward compatibility only. The following code fragment works in most cases.

```
int length;
char *p;

while((c = getopt(argc, argv, ``0123456789'')) != -1) {
   switch (c) {
   case '0': case '1': case '2': case '3':
   case '4': case '5': case '6': case '7':
   case '8': case '9':
      p = argv[optind - 1];
      if (p[0] == '-' && p[1] == ch && !p[2])
         length = atoi(++p);
      else
         length = atoi(argv[optind] + 1);
      break;
   }
}
```

GETSUBOPT(3)

NAME

getsubopt - get sub options from an argument

SYNOPSIS

```
#include <stdlib.h>

extern char *suboptarg

int
getsubopt(char **optionp, char * const *tokens,
char **valuep)
```

DESCRIPTION

The **getsubopt()** function parses a string containing tokens delimited by one or more tab, space or comma (',') character. It is intended for use in parsing groups of option arguments provided as part of a utility command line.

The argument *optionp* is a pointer to a pointer to the string. The argument tokens is a pointer to a `NULL`-terminated array of pointers to strings.

The **getsubopt()** function returns the zero-based offset of the pointer in the tokens array referencing a string which matches the first token in the string, or, -1 if the string contains no tokens or tokens does not contain a matching string.

If the token is of the form "name=value" the location referenced by *valuep* will be set to point to the start of the "value" portion of the token.

On return from **getsubopt()**, *optionp* will be set to point to the start of the next token in the string, or the null at the end of the string if no more tokens are present. The external variable `suboptarg` will be set to point to the start of the current token, or `NULL` if no tokens were present. The argument *valuep* will be set to point to the "value" portion of the token, or `NULL` if no "value" portion was present.

EXAMPLE

```
char *tokens[] = {
   #define ONE     0
       "one",
   #define TWO     1
```

```
        "two",
        NULL
};

...

extern char *optarg, *suboptarg;
char *options, *value;

while ((ch = getopt(argc, argv, "ab:")) != -1) {
   switch(ch) {
   case 'a':
      /* process ``a'' option */
      break;
   case 'b':
      options = optarg;
      while (*options) {
         switch(getsubopt(&options,
       tokens, &value)) {
         case ONE:
            /* process "one" sub option */
            break;
         case TWO:
            /* process "two" sub option */
            if (!value)
               error("no value for two");
            i = atoi(value);
            break;
         case -1:
            if (suboptarg)
               error("illegal sub option %s",
                     suboptarg);
            else
               error("missing sub option");
            break;
 }
      }
      break;
   }
}
```

SEE ALSO

getopt(3), strsep(3)

IMAXABS(3)

NAME

imaxabs - return the absolute value of a intmax_t integer

SYNOPSIS

```
#include <inttypes.h>

intmax_t
imaxabs(intmax_t j)
```

DESCRIPTION

The **imaxabs()** function returns the absolute value of the `intmax_t` integer *j*.

SEE ALSO

abs(3), cabs(3), labs(3), llabs(3), floor(3), math(3)

STANDARDS

The **imaxabs()** function conforms to ISO/IEC 9899:1999 (“ISO C99”).

IMAXDIV(3)

NAME

imaxdiv - return quotient and remainder from division

SYNOPSIS

```
#include <inttypes.h>

imaxdiv\_t
imaxdiv(intmax\_t num, intmax\_t denom)
```

DESCRIPTION

The **imaxdiv()** function computes the value *num*/*denum* and returns the quotient and remainder in a structure named `imaxdiv_t` that contains two `intmax_t` integer members named `quot` (the quotient) and `rem` (the remainder).

If the result cannot be represented, the values of `quot` and `rem` are undefined.

SEE ALSO

div(3), ldiv(3), lldiv(3)

STANDARDS

The **imaxdiv()** function conforms to ISO/IEC 9899:1999 ("ISO C99").

LABS(3)

NAME

labs - return the absolute value of a long integer

SYNOPSIS

```
#include <stdlib.h>

long
labs(long j)
```

DESCRIPTION

The **labs()** function returns the absolute value of the `long` integer *j*.

SEE ALSO

abs(3), cabs(3), imaxabs(3), floor(3), math(3)

STANDARDS

The **labs()** function conforms to ISO/IEC 9899:1990 ("ISO C90").

ISSUES

The absolute value of the most negative integer remains negative.

LDIV(3)

NAME

ldiv - return quotient and remainder from division

SYNOPSIS

```
#include <stdlib.h>

ldiv_t
ldiv(long num, long denom)
```

DESCRIPTION

The **ldiv()** function computes the value *num*/*denom* and returns the quotient and remainder in a structure named `ldiv_t` that contains two `long` integer members named `quot` and `rem`. Unlike the implementation defined semantics of normal `long` division, **ldiv()** provides precise semantics on the `quot` and `rem` values.

The sign of `quot` is the same as the algebraic quotient.

If the division is inexact, the magnitude of the resulting quotient is the largest integer less than the magnitude of the algebraic quotient.

If the result cannot be represented, the values of `quot` and `rem` are undefined.

If the result can be represented, then `quot * `*denom*` + rem` will equal *num*.

SEE ALSO

div(3), imaxdiv(3), lldiv(3), math(3)

STANDARDS

The **ldiv()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

LLABS(3)

NAME

llabs - returns absolute value

SYNOPSIS

```
#include <stdlib.h>

long long
long labs(long long j)
```

DESCRIPTION

The **llabs()** function returns the absolute value of the `long long` integer *j*.

SEE ALSO

abs(3), cabs(3), imaxabs(3), labs(3), floor(3), math(3)

STANDARDS

The **llabs()** function conforms to ISO/IEC 9899:1999 ("ISO C99").

ISSUES

The absolute value of the most negative integer remains negative.

LLDIV(3)

NAME

lldiv - return quotient and remainder from division

SYNOPSIS

```
#include <stdlib.h>

lldiv_t
ldiv(long long num, long long denom)
```

DESCRIPTION

The **lldiv()** function computes the value *num*/*denom* and returns the quotient and remainder in a structure named `lldiv_t` that contains two `long long` integer members named `quot` and `rem`. Unlike the implementation defined semantics of normal `long long` division, **lldiv()** provides precise semantics on the `quot` and `rem` values.

The sign of `quot` is the same as the algebraic quotient.

If the division is inexact, the magnitude of the resulting quotient is the largest integer less than the magnitude of the algebraic quotient.

If the result cannot be represented, the values of `quot` and `rem` are undefined.

If the result can be represented, then `quot * `*denom*` + rem` will equal *num*.

SEE ALSO

div(3), imaxdiv(3), ldiv(3), math(3)

MALLOC(3)

NAME

malloc - general memory allocation function

SYNOPSIS

```
#include <stdlib.h>

void *
malloc(size_t size)
```

DESCRIPTION

The **malloc()** function allocates uninitialized space for an object whose size is specified by *size*. The **malloc()** function maintains multiple lists of free blocks according to various sizes, allocating space from the appropriate list.

The allocated space is suitably aligned (after possible pointer coercion) for storage of any type of object. If the space is of pagesize or larger, the memory returned will be page-aligned.

RETURN VALUES

The **malloc()** function returns a pointer to the allocated space if successful; otherwise a null pointer is returned.

SEE ALSO

free(3), calloc(3), alloca(3), realloc(3), memory(3)

STANDARDS

The **malloc()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

MEMORY(3)

NAME

malloc, free, realloc, calloc, alloca - general memory allocation operations

SYNOPSIS

```
#include <stdlib.h>

void *
malloc(size_t size)

void
free(void *ptr)

void *
realloc(void *ptr, size_t size)

void *
calloc(size_t nelem, size_t elsize)

void *
alloca(size_t size)

void *
__malloc31(size_t size)

void *
__free31(void *ptr);

void *
__malloc24(size_t size)

void *
__free24(void *ptr);
```

DESCRIPTION

These functions allocate and free memory for the calling process. They are described in the individual manual pages.

SEE ALSO

calloc(3), free(3), malloc(3), realloc(3), alloca(3), __malloc31(3), __free31(3), __malloc24(3), __free24(3),

STANDARDS

These functions, with the exception of **alloca()**, **__malloc31()**, **__free31()**, **__malloc24()** and **__free24()** conform to ISO/IEC 9899:1990 ("ISO C90").

STRFMON(3)

NAME

strfmon - convert monetary value to string

SYNOPSIS

```
#include <monetary.h>

ssize_t
strfmon(char * restrict s, size_t maxsize, const char * restrict format,
        ...);
```

DESCRIPTION

The **strfmon()** function places characters into the array pointed to by *s* as controlled by the string pointed to by *format*. No more than *maxsize* bytes are placed into the array.

The *format* string is composed of zero or more directives: ordinary characters (not %), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the % character. After the %, the following appear in sequence:

- Zero or more of the following flags:

=f	A '=' character followed by another character *f* which is used as the numeric fill character.
^	Do not use grouping characters, regardless of the current locale default.
+	Represent positive values by prefixing them with a positive sign, and negative values by prefixing them with a negative sign. This is the default.
(	Enclose negative values in parentheses.
!	Do not include a currency symbol in the output.
-	Left justify the result. Only valid when a field width is specified.

- An optional minimum field width as a decimal number. By default, there is no minimum width.

- A '#' sign followed by a decimal number specifying the maximum expected number of digits after the radix character.
- A '.' character followed by a decimal number specifying the number the number of digits after the radix character.
- One of the following conversion specifiers:

`i`	The `double` argument is formatted as an international monetary amount. The double value is in the compilations default floating point format.
`n`	The `double` argument is formatted as a national monetary amount. The double value is in the compilation's default floating point format.
`%`	A '%' character is written.

The `double` arguments passed for formating are in the floating point format currently in operation, either BFP or HFP. See the __isBFP(3) function for more information regarding the current floating point mode.

RETURN VALUES

If the total number of resulting bytes including the terminating `NUL` byte is not more than *maxsize*, **strfmon()** returns the number of bytes placed into the array pointed to by *s*, not including the terminating `NUL` byte. Otherwise, -1 is returned, the contents of the array are indeterminate, and `errno` is set to indicate the error.

ERRORS

The **strfmon()** function will fail if:

`[E2BIG]`	Conversion stopped due to lack of space in the buffer.
`[EINVAL]`	The format string is invalid.
`[ENOMEM]`	Not enough memory for temporary buffers.

SEE ALSO

localeconv(3), __isBFP(3)

STANDARDS

The **strfmon()** function conforms to IEEE Std 1003.1-2001 ("POSIX.1").

ISSUES

The **strfmon()** function does not correctly handle multibyte characters in the format argument.

QSORT(3)

NAME

qsort, heapsort, mergesort - sort functions

SYNOPSIS

```
#include <stdlib.h>

void
qsort(void *base, size_t nmemb, size_t size,
int (*compar)(const void *, const void *))

int
heapsort(void *base, size_t nmemb, size_t size,
int (*compar)(const void *, const void *))

int
mergesort(void *base, size_t nmemb, size_t size,
int (*compar)(const void *, const void *))
```

DESCRIPTION

The **qsort()** function is a modified partition-exchange sort, or quicksort. The **heapsort()** function is a modified selection sort. The **mergesort()** function is a modified merge sort with exponential search intended for sorting data with pre-existing order.

The **qsort()** and **heapsort()** functions sort an array of *nmemb* objects, the initial member of which is pointed to by *base*. The size of each object is specified by size. **mergesort()** behaves similarly, but requires that size be greater than `sizeof(void *) / 2`.

The contents of the array base are sorted in ascending order according to a comparison function pointed to by *compar*, which requires two arguments pointing to the objects being compared.

The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

The functions **qsort()** and **heapsort()** are not stable, that is, if two members compare as equal, their order in the sorted array is undefined. The function **mergesort()** is stable.

The **qsort()** function is an implementation of C.A.R. Hoare's "quicksort" algorithm, a variant of partition-exchange sorting; in particular, see D.E. Knuth's Algorithm Q. **qsort()** takes O N lg N average time. This implementation uses median selection to avoid its O N**2 worst-case behavior.

The **heapsort()** function is an implementation of J.W.J. William's "heapsort" algorithm, a variant of selection sorting; in particular, see D.E. Knuth's Algorithm H. **heapsort()** takes O N lg N worst-case time. Its only advantage over **qsort()** is that it uses almost no additional memory; while **qsort()** does not allocate memory, it is implemented using recursion.

The function **mergesort()** requires additional memory of size *nmemb* * *size* bytes; it should be used only when space is not at a premium. **mergesort()** is optimized for data with pre-existing order; its worst case time is O N lg N; its best case is O N.

Normally, **qsort()** is faster than **mergesort()** is faster than **heapsort()**. Memory availability and pre-existing order in the data can make this untrue.

RETURN VALUES

The **qsort()** function returns no value.

Upon successful completion, **heapsort()** and **mergesort()** return 0. Otherwise, they return -1 and the global variable `errno` is set to indicate the error.

ERRORS

The **heapsort()** function succeeds unless:

`[EINVAL]` The size argument is zero, or, the size argument to **mergesort()** is less than "`sizeof(void *) / 2`".

`[ENOMEM]` **heapsort()** or **mergesort()** were unable to allocate memory.

COMPATIBILITY

Previous versions of **qsort()** did not permit the comparison routine itself to call qsort(3). This is no longer true.

SEE ALSO

radixsort(3)

Hoare, C.A.R., "Quicksort", The Computer Journal, 5:1, pp. 10-15, 1962.

Williams, J.W.J, "Heapsort", Communications of the ACM, 7:1, pp. 347-348, 1964.

Knuth, D.E., "Sorting and Searching", The Art of Computer Programming, Vol. 3, pp. 114-123, 145-149, 1968.

Mcilroy, P.M., "Optimistic Sorting and Information Theoretic Complexity", Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, January 1992.

Bentley, J.L., "Engineering a Sort Function", bentley@research.att.com, January 1992.

STANDARDS

The **qsort()** function conforms to ISO/IEC 9899:1990 ("ISO C90").

RADIXSORT(3)

NAME

radixsort - radix sort

SYNOPSIS

```
#include <limits.h>
#include <stdlib.h>

int
radixsort(const unsigned char **base, int nmemb,
const unsigned char *table, unsigned endbyte)

int
sradixsort(const unsigned char **base, int nmemb,
const unsigned char *table, unsigned endbyte)
```

DESCRIPTION

The **radixsort()** and **sradixsort()** functions are implementations of radix sort.

These functions sort an array of pointers to byte strings, the initial member of which is referenced by *base*. The byte strings may contain any values; the end of each string is denoted by the user-specified value *endbyte*.

Applications may specify a sort order by providing the table argument. If non-`NULL`, table must reference an array of `UCHAR_MAX + 1` bytes which contains the sort weight of each possible byte value. The end-of-string byte must have a sort weight of 0 or 255 (for sorting in reverse order). More than one byte may have the same sort weight. The table argument is useful for applications to use when to sorting different characters equally. For example, providing a table with the same weights for A-Z as for a-z will result in a case-insensitive sort. If table is `NULL`, the contents of the array are sorted in ascending order according to the ASCII order of the byte strings they reference and *endbyte* has a sorting weight of 0.

The **sradixsort()** function is stable, that is, if two elements compare as equal, their order in the sorted array is unchanged. The **sradixsort()** function uses additional memory sufficient to hold *nmemb* pointers.

The **radixsort()** function is not stable, but uses no additional memory.

These functions are variants of most-significant-byte radix sorting; in particular, see D.E. Knuth's Algorithm R and section 5.2.5, exercise 10. They take linear time relative to the number of bytes in the strings.

RETURN VALUES

Upon successful completion 0 is returned. Otherwise, -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

`[EINVAL]` The value of the *endbyte* element of table is not 0 or 255.

Additionally, the **sradixsort()** function may fail and set `errno` for any of the errors specified for the library routine malloc(3).

SEE ALSO

qsort(3)

Knuth, D.E., "Sorting and Searching", The Art of Computer Programming, Vol. 3, pp. 170-178, 1968.

Paige, R., "Three Partition Refinement Algorithms", SIAM J. Comput., No. 6, Vol. 16, 1987.

McIlroy, P., "Computing Systems", Engineering Radix Sort, Vol. 6:1, pp. 5-27, 1993.

RAND(3)

NAME

rand, srand - bad random number generator

SYNOPSIS

```
#include <stdlib.h>

void
srand(unsigned seed)

int
rand(void)
```

DESCRIPTION

These interfaces are obsoleted by random(3).

The **rand()** function computes a sequence of pseudo-random integers in the range of `0` to `RAND_MAX` (as defined by the header file `<stdlib.h>`).

The **srand()** function sets its argument as the seed for a new sequence of pseudo-random numbers to be returned by **rand()**. These sequences are repeatable by calling **srand()** with the same seed value.

If no seed value is provided, the functions are automatically seeded with a value of 1.

SEE ALSO

random(3)

STANDARDS

The **rand()** and **srand()** functions conform to ISO/IEC 9899:1990 ("ISO C90").

RANDOM(3)

NAME

random, srandom, initstate, setstate - better random number generator; routines for changing generators

SYNOPSIS

```
#include <stdlib.h>

long
random(void)

void
srandom(unsigned long seed)

char *
initstate(unsigned long seed, char *state, long n)

char *
setstate(char *state)
```

DESCRIPTION

The **random()** function uses a non-linear additive feedback random number generator employing a default table of size 31 long integers to return successive pseudo-random numbers in the range from 0 to (2**31)-1. The period of this random number generator is very large, approximately 16*((2**31)-1).

The **random()** and **srandom()** functions have (almost) the same calling sequence and initialization properties as the rand(3) and srand(3) functions. The difference is that rand(3) produces a much less random sequence – in fact, the low dozen bits generated by rand go through a cyclic pattern. All the bits generated by **random()** are usable. For example, '**random()**&01' will produce a random binary value.

Like rand(3), **random()** will by default produce a sequence of numbers that can be duplicated by calling **srandom()** with '1' as the seed.

The **initstate()** routine allows a state array, passed as an argument, to be initialized for future use. The size of the state array (in bytes) is used by **initstate()** to decide how sophisticated a random number generator it should use – the more state, the better the random numbers will be. Current "optimal" values for the amount of state information are 8, 32, 64, 128, and 256 bytes; other amounts will be rounded down to the nearest known amount. Using less than 8 bytes will cause an error. The

seed for the initialization (which specifies a starting point for the random number sequence, and provides for restarting at the same point) is also an argument. The **initstate()** function returns a pointer to the previous state information array.

Once a state has been initialized, the **setstate()** routine provides for rapid switching between states. The **setstate()** function returns a pointer to the previous state array; its argument state array is used for further random number generation until the next call to **initstate()** or **setstate()**.

Once a state array has been initialized, it may be restarted at a different point either by calling **initstate()**, with the desired seed, the state array, and its size, or by calling both **setstate()** with the state array and **srandom()** with the desired seed. The advantage of calling both **setstate()** and **srandom()** is that the size of the state array does not have to be remembered after it is initialized.

With 256 bytes of state information, the period of the random number generator is greater than 2**69 which should be sufficient for most purposes.

DIAGNOSTICS

If **initstate()** is called with less than 8 bytes of state information, or if **setstate()** detects that the state information has been garbled, error messages are printed on the standard error output.

SEE ALSO

rand(3), srand(3), urandom(4)

ISSUES

About 2/3 the speed of rand(3).

The historical implementation used to have a very weak seeding; the random sequence did not vary much with the seed. The current implementation employs a better pseudo-random number generator for the initial state calculation.

REALLOC(3)

NAME

realloc - reallocation of memory function

SYNOPSIS

```
#include <stdlib.h>

void *
realloc(void *ptr, size_t size)
```

DESCRIPTION

The **realloc()** function changes the size of the object pointed to by *ptr* to the size specified by *size*. The contents of the object are unchanged up to the lesser of the new and old sizes. If the new size is larger, the value of the newly allocated portion of the object is indeterminate. If *ptr* is a null pointer, the **realloc()** function behaves like the malloc(3) function for the specified size. Otherwise, if *ptr* does not match a pointer earlier returned by the calloc(3), malloc(3), or **reallot()** function, or if the space has been deallocated by a call to the free or **realloc()** function, unpredictable and usually detrimental behavior will occur. If the space cannot be allocated, the object pointed to by *ptr* is unchanged. If size is zero and *ptr* is not a null pointer, the object it points to is freed.

The **realloc()** function returns either a null pointer or a pointer to the possibly moved allocated space.

SEE ALSO

alloca(3), calloc(3), free(3), malloc(3),

STANDARDS

The **realloc()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

REALPATH(3)

NAME

realpath - returns the canonicalized absolute `//HFS:`-style pathname

SYNOPSIS

```
#include <sys/param.h>
#include <stdlib.h>

char *
realpath(const char *pathname, char resolved_path[MAXPATHLEN]);
```

DESCRIPTION

The **realpath()** function resolves all symbolic links, extra "/" characters and references to /./ and /../ in the `//HFS:`-style path specified by *pathname*, and copies the resulting absolute pathname into the memory referenced by *resolved_path*. The *resolved_path* argument must refer to a buffer capable of storing at least `MAXPATHLEN` characters.

The **realpath()** function will resolve both absolute and relative paths and return the absolute pathname corresponding to *pathname*. All but the last component of *pathname* must exist when **realpath()** is called.

RETURN VALUES

The **realpath()** function returns *resolved_path* on success. If an error occurs, *realpath()* returns `NULL`, and *resolved_path* contains the pathname which caused the problem.

ERRORS

The function **realpath()** may fail and set the external variable `errno` for any of the errors specified for the library functions chdir(2), close(2), fchdir(2), lstat(2), open(2), readlink(2) and getcwd(3).

SEE ALSO

getcwd(3)

STRTOD(3)

NAME

strtod - convert string to double

SYNOPSIS

```
#include <stdlib.h>

double
strtod(const char *nptr, char **endptr);

float
strtof(const char *nptr, char **endptr);

long double
strtold(const char *nptr, char **endptr);
```

DESCRIPTION

These functions function convert the initial portion of the string pointed to by *nptr* to `double`, `float`, and `long double` representation, respectively.

The expected form of the string is an optional plus ("`+`") or minus sign ("`-`") followed by a sequence of digits optionally containing a decimal-point character, optionally followed by an exponent. An exponent consists of an "`E`" or "`e`", followed by an optional plus or minus sign, followed by a sequence of digits.

Alternatively, if the portion of the string following the optional plus or minus sign begins with "`INFINITY`" or "`NAN`", ignoring case, it is interpreted as an infinity or a quiet NaN, respectively. HFP values cannot represent infinity or NaN; in that case it is interpreted as `HUGE_VAL` and `0.0`. For BFP values the syntax "NAN(s)", where *s* is an alphanumeric string, produces the same value as the call **nan("s")** (respectively, **nanf("s")** and **nanl("s")**.)

Leading white-space characters in the string (as defined by the isspace(3) function) are skipped.

These functions use the **__isBFP()** function to determine if a BFP (IEEE) or HFP value is to be returned.

RETURN VALUES

The **strtod()**, **strtof()**, **strtold()** functions returns the converted value, if any.

If *endptr* is not `NULL`, a pointer to the character after the last character used in the conversion is stored in the location referenced by *endptr*.

If no conversion is performed, zero is returned and the value of *nptr* is stored in the location referenced by *endptr*.

If the correct value would cause overflow, plus or minus `HUGE_VAL`, `HUGE_VALF`, or `HUGE_VALL` is returned (according to the sign of the value), and `ERANGE` is stored in `errno`. If the correct value would cause underflow, zero is returned and `ERANGE` is stored in `errno`.

ERRORS

`[ERANGE]` Overflow or underflow occurred.

SEE ALSO

atof(3), atoi(3), atol(3), nan(3) strtol(3), strtoul(3)

STANDARDS

The **strtod()** function conforms to ISO/IEC 9899:1990 ("ISO C90").

STRTOL(3)

NAME

strtol, strtoll, strtoimax, strtoq - convert string value to a `long`, `long long`, `intmax_t` or `quad_t` integer

SYNOPSIS

```
#include <stdlib.h>
#include <limits.h>

long
strtol(const char *nptr, char **endptr, int base)

long
strtoll(const char *nptr, char **endptr, int base)

#include <inttypes.h>

intmax_t
strtoimax(const char * restrict nptr,
          char ** restrict endptr, int base)

#include <sys/types.h>
#include <stdlib.h>
#include <limits.h>

quad_t
strtoq(const char *nptr, char **endptr, int base)
```

DESCRIPTION

The **strtol()** function converts the string in *nptr* to a `long` value. The **strtoll()** functions converts the string in *nptr* to a `long long` value. The **strtoimax()** function converts the string in *nptr* to a `intmax_t` value. The **strtoq()** function converts the string in *nptr* to a `quad_t` value. The conversion is done according to the given *base*, which must be between 2 and 36 inclusive, or be the special value 0.

The string may begin with an arbitrary amount of white space (as determined by isspace(3)) followed by a single optional '+' or '-' sign. If *base* is zero or 16, the string may then include a '0x' prefix, and the number will be read in base 16. Otherwise, a zero base is taken as 10 (decimal) unless the next character is '0', in which case it is taken as 8 (octal).

The remainder of the string is converted to a long value in the obvious manner, stopping at the first character which is not a valid digit in the given base. (In bases above 10, the letter 'A' in either upper or lower case represents 10, 'B' represents 11, and so forth, with 'Z' representing 35.)

If *endptr* is non nil, **strtol()** stores the address of the first invalid character in `*endptr`. If there were no digits at all, however, **strtol()** stores the original value of *nptr* in `*endptr`. Thus, if `*nptr` is not `'\0'` but `**endptr` is `'\0'` on return, the entire string was valid.

RETURN VALUES

The **strtol()**, **strtoll()** and **strtoimax()** functions returns the result of the conversion, unless the value would underflow or overflow. If an underflow occurs, **strtol()** returns `LONG_MIN`, **strtoll()** returns `LLONG_MIN` and **strtoimax()** returns `INTMAX_MIN`. If an overflow occurs, **strtol()** returns `LONG_MAX`, **strtoll()** returns `LLONG_MAX` and **strtoimax()** returns `INTMAX_MAX`. If an overflow or underflow occurs, `errno` is set to `ERANGE`.

ERRORS

`[ERANGE]` The given string was out of range; the value converted has been clamped.

SEE ALSO

atof(3), atoi(3), atol(3), strtod(3), strtoul(3)

STANDARDS

The **strtol()** function conforms to ISO/IEC 9899:1990 ("ISO C90"). The **strtoll()** and **strtoimax()** functions conform to ISO/IEC 9899:1999 ("ISO C99"). The **strtoq()** function is deprecated.

ISSUES

Ignores the current locale.

STRTOUL(3)

NAME

strtoul, strtoull, strtoumax, strtouq - convert a string to an `unsigned long`, `unsigned long long`, `uintmax_t` or `uquad_t` integer

SYNOPSIS

```
#include <stdlib.h>
#include <limits.h>

unsigned long
strtoul(const char *nptr, char **endptr, int base);

unsigned long long
strtoull(const char *nptr, char **endptr, int base);

#include <inttypes.h>

uintmax_t
strtoumax(const char * restrict nptr,
          char ** restrict endptr, int base);


#include <sys/types.h>
#include <stdlib.h>
#include <limits.h>

u_quad_t
strtouq(const char *nptr, char **endptr, int base);
```

DESCRIPTION

The **strtoul()** function converts the string in *nptr* to an `unsigned long` value. The **strtoull()** function converts the string in *nptr* to an `unsigned long long` value. The **strtouq()** function converts the string in *nptr* to a `u_quad_t` value. The conversion is done according to the given *base*, which must be between 2 and 36 inclusive, or be the special value 0.

The string may begin with an arbitrary amount of white space (as determined by isspace(3)) followed by a single optional '+' or '-' sign. If base is zero or 16, the string may then include a '0x' prefix, and the number will be read in base 16; otherwise, a zero base is taken as 10 (decimal) unless the next character is '0', in which case it is taken as 8 (octal).

The remainder of the string is converted to an `unsigned long` value in the obvious manner, stopping at the end of the string or at the first character that does not produce a valid digit in the given base. (In bases above 10, the letter 'A' in either upper or lower case represents 10, 'B' represents 11, and so forth, with 'Z' representing 35.)

If *endptr* is non nil, **strtoul()** stores the address of the first invalid character in `*endptr`. If there were no digits at all, however, **strtoul()** stores the original value of *nptr* in `*endptr`. (Thus, if `*nptr` is not `'\0'` but `**endptr` is `'\0'` on return, the entire string was valid.)

RETURN VALUES

The **strtoul()**, **strtoull()** and **strtoumax()** functions return either the result of the conversion or, if there was a leading minus sign, the negation of the result of the conversion, unless the original (non-negated) value would overflow; in the latter case, **strtoul()** returns `ULONG_MAX`, **strtoull()** returns `ULLONG_MAX` and **strtoumax()** returns `UINTMAX_MAX`. If the conversion would overflow, the global variable `errno` is set to `ERANGE`.

ERRORS

`[ERANGE]` The given string was out of range; the value converted has been clamped.

SEE ALSO

strtol(3)

STANDARDS

The **strtoul()** function conforms to ISO/IEC 9899:1990 ("ISO C90"). The **strtoull()** and **strtoumax()** functions conform to ISO/IEC 9899:1999 ("ISO C99"). The **strtouq()** function is deprecated.

ISSUES

Ignores the current locale.

SYSCONF(3)

NAME

sysconf – get configurable system variables

SYNOPSIS

```
#include <unistd.h>

long
sysconf(int name);
```

DESCRIPTION

The **sysconf()** function provides a method for applications to determine the current value of a configurable system limit or option variable. The name argument specifies the system variable to be queried. Symbolic constants for each name value are found in the include file `<unistd.h>`.

The available values are as follows:

_SC_ARG_MAX The maximum bytes of argument to execve(2).

_SC_CHILD_MAX The maximum number of simultaneous processes per user id.

_SC_CLK_TCK The frequency of the statistics clock in ticks per second.

_SC_JOB_CONTROL Return 1 if job control is available on this system, otherwise -1.

_SC_NGROUPS_MAX The maximum number of supplemental groups.

_SC_OPEN_MAX The maximum number of open files per user id.

_SC_SAVED_IDS Returns 1 if saved set-group and saved set-user ID is available, otherwise -1.

_SC_MMAP_MEM_MAX_NP Maximum area (in pages) of date space that can allocated to mmap().

_SC_TTY_GROUP Return the group number associated with the TTYGROUP setting.

_SC_THREADS_MAX_NP The maximum number of threads allowed to be created by **pthread_create()**.

_SC_THREADS_TASKS_MAX_NP The maximum number of MVS TASKs that can be used to handle threads created by **pthread_create()**.

_SC_TZNAME_MAX The minimum maximum number of types supported for the name of a timezone.

_SC_VERSION The version of IEEE Std 1003.1 ("POSIX.1") with which the system attempts to comply.

_SC_2_CHAR_TERM Return 1 if the system supports at least one terminal type capable of all operations described in IEEE Std 1003.2 ("POSIX.2"), otherwise -1.

_SC_PAGESIZE The size of the system page in bytes.

_SC_PAGE_SIZE The size of the system page in bytes.

RETURN VALUES

If the call to **sysconf()** is not successful, `-1` is returned and `errno` is set appropriately. Otherwise, if the variable s associated with functionality that is not supported, `-1` is returned and `errno` is not modified. Otherwise, the current variable value is returned.

ERRORS

The following error may be reported:

`[EINVAL]` The value of the name argument is invalid.

STANDARDS

Except for the fact that values returned by sysconf() may change over the lifetime of the calling process, this function conforms to IEEE Std 1003.1-1988 ("POSIX.1").

SYSTEM(3)

NAME

system - pass a command to the POSIX shell

SYNOPSIS

```
#include <stdlib.h>

int
system(const char *string);
```

DESCRIPTION

The **system()** function hands the argument string to the POSIX command interpreter `sh`. The calling process waits for the shell to finish executing the command, ignoring `SIGINT` and `SIGQUIT`, and blocking `SIGCHLD`.

If *string* is a `NULL` pointer, **system()** will return non-zero if the command interpreter (sh) is available, and zero if it is not.

The **system()** function returns the exit status of the shell as returned by waitpid(2), or -1 if an error occurred when invoking fork(2) or waitpid(2). A return value of 127 means the execution of the shell failed.

SEE ALSO

execve(2), fork(2), waitpid(2), popen(3)

STANDARDS

The **system()** function conforms to ISO/IEC 9899:1990 ("ISO C90"), and is expected to be IEEE Std 1003.2 ("POSIX.2") compatible.

Standard Time library

The ANSI standard defines several functions for manipulating time values, which are found in the Standard Time Library.

CTIME(3)

NAME

asctime, asctime_r, ctime, ctime_r, difftime, gmtime, gmtime_r, localtime, localtime_r, mktime, timegm - transform binary date and time values

SYNOPSIS

```
#include <time.h>

extern char *tzname[2];

char *
     ctime(const time_t *clock);

double
difftime(time_t time1, time_t time0);

char *
asctime(const struct tm *tm);

struct tm *
localtime(const time_t *clock);

struct tm *
gmtime(const time_t *clock);

time_t
mktime(struct tm *tm);

time_t
timegm(struct tm *tm);

char *
ctime_r(const time_t *clock, char *buf);

struct tm *
localtime_r(const time_t *clock, struct tm *result);

struct tm *
gmtime_r(const time_t *clock, struct tm *result);

char *
asctime_r(const struct tm *tm, char *buf);
```

DESCRIPTION

The functions **ctime()**, **gmtime()** and **localtime()** all take as an argument a time value representing the time in seconds since the Epoch (00:00:00 UTC, January 1, 1970; see time(3)).

The function **localtime()** converts the time value pointed at by *clock*, and returns a pointer to a `struct tm` (described below) which contains the broken-out time information for the value after adjusting for the current time zone (and any other factors such as Daylight Saving Time). Time zone adjustments are performed as specified by the TZ environment variable (see tzset(3)). The function **localtime()** uses tzset(3) to initialize time conversion information if tzset(3) has not already been called by the process.

After filling in the tm structure, **localtime()** sets the `tm_isdst`'th element of `tzname` to a pointer to a string that's the time zone abbreviation to be used with **localtime()**'s return value.

The function **gmtime()** similarly converts the time value, but without any time zone adjustment, and returns a pointer to a tm structure (described below).

The **ctime()** function adjusts the time value for the current time zone in the same manner as **localtime()**, and returns a pointer to a 26-character string of the form:

```
Thu Nov 24 18:22:48 1986\n\0
```

All the fields have constant width.

The **ctime_r()** function provides the same functionality as **ctime()** except the caller must provide the output buffer *buf* to store the result, which must be at least 26 characters long. The **localtime_r()** and **gmtime_r()** functions provide the same functionality as **localtime()** and **gmtime()** respectively, except the caller must provide the output buffer *result*.

The **asctime()** function converts the broken down time in the structure tm pointed at by `*tm` to the form shown in the example above.

The **asctime_r()** function provides the same functionality as **asctime()** except the caller must provide the output buffer *buf* to store the reuslt, which must be at least 26 characters long.

The functions **mktime()** and **timegm()** convert the broken-down time, in the structure pointer to by *tm* into a time value with the same encoding as that of the values returned by the time(3) function (that is, seconds from the Epoch, UTC). The **mktime()** function interprets the input structure according to the current timezone setting (see tzset(3)). The **timegm()** function interprets the input structure as representing Universal Coordinated Time (UTC).

The original values of the tm_wday and tm_yday components of the structure are Ignored, and the original values of the other components are not restricted to their normal ranges, and will be normalized if needed. For example, October 40 is changed into November 9, a `tm_hour` of -1 means 1 hour before midnight, `tm_mday` of 0 means the day preceding the current month, and `tm_mon` of -2 means 2 months before January of `tm_year`. (A positive or zero value for `tm_isdst` causes **mktime()** to presume initially that summer time (for example, Daylight Saving Time) is or is not in effect for the specified time, respectively. A negative value for `tm_isdst` causes the **mktime()** function to attempt to divine whether summer time is in effect for the specified time. The `tm_isdst` and `tm_gmtoff` members are forced to zero by **timegm()**.)

On successful completion, the values of the `tm_wday` and `tm_yday` components of the structure are set appropriately, and the other components are set to represent the specified calendar time, but with their values forced to their normal ranges; the final value of `tm_mday` is not set until `tm_mon` and `tm_year` are determined. The **mktime()** function returns the specified calendar time; if the calendar time cannot be represented, it returns -1;

The **difftime()** function returns the difference between two calendar times, (*time1* - *time0*), expressed in seconds.

External declarations as well as the tm structure definition are in the `<time.h>` include file. The tm structure includes at least the following fields:

```
int tm_sec;      /* seconds (0 - 60) */
int tm_min;      /* minutes (0 - 59) */
int tm_hour;     /* hours (0 - 23) */
int tm_mday;     /* day of month (1 - 31) */
int tm_mon;      /* month of year (0 - 11) */
int tm_year;     /* year - 1900 */
int tm_wday;     /* day of week (Sunday = 0) */
int tm_yday;     /* day of year (0 - 365) */
int tm_isdst;    /* is summer time in effect? */
char *tm_zone;   /* abbreviation of timezone name */
long tm_gmtoff;  /* offset from UTC in seconds */
```

The field `tm_isdst` is non-zero if summer time is in effect.

The field `tm_gmtoff` is the offset (in seconds) of the time represented from UTC, with positive values indicating east of the Prime Meridian.

SEE ALSO

gettimeofday(2), getenv(3), time(3), tzset(3)

STANDARDS

The **asctime()**, **ctime()**, **difftime()**, **gmtime()**, **localtime()**, and **mktime()** functions conform to ISO/IEC 9899:1990 ("ISO C90"), and conform to ISO/IEC 9945-1:1996 ("POSIX.1") provided the selected local timezone does not contain a leap-second table.

The **asctime_r()**, **ctime_r()**, **gmtime_r()**, and **localtime_r()** functions are expected to conform to ISO/IEC 9945-1:1996 ("POSIX.1") (again provided the selected local timezone does not contain a leap-second table).

The **timegm()** function is not specified by any standard; its function cannot be completely emulated using the standard functions described above.

ISSUES

Except for **difftime()** and **mktime()** and the _r() variants of the other functions, these functions leave their result in an internal static object and return a pointer to that object. Subsequent calls to these function will modify the same object.

The C standard provides no mechanism for a program to modify its current local timezone setting, and the POSIX-standard method is not reentrant. (However, thread-safe implementations are provided in the POSIX threaded environment.)

The `tm_zone` field of a returned tm structure points to a static array of characters, which will also be overwritten by any subsequent calls (as well as by subsequent calls to tzset(3) and tzsetwall(3)).

Use of the external variable `tzname` is discouraged; the `tm_zone` entry in the tm structure is preferred.

STRFTIME(3)

NAME

strftime - format date and time

SYNOPSIS

```
#include <time.h>

size_t
strftime(char *buf, size_t maxsize, const char *format,
const struct tm *timeptr)
```

DESCRIPTION

The **strftime()** function formats the information from *timeptr* into the buffer *buf* according to the string pointed to by format.

The format string consists of zero or more conversion specifications and ordinary characters. All ordinary characters are copied directly into the buffer. A conversion specification consists of a percent sign '`%`' and one other character.

No more than *maxsize* characters will be placed into the array. If the total number of resulting characters, including the terminating null character, is not more than *maxsize*, **strftime()** returns the number of characters in the array, not counting the terminating null. Otherwise, zero is returned and the buffer content is indeterminate.

Each conversion specification is replaced by the characters as follows which are then copied into the buffer.

`%A` is replaced by national representation of the full weekday name.

`%a` is replaced by national representation of the abbreviated weekday name, where the abbreviation is the first three characters.

`%B` is replaced by national representation of the full month name.

`%b` is replaced by national representation of the abbreviated month name, where the abbreviation is the first three characters.

`%C` is replaced by (year / 100) as decimal number; single digits are preceded by a zero.

`%c` is replaced by national representation of time and date (the format is similar with produced by asctime(3)).

%D is equivalent to "%m/%d/%y".

%d is replaced by the day of the month as a decimal number (01-31).

%E* POSIX locale extensions. The sequences %Ec %EC %Ex %Ey %EY %Od %Oe %OH %OI %Om %OM %OS %Ou %OU %OV %Ow %OW %Oy are supposed to provide alternate representations.

%e is replaced by the day of month as a decimal number (1-31); single digits are preceded by a blank.

%G is replaced by a year as a decimal number with century. This year is the one that contains the greater part of the week (Monday as the first day of the week).

%g is replaced by the same year as in "%G", but as a decimal number without century (00-99).

%H is replaced by the hour (24-hour clock) as a decimal number (00-23).

%h the same as %b.

%I is replaced by the hour (12-hour clock) as a decimal number (01-12).

%j is replaced by the day of the year as a decimal number (001-366).

%k is replaced by the hour (24-hour clock) as a decimal number (0-23); single digits are preceded by a blank.

%l is replaced by the hour (12-hour clock) as a decimal number (1-12); single digits are preceded by a blank.

%M is replaced by the minute as a decimal number (00-59).

%m is replaced by the month as a decimal number (01-12).

%n is replaced by a newline.

%O* the same as %E*.

%p is replaced by national representation of either "ante meridiem" or "post meridiem" as appropriate.

%R is equivalent to "%H:%M".

%r is equivalent to "%I:%M:%S %p".

%S is replaced by the second as a decimal number (00-60).

%s is replaced by the number of seconds since the Epoch, UTC (see mktime(3)).

%T is equivalent to "%H:%M:%S".

%t is replaced by a tab.

%U is replaced by the week number of the year (Sunday as the first day of the week) as a decimal number (00-53).

%u is replaced by the weekday (Monday as the first day of the week) as a decimal number (1-7).

%V is replaced by the week number of the year (Monday as the first day of the week) as a decimal number (01-53). If the week containing January 1 has four or more days in the new year, then it is week 1; otherwise it is the last week of the previous year, and the next week is week 1.

%v is equivalent to "%e-%b-%Y".

%W is replaced by the week number of the year (Monday as the first day of the week) as a decimal number (00-53).

%w is replaced by the weekday (Sunday as the first day of the week) as a decimal number (0-6).

%X is replaced by national representation of the time.

%x is replaced by national representation of the date.

%Y is replaced by the year with century as a decimal number.

%y is replaced by the year without century as a decimal number (00-99).

%Z is replaced by the time zone name.

%+ is replaced by national representation of the date and time (the format is similar with produced by date(1)).

%% is replaced by '%'.

SEE ALSO

ctime(3), printf(3), strptime(3)

STANDARDS

The **strftime()** function conforms to ISO/IEC 9899:1990 ("ISO C90") with a lot of extensions including '%C', '%D', '%E*', '%e', '%G', '%g', '%h', '%k', '%l', '%n', '%O*', '%R', '%r', '%s', '%T', '%t', '%u', '%V', '%+'.

The peculiar week number and year in the replacements of '%G', '%g' and '%V' are defined in ISO 8601: 1988.

STRPTIME(3)

NAME

strptime - parse date and time string

SYNOPSIS

```
#include <time.h>

const char *
strptime(const char *buf, const char *format,
struct tm *timeptr)
```

DESCRIPTION

The **strptime()** function parses the string in the buffer *buf* according to the string pointed to by *format*, and fills in the elements of the structure pointed to by *timeptr*. Thus, it can be considered the reverse operation of strftime(3).

The format string consists of zero or more conversion specifications and ordinary characters. All ordinary characters are matched exactly with the buffer, where white space in the format string will match any amount of white space in the buffer. All conversion specifications are identical to those described in strftime(3).

RETURN VALUES

Upon successful completion, **strptime()** returns the pointer to the first character in *buf* that has not been required to satisfy the specified conversions in format. It returns `NULL` if one of the conversions failed.

SEE ALSO

scanf(3), strftime(3)

TIME2POSIX(3)

NAME

time2posix, posix2time - convert seconds since the Epoch

SYNOPSIS

```
#include <time.h>

time_t
time2posix(const time_t *t)

time_t
posix2time(const time_t *t)
```

DESCRIPTION

IEEE Std1003.1-1988 ("POSIX") legislates that a time_t value of 536457599 shall correspond to "Wed Dec 31 23:59:59 GMT 1986." This effectively implies that POSIX `time_t`'s cannot include leap seconds and, therefore, that the system time must be adjusted as each leap occurs.

If the time package is configured with leap-second support enabled, however, no such adjustment is needed and `time_t` values continue to increase over leap events (as a true 'seconds since...' value). This means that these values will differ from those required by POSIX by the net number of leap seconds inserted since the Epoch.

Typically this is not a problem as the type `time_t` is intended to be (mostly) opaque - `time_t` values should only be obtained-from and passed-to functions such as time(2), localtime(3), mktime(3) and difftime(3). However, IEEE Std1003.1-1988 ("POSIX") gives an arithmetic expression for directly computing a `time_t` value from a given date/time, and the same relationship is assumed by some (usually older) applications. Any programs creating/dissecting `time_t`'s using such a relationship will typically not handle intervals over leap seconds correctly.

The **time2posix()** and **posix2time()** functions are provided to address this `time_t` mismatch by converting between local `time_t` values and their POSIX equivalents. This is done by accounting for the number of time-base changes that would have taken place on a POSIX system as leap seconds were inserted or deleted. These converted values can then be used in lieu of correcting the older applications, or when communicating with POSIX-compliant systems.

The **time2posix()** function is single-valued. That is, every local `time_t` corresponds to a single POSIX `time_t`. The **posix2time()** function is less well-behaved: for a

positive leap second hit the result is not unique, and for a negative leap second hit the corresponding POSIX `time_t` doesn't exist, so an adjacent value is returned. Both of these are good indicators of the inferiority of the POSIX representation.

The following table summarizes the relationship between `time_t` and its conversion to, and back from, the POSIX representation over the leap second inserted at the end of June, 1993.

DATE TIME T X=time2posix(T) posix2time(X)

93/06/30 23:59:59 A+0 B+0 A+0

93/06/30 23:59:60 A+1 B+1 A+1 or A+2

93/07/01 00:00:00 A+2 B+1 A+1 or A+2

93/07/01 00:00:01 A+3 B+2 A+3

A leap second deletion would look like...

DATE TIME T X=time2posix(T) posix2time(X)

??/06/30 23:59:58 A+0 B+0 A+0

??/07/01 00:00:00 A+1 B+2 A+1

??/07/01 00:00:01 A+2 B+3 A+2

[Note: posix2time(B+1) → A+0 or A+1]

If leap-second support is not enabled, local `time_t`'s and POSIX `time_t`'s are equivalent, and both **time2posix()** and **posix2time()** degenerate to the identity function.

SEE ALSO

difftime(3), localtime(3), mktime(3), time(3)

TZSET(3)

NAME

tzset - initialize time conversion information

SYNOPSIS

```
#include <time.h>

void
tzset(void);
```

DESCRIPTION

The **tzset()** function initializes time conversion information used by the library routine localtime(3). The environment variable `TZ` specifies how this is done.

If `TZ` does not appear in the environment, or `TZ` appears in the environment but its value is a null string, Coordinated Universal Time (UTC) is used (without leap second correction).

If `TZ` appears in the environment and its value begins with a colon (‘`:`’), the rest of its value is used as a pathname of a tzfile(5)-format file from which to read the time conversion information. If the first character of the pathname is a slash (‘`/`’) it is used as an absolute pathname; otherwise, it is used as a pathname relative to the system time conversion information directory.

If its value does not begin with a colon, it is first used as the pathname of a file (as described above) from which to read the time conversion information. If that file cannot be read, the value is then interpreted as a direct specification (the format is described below) of the time conversion information.

If the `TZ` environment variable does not specify a tzfile(5)-format file and cannot be interpreted as a direct specification, UTC is used.

SPECIFICATION FORMAT

When `TZ` is used directly as a specification of the time conversion information, it must have the following syntax (spaces inserted for clarity):

std offset [*dst* [*offset*] [**,** *rule*]]

Where:

std and *dst* Three or more bytes that are the designation for the standard (*std*) or summer (*dst*) time zone. Only *std* is required; if *dst* is missing, then summer time does not apply in this locale. Upper and lowercase letters are explicitly allowed. Any characters except a leading colon (‘`:`’), digits, comma (‘`,`’), minus (‘`-`’), plus (‘`+`’), and `NUL` are allowed.

offset Indicates the value one must add to the local time to arrive at Coordinated Universal Time. The offset has the form:

hh[`:`*mm*[`:`*ss*]]

The minutes (*mm*) and seconds (*ss*) are optional. The hour (*hh*) is required and may be a single digit. The offset following `std` is required. If no offset follows *dst*, summer time is assumed to be one hour ahead of standard time. One or more digits may be used; the value is always interpreted as a decimal number. The hour must be between zero and 24, and the minutes (and seconds) – if present – between zero and 59. If preceded by a (’`-`’) the time zone shall be east of the Prime Meridian; otherwise it shall be west (which may be indicated by an optional preceding (’`+`’)).

rule Indicates when to change to and back from summer time. The rule has the form:

date`/`*time*`,`*date*`/`*time*

where the first date describes when the change from standard to summer time occurs and the second date describes when the change back happens. Each time field describes when, in current local time, the change to the other time is made.

The format of *date* is one of the following:

`J` *n* The Julian day *n* ($1 \leq n \leq 365$). Leap days are not counted; that is, in all years – including leap years – February 28 is day 59 and March 1 is day 60. It is impossible to explicitly refer to the occasional February 29.

n The zero-based Julian day ($0 \leq n \leq 365$). Leap days are counted, and it is possible to refer to February 29.

`M` *m*`.`*n*`.`*d* The d^{th} day ($0 \leq d \leq 6$) of week *n* of month *m* of the year ($1 \leq n \leq 5$), ($1 \leq m \leq 12$), where week 5 means “the last *d* day in month *m*” which may occur in either the fourth or the fifth week). Week 1 is the first week in which the *d*’th day occurs. Day zero is Sunday.

The *time* has the same format as *offset* except that no leading sign (’`-`’) or (’`+`’) is allowed. The default, if *time* is not given, is `02:00:00`.

For compatibility with System V Release 3.1, a semicolon (‘;’) may be used to separate the rule from the rest of the specification.

SEE ALSO

gettimeofday(2), ctime(3), getenv(3), time(3), tzfile(5)

TZFILE(5)

NAME

tzfile - timezone information

DESCRIPTION

The time zone information files used by tzset(3) begin with the magic characters "`TZif`" to identify them as time zone information files, followed by sixteen bytes reserved for future use, followed by four four byte values written in a "standard" byte order (the high-order byte of the value is written first). These values are, in order:

`tzh_ttisgmtcnt`	The number of UTC/local indicators stored in the file.
`tzh_ttisstdcnt`	The number of standard/wall indicators stored in the file.
`tzh_leapcnt`	The number of leap seconds for which data is stored in the file.
`tzh_timecnt`	The number of "transition times" for which data is stored in the file.
`tzh_typecnt`	The number of "local time types" for which data is stored in the file (must not be zero).
`tzh_charcnt`	The number of characters of "time zone abbreviation strings" stored in the file.

The above header is followed by `tzh_timecnt` four-byte values of type `long`, sorted in ascending order. These values are written in "standard" byte order. Each is used as a transition time (as returned by time(3)) at which the rules for computing local time change. Next come `tzh_timecnt` one-byte values of type `unsigned char`; each one tells which of the different types of "local time" types described in the file is associated with the same-indexed transition time. These values serve as indices into an array of `ttinfo` structures that appears next in the file; these structures are defined as follows:

```
struct ttinfo {
   long          tt_gmtoff;
   int           tt_isdst;
   unsigned int  tt_abbrind;
};
```

Each structure is written as a four-byte value for `tt_gmtoff` of type `long`, in a standard byte order, followed by a one-byte value for `tt_isdst` and a one-byte value for `tt_abbrind`. In each structure, `tt_gmtoff` gives the number of seconds to be added to UTC, `tt_isdst` tells whether `tm_isdst` should be set by localtime(3) and `tt_abbrind` serves as an index into the array of time zone abbreviation characters that follow the `ttinfo` structure(s) in the file.

Then there are `tzh_leapcnt` pairs of four-byte values, written in standard byte order; the first value of each pair gives the time (as returned by time(3)) at which a leap second occurs; the second gives the total number of leap seconds to be applied after the given time. The pairs of values are sorted in ascending order by time.

Then there are `tzh_ttisstdcnt` standard/wall indicators, each stored as a one-byte value; they tell whether the transition times associated with local time types were specified as standard time or wall clock time, and are used when a time zone file is used in handling POSIX-style time zone environment variables.

Finally there are `tzh_ttisgmtcnt` UTC/local indicators, each stored as a one-byte value; they tell whether the transition times associated with local time types were specified as UTC or local time, and are used when a time zone file is used in handling POSIX-style time zone environment variables.

localtime(3) uses the first standard-time `ttinfo` structure in the file (or simply the first `ttinfo` structure in the absence of a standard-time structure) if either `tzh_timecnt` is zero or the time argument is less than the first transition time recorded in the file.

SEE ALSO

ctime(3), time2posix(3)

String Library

The ANSI C standard defines functions for concatenating, searching, copying and general manipulation of strings. These are found in the String Library.

The Systems/C library also includes the string manipulation functions typically found on BSD UNIX variants.

BCMP(3)

NAME

bcmp - compare byte string

SYNOPSIS

```
#include <string.h>

int
bcmp(const void *b1, const void *b2, size_t len)
```

DESCRIPTION

The **bcmp()** function compares byte string *b1* against byte string *b2*, returning zero if they are identical, non-zero otherwise. Both strings are assumed to be len bytes long. Zero-length strings are always identical.

The strings may overlap.

SEE ALSO

bcmp(3), memcmp(3), strcasecmp(3), strcmp(3), strcoll(3), strxfrm(3)

BCOPY(3)

NAME

bcopy - copy byte string

SYNOPSIS

```
#include <string.h>

void
bcopy(const void *src, void *dst, size_t len)
```

DESCRIPTION

The **bcopy()** function copies len bytes from string *src* to string *dst*. The two strings may overlap. If *len* is zero, no bytes are copied.

SEE ALSO

memccpy(3), memcpy(3), memmove(3), strcpy(3), strncpy(3)

BSTRING(3)

NAME

bcmp, bcopy, bzero, memccpy, memchr, memcmp, memcpy, memmove, memset - byte string operations

SYNOPSIS

```
#include <string.h>

int
bcmp(const void *b1, const void *b2, size_t len)

void
bcopy(const void *src, void *dst, size_t len)

void
bzero(void *b, size_t len)

void *
memchr(const void *b, int c, size_t len)

int
memcmp(const void *b1, const void *b2, size_t len)

void *
memccpy(void *dst, const void *src, int c, size_t len)

void *
memcpy(void *dst, const void *src, size_t len)

void *
memmove(void *dst, const void *src, size_t len)

void *
memset(void *b, int c, size_t len)
```

DESCRIPTION

These functions operate on variable length strings of bytes. They do not check for terminating null bytes as the routines listed in string(3) do.

See the specific manual pages for more information.

SEE ALSO

bcmp(3), bcopy(3), bzero(3), memccpy(3), memchr(3), memcmp(3), memcpy(3), memmove(3), memset(3)

STANDARDS

The functions **memchr()**, **memcmp()**, **memcpy()**, **memmove()**, and **memset()** conform to ISO/IEC 9899:1990 ("ISO C90").

BZERO(3)

NAME

bzero - write zeroes to a byte string

SYNOPSIS

```
#include <string.h>

void
bzero(void *b, size_t len)
```

DESCRIPTION

The **bzero()** function writes *len* zero bytes to the string b. If *len* is zero, **bzero()** does nothing.

SEE ALSO

memset(3), swab(3)

FFS(3)

NAME

ffs,ffsl - find first bit set in a bit string

SYNOPSIS

```
#include <string.h>

int
ffs(int value)

int
ffsl(long value)

int
ffsll(long long value)

int
fls(int value)

int
flsl(long value)

int
flsll(long long value)
```

DESCRIPTION

The **ffs()**, **ffsl()** and **ffsll()** functions find the first (least significant bit) bit set in *value* and return the index of that bit.

The **fls()**, **flsl()** and **flsll()** functions find the last (most significant bit) bit set in *value* and return the index of that bit.

Bits are numbered starting from 1, starting at the rightmost bit. A return value of 0 means that the argument was zero.

SEE ALSO

bitstring(3)

INDEX(3)

NAME

index - locate character in string

SYNOPSIS

```
#include <string.h>

char *
index(const char *s, int c)
```

DESCRIPTION

The **index()** function locates the first character matching *c* (converted to a char) in the null-terminated string *s*.

RETURN VALUES

A pointer to the character is returned if it is found; otherwise `NULL` is returned. If *c* is `'\0'`, **index()** locates the terminating `'\0'`.

SEE ALSO

memchr(3), rindex(3), strchr(3), strcspn(3), strpbrk(3), strrchr(3), strsep(3), strspn(3), strstr(3), strtok(3)

MEMCCPY(3)

NAME

memccpy - copy string until character found

SYNOPSIS

```
#include <string.h>

void *
memccpy(void *dst, const void *src, int c, size_t len)
```

DESCRIPTION

The **memccpy()** function copies bytes from string *src* to string *dst*. If the character *c* (as converted to an unsigned char) occurs in the string *src*, the copy stops and a pointer to the byte after the copy of *c* in the string *dst* is returned. Otherwise, *len* bytes are copied, and a `NULL` pointer is returned.

SEE ALSO

bcopy(3), memcpy(3), memmove(3), strcpy(3)

MEMCHR(3)

NAME

memchr - locate byte in byte string

SYNOPSIS

```
#include <string.h>

void *
memchr(const void *b, int c, size_t len)
```

DESCRIPTION

The **memchr()** function locates the first occurrence of *c* (converted to an unsigned char) in string *b*.

RETURN VALUES

The **memchr()** function returns a pointer to the byte located, or `NULL` if no such byte exists within *len* bytes.

SEE ALSO

index(3), rindex(3), strchr(3), strcspn(3), strpbrk(3), strrchr(3), strsep(3), strspn(3), strstr(3), strtok(3)

STANDARDS

The **memchr()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

MEMCMP(3)

NAME

memcmp - compare byte string

SYNOPSIS

```
#include <string.h>

int
memcmp(const void *b1, const void *b2, size_t len)
```

DESCRIPTION

The **memcmp()** function compares byte string *b1* against byte string *b2*. Both strings are assumed to be *len* bytes long.

RETURN VALUES

The **memcmp()** function returns zero if the two strings are identical, otherwise returns the difference between the first two differing bytes treated as `unsigned char` values, so that `'\200'` is greater than `'\0'`, for example. Zero-length strings are always identical.

SEE ALSO

bcmp(3), strcasecmp(3), strcmp(3), strcoll(3), strxfrm(3)

STANDARDS

The **memcmp()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

MEMCPY(3)

NAME

memcpy - copy byte string

SYNOPSIS

```
#include <string.h>

void *
memcpy(void *dst, const void *src, size_t len)

void *
mempcpy(void *dst, const void *src, size_t len);
```

DESCRIPTION

The **memcpy()** and **mempcpy()** functions copy *len* bytes from string *src* to string *dst*. If *src* and *dst* overlap the result is undefined.

RETURN VALUES

The **memcpy()** function returns the original value of dst.

The **mempcpy()** function returns a pointer to the byte after the last written byte.

SEE ALSO

bcopy(3), memccpy(3), memmove(3), strcpy(3)

STANDARDS

The **memcpy()** function conforms to ISO/IEC 9899:1990 ("ISO C90").

ISSUES

In this implementation the **memcpy()** function is implemented using bcopy(3), and therefore the strings may overlap. On other systems, or when using the built-in **memcpy()** function, copying overlapping strings may produce unpredictable resutls.

The built-in **memcpy()** is expanded, the implementation uses the MVC instruction which does produce the same results as the library function when the strings overlap.

MEMMEM(3)

NAME

memmem - locate a byte substring in a byte string

SYNOPSIS

```
#include <string.h>

void *
memmem(const char *big, size_t big_len, const char *little,
         size_t little_len);
```

DESCRIPTION

The **memmem()** function locates the first occurrence of the byte string *little* in the byte string *big*.

RETURN VALUES

If *big_len* is smaller than *little_len*, if *little_len* is 0, if *big_len* is 0 or if *little* occurs nowhere in *big*, `NULL` is returned; otherwise a pointer to the first character of the first occurrence of *little* is returned.

SEE ALSO

memchar(3), strchr(3), strstr(3)

MEMMOVE(3)

NAME

memmove - copy byte string

SYNOPSIS

```
#include <string.h>

void *
memmove(void *dst, const void *src, size_t len)
```

DESCRIPTION

The **memmove()** function copies *len* bytes from string *src* to string *dst*. The two strings may overlap; the copy is always done in a non-destructive manner.

RETURN VALUES

The **memmove()** function returns the original value of *dst*.

SEE ALSO

bcopy(3), memccpy(3), memcpy(3), strcpy(3)

STANDARDS

The **memmove()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

MEMSET(3)

NAME

memset - write a byte to byte string

SYNOPSIS

```
#include <string.h>

void *
memset(void *b, int c, size_t len)
```

DESCRIPTION

The **memset()** function writes *len* bytes of value *c* (converted to an unsigned char) to the string *b*.

RETURNS

The **memset()** function returns its first argument.

SEE ALSO

bzero(3), swab(3)

STANDARDS

The **memset()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

RINDEX(3)

NAME

rindex - locate character in string

SYNOPSIS

```
#include <string.h>

char *
rindex(const char *s, int c)
```

DESCRIPTION

The **rindex()** function locates the last character matching *c* (converted to a `char`) in the null-terminated string *s*.

RETURN VALUES

A pointer to the character is returned if it is found; otherwise `NULL` is returned. If *c* is `'\0'`, **rindex()** locates the terminating `'\0'`.

SEE ALSO

index(3), memchr(3), strchr(3), strcspn(3), strpbrk(3), strrchr(3), strep(3), strspn(3), strstr(3), strtok(3)

STRCASECMP(3)

NAME

strcasecmp, strncasecmp - compare strings, ignoring case

SYNOPSIS

```
#include <strings.h>

int
strcasecmp(const char *s1, const char *s2)

int
strncasecmp(const char *s1, const char *s2, size_t len)
```

DESCRIPTION

The **strcasecmp()** and **strncasecmp()** functions compare the null-terminated strings *s1* and *s2* and return an integer greater than, equal to, or less than 0, according as *s1* is lexicographically greater than, equal to, or less than *s2* after translation of each corresponding character to lower-case. The strings themselves are not modified. The comparison is done using unsigned characters, so that `'\200'` is greater than `'\0'`.

The **strncasecmp()** compares at most *len* characters.

SEE ALSO

bcmp(3), memcmp(3), strcmp(3), strcoll(3), strxfrm(3)

NOTES

The **strcasecmp()** and **strncasecmp()** function prototypes existed previously in `<string.h>` before they were moved to `<strings.h>` for IEEE Std 1003.1-2001 ("POSIX.1") compliance.

STRCAT(3)

NAME

strcat - concatenate strings

SYNOPSIS

```
#include <string.h>

char *
strcat(char *s, const char *append)

char *
strncat(char *s, const char *append, size_t count)
```

DESCRIPTION

The **strcat()** and **strncat()** functions append a copy of the null-terminated string *append* to the end of the null-terminated string *s*, then add a terminating `'\0'`. The string *s* must have sufficient space to hold the result.

The **strncat()** function appends not more than *count* characters from *append*, and then adds a terminating `'\0'`.

RETURN VALUES

The **strcat()** and **strncat()** functions return the pointer *s*.

SEE ALSO

bcopy(3), memccpy(3), memcpy(3), memmove(3), strcpy(3)

STANDARDS

The **strcat()** and **strncat()** functions conform to ISO/IEC 9899:1990 (“ISO C90”).

STRCHR(3)

NAME

strchr - locate character in string

SYNOPSIS

```
#include <string.h>

char *
strchr(const char *s, int c)

char *
strchrnul(const char *s, int c)
```

DESCRIPTION

The **strchr()** function locates the first occurrence of *c* in the string pointed to by *s*. The terminating NUL character is considered part of the string. If *c* is `'\0'`, **strchr()** locates the terminating `'\0'`.

The **strchrnul()** function is identical to **strchr()** except that if *c* is not found in *s* a pointer to the terminating `'\0'` is returned.

RETURN VALUES

The function **strchr()** returns a pointer to the located character, or `NULL` if the character does not appear in the string.

strchrnul() returns a pointer to the terminating `'\0'` if the character does not appear in the string.

SEE ALSO

index(3), memchr(3), index(3), strcspn(3), strpbrk(3), strrchr(3), strep(3), strspn(3), strstr(3), strtok(3)

STANDARDS

The **strchr()** function conforms to ISO/IEC 9899:1990 ("ISO C90").

The **strchrnul()** function is an extension.

STRCMP(3)

NAME

strcmp, strncmp, - compare strings

SYNOPSIS

```
#include <string.h>

int
strcmp(const char *s1, const char *s2)

int
strncmp(const char *s1, const char *s2, size_t len)
```

DESCRIPTION

The **strcmp()** and **strncmp()** functions lexicographically compare the null- terminated strings *s1* and *s2*.

The **strncmp()** function compares not more then *len* characters.

RETURN VALUES

The **strcmp()** and **strncmp()** return an integer greater than, equal to, or less than 0, accordingly as the string *s1* is greater than, equal to, or less than the string *s2*. The comparison is done using unsigned characters, so that `'\200'` is greater than `'\0'`.

SEE ALSO

bcmp(3), memcmp(3), strcasecmp(3), strcoll(3), strxfrm(3)

STANDARDS

The **strcmp()** and **strncmp()** functions conform to ISO/IEC 9899:1990 ("ISO C90").

STRCOLL(3)

NAME

strcoll - compare strings according to current collation

SYNOPSIS

```
#include <string.h>

int
strcoll(const char *s1, const char *s2)
```

DESCRIPTION

The **strcoll()** function lexicographically compares the null-terminated strings *s1* and *s2* according to the current locale collation if any. If no locale is current, **stroll()** calls **strcmp()**.

The **stroll()** function returns an integer greater than, equal to, or less than 0, accordingly as *s1* is greater than, equal to, or less than *s2*.

SEE ALSO

setlocale(3), strcmp(3), strxfrm(3)

STANDARDS

The **strcoll()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

STRCPY(3)

NAME

strcpy - copy strings

SYNOPSIS

```
#include <string.h>

char *
stpcpy(char * restrict dst, const char * restrict src);

char *
stpncpy(char * restrict dst, const char * restrict src, size_t len);

char *
strcpy(char * restrict dst, const char * restrict src)

char *
strncpy(char * restrict dst, const char * restrict src, size_t len)
```

DESCRIPTION

The **stpcpy()** and strcpy() functions copy the string *src* to *dst* (including the terminating `'\0'` character.)

The **strncpy()** and **stpncpy()** functions copy at most *len* characters from *src* into *dst*. If *src* is less than *len* characters long, the remainder of dst is filled with `'\0'` characters. Otherwise, *dst* is not terminated.

RETURN VALUES

The **strcpy()** and **strncpy()** functions return *dst*. The **stpcpy()** and **stpncpy()** functions return a pointer to the terminating `'\0'` character of *dst*. If **stpncpy()** does not terminate *dst* with a `NUL` character, it instead returns a pointer to *dst*`[n]` (which does not necessarily refer to a valid memory location.)

EXAMPLES

The following sets `chararray` to `"abc\0\0\0"`:

```
(void)strncpy(chararray, "abc", 6);
```

The following sets `chararray` to `"abcdef"`:

```
char chararray[6]

(void)strncpy(chararray, "abcdefgh", sizeof(chararray));
```

Note that it does not `NUL` terminate chararray because the length of the source string is greater than or equal to the length argument.

The following copies as many characters from `input` to `buf` as will fit and `NUL` terminates the result. Because **strncpy()** does not guarantee to `NUL` terminate the string itself, this must be done explicitly.

```
char buf[1024];

(void)strncpy(buf, input, sizeof(buf) - 1);
buf[sizeof(buf) - 1] = '\0';
```

This could be better achieved using strlcpy(3), as shown in the following example:

```
(void)strlcpy(buf, input, sizeof(buf));
```

Note that because strlcpy(3) is not defined in any standards, it should only be used when portability is not a concern.

SEE ALSO

bcopy(3), memccpy(3), memcpy(3), memmove(3), strlcpy(3)

STANDARDS

The **strcpy()** and **strncpy()** functions conform to ISO/IEC 9899:1990 ("ISO C90"). The **stpcpy()** and **stpncpy()** functions conform to IEEE Std 1003.1-2008 ("POSIX.1").

STRCSPN(3)

NAME

strcspn - span the complement of a string

SYNOPSIS

```
#include <string.h>

size_t
strcspn(const char *s, const char *charset)
```

DESCRIPTION

The **strcspn()** function spans the initial part of the null-terminated string *s* as long as the characters from *s* do not occur in string *charset* (it spans the complement of *charset*).

RETURN VALUES

The **strcspn()** function returns the number of characters spanned.

SEE ALSO

index(3), memchr(3), rindex(3), strchr(3), strpbrk(3), strrchr(3), strep(3), strspn(3), strstr(3), strtok(3)

STANDARDS

The **strcspn()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

STRDUP(3)

NAME

strdup - save a copy of a string

SYNOPSIS

```
#include <string.h>

char *
strdup(const char *str)
```

DESCRIPTION

The **strdup()** function allocates sufficient memory for a copy of the string *str*, does the copy, and returns a pointer to it. The pointer may subsequently be used as an argument to the function free(3).

If insufficient memory is available, `NULL` is returned.

SEE ALSO

free(3), malloc(3)

STRERROR(3)

NAME

perror, strerror, sys_errlist, sys_nerr - system error messages

SYNOPSIS

```
#include <stdio.h>

void
perror(const char *string)

extern const char * const sys_errlist[];
extern const int sys_nerr;

#include <string.h>

char *
strerror(int errnum)
```

DESCRIPTION

The **strerror()** and **perror()** functions look up the error message string corresponding to an error number.

The **strerror()** function accepts an error number argument *errnum* and returns a pointer to the corresponding message string.

The **perror()** function finds the error message corresponding to the current value of the global variable `errno` (intro(2)) and writes it, followed by a newline character, to the standard error file descriptor. If the argument string is non-`NULL` and does not point to the null character, this string is prepended to the message string and separated from it by a colon and space ("`:  `"); otherwise, only the error message string is printed.

If *errnum* is not a recognized error number, the error message string will contain "`Unknown error:  `" followed by the error number in decimal.

The message strings can be accessed directly using the external array `sys_errlist`. The external value `sys_nerr` contains a count of the messages in `sys_errlist`. The use of these variables is deprecated; **strerror()** should be used instead.

ISSUES

For unknown error numbers, the **strerror()** function will return its result in a static buffer which may be overwritten by subsequent calls.

Programs that use the deprecated `sys_errlist` variable often fail to compile because they declare it inconsistently.

STRING(3)

NAME

strcat, strncat, strchr, strrchr, strcmp, strncmp, strcasecmp, strncasecmp, strcpy, strncpy, strerror, strlen, strnlen, strpbrk, strsep, strspn, strcspn, strstr, strtok, index, rindex - string specific functions

SYNOPSIS

```
#include <string.h>

char *
strcat(char *s, const char * append)

char *
strncat(char *s, const char *append, size_t count)

char *
strchr(const char *s, int c)

char *
strrchr(const char *s, int c)

int
strcmp(const char *s1, const char *s2)

int
strncmp(const char *s1, const char *s2, size_t count)

int
strcasecmp(const char *s1, const char *s2)

int
strncasecmp(const char *s1, const char *s2,
size_t count)

size_t
strnlen(const char *s, size_t maxlen)

char *
strcpy(char *dst, const char *src)

char *
strncpy(char *dst, const char *src, size_t count)
```

```
char *
strerror(int errno)

size_t
strlen(const char *s)

char *
strpbrk(const char *s, const char *charset)

char *
strsep(char **stringp, const char *delim)

size_t
strspn(const char *s, const char *charset)

size_t
strcspn(const char *s, const char *charset)

char *
strstr(const char *big, const char *little)

char *
strtok(char *s, const char *delim)

char *
index(const char *s, int c)

char *
rindex(const char *s, int c)
```

DESCRIPTION

The string functions manipulate strings terminated by a null byte.

See the specific manual pages for more information. For manipulating variable length generic objects as byte strings (without the null byte check), see bstring(3).

Except as noted in their specific manual pages, the string functions do not test the destination for size limitations.

SEE ALSO

bstring(3), index(3), rindex(3), strcasecmp(3), strcat(3), strchr(3), strcmp(3), strcpy(3), strcspn(3), strerror(3), strlen(3), strpbrk(3), strrchr(3), strsep(3), strspn(3), strstr(3), strtok(3)

STANDARDS

The **strcat()**, **strncat()**, **strchr()**, **strrchr()**, **strcmp()**, **strncmp()**, **strcpy()**, **strncpy()**, **strerror()**, **strlen()**, **strpbrk()**, **strsep()**, **strspn()**, **strcspn()**, **strstr()**, and **strtok()** functions conform to ISO/IEC 9899:1990 ("ISO C90").

STRLCPY(3)

NAME

strlcpy, strlcat - size-bounded string copying and concatenation

SYNOPSIS

```
#include <string.h>

size_t
strlcpy(char *dst, const char *src, size_t size);

size_t
strlcat(char *dst, const char *src, size_t size);
```

DESCRIPTION

The **strlcpy()** and **strlcat()** functions copy and concatenate strings respectively. They are designed to be safer, more consistent, and less error prone replacements for strncpy(3) and strncat(3). Unlike those functions, **strlcpy()** and **strlcat()** take the full size of the buffer (not just the length) and guarantee to `NUL`-terminate the result (as long as size is larger than `0` or, in the case of **strlcat()**, as long as there is at least one byte free in *dst*). Note that a byte should be included for the `NUL` in *size*. Also note that **strlcpy()** and **strlcat()** only operate on true "C" strings. This means that for **strlcpy()** *src* must be `NUL`-terminated and for **strlcat()** both *src* and *dst* must be `NUL`-terminated.

The **strlcpy()** function copies up to *size* `- 1` characters from the `NUL`-terminated string *src* to *dst*, `NUL`-terminating the result.

The **strlcat()** function appends the `NUL`-terminated string *src* to the end of *dst*. It will append at most *size* `- strlen(`*dst*`) - 1` bytes, `NUL`-terminating the result.

RETURN VALUES

The **strlcpy()** and **strlcat()** functions return the total length of the string they tried to create. For **strlcpy()** that means the length of *src*. For **strlcat()** that means the initial length of *dst* plus the length of *src*. While this may seem somewhat confusing it was done to make truncation detection simple.

Note however, that if **strlcat()** traverses *size* characters without finding a `NUL`, the length of the string is considered to be *size* and the destination string will not be

NUL-terminated (since there was no space for the NUL). This keeps **strlcat()** from running off the end of a string. In practice this should not happen (as it means that either *size* is incorrect or that *dst* is not a proper "C" string). The check exists to prevent potential security problems in incorrect code.

EXAMPLES

The following code fragmen illustrates the simple case:

```
char *s, *p, buf[BUFSIZ];

...

(void)strlcpy(buf, s, sizeof(buf));
(void)strlcat(buf, p, sizeof(buf));
```

To detect truncation, perhaps while building a pathlen, something like the following might be used:

```
char *dir, *file, pname[MAXPATHLEN];

...

if (strlcpy(pname, dir, sizeof(pname)) >= sizeof(pname))
        goto toolong;
if (strlcat(pname, file, sizeof(pname)) >= sizeof(pname))
        goto toolong;
```

Since we know how many characters we copied the first time, we can speed things up a bit by using a copy instead of an append:

```
char *dir, *file, pname[MAXPATHLEN];
size_t n;

...

n = strlcpy(pname, dir, sizeof(pname));
if (n >= sizeof(pname))
        goto toolong;
if (strlcpy(pname + n, file, sizeof(pname) - n) >= sizeof(pname) - n)
        goto toolong;
```

However, one may question the validity of such optimizations, as they defeat the whole purpose of strlcpy() and strlcat().

SEE ALSO

snprintf(3), strncat(3), strncpy(3)

STRLEN(3)

NAME

strlen, strnlen - find length of string

SYNOPSIS

```
#include <string.h>

size_t
strlen(const char *s)

size_t
strnlen(const char *s, size_t maxlen);
```

DESCRIPTION

The **strlen()** function computes the length of the string *s*. The **strnlen()** function attempts to compute the length of *s*, but never scans beyond the first *maxlen* bytes of *s*.

RETURN VALUES

The **strlen()** function returns the number of characters that precede the terminating NUL character. The **strnlen()** function returns either the same result as **strlen()** or *maxlen*, whichever is smaller.

SEE ALSO

string(3), wcslen(3), wcswidth(3)

STANDARDS

The **strlen()** function conforms to ISO/IEC 9899:1990 (“ISO C90”). The **strnlen()** function conforms to IEEE Std 1003.1-2008 (”POSIX.1”).

STRPBRK(3)

NAME

strpbrk - locate multiple characters in string

SYNOPSIS

```
#include <string.h>

char *
strpbrk(const char *s, const char *charset)
```

DESCRIPTION

The **strpbrk()** function locates in the null-terminated string *s* the first occurrence of any character in the string *charset* and returns a pointer to this character. If no characters from *charset* occur anywhere in *s* then **strpbrk()** returns `NULL`.

SEE ALSO

index(3), memchr(3), rindex(3), strchr(3), strcspn(3), strrchr(3), strep(3), strspn(3), strstr(3), strtok(3)

STANDARDS

The **strpbrk()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

STRRCHR(3)

NAME

strrchr - locate character in string

SYNOPSIS

```
#include <string.h>

char *
strrchr(const char *s, int c)
```

DESCRIPTION

The **strrchr()** function locates the last occurrence of *c* (converted to a char) in the string *s*. If *c* is `'\0'`, **strrchr()** locates the terminating `'\0'`.

RETURN VALUES

The **strrchr()** function returns a pointer to the character, or a null pointer if *c* does not occur anywhere in *s*.

SEE ALSO

index(3), memchr(3), rindex(3), strchr(3), strcspn(3), strpbrk(3), strep(3), strspn(3), strstr(3), strtok(3)

STANDARDS

The **strrchr()** function conforms to ISO/IEC 9899:1990 ("ISO C90").

STRSEP(3)

NAME

strsep - separate strings

SYNOPSIS

```
#include <string.h>

char *
strsep(char **stringp, const char *delim)
```

DESCRIPTION

The **strsep()** function locates, in the string referenced by `*stringp`, the first occurrence of any character in the string *delim* (or the terminating `'\0'` character) and replaces it with a `'\0'`. The location of the next character after the delimiter character (or `NULL`, if the end of the string was reached) is stored in `*stringp`. The original value of `*stringp` is returned.

An "empty" field, i.e. one caused by two adjacent delimiter characters, can be detected by comparing the location referenced by the pointer returned in `*stringp` to `'\0'`.

If `*stringp` is initially `NULL`, **strsep()** returns `NULL`.

EXAMPLES

The following uses **strsep()** to parse a string, containing tokens delimited by white space, into an argument vector:

```
char **ap, *argv[10], *inputstring;
for (ap = argv;
     (*ap = strsep(&inputstring, " \t")) != NULL;)
   if (**ap != '\0')
      if (++ap >= &argv[10])
         break;
```

STRSPN(3)

NAME

strspn - span a string

SYNOPSIS

```
#include <string.h>

size_t
strspn(const char *s, const char *charset)
```

DESCRIPTION

The **strspn()** function spans the initial part of the null-terminated string *s* as long as the characters from *s* occur in string *charset*.

RETURN VALUES

The **strspn()** function returns the number of characters spanned.

SEE ALSO

index(3), memchr(3), rindex(3), strchr(3), strcspn(3), strpbrk(3), strrchr(3), strsep(3), strstr(3), strtok(3)

STANDARDS

The **strspn()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

STRSTR(3)

NAME

strstr, strcasestr, strnstr - locate a substring in a string

SYNOPSIS

```
#include <string.h>

char *
strstr(const char *big, const char *little)

char *
strcasestr(const char *big, const char *little);

char *
strnstr(const char *big, const char *little, size_t len);
```

DESCRIPTION

The **strstr()** function locates the first occurrence of the null-terminated string *little* in the null-terminated string *big*.

The **strcasestr()** function is similar to **strstr()**, but ignores the case of both strings.

The **strnstr()** function locates the first occurrence of the null-terminated string *little* in the string *big*, where not more than *len* characters are searched. Characters that appear after a ‘
0’ character are not searched. Since the **strnstr()** function is a Systems/C specific API, it should only be used when portability is not a concern.

RETURN VALUES

If *little* is the empty string, **strstr()** returns *big*. If *little* occurs nowhere in *big*, `NULL` is returned. Otherwise a pointer to the first character of the first occurrence of *little* is returned.

SEE ALSO

index(3), memchr(3), index(3), strchr(3), strcspn(3), strpbrk(3), strrchr(3), strsep(3), strspn(3), strtok(3)

STANDARDS

The **strstr()** function conforms to ISO/IEC 9899:1990 ("ISO C90").

STRTOK(3)

NAME

strtok, strtok_r - string tokens

SYNOPSIS

```
#include <string.h>

char *
strtok(char *str, const char *sep)

char *
strtok_r(char *str, const char *sep, char **last)
```

DESCRIPTION

This interface is obsoleted by strsep(3).

The **strtok()** function is used to isolate sequential tokens in a null-terminated string, *str*. These tokens are separated in the string by at least one of the characters in *sep*. The first time that **strtok()** is called *str* should be specified. Subsequent calls, intended to obtain further tokens from the same string, should pass a null pointer instead. The separator string, *sep*, must be supplied each time, and may change between calls.

The **strtok_r()** function is a reentrant version of **strtok()**. The context pointer *last* must be provided on each call. **strtok_r()** may also be used to nest two parsing loops within one another, as long as separate context pointers are used.

The **strtok()** and **strtok_r()** functions return a pointer to the beginning of each subsequent token in the string, after replacing the token itself with a `NUL` character. When no more tokens remain, a null pointer is returned.

EXAMPLE

The following uses **strtok_r()** to parse two strings using separate contexts:

```
char test[80], blah[80];
char *sep = "\/:;=-";
char *word, *phrase, *brkt, *brkb;
```

```
strcpy(test,
    "This;is.a:test:of=the/string\tokenizer-function.");

for (word = strtok_r(test, sep, &brkt);
     word;
     word = strtok_r(NULL, sep, &brkt))
{
   strcpy(blah, "blah:blat:blab:blag");

   for (phrase = strtok_r(blah, sep, &brkb);
        phrase;
        phrase = strtok_r(NULL, sep, &brkb))
   {
      printf("So far we're at %s:%s\n",
             word, phrase);
   }
}
```

SEE ALSO

index(3), memchr(3), rindex(3), strchr(3), strcspn(3), strpbrk(3), strrchr(3), strsep(3), strspn(3), strstr(3)

STANDARDS

The **strtok()** function conforms to ISO/IEC 9899:1990 ("ISO C90").

ISSUES

The System V **strtok()**, if handed a string containing only delimiter characters, will not alter the next starting point, so that a call to **strtok()** with a different (or empty) delimiter string may return a non-`NULL` value. Since this implementation always alters the next starting point, such a sequence of calls would always return `NULL`.

STRXFRM(3)

NAME

strxfrm - transform a string under locale

SYNOPSIS

```
#include <string.h>

size_t
strxfrm(char *dst, const char *src, size_t n)
```

DESCRIPTION

The **strxfrm()** function transforms a null-terminated string pointed to by *src* according to the current locale collation if any, then copies not more than *n*−1 characters of the resulting string into *dst*, terminating it with a null character and then returns the resulting length. Comparing two strings using **strcmp()** after **strxfrm()** is equal to comparing two original strings with **strcoll()**.

ISSUES

Sometimes the behavior of this function is unpredictable.

SEE ALSO

setlocale(3), strcmp(3), strcoll(3)

STANDARDS

The **strxfrm()** function conforms to ISO/IEC 9899:1990 (“ISO C90”).

SWAB(3)

NAME

swab - swap adjacent bytes

SYNOPSIS

```
#include <string.h>

void
swab(const void *src, void *dst, size_t len)
```

DESCRIPTION

The function **swab()** copies *len* bytes from the location referenced by *src* to the location referenced by *dst*, swapping adjacent bytes.

The argument *len* must be even number.

SEE ALSO

bzero(3), memset(3)

WCSWIDTH(3)

NAME

wcswidth - number of column positions in wide-character string

SYNOPSIS

```
int
wcswidth(const wchar_t *pwcs, size_t n);
```

DESCRIPTION

The **wcswidth()** function determins the number of column positions required for the first *n* characters of *pwcs*, or until a null wide character (`L'\0'`) is encountered.

RETURN VALUES

The **wcswidth()** function returns 0 if *pwcs* is an empty string (`L""`), -1 if a non-printing wide character is encountered, otherwise it returns the number of column positions occupied.

SEE ALSO

iswprint(3), wcwidth(3)

STANDARDS

The **wcswidth()** function conforms to ISO/IEC 9899:1999 (“ISO C99”).

WMEMCHR(3)

NAME

wmemchr, wmemcmp, wmemcpy, wmemmove, wmemset, wcscat, wcschr, wcscmp, wcscpy, wcscspn, wcslcat, wcslcpy, wcslen, wcsncat, wcsncmp, wcsncpy, wcsnlen, wcspbrk, wcsrchr, wcsspn, wcsstr - wide character string manipulation operations

SYNOPSIS

```
#include <wchar.h>

wchar_t *
wmemchr(const wchar_t *s, wchar_t c, size_t n);

int
wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);

wchar_t *
wmemcpy(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);

wchar_t *
wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);

wchar_t *
wmemset(wchar_t *s, wchar_t c, size_t n);

wchar_t *
wcscat(wchar_t * restrict s1, const wchar_t * restrict s2);

wchar_t *
wcschr(const wchar_t *s, wchar_t c);

int
wcscmp(const wchar_t *s1, const wchar_t *s2);

wchar_t *
wcscpy(wchar_t * restrict s1, const wchar_t * restrict s2);

size_t
wcsnlen(const wchar_t *s, size_t maxlen);

size_t
wcscspn(const wchar_t *s1, const wchar_t *s2);

size_t
```

```
wcslcat(wchar_t *s1, const wchar_t *s2, size_t n);

size_t
wcslcpy(wchar_t *s1, const wchar_t *s2, size_t n);

size_t
wcslen(const wchar_t *s);

wchar_t *
wcsncat(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);

int
wcsncmp(const wchar_t *s1, const wchar_t * s2, size_t n);

wchar_t *
wcsncpy(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);

wchar_t *
wcspbrk(const wchar_t *s1, const wchar_t *s2);

wchar_t *
wcsrchr(const wchar_t *s, wchar_t c);

size_t
wcsspn(const wchar_t *s1, const wchar_t *s2);

wchar_t *
wcsstr(const wchar_t *s1, const wchar_t *s2);
```

DESCRIPTION

The functions implement string manipulation operations over wide character strings. For a detailed description, refer to documents for the respective single-byte counterpart, such as memchr(3).

SEE ALSO

memchr(3), memcmp(3), memcpy(3), memmove(3), memset(3), strcat(3), strchr(3), strcmp(3), strcpy(3), strcspn(3), strlcat(3), strlcpy(3), strlen(3), strncat(3), strncmp(3), strncpy(3), strnlen(3). strpbrk(3), strrchr(3), strspn(3), strstr(3)

STANDARDS

These functions conform to ISO/IEC 9899:1999 (“ISO C99”), with the exception of **wcsnlen()**, which conforms to IEEE Std 1003.1-2008 (”POSIX.1”); and **wcslcat()** and **wcslcpy()**, which are extensions.

Regular Expression Library

The Systems/C library includes support for POSIX regular expressions in the regular expression library. Regular expressions are a very powerful programming paradigm used for text pattern matching.

REGEX(3)

NAME

regcomp, regexec, regerror, regfree, __reglen, __regcpy - regular-expression library

SYNOPSIS

```
#include <sys/types.h>
#include <regex.h>

int regcomp(regex_t *preg, const char *pattern,
int cflags);

int regexec(const regex_t *preg, const char *string,
size_t nmatch, regmatch_t pmatch[],
int eflags);

size_t regerror(int errcode, const regex_t *preg,
char *errbuf, size_t errbuf_size);

void regfree(regex_t *preg);

size_t __reglen(const regex_t *r);

void __regcpy(regex_t *dst, const regex_t *src);
```

DESCRIPTION

These routines implement POSIX 1003.2 regular expressions ("RE"s); see re_format(7). **regcomp()** compiles an RE written as a string into an internal form, **regexec()** matches that internal form against a string and reports results, **regerror()** transforms error codes from either into human-readable messages, and **regfree()** frees any dynamically-allocated storage used by the internal form of an RE.

Dignus-specific extensions **__reglen()** and **__regcpy()** make it possible to copy a `regex_t` and its internal components into user-allocated memory. **__reglen()** returns the length that will be needed to represent its argument as one contiguous block of memory, while **__regcpy()** will fill in that memory. **regfree()** should not be called on a pointer that has been the destination of **__regcpy()**. Instead, the memory should be freed according to whatever specific technique was used to allocate it.

Example of copying:

```
regex_t src;
regcomp(&src, ...);
regex_t *dst = my_malloc(__reglen(&src));
__regcpy(dst, &src);
/* optionally you can regfree(&src) now, and continue to use dst */
...
my_free(dst);
```

The header `<regex.h>` declares two structure types, `regex_t` and `rematch_t`, the former for compiled internal forms and the latter for match reporting. It also declares the four functions, a type `regoff_t`, and a number of constants with names starting with "`REG_`".

regcomp() compiles the regular expression contained in the pattern string, subject to the flags in *cflags*, and places the results in the `regex_t` structure pointed to by *preg*. *Cflags* is the bitwise OR of zero or more of the following flags:

`REG_EXTENDED`	Compile modern ("extended") REs, rather than the obsolete ("basic") REs that are the default.
`REG_BASIC`	This is a synonym for `0`, provided as a counterpart to `REG_EXTENDED` to improve readability.
`REG_NOSPEC`	Compile with recognition of all special characters turned off. All characters are thus considered ordinary, so the "RE" is a literal string. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. `REG_EXTENDED` and `REG_NOSPEC` may not be used in the same call to regcomp.
`REG_ICASE`	Compile for matching that ignores upper/lower case distinctions. See re_format(7).
`REG_NOSUB`	Compile for matching that need only report success or failure, not what was matched.
`REG_NEWLINE`	Compile for newline-sensitive matching. By default, newline is a completely ordinary character with no special meaning in either REs or strings. With this flag, '`[^`' bracket expressions and '`.`' never match newline, a '`^`' anchor matches the null string after any newline in the string in addition to its normal function, and the '`$`' anchor matches the null string before any newline in the string in addition to its normal function.

REG_PEND — The regular expression ends, not at the first NUL, but just before the character pointed to by the re_endp member of the structure pointed to by *preg*. The re_endp member is of type const char *. This flag permits inclusion of NULs in the RE; they are considered ordinary characters. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems.

When successful, **regcomp()** returns 0 and fills in the structure pointed to by *preg*. One member of that structure (other than re_endp) is publicized: re_nsub, of type size_t, contains the number of parenthesize subexpressions within the RE (except that the value of this member is undefined if the REG_NOSUB flag was used). If **regcomp()** fails, it returns a non-zero error code; see DIAGNOSTICS below.

regexec() matches the compiled RE pointed to by *preg* against the *string*, subject to the flags in *eflags*, and reports results using *nmatch*, *pmatch*, and the returned value. The RE must have been compiled by a previous invocation of **regcomp()**. The compiled form is not altered during execution of **regexec()**, so a single compiled RE can be used simultaneously by multiple threads.

By default, the NUL-terminated string pointed to by string is considered to be the text of an entire line, minus any terminating newline. The *eflags* argument is the bitwise OR of zero or more of the following flags:

REG_NOTBOL — The first character of the string is not the beginning of a line, so the '^' anchor should not match before it. This does not affect the behavior of newlines under REG_NEWLINE.

REG_NOTEOL — The NUL terminating the string does not end a line, so the '$' anchor should not match before it. This does not affect the behavior of newlines under REG_NEWLINE.

REG_STARTEND — The string is considered to start at string + pmatch[0].rm_so and to have a terminating NUL located at string + pmatch[0].rm_eo (there need not actually be a NUL at that location), regardless of the value of *nmatch*. See below for the definition of *pmatch* and *nmatch*. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. Note that a non-zero rm_so does not imply REG_NOTBOL; REG_STARTEND affects only the location of the string, not how it is matched.

See re_format(7) for a discussion of what is matched in situations where an RE or a portion thereof could match any of several substrings of *string*.

Normally, **regexec()** returns 0 for success and the non-zero code `REG_NOMATCH` for failure. Other non-zero error codes may be returned in exceptional situations; see DIAGNOSTICS.

If `REG_NOSUB` was specified in the compilation of the RE, or if *nmatch* is 0, regexec ignores the *pmatch* argument (but see below for the case where `REG_STARTEND` is specified). Otherwise, *pmatch* points to an array of *nmatch* structures of type `regmatch_t`. Such a structure has at least the members `rm_so` and `rm_eo`, both of type `regoff_t` (a signed arithmetic type at least as large as an `off_t` and a `ssize_t`), containing respectively the offset of the first character of a substring and the offset of the first character after the end of the substring. Offsets are measured from the beginning of the *string* argument given to **regexec()**. An empty substring is denoted by equal offsets, both indicating the character following the empty substring.

The 0^{th} member of the *pmatch* array is filled in to indicate what substring of *string* was matched by the entire RE. Remaining members report what substring was matched by parenthesized subexpressions within the RE; member `i` reports subexpression `i`, with subexpressions counted (starting at 1) by the order of their opening parentheses in the RE, left to right. Unused entries in the array– corresponding either to subexpressions that did not participate in the match at all, or to subexpressions that do not exist in the RE (that is, `i > preg->re_nsub`) – have both `rm_so` and `rm_eo` set to `-1`. If a subexpression participated in the match several times, the reported substring is the last one it matched. (Note, as an example in particular, that when the RE '`(b*)+`' matches '`bbb`', the parenthesized subexpression matches each of the three '`b`'s and then an infinite number of empty strings following the last '`b`', so the reported substring is one of the empties.)

If `REG_STARTEND` is specified, *pmatch* must point to at least one `regmatch_t` (even if *nmatch* is `0` or `REG_NOSUB` was specified), to hold the input offsets for `REG_STARTEND`. Use for output is still entirely controlled by *nmatch*; if *nmatch* is `0` or `REG_NOSUB` was specified, the value of `pmatch[0]` will not be changed by a successful **regexec()**.

regerror() maps a non-zero *errcode* from either **regcomp()** or **regexec()** to a human-readable, printable message. If *preg* is non-`NULL`, the error code should have arisen from use of the `regex_t` pointed to by *preg*, and if the error code came from regcomp, it should have been the result from the most recent regcomp using that `regex_t`. (Regerror may be able to supply a more detailed message using information from the `regex_t`.) Regerror places the `NUL`-terminated message into the buffer pointed to by *errbuf*, limiting the length (including the `NUL`) to at most *errbuf_size* bytes. If the whole message won't fit, as much of it as will fit before the terminating `NUL` is supplied. In any case, the returned value is the size of buffer needed to hold the whole message (including terminating `NUL`). If *errbuf_size* is `0`, *errbuf* is ignored but the return value is still correct.

If the *errcode* given to **regerror()** is first ORed with `REG_ITOA`, the "message" that results is the printable name of the error code, e.g. `REG_NOMATCH`, rather than an explanation thereof. If *errcode* is `REG_ATOI`, then *preg* shall be non-`NULL` and the `re_endp` member of the structure it points to must point to the printable name of

an error code; in this case, the result in *errbuf* is the decimal digits of the numeric value of the error code (`0` if the name is not recognized). `REG_ITOA` and `REG_ATOI` are intended primarily as debugging facilities; they are extensions, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. Be warned also that they are considered experimental and changes are possible.

regfree() frees any dynamically-allocated storage associated with the compiled RE pointed to by *preg*. The remaining `regex_t` is no longer a valid compiled RE and the effect of supplying it to regexec or regerror is undefined.

None of these functions references global variables except for tables of constants; all are safe for use from multiple threads if the arguments are safe.

IMPLEMENTATION CHOICES

There are a number of decisions that 1003.2 leaves up to the implementor, either by explicitly saying s"undefined" or by virtue of them being forbidden by the RE grammar.

This implementation treats them as follows.

See re_format(7) for a discussion of the definition of case-independent matching.

There is no particular limit on the length of REs, except insofar as memory is limited. Memory usage is approximately linear in RE size, and largely insensitive to RE complexity, except for bounded repetitions. See ISSUES for one short RE using them that will run almost any system out of memory.

A backslashed character other than one specifically given a magic meaning by 1003.2 (such magic meanings occur only in obsolete ["basic"] REs) is taken as an ordinary character.

Any unmatched `[` is a `REG_EBRACK` error.

Equivalence classes cannot begin or end bracket-expression ranges. The endpoint of one range cannot begin another.

`RE_DUP_MAX`, the limit on repetition counts in bounded repetitions, is 255.

A repetition operator (`?`, `*`, `+`, or bounds) cannot follow another repetition operator. A repetition operator cannot begin an expression or subexpression or follow '`^`' or '`|`'.

'`|`' cannot appear first or last in a (sub)expression or after another '`|`', i.e. an operand of '`|`' cannot be an empty subexpression. An empty parenthesized subexpression, '`()`', is legal and matches an empty (sub)string. An empty string is not a legal RE.

A '{' followed by a digit is considered the beginning of bounds for a bounded repetition, which must then follow the syntax for bounds. A '{' not followed by a digit is considered an ordinary character.

'^' and '$' beginning and ending subexpressions in obsolete ("basic") REs are anchors, not ordinary characters.

SEE ALSO

re_format(7)

POSIX 1003.2, sections 2.8 (Regular Expression Notation) and B.5 (C Binding for Regular Expression Matching).

DIAGNOSTICS

Non-zero error codes from **regcomp()** and **regexec()** include the following:

REG_NOMATCH	**regexec()** failed to match
REG_BADPAT	invalid regular expression
REG_ECOLLATE	invalid collating element
REG_ECTYPE	invalid character class
REG_EESCAPE	\ applied to unescapable character
REG_ESUBREG	invalid backreference number
REG_EBRACK	brackets [ ] not balanced
REG_EPAREN	parentheses ( ) not balanced
REG_EBRACE	braces { } not balanced
REG_BADBR	invalid repetition count(s) in { }
REG_ERANGE	invalid character range in [ ]
REG_ESPACE	ran out of memory
REG_BADRPT	?, *, or + operand invalid
REG_EMPTY	empty (sub)expression
REG_ASSERT	"can't happen"–you found a bug
REG_INVARG	invalid argument, e.g. negative-length string

ISSUES

There is one known functionality issue. The implementation of Internationalization is incomplete: the locale is always assumed to be the default one of 1003.2, and only the collating elements etc. of that locale are available.

Regcomp implements bounded repetitions by macro expansion, which is costly in time and space if counts are large or bounded repetitions are nested. An RE like, say, "`((((a{1,100}){1,100}){1,100}){1,100}){1,100}`" will (eventually) run almost any existing machine out of swap space.

Due to a mistake in 1003.2, things like '`a)b`' are legal REs because '`)`' is a special character only in the presence of a previous unmatched '`(`'. This can't be fixed until the spec is fixed.

The standard's definition of back references is vague. For example, does '`a\(\(b\)*\2\)*d`' match '`abbbd`'? Until the standard is clarified, behavior in such cases should not be relied on.

RE_FORMAT(7)

NAME

re_format - POSIX 1003.2 regular expressions

DESCRIPTION

Regular expressions ("RE"s), as defined in POSIX 1003.2, come in two forms: modern REs (roughly those of the POSIX utility egrep; 1003.2 calls these "extended" REs) and obsolete REs (roughly those of ed; 1003.2 "basic" Res). Obsolete REs mostly exist for backward compatibility in some old programs; they will be discussed at the end. 1003.2 leaves some aspects of RE syntax and semantics open; '*' marks decisions on these aspects that may not be fully portable to other 1003.2 implementations.

A (modern) RE is one* or more non-empty* *branches*, separated by '|'. It matches anything that matches one of the branches.

A *branch* is one* or more *pieces*, concatenated. It matches a match for the first, followed by a match for the second, etc.

A *piece* is an *atom* possibly followed by a single* '*', '+', '?', or *bound*. An atom followed by '*' matches a sequence of 0 or more matches of the atom. An atom followed by '+' matches a sequence of 1 or more matches of the atom. An atom followed by '?' matches a sequence of 0 or 1 matches of the atom.

A *bound* is '{' followed by an unsigned decimal integer, possibly followed by ',' possibly followed by another unsigned decimal integer, always followed by '}'. The integers must lie between 0 and RE_DUP_MAX (255*) inclusive, and if there are two of them, the first may not exceed the second. An atom followed by a bound containing one integer *i* and no comma matches a sequence of exactly *i* matches of the atom. An atom followed by a bound containing one integer *i* and a comma matches a sequence of *i* or more matches of the atom. An atom followed by a bound containing two integers *i* and *j* matches a sequence of *i* through *j* (inclusive) matches of the atom.

An atom is a regular expression enclosed in '()' (matching a match for the regular expression), an empty set of '()' (matching the null string)*, a *bracket expression* (see below), '.' (matching any single character), '^' (matching the null string at the beginning of a line), '$' (matching the null string at the end of a line), a '\' followed by one of the characters '^.[$()|*+?{\' (matching that character taken as an ordinary character), a '\' followed by any other character* (matching that character taken as an ordinary character, as if the '\' had not been present*), or a single character with no other significance (matching that character). A '{' followed by a character other than a digit is an ordinary character, not the beginning of a bound*. It is illegal to end an RE with '\'.

A *bracket expression* is a list of characters enclosed in '`[]`'. It normally matches any single character from the list (but see below). If the list begins with '`^`', it matches any single character (but see below) not from the rest of the list. If two characters in the list are separated by '`-`', this is shorthand for the full *range* of characters between those two (inclusive) in the collating sequence, e.g. '`[0-9]`' in ASCII matches any decimal digit. It is illegal* for two ranges to share an endpoint, e.g. '`a-c-e`'. Ranges are very collating-sequence-dependent, and portable programs should avoid relying on them.

To include a literal '`]`' in the list, make it the first character (following a possible '`^`'). To include a literal '`-`', make it the first or last character, or the second endpoint of a range. To use a literal '`-`' as the first endpoint of a range, enclose it in '`[.`' and '`.]`' to make it a collating element (see below). With the exception of these and some combinations using '`[`' (see next paragraphs), all other special characters, including '\', lose their special significance within a bracket expression.

Within a bracket expression, a collating element (a character, a multi-character sequence that collates as if it were a single character, or a collating-sequence name for either) enclosed in '`[.`' and '`.]`' stands for the sequence of characters of that collating element. The sequence is a single element of the bracket expression's list. A bracket expression containing a multi-character collating element can thus match more than one character, e.g. if the collating sequence includes a '`ch`' collating element, then the RE '`[[.ch.]]*c`' matches the first five characters of '`chchcc`'.

Within a bracket expression, a collating element enclosed in '`[=`' and '`=]`' is an equivalence class, standing for the sequences of characters of all collating elements equivalent to that one, including itself. (If there are no other equivalent collating elements, the treatment is as if the enclosing delimiters were '`[.`' and '`.]`'.) For example, if '`x`' and '`y`' are the members of an equivalence class, then '`[[=x=]]`', '`[[=y=]]`', and '`[xy]`' are all synonymous. An equivalence class may not* be an endpoint of a range.

Within a bracket expression, the name of a character class enclosed in '`[:`' and '`:]`' stands for the list of all characters belonging to that class. Standard character class names are:

```
alnum  digit  punct
alpha  graph  space
blank  lower  upper
cntrl  print  digit
```

These stand for the character classes defined in ctype(3). A locale may provide others. A character class may not be used as an endpoint of a range.

There are two special cases* of bracket expressions: the bracket expressions '`[[:<:]]`' and '`[[:>:]]`' match the null string at the beginning and end of a word respectively. A word is defined as a sequence of word characters which is neither preceded nor followed by word characters. A word character is an alnum character (as defined by ctype(3)) or an underscore. This is an extension, compatible with

but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems.

In the event that an RE could match more than one substring of a given string, the RE matches the one starting earliest in the string. If the RE could match more than one substring starting at that point, it matches the longest. Subexpressions also match the longest possible substrings, subject to the constraint that the whole match be as long as possible, with subexpressions starting earlier in the RE taking priority over ones starting later. Note that higher-level subexpressions thus take priority over their lower-level component subexpressions.

Match lengths are measured in characters, not collating elements. A null string Is considered longer than no match at all. For example, '`bb*`' matches the three middle characters of '`abbbc`', '`(wee|week)(knights|nights)`' matches all ten characters of '`weeknights`', when '`(.*).*`' is matched against '`abc`' the parenthesized subexpression matches all three characters, and when '`(a*)*`' is matched against '`bc`' both the whole RE and the parenthesized subexpression match the null string.

If case-independent matching is specified, the effect is much as if all case distinctions had vanished from the alphabet. When an alphabetic that exists in multiple cases appears as an ordinary character outside a bracket expression, it is effectively transformed into a bracket expression containing both cases, e.g. '`x`' becomes '`[xX]`'. When it appears inside a bracket expression, all case counterparts of it are added to the bracket expression, so that (e.g.) '`[x]`' becomes '`[xX]`' and '`[^x]`' becomes '`[^xX]`'.

No particular limit is imposed on the length of REs*. Programs intended to be portable should not employ REs longer than 256 bytes, as an implementation can refuse to accept such REs and remain POSIX-compliant.

Obsolete ("basic") regular expressions differ in several respects. '`|`', '`+`', and '`?`' are ordinary characters and there is no equivalent for their functionality. The delimiters for bounds are '\{' and '\}', with '{' and '}' by themselves ordinary characters. The parentheses for nested subexpressions are '\(' and '\)', with '`(`' and '`)`' by themselves ordinary characters. '`^`' is an ordinary character except at the beginning of the RE or* the beginning of a parenthesized subexpression, '`$`' is an ordinary character except at the end of the RE or* the end of a parenthesized subexpression, and '`*`' is an ordinary character if it appears at the beginning of the RE or the beginning of a parenthesized subexpression (after a possible leading '`^`'). Finally, there is one new type of atom, a *back reference*: '\' followed by a non-zero decimal digit *d* matches the same sequence of characters matched by the d^{th} parenthesized subexpression (numbering subexpressions by the positions of their opening parentheses, left to right), so that (e.g.) '`\([bc]\)\1`' matches '`bb`' or '`cc`' but not '`bc`'.

SEE ALSO

regex(3)

POSIX 1003.2, section 2.8 (Regular Expression Notation).

ISSUES

The current 1003.2 spec says that '`)`' is an ordinary character in the absence of an unmatched '`(`'; this was an unintentional result of a wording error, and change is likely. Avoid relying on it.

Back references are a dreadful botch, posing major prob lems for efficient implementations. They are also somewhat vaguely defined (does '`a\(\(b\)*\2\)*d`' match '`abbbd`'?). Avoid using them.

1003.2's specification of case-independent matching is vague. The "one case implies all cases" definition given above is current consensus among implementors as to the right interpretation.

Net Library

The Net library provides the set of functions related to network operations and TCP/IP.

ADDR2ASCII(3)

NAME

addr2ascii, ascii2addr - Generic address formatting routines

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>

char *
addr2ascii(int af, const void *addrp, int len, char *buf)

int
ascii2addr(int af, const char *ascii, void *result)
```

DESCRIPTION

The routines **addr2ascii()** and **ascii2addr()** are used to convert network addresses between binary form and a printable form appropriate to the address family. Both functions take an *af* argument, specifying the address family to be used in the conversion process. (Currently, only the `AF_INET` and `AF_LINK` address families are supported.)

The **addr2ascii()** function is used to convert binary, network-format addresses into printable form. In addition to *af*, there are three other arguments. The *addrp* argument is a pointer to the network address to be converted. The *len* argument is the length of the addresss. The buf argument is an optional pointer to a caller-allocated buffer to hold the result; if a `NULL` pointer is passed, **addr2ascii()** uses a statically-allocated buffer.

The **ascii2addr()** function performs the inverse operation to **addr2ascii()**. In addition to *af*, it takes two parameters, *ascii* and *result*. The *ascii* parameter is a pointer to the string which is to be converted into binary. The *result* parameter is a pointer to an appropriate network address structure for the specified family.

The following gives the appropriate structure to use for binary addresses in the specified family:

`AF_INET`	`struct in_addr` (in `<netinet/in.h>`)
`AF_LINK`	`struct sockaddr_dl` (in `<net/if_dl.h>`)

RETURN VALUES

The **addr2ascii()** function returns the address of the buffer it was passed, or a static buffer if the `NULL` pointer was passed; on failure it returns a `NULL` pointer. The **ascii2addr()** function returns the length of the binary address in bytes, or -1 on failure.

EXAMPLES

The inet(3) functions **inet_ntoa()** and **inet_aton()** could be implemented thusly:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

char *
inet_ntoa(struct in_addr addr)
{
   return addr2ascii(AF_INET, &addr, sizeof addr, 0);
}

int
inet_aton(const char *ascii, struct in_addr *addr)
{
   return (ascii2addr(AF_INET, ascii, addr)
               == sizeof(*addr));
}
```

In actuality, this cannot be done because add2ascii(0 and **ascii2addr()** are implemented in terms of the inet(3) functions, rather than the other way around.

ERRORS

When a failure is returned, `errno` is set to one of the following values:

`[ENAMETOOLONG]` The **addr2ascii()** routine was passed a *len* parameter which was inappropriate for the address family given by *af*.

`[EPROTONOSUPPORT]` Either routine was passed an af parameter other than `AF_INET` or `AF_LINK`.

`[EINVAL]` The string passed to **ascii2addr()** was improperly formatted for address family *af*.

SEE ALSO

inet(3), linkaddr(3), inet(4)

BYTEORDER(3)

NAME

htonl, htons, ntohl, ntohs - convert values between host and network byte order

SYNOPSIS

```
#include <arpa/inet.h>

or

#include <netinet/in.h>

uint32_t
htonl(uint32_t hostlong)

uint16_t
htons(uint16_t hostshort)

uint32_t
ntohl(uint32_t netlong)

uint16_t
ntohs(uint16_t netshort)
```

DESCRIPTION

These routines convert 16 and 32 bit quantities between network byte order and host byte order. On machines which have a byte order which is the same as the network order, these routines are defined as null macros.

These routines are most often used in conjunction with Internet addresses and ports as returned by gethostbyname(3) and getservent(3).

SEE ALSO

gethostbyname(3), getservent(3)

ETHERS(3)

NAME

ethers, ethers_line, ether_aton, ether_ntoa, ether_ntohost, ether_hostton - Ethernet address conversion and lookup routines.

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <net/ethernet.h>

int
ether_line(char *l, struct ether_addr *e, char *hostname)

struct ether_addr *
ether_aton(char *a)

char *
ether_ntoa(struct ether_addr *n)

int
ether_ntohost(char *hostname, struct ether_addr *e)

int
ether_hostton(char *hostname, struct ether_addr *e)
```

DESCRIPTION

These functions operate on ethernet addresses using an `ether_addr` structure, which is defined in the header file `<netinet/if_ether.h>`:

```
/*
 * The number of bytes in an ethernet (MAC)
 * address.
 */
#define ETHER_ADDR_LEN  6

/*
 * Structure of a 48-bit Ethernet address.
 */
struct ether_addr {
```

```
    u_char octet[ETHER_ADDR_LEN];
}
```

The function **ether_line()** scans *l*, an string in ethers(5) format and sets *e* to the ethernet address specified in the string and *hostname* to the hostname. This function is used to parse lines from the ethers specification file (typically `/etc/ethers` on UNIX hosts) into their component parts.

The **ether_aton()** function converts a string representation of an ethernet address into an `ether_addr` structure. Likewise, **ether_ntoa()** converts an ethernet address specified as an `ether_addr` structure into a string.

The **ether_ntohost()** and **ether_hostton()** functions map ethernet addresses to their corresponding hostnames as specified in the ethers specification file (typically /etc/ethers on UNIX hosts.) **ether_ntohost()** converts from ethernet address to hostname, and **ether_hostton()** converts from hostname to ethernet address.

RETURN VALUES

ether_line() returns zero on success and non-zero if it was unable to parse any part of the supplied line *l*. It returns the extracted ethernet address in the supplied `ether_addr` structure *e* and returns the hostname in the supplied string *hostname*.

On success, **ether_ntoa()** returns a pointer to a string containing the string representation of an ether address. If it is unable to convert the supplied `ether_addr` structure, it returns a `NULL` pointer. Likewise, **ether_aton()** returns a pointer to an `ether_addr` structure on success and a `NULL` pointer on failure.

The **ether_ntohost()** and **ether_hostton()** functions both return zero on success and non-zero if they were unable to find a match in the ethers database file.

FILES

These functions use the algorithm outlined in the IBM TCP/IP documentation to locate the `ETC.ETHERS` file; using the location specified in the `TCPIP.DATA` file.

NOTES

The user must insure that the hostname strings passed to the **ether_line()**, **ether_ntohost()** and **ether_hostton()** functions are large enough to contain the returned hostnames.

ISSUES

The **ether_aton()** and **ether_ntoa()** functions return values that are stored in static memory areas which may be overwritten the next time they are called.

GAI_STRERROR(3)

NAME

gai_strerror – get error message string from EAI_xxx error code

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

const char *
gai_strerror(int ecode);
```

DESCRIPTION

The **gai_strerror()** function returns an error message string corresponding to the error code returned by getaddrinfo(3) or getnameinfo(3).

The following error codes and their meaning are defined in `<netdb.h>`:

`EAI_AGAIN`	temporary failure in name resolution
`EAI_BADFLAGS`	invalid value for *ai_flags*
`EAI_BADHINTS`	invalid value for *hints*
`EAI_FAIL`	non-recoverable failure in name resolution
`EAI_FAMILY`	*ai_family* not supported
`EAI_MEMORY`	memory allocation failure
`EAI_NONAME`	hostname or servname not provided, or not known
`EAI_PROTOCOL`	resolved protocol is unknown
`EAI_SERVICE`	servname not supported for *ai_socktype*
`EAI_SOCKTYPE`	*ai_socktype* not supported
`EAI_SYSTEM`	system error returned in `errno`

RETURN VALUES

The **gai_strerror()** function returns a pointer to the error message string corresponding to ecode. If *ecode* is out of range, an implementation-specific error message string is returned.

SEE ALSO

getaddrinfo(3), getnameinfo(3)

GETADDRINFO(3)

NAME

getaddrinfo, freeaddrinfo – socket address structure to host and service name

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int
getaddrinfo(const char *hostname, const char *servname,
            const struct addrinfo *hints, struct addrinfo **res);

void
freeaddrinfo(struct addrinfo *ai);
```

DESCRIPTION

The **getaddrinfo()** function is used to get a list of IP addresses and port numbers for host hostname and service servname. It is a replacement for and provides more flexibility than the gethostbyname(3) and getservbyname(3) functions.

The *hostname* and *servname* arguments are either pointers to `NUL`-terminated strings or the null pointer. An acceptable value for *hostname* is either a valid host name or a numeric host address string consisting of a dotted decimal IPv4 address or an IPv6 address. The *servname* is either a decimal port number or a service name. At least one of *hostname* and *servname* must be non-null.

hints is an optional pointer to a struct addrinfo, as defined by `<netdb.h>`:

```
struct addrinfo {
int ai_flags;      /* input flags */
int ai_family;      /* protocol family for socket */
int ai_socktype;      /* socket type */
int ai_protocol;      /* protocol for socket */
socklen_t ai_addrlen;   /* length of socket-address */
struct sockaddr *ai_addr; /* socket-address for socket */
char *ai_canonname;     /* canonical name for service location */
struct addrinfo *ai_next; /* pointer to next in list */
};
```

This structure can be used to provide hints concerning the type of socket that the caller supports or wishes to use. The caller can supply the following structure elements in hints:

ai_family The protocol family that should be used. When *ai_family* is set to `PF_UNSPEC`, it means the caller will accept any protocol family supported by the operating system.

ai_socktype Denotes the type of socket that is wanted: `SOCK_STREAM`, `SOCK_DGRAM`, or `SOCK_RAW`. When *ai_socktype* is zero the caller will accept any socket type.

ai_protocol Indicates which transport protocol is desired, `IPPROTO_UDP` or `IPPROTO_TCP`. If *ai_protocol* is zero the caller will accept any protocol.

ai_flags *ai_flags* is formed by OR'ing the following values:

`AI_CANONNAME` If the `AI_CANONNAME` bit is set, a successful call to **getaddrinfo()** will return a NUL-terminated string containing the canonical name of the specified hostname in the *ai_canonname* element of the first addrinfo structure returned.

`AI_NUMERICHOST` If the `AI_NUMERICHOST` bit is set, it indi cates that hostname should be treated as a numeric string defining an IPv4 or IPv6 address and no name resolution should be attempted.

`AI_PASSIVE` If the `AI_PASSIVE` bit is set it indicates that the returned socket address structure is intended for use in a call to bind(2). In this case, if the hostname argument is the null pointer, then the IP address portion of the socket address structure will be set to `INADDR_ANY` for an IPv4 address or `IN6ADDR_ANY_INIT` for an IPv6 address.

If the `AI_PASSIVE` bit is not set, the returned socket address structure will be ready for use in a call to connect(2) for a connection-oriented protocol or connect(2), sendto(2), or sendmsg(2) if a connectionless protocol was chosen. The IP address portion of the socket address structure will be set to the loopback address if hostname is the null pointer and `AI_PASSIVE` is not set.

All other elements of the addrinfo structure passed via hints must be zero or the null pointer.

If hints is the null pointer, **getaddrinfo()** behaves as if the caller provided a struct addrinfo with *ai_family* set to `PF_UNSPEC` and all other elements set to zero or `NULL`.

After a successful call to getaddrinfo(), **res* is a pointer to a linked list of one or more addrinfo structures. The list can be traversed by following the *ai_next* pointer in each addrinfo structure until a null pointer is encountered. The three members *ai_family*, *ai_socktype*, and *ai_protocol* in each returned addrinfo structure are suitable for a call to socket(2). For each addrinfo structure in the list, the *ai_addr* member points to a filled-in socket address structure of length *ai_addrlen*.

This implementation of **getaddrinfo()** allows numeric IPv6 address notation with scope identifier, as documented in chapter 11 of draft-ietf-ipv6-scoping-arch-02.txt. By appending the percent character and scope identifier to addresses, one can fill the sin6_scope_id field for addresses. This would make management of scoped addresses easier and allows cut-and-paste input of scoped addresses.

At this moment the code supports only link-local addresses with the fo mat. The scope identifier is hardcoded to the name of the hardware interface associated with the link (such as ne0). An example is "`fe80::1%ne0`", which means "`fe80::1` on the link associated with the ne0 interface".

The current implementation assumes a one-to-one relationship between the interface and link, which is not necessarily true from the specification.

All of the information returned by **getaddrinfo()** is dynamically allocated: the addrinfo structures themselves as well as the socket address structures and the canonical host name strings included in the addrinfo structures.

Memory allocated for the dynamically allocated structures created by a successful call to **getaddrinfo()** is released by the **freeaddrinfo()** function. The *ai* pointer should be a addrinfo structure created by a call to **getaddrinfo()**.

RETURN VALUES

getaddrinfo() returns zero on success or one of the error codes listed in gai_strerror(3) if an error occurs.

EXAMPLES

The following code tries to connect to "`www.kame.net`" service "`http`" via a stream socket. It loops through all the addresses available, regardless of address family. If the destination resolves to an IPv4 address, it will use an `AF_INET` socket. Similarly, if it resolves to IPv6, an `AF_INET6` socket is used. Observe that there is no hardcoded reference to a particular address family. The code works even if getaddrinfo() returns addresses that are not IPv4/v6.

```
struct addrinfo hints, *res, *res0;
int error;
```

```
int s;
const char *cause = NULL;

memset(&hints, 0, sizeof(hints));
hints.ai_family = PF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
error = getaddrinfo("www.kame.net", "http", &hints, &res0);
if (error) {
errx(1, "%s", gai_strerror(error));
/*NOTREACHED*/
}
s = -1;
for (res = res0; res; res = res->ai_next) {
s = socket(res->ai_family, res->ai_socktype,
    res->ai_protocol);
if (s < 0) {
cause = "socket";
continue;
}

if (connect(s, res->ai_addr, res->ai_addrlen) < 0) {
cause = "connect";
close(s);
s = -1;
continue;
}

break;  /* okay we got one */
}
if (s < 0) {
err(1, "%s", cause);
/*NOTREACHED*/
}
freeaddrinfo(res0);
```

The following example tries to open a wildcard listening socket onto service "`http`", for all the address families available.

```
struct addrinfo hints, *res, *res0;
int error;
int s[MAXSOCK];
int nsock;
const char *cause = NULL;

memset(&hints, 0, sizeof(hints));
hints.ai_family = PF_UNSPEC;
```

```
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;
error = getaddrinfo(NULL, "http", &hints, &res0);
if (error) {
errx(1, "%s", gai_strerror(error));
/*NOTREACHED*/
}
nsock = 0;
for (res = res0; res && nsock < MAXSOCK; res = res->ai_next) {
s[nsock] = socket(res->ai_family, res->ai_socktype,
    res->ai_protocol);
if (s[nsock] < 0) {
cause = "socket";
continue;
}

if (bind(s[nsock], res->ai_addr, res->ai_addrlen) < 0) {
cause = "bind";
close(s[nsock]);
continue;
}
(void) listen(s[nsock], 5);

nsock++;
}
if (nsock == 0) {
err(1, "%s", cause);
/*NOTREACHED*/
}
freeaddrinfo(res0);
```

SEE ALSO

bind(2), connect(2), send(2), socket(2), gethostbyname(3), getnameinfo(3), getservbyname(3), resolver(3)

R. Gilligan, S. Thomson, J. Bound, J. McCann, and W. Stevens, Basic Socket Interface Extensions for IPv6, RFC 3493, February 2003.

S. Deering, B. Haberman, T. Jinmei, E. Nordmark, and B. Zill, IPv6 Scoped Address Architecture, internet draft, draft-ietf-ipv6-scoping- arch-02.txt, work in progress material.

Craig Metz, "Protocol Independence Using the Sockets API", Proceedings of the freenix track: 2000 USENIX annual technical conference, June 2000.

STANDARDS

The getaddrinfo() function is defined by the IEEE Std 1003.1g-2000 ("POSIX.1") draft specification and documented in RFC 3493, "Basic Socket Interface Extensions for IPv6".

GETHOSTBYNAME(3)

NAME

gethostbyname, gethostbtname2, gethostbyaddr, gethostent, sethostent, endhostent, herror, hstrerror - get network host entry.

SYNOPSIS

```
#include <netdb.h>

extern int h_errno;

struct hostent *
gethostbyname(const char *name)

struct hostent *
gethostbyname2(const char *name, int af)

struct hostent *
gethostbyaddr(const char *addr, int len, int type)

struct hostent *
gethostent(void)

void
sethostent(int stayopen)

void
endhostent(void)

void
herror(const char *string)

const char *
hstrerror(int err)
```

DESCRIPTION

The **gethostbyname()**, **gethostbyname2()** and **gethostbyaddr()** functions each return a pointer to an object with the following structure describing an internet host referenced by name or by address, respectively. This structure contains either the information obtained from the name server, or broken-out fields from a line in the host information file. If the local name server is not running, these routines do a lookup in the host information file.

```
struct hostent {
char  *h_name;       /* official name of host */
char  **h_aliases;   /* alias list        */
int   h_addrtype;    /* host address type */
int   h_length;      /* length of address */
char  **h_addr_list; /* list of addresses */
                     /*  from name server */
}
#define h_addr  h_addr_list[0] /* address,for */
                               /*   backwards      */
                               /*   compatibility  */
```

The members of this structure are:

`h_name`	Official name of the host.
`h_aliases`	A `NULL`-terminated array of alternate names for the host.
`h_addrtype`	The type of address being returned; usually `AF_INET`.
`h_length`	The length, in bytes, of the address.
`h_addr_list`	A `NULL`-terminated array of network addresses for the host. Host addresses are returned in network byte order.
`h_addr`	The first address in `h_addr_list`; this is for backward compatibility.

When using the nameserver, **gethostbyname()** and **gethostaddr()** will search for the named host in the current domain and its parents unless the name ends in a dot. If the name contains no dot, and if the environment variable "`HOSTALIASES`" contains the name of an alias file, the alias file will first be searched for an alias matching the input name.

The **gethostbyname2()** function is an evolution of **gethostbyname()** which is intended to allow lookups in address families other than `AF_INET`, for example `AF_INET6`. Currently the *af* argument must be specified as `AF_INET` else the function will return `NULL` after having set `h_errno` to `NETDB_INTERNAL`.

The **sethostent()** function may be used to request the use of a connected TCP socket for queries. If the *stayopen* flag is non-zero, this sets the option to send all queries to the name server using TCP and to retain the connection after each call to **gethostbyname()**, **gethostbyname2()** or **gethostbyaddr()**. Otherwise, queries are performed using UDP datagrams.

The **endhostent()** function closes the TCP connection.

The **herror()** function writes a message to the diagnostic output consisting of the string parameter *string*, the constant string `":  "`, and a message corresponding to the value of `h_errno`.

The **hstrerror()** function returns a string which is the message text corresponding to the value of the *err* parameter.

FILES

These functions use the algorithm described in the IBM TCP/IP documentation to locate the `TCPIP.DATA`, `ETC.HOSTS`, `ETC.HOST.CONF` and `ETC.RESOLV.CONF` files.

DIAGNOSTICS

Error return status from **gethostbyname()**, **gethostbyname2()** and **gethostbyaddr()** is indicated by return of a `NULL` pointer. The external integer `h_errno` may then be checked to see whether this is a temporary failure or an invalid or unknown host. The routine **herror()** can be used to print an error message describing the failure. If its argument string is non-`NULL`, it is printed, followed by a colon and a space. The error message is printed with a trailing newline.

The variable `h_errno` can have the following values:

`HOST_NOT_FOUND`	No such host is known.
`TRY_AGAIN`	This is usually a temporary error and means that the local server did not receive a response from an authoritative server. A retry at some later time may succeed.
`NO_RECOVERY`	Some unexpected server failure was encountered. This is a non-recoverable error.
`NO_DATA`	The requested name is valid but does not have an IP address; this is not a temporary error. This means that the name is known to the name of the server but there is no address associated with this name. Another type of request to the name server using this domain name will result in an answer; for example, a mail-forwarder may be registered for this domain.

SEE ALSO

resolver(3), IBM TCP/IP documentation

CAVEAT

The **gethostent()** function reads the next line of the `ETC.HOSTS` file, opening the file if necessary.

The **sethostent()** function opens and/or rewinds the `ETC.HOSTS` file. If the *stayopen* argument is non-zero, the file will not be closed after each call to **gethostbyname()**, **gethostbyname2()** or **gethostbyaddr()**.

The **endhostent()** function closes the file.

ISSUES

These functions use static data storage; if the data is needed for future use, it should be copied before any subsequent calls overwrite it. Only the Internet address format is currently understood.

__NSSWITCH_LINE(3)

NAME

__nsswitch_line - process an nsswitch configuration string

SYNOPSIS

```
#include <nsswitch.h>

int
__nsswitch_line(const char *line)
```

DESCRIPTION

The **__nsswitch_line()**, function processes a string as if it were a line of text in the Unix `nsswitch.conf` configuration file. It is useful for specifying the order in which DNS resolution techniques should be used by functions such as **gethostbyname()**. The syntax of the string is "*database*: *method1 method2 ...*". The only database that is currently useful is `hosts`, which can use the following methods:

`files`	Use `ETC.HOSTS` file for hostname lookup.
`dns`	Use the DNS resolver built into the Dignus runtime.
`ibm`	Use IBM's `EZASMI` or `BPX` interface for hostname lookup.

The default is `"hosts: ibm files dns"`. Which indicates the library should try the IBM resolver first, then look for host files, then use the Dignus resolver library. To limit the search to just the IBM resolver, set the hosts line to `"hosts: ibm"`.

DIAGNOSTICS

On success, **__nsswitch_line()** returns 1. If the string does not conform to the requirements, it returns 0 and sets `errno` to `EINVAL`.

NOTES

The name `ezasmi` was previously used to indicate the IBM `EZASMI` interface should be employed. With the change of the Dignus runtime to use the `BPX` socket interface, the name was changed to `ibm`. The name `ezasmi` is still accepted, and is equivalent to `ibm`.

SEE ALSO

gethostbyname(3), IBM TCP/IP documentation

GETIPNODEBYNAME(3)

NAME

getipnodebyname, getipnodebyaddr, freehostent - nodenametoaddress and addresstonodename translation

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

struct hostent *
getipnodebyname(const char *name, int af, int flags, int *error_num);

struct hostent *
getipnodebyaddr(const void *src, size_t len, int af, int *error_num);

void
freehostent(struct hostent *ptr);
```

DESCRIPTION

getipnodebyname() and **getipnodebyaddr()** functions are very similar to gethostbyname(3), gethostbyname2(3) and gethostbyaddr(3). The functions cover all the functionalities provided by the older ones, and provide better interface to programmers. The functions require additional arguments, *af*, and *flags*, for specifying address family and operation mode. The additional arguments allow the programmer to get the address for a nodename, for specific address family (such as `AF_INET` or `AF_INET6`). The functions also require an additional pointer argument, *error_num* to return the appropriate error code, to support thread safe error code returns.

The type and usage of the return value, `struct hostent` is described in gethostbyname(3).

For **getipnodebyname()**, the *name* argument can be either a node name or a numeric address string (i.e. a dotted-decimal IPv4 address or an IPv6 hex address.) The *af* argument specifies the address family, either `AF_INET` or `AF_INET6`. The *flags* argument specifies the types of addresses that are searched for, and the types of addresses that are returend. Note that a specifal *flags* value of `AI_DEFAULT` (defined blow) should handle most applications. That is, porting simple applications to use IPv6 replaces the call

```
hptr = gethostbyname(name);
```

with

```
hptr = getipnodebyname(name, AF_INET6, AI_DEFAULT, &error_num);
```

Applications desiring finer control over the types of addresses searched for and returned can specify other combinations of the *flags* argument.

A *flags* of `0` implies a strict interpretation of the *af* argument:

- If *flags* is `0` and *af* is `AF_INET`, then the caller wants only IPv4 addresses. A query is made for `A` records. If successful, the IPv4 addresses are returned and the `h_length` member of the `hostent` structure will be 4, else the function returns a `NULL` pointer.
- If *flags* is 0 and if af is `AF_INET6`, then the caller wants only IPv6 addresses. A query is made for `AAAA` records. If successful, the IPv6 addresses are returned and the `h_length` member of the `hostent` structure will be 16, else the function returns a `NULL` pointer.

Other constants can be logically-ORed into the *flags* argument to modify behavior of the function.

- If the `AI_V4MAPPED` flag is specified along with an *af* of `AF_INET6`, then the caller will accept IPv4-mapped IPv6 addresses. That is, if no `AAAA` records are found then a query is made for `A` records and any found are returned as IPv4-mapped IPv6 addresses (`h_length` will be 16). The `AI_V4MAPPED` flag is ignored unless *af* equals `AF_INET6`.
- The `AI_V4MAPPED_CFG` flag is the same as the `AI_V4MAPPED` flag only if the underlying system supports IPv4-mapped IPv6 address.
- If the `AI_ALL` flag is used in conjunction with the `AI_V4MAPPED` flag, and only used with the IPv6 address family. When `AI_ALL` is logically OR'd with `AI_V4MAPPED` flag then the caller wants all addresses: IPv6 and IPv4-mapped IPv6. A query is first made for `AAAA` records and if successful, the IPv6 addresses are returned. Another query is then made for `A` records and any found are returned as IPv4-mapped IPv6 addresses. `h_length` will be 16. Only if both queries fail does the function return a `NULL` pointer. This flag is ignored unless af equals `AF_INET6`. If both `AI_ALL` and `AI_V4MAPPED` are specified, `AI_ALL` takes precedence.
- The `AI_ADDRCONFIG` flag specifies that a query for `AAAA` records should occur only if the node has at least one IPv6 source address configured and a query for `A` records should occur only if the node has at least one IPv4 source address configured.

For example, if the node has no IPv6 source addresses configured, and *af* equals `AF_INET6`, and the node name being looked up has both `AAAA` and `A` records, then: (a) if only `AI_ADDRCONFIG` is specified, the function returns a `NULL` pointer; (b) if `AI_ADDRCONFIG | AI_V4MAPPED` is specified, the `A` records are returned as IPv4-mapped IPv6 addresses;

The special flags value of `AI_DEFAULT` is defined as

```
#define  AI_DEFAULT  (AI_V4MAPPED_CFG | AI_ADDRCONFIG)
```

Note that the **getipnodebyname()** function must allow the *name* argument to be either a node name or a literal address string (i.e. a dotted-decimal IPv4 address or an IPv6 hex address.) this saves applicats from having to call inet_pton(3) to handle literal address strings. When the *name* argument is a literal address strings, the *flags* argument is always ignored.

There are four scenarios based on the type of literal address string and the value of the *af* argument. The two simple cases are when name is a dotted-decimal IPv4 address and *af* equals `AF_INET`, or when name is an IPv6 hex address and *af* equals `AF_INET6`. The members of the returned `hostent` structure are: `h_name` points to a copy of the name argument, `h_aliases` is a `NULL` pointer, `h_addrtype` is a copy of the *af* argument, `h_length` is either 4 (for `AF_INET`) or 16 (for `AF_INET6`), `h_addr_list[0]` is a pointer to the 4-byte or 16-byte binary address, and `h_addr_list[1]` is a `NULL` pointer.

When *name* is a dotted-decimal IPv4 address and *af* equals `AF_INET6`, and `AI_V4MAPPED` is specified, an IPv4-mapped IPv6 address is returned: `h_name` points to an IPv6 hex address containing the IPv4-mapped IPv6 address, `h_aliases` is a `NULL` pointer, `h_addrtype` is `AF_INET6`, `h_length` is 16, `h_addr_list[0]` is a pointer to the 16-byte binary address, and `h_addr_list[1]` is a `NULL` pointer.

It is an error when *name* is an IPv6 hex address and *af* equals `AF_INET`. The function's return value is a `NULL` pointer and the value pointed to by *error_num* equals `HOST_NOT_FOUND`.

getipnodebyaddr() takes almost the same argument as gethostbyaddr(3), but adds a pointer to return an error number. Additionally it takes care of IPv4-mapped IPv6 addresses, and IPv4-compatible IPv6 addresses.

getipnodebyname() and **getipnodebyaddr()** dynamically allocate the structure to be returned to the caller. **freehostent()** reclaims the memory region allocated and returned by **getipnodebyname()** or **getipnodebyaddr()**.

DIAGNOSTICS

getipnodebyname() and **getipnodebyaddr()** return `NULL` on errors. The integer values pointed by *error_num* may then be checked to see if this is a temporary failure

or an invalid or unknown host. The meanings of each error code are described in gethostbyname(3).

SEE ALSO

gethostbyname(3), gethostbyaddr(3)

R. Gilligan, S. Thomson, J. Bound, and W. Stevens, "Basic Socket Interface Extensions for IPv6", RFC2553, March 1999.

STANDARDS

getipnodebyname() and **getipnodebyaddr()** are documented in "Basic Socket Interface Extensions for IPv6" (RFC2553).

ISSUES

getipnodebyname() and **getipnodebyaddr()** do not handle scoped IPv6 address properly. If you use these functions, your program will not be able to handle scoped IPv6 addresses. For IPv6 address manipulation, getaddrinfo(3) and getnameinfo(3) are recommended.

The current implementation is not thread-safe.

GETNAMEINFO(3)

NAME

getnameinfo – socket address structure to hostname and service name

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int
getnameinfo(const struct sockaddr *sa, socklen_t salen, char *host,
    size_t hostlen, char *serv, size_t servlen, int flags);
```

DESCRIPTION

The **getnameinfo()** function is used to convert a sockaddr structure to a pair of host name and service strings. It is a replacement for and provides more flexibility than the gethostbyaddr(3) and getservbyport(3) functions and is the converse of the getaddrinfo(3)) function.

The *sockaddr* structure *sa* should point to either a `sockaddr_in` or `sockaddr_in6` structure (for IPv4 or IPv6 respectively) that is *salen* bytes long.

The host and service names associated with sa are stored in *host* and *serv* which have length parameters *hostlen* and *servlen*. The maximum value for hostlen is `NI_MAXHOST` and the maximum value for servlen is `NI_MAXSERV`, as defined by `<netdb.h>`. If a length parameter is zero, no string will be stored. Otherwise, enough space must be provided to store the host name or service string plus a byte for the `NUL` terminator.

The flags argument is formed by OR'ing the following values:

`NI_NOFQDN`	A fully qualified domain name is not required for local hosts. The local part of the fully qualified domain name is returned instead.
`NI_NUMERICHOST`	Return the address in numeric form, as if calling inet_ntop(3)), instead of a host name.
`NI_NAMEREQD`	A name is required. If the host name cannot be found in DNS and this flag is set, a non-zero error code is returned. If the host name is not found and the flag is not set, the address is returned in numeric form.

NI_NUMERICSERV — The service name is returned as a digit string representing the port number.

NI_DGRAM — Specifies that the service being looked up is a datagram service, and causes getservbyport(3)) to be called with a second argument of "udp" instead of its default of "tcp". This is required for the few ports (512-514) that have different services for UDP and TCP.

This implementation allows numeric IPv6 address notation with scope identifier, as documented in chapter 11 of draft-ietf-ipv6-scoping-arch-02.txt. IPv6 link-local address will appear as a string like "fe80::1%ne0". Refer to getaddrinfo(3) for more information.

RETURN VALUES

getnameinfo() returns zero on success or one of the error codes listed in gai_strerror(3) if an error occurs.

EXAMPLES

The following code tries to get a numeric host name, and service name, for a given socket address. Observe that there is no hardcoded reference to a particular address family.

```
struct sockaddr *sa;    /* input */
char hbuf[NI_MAXHOST], sbuf[NI_MAXSERV];

if (getnameinfo(sa, sa->sa_len, hbuf, sizeof(hbuf), sbuf,
    sizeof(sbuf), NI_NUMERICHOST | NI_NUMERICSERV)) {
errx(1, "could not get numeric hostname");
/*NOTREACHED*/
}
printf("host=%s, serv=%s\n", hbuf, sbuf);
```

The following version checks if the socket address has a reverse address mapping:

```
struct sockaddr *sa;    /* input */
char hbuf[NI_MAXHOST];

if (getnameinfo(sa, sa->sa_len, hbuf, sizeof(hbuf), NULL, 0,
    NI_NAMEREQD)) {
errx(1, "could not resolve hostname");
```

```
/*NOTREACHED*/
}
printf("host=%s\n", hbuf);
```

SEE ALSO

gai_strerror(3), getaddrinfo(3), gethostbyaddr(3), getservbyport(3) inet_ntop(3), resolver(3)

R. Gilligan, S. Thomson, J. Bound, and W. Stevens, Basic Socket Interface Extensions for IPv6, RFC 2553, March 1999.

S. Deering, B. Haberman, T. Jinmei, E. Nordmark, and B. Zill, IPv6 Scoped Address Architecture, internet draft, draft-ietf-ipv6-scoping- arch-02.txt, work in progress material.

Craig Metz, "Protocol Independence Using the Sockets API", Proceedings of the freenix track: 2000 USENIX annual technical conference, June 2000.

STANDARDS

The **getnameinfo()** function is defined by the IEEE Std 1003.1g-2000 ("POSIX.1") draft specification and documented in RFC 2553, "Basic Socket Interface Extensions for IPv6".

CAVEATS

getnameinfo() can return both numeric and FQDN forms of the address specified in *sa*. There is no return value that indicates whether the string returned in *host* is a result of binary to numeric-text translation (like inet_ntop(3)), or is the result of a DNS reverse lookup. Because of this, malicious parties could set up a PTR record as follows:

```
1.0.0.127.in-addr.arpa. IN PTR  10.1.1.1
```

and trick the caller of **getnameinfo()** into believing that *sa* is 10.1.1.1 when it is actually 127.0.0.1.

To prevent such attacks, the use of `NI_NAMEREQD` is recommended when the result of **getnameinfo()** is used for access control purposes:

```
struct sockaddr *sa;
socklen_t salen;
```

```
char addr[NI_MAXHOST];
struct addrinfo hints, *res;
int error;

error = getnameinfo(sa, salen, addr, sizeof(addr),
    NULL, 0, NI_NAMEREQD);
if (error == 0) {
memset(&hints, 0, sizeof(hints));
hints.ai_socktype = SOCK_DGRAM; /*dummy*/
hints.ai_flags = AI_NUMERICHOST;
if (getaddrinfo(addr, "0", &hints, &res) == 0) {
/* malicious PTR record */
freeaddrinfo(res);
printf("bogus PTR record\n");
return -1;
}
/* addr is FQDN as a result of PTR lookup */
} else {
/* addr is numeric string */
error = getnameinfo(sa, salen, addr, sizeof(addr),
    NULL, 0, NI_NUMERICHOST);
}
```

GETNETENT(3)

NAME

getnetent, getnetbyaddr, getnetbyname, setnetent, endnetent - get network entry

SYNOPSIS

```
#include <netdb.h>
struct netent *
getnetent(void)

struct netent *
getnetbyname(const char *name)

struct netent *
getnetbyaddr(unsigned long net, int type)

void
setnetent(int stayopen)

void
endnetent(void)
```

DESCRIPTION

The **getnetent()**, **getnetbyname()**, and **getnetbyaddr()** functions each return a pointer to an object with the following structure, containing the broken-out fields of a line in the network data base.

```
struct netent {
   char           *n_name;     /* official    */
                               /* name of net */
   char           **n_aliases; /* alias list  */
   int            n_addrtype;  /* net         */
                               /* number type */
   unsigned long  n_net;       /* net number  */
};
```

The members of this structure are:

`n_name` The official name of the network.

`n_aliases`	A zero terminated list of alternate names for the network.
`n_addrtype`	The type of the network number returned; currently only `AF_INET`.
`n_net`	The network number. Network numbers are return in machine

The **getnetent()** function reads the next line of the file, opening the file if necessary.

The **setnetent()** function opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to **getnetbyname()** or **getnetbyaddr()**.

The **endnetent()** function closes the file.

The **getnetbyname()** function and **getnetbyaddr()** sequentially search from the beginning of the file until a matching net name or net address and type is found, or until `EOF` is encountered. The *type* must be `AF_INET`. Network numbers are supplied in host order.

FILES

These functions use the algorithm described in the IBM TCP/IP documentation to locate the `TCPIP.DATA`, and `ETC.NETWORKS` files.

DIAGNOSTICS

A `NULL` pointer (0) is returned on `EOF` or error.

SEE ALSO

RFC 1101

ISSUES

The data space used by these functions is static; if future use requires the data, it should be copied before any subsequent calls to these functions overwrite it. Only Internet network numbers are currently understood. Expecting network numbers to fit in no more than 32 bits is probably naive.

GETPROTOENT(3)

NAME

getprotoent, getprotobynumber, getprotobyname, setprotoent, endprotoent - get protocol entry.

SYNOPSIS

```
#include <netdb.h>
struct protoent *
getprotoent(void)

struct protent *
getprotobyname(const char *name)

struct protoent *
getprotobynumber(int proto)

void
setprotoent(int stayopen)

void
endprotoent(void)
```

DESCRIPTION

The **getprotoent()**, **getprotobyname()**, and **getprotobynumber()** functions each return a pointer to an object with the following structure containing the broken-out fields of a line in the network protocol data base.

```
struct protoent {
   char *p_name;      /* official name */
                      /*   of protocol */
   char **p_aliases;  /* alias list */
   int p_proto;      /* protocol number */
};
```

The members of this structure are:

`p_name`	The official name of the protocol.
`p_aliases`	A zero terminated list of alternate names for the protocol.

`p_proto` The protocol number.

The **getprotoent()** function reads the next line of the file, opening the file if necessary.

The **setprotoent()** function opens and rewinds the file. If the *stayopen* flag is nonzero, the net data base will not be closed after each call to **getprotobyname()** or **getprotobynumber()**.

The **endprotoen()** function closes the file.

The getprotobyname(0 function and getprotobynumber(0 function sequentially search from the beginning of the file until a matching protocol name or protocol number is found, or until `EOF` is encountered.

RETURN VALUES

The `NULL` pointer (0) is returned on `EOF` or error.

FILES

These functions use the algorithm described in the IBM TCP/IP documentation to locate the `TCPIP.DATA`, and `ETC.PROTOCOLS` files.

ISSUES

These functions use a static data space; if the data is needed for future use, it should be copied before any subsequent calls overwrite it. Only the Internet protocols are currently understood.

GETSERVENT(3)

NAME

getservent, getservbyport, getservbyname, setservent, endservent - get service entry.

SYNOPSIS

```
#include <netdb.h>
struct servent *
getservent()

struct servent *
getservbyname(const char *name, const char *proto)

struct servent *
getservbyport(int port, const char *proto)

void
setservent(int stayopen)

void
endservent(void)
```

DESCRIPTION

The **getservent()**, **getservbyname()**, and **getservbyport()** functions each return a pointer to an object with the following structure containing the broken-out fields of a line in the network services data base.

```
struct servent {
   char  *s_name;     /* official name */
                      /*    of service */
   char  **s_aliases; /* alias list */
   int   s_port;      /* port service */
                      /*   resides at */
   char  *s_proto;    /* protocol to use */
};
```

The members of this structure are:

`s_name` The official name of the service.

`s_aliases`	A zero terminated list of alternate names for the service.
`s_port`	The port number at which the service resides. Port numbers are returned in network byte order.
`s_proto`	The name of the protocol to use when contacting the service.

The **getservent()** function reads the next line of the file, opening the file if necessary.

The **setservent()** function opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to **getservbyname()** or **getservbyent()**.

The **endservent()** function closes the file.

The **getservbyname()** and **getservbyport()** functions sequentially search from the beginning of the file until a matching protocol name or port number is found, or until `EOF` is encountered. If a protocol name is also supplied (non-`NULL`), searches must also match the protocol.

FILES

These functions use the algorithm described in the IBM TCP/IP documentation to locate the `TCPIP.DATA`, and `ETC.SERVICES` files. If that file cannot be located, these functions will look for `//HFS:/etc/services`.

DIAGNOSTICS

The `NULL` pointer (0) is returned on `EOF` or error.

SEE ALSO

getprotoent(3)

ISSUES

These functions use static data storage; if the data is needed for future use, it should be copied before any subsequent calls overwrite it. Expecting port numbers to fit in a 32 bit quantity is probably naive.

INET(3)

NAME

inet_aton, inet_addr, inet_network, inet_ntoa, inet_makeaddr, inet_lnaof, inet_netof - Internet address manipulation routines.

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int
inet_aton(const char *cp, struct in_addr *pin)

unsigned long
inet_addr(const char *cp)

unsigned long
inet_network(const char *cp)

char *
inet_ntoa(struct in_addr in)

struct in_addr
inet_makeaddr(unsigned long net, unsigned long lna)

unsigned long
inet_lnaof(struct in_addr in)

unsigned long
inet_netof(struct in_addr in)
```

DESCRIPTION

The routines **inet_aton()**, **inet_addr()** and **inet_network()** interpret character strings representing numbers expressed in the Internet standard '.' notation. The **inet_aton()** routine interprets the specified character string as an Internet address, placing the address into the structure provided. It returns 1 if the string was successfully interpreted, or 0 if the string is invalid. The inet_addr(0 and inet_network(0 functions return numbers suitable for use an Internet addresses and Internet network numbers, respectively. The routine **inet_ntoa()** takes an Internet address and returns a string representing the address in '.' notation. The

routine **inet_makeaddr()** takes an Internet network number and a local network address and constructs an Internet address from it. The routines **inet_netof()** and **inet_lnaof()** break apart Internet host addresses, returning the network number and local network address part, respectively.

All Internet addresses are returned in network order (bytes ordered from left to right.) All network numbers and local address parts are returned as machine format integer values.

INTERNET ADDRESSES

Values specified using the '.' notation take one of the following forms:

a.b.c.d
a.b.c
a.b
a

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address. Note that when an Internet addresses is viewed as a 32-bit integer quantity appear in host byte order.

When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right-most two bytes of the network address. This makes the three part address format convenient for specifying Class B network addresses as "128.net.host".

When a two part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right most three bytes of the network address. This makes the two part address format convenient for specifying Class A network addresses as "net.host".

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as "parts" in a '.' notation may be decimal, octal, or hexadecimal, as specified in the C language (I.e., a leading 0x or 0X implies hexadecimal; otherwise, a leading 0 implies octal; otherwise the number is interpreted as decimal.)

The **inet_aton()** and **inet_ntoa()** functions are semi-deprecated in favor of the addr2ascii(3) family. However, since those functions are not yet widely implemented, portable programs cannot rely on their presence and will continue to use the inet(3) functions for some time.

DIAGNOSTICS

The constant `INADDR_NONE` is returned by **inet_addr()** and **inet_network()** for malformed requests.

SEE ALSO

addr2ascii(3), gethostbyname(3), getnetent(3)

ISSUES

The value `INADDR_NONE` (`0xffffffff`) is a valid broadcast address, but **inet_addr()** cannot return that value without indicating failure. The newer **inet_aton()** function does not share this problem. The problem of host byte ordering versus network byte ordering is confusing. The string returned by **inet_ntoa()** resides in a static memory area.

inet_addr() should return a `struct in_addr` .

NS(3)

NAME

ns_addr, ns_ntoa - Xerox NS address conversion routines

SYNOPSIS

```
#include <sys/types.h>
#include <nets/ns.h>

struct ns_addr
ns_addr(char *cp)

char *
ns_ntoa(struct ns_addr ns)
```

DESCRIPTION

The routine **ns_addr()** interprets character strings representing XNS addresses, returning binary information suitable for use in system calls. The routine **ns_ntoa()** takes XNS addresses and returns strings representing the address in a notation in common use in the Xerox Development Environment:

<*network number*>.<*host number*>.<*port number*>

Trailing zero fields are suppressed, and each number is printed in hexadecimal, in a format suitable for input to **ns_addr()**. Any fields lcaking super-decimal digits will have a trailing 'H' appended.

Unfortunately, no universal standard exists for representing XNS addresses. An effort has been made to insure that **ns_addr()** be compatible with most formats in common use. It will first separate an address into 1 to 3 fields using a single delimiter chosen from period '.' , colon ':' or pound-sign '#'. Each field is then examined for byte separators (colon or period.) If there are byte separators, each subfield separated is taken to be a small hexadecimal number, and the entirety is taken as a network-byte-ordered quantity to be zero extended in the high-network-order bytes. Next, the field is inspected for hyphens, in which case the field is assumed to be a number in decimal notation with hyphens separating the millennia. Next, the field is assumed to be a number: It is interpreted as hexadecimal if there is a leading '0x' (as in C), a trailing 'H' (as in Mesa), or there are any super-decimal digits present. It is interpreted as octal if there is a leading '0' and there are no super-octal digits. Otherwise, it is converted as a decimal number.

RETURN VALUES

None. (See ISSUES.)

ISSUES

The string returned by **ns_ntoa()** resides in a static memory area. The function **ns_addr()** should diagnose improperly formed input, and there should be an unambiguous way to recognize this.

RESOLVER(3)

NAME

res_query, res_search, res_mkquery, res_send, res_init, dn_comp, dn_expand - resolver routines

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int
res_query(const char *dname, int class, int type,
u_char *answer, int anslen)

int
res_search(const char *dname, int class, int type,
u_char *answer, int anslen)

int
res_mkquery(int op, const char *dname, int class,
int type, const u_char *data, int datalen,
const u_char *newer_in, u_char *buf,
int buflen)

int
res_send(const u_char *msg, int msglen, u_char *answer,
int anslen)

int
res_init()

dn_comp(const char *exp_dn, u_char *comp_dn, int length,
u_char **dnptrs, u_char **lastdnptr)

int
dn_expand(const u_char *msg, const u_char *eomorig,
const u_char *comp_dn, char *exp_dn,
int length)
```

DESCRIPTION

These routines are used for making, sending and interpreting query and reply messages with Internet domain name servers.

Global configuration and state information that is used by the resolver routines is kept in the structure `_res`. Most of the values have reasonable defaults and can be ignored. Options stored in `_res.otions` are defined in `resolv.h` and are as follows. Options are stored as a simple bit mask containing the bitwise 'or' of the options enabled.

`RES_INIT`	True if the initial name server address and default domain name are initialized (i.e., **res_init()** has been called.)
`RES_DEBUG`	Print debugging messages.
`RES_AAONLY`	Accept authoritative answers only. With this option, **res_send()** should continue until it finds an authoritative answer or finds an error. Currently this is not implemented.
`RES_USEVC`	Use TCP connections for queries instead of UDP datagrams.
`RES_STAYOPEN`	Used with `RES_USEVC` to keep the TCP connection open between queries. This is useful only in programs that regularly do many queries. UDP should be the normal mode used.
`RES_IGNTC`	Unused currently (ignore truncation errors, I.e., don't retry with TCP.)
`RES_RECURSE`	Set the recursion-desired bit in queries. This is the default. (**res_send()** does not do iterative queries and expects the name server to handle recursion.)
`RES_DEFNAMES`	If set, **res_search()** will append the default domain name to single-component names (those that do not contain a dot.) This option is enabled by default.
`RES_DNSRCH`	If this option is set, **res_search()** will search for host names in the current domain and in parent domains. This is used by the standard host lookup routine gethostbyname(3). This option is enabled by default.
`RES_NOALIASES`	This option turns off the user level aliasing feature controlled by the "`HOSTALIASES`" environment variable. Network demons should set this option.

The **res_init()** routine reads the configuration file to get the default domain name, search list and the Internet address of the local name server(s). If no server is configured the host running the resolver is tried. The current domain name is defined

by the hostname if not specified in the configuration file; it can be overridden by the environment variable `LOCALDOMAIN`. This environment variable may contain several blank-separated tokens if you with to override the search list on a per-process basis. This is similar to the search command in the configuration file. Another environment variable "`RES_OPTIONS`" can be set to override certain internal resolver options which are otherwise set by changing fields in the `_res` structure or are inherited from the configuration file. Initialization normally occurs on the first call to one of the following routines.

The **res_query()** function provides an interface to the server query mechanism. It constructs a query, sends it to the local server, awaits a response, and makes preliminary checks on the reply. The query requests information of the specified *type* and *class* for the specified fully-qualified domain name *dname*. The reply message is left in the *answer* buffer with length *anslen* supplied by the caller.

The **res_search()** routine makes a query and awaits a response like **res_query()**, but in addition, it implements the default and search rules controlled by the `RES_DEFNAMES` and `RES_DNSRCH` options. It returns the first successful reply.

The remaining routines are lower-level routines used by **res_query()**. The **res_mkquery()** function constructs a standard query message and places it in *buf*. It returns the size of the query, or `-1` if the query is larger than *buflen*. The query type op is usually `QUERY`, but can be any of the query types defined in `<arpa/nameserv.h>`. The domain name for the query is given by *dname*. *Newer_in* is currently unused but is intended for making update messages.

The **res_send()** routine sends a pre-formatted query and returns an answer. It will call **res_init()** if `RES_INIT` is not set, send the query to the local name server, and handle timeouts and retries. The length of the reply message is returned, or `-1` if there were errors.

The **dn_comp()** function compresses the domain name *exp_dn* and stores it in *comp_dn*. The size of the compressed name is returned or -1 if there were errors. The size of the array pointed to be *comp_dn* is given by *length* The compression uses an array of pointers *dnptrs* to previously-compressed names in the current message. The first pointer points to the beginning of the message and the list ends with `NULL`. The limit to the array is specified by *lastdnptr*. A side effect of **dn_comp()** is to update the list of pointers for labels inserted into the message as the name is compressed. If *dnptr* is `NULL`, the list of labels is not updated.

The **dn_expand()** entry expands the compressed domain name *comp_dn* to a full domain name. The compressed name is contained in a query or reply message; *msg* is a pointer to the beginning of the message. The uncompressed name is placed in the buffer indicated by *exp_dn* which is of size *length*. The size of the compressed name is returned or `-1` if there was an error.

FILES

These functions use the algorithm described in the IBM TCP/IP documentation to locate the `TCPIP.DATA`, and `ETC.RESOLV.CONF` files.

SEE ALSO

gethostbyname(3)

RFC1032, RFC1033, RFC1034, RFC1035, RFC974

Thread Library

PTHREAD(3)

NAME

pthread – POSIX thread functions

SYNOPSIS

```
#include <pthread.h>
```

DESCRIPTION

POSIX threads are a set of functions that support applications with requirements for multiple flows of control, called threads, within a process. Multithreading is used to improve the performance of a program.

The POSIX thread functions are summarized in this section in the following groups:

- Thread Routines
- Attribute Object Routines
- Mutex Routines
- Condition Variable Routines
- Read/Write Lock Routines
- Per-Thread Context Routines
- Cleanup Routines

Thread Routines

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
            void *(*start_routine)(void *), void *arg)
```

Creates a new thread of execution.

```
int pthread_cancel(pthread_t thread)
```

Cancels execution of a thread.

```
int pthread_detach(pthread_t thread)
```

Marks a thread for deletion.

```
int pthread_equal(pthread_t t1, pthread_t t2)
```

Compares two thread IDs.

```
void pthread_exit(void *value_ptr)
```

Terminates the calling thread.

```
int pthread_join(pthread_t thread, void **value_ptr)
```

Causes the calling thread to wait for the termination of the specified thread.

```
int pthread_kill(pthread_t thread, int sig)
```

Delivers a signal to a specified thread.

```
int pthread_main_np(void)
```

Determines if the thread is the initial thread.

```
int pthread_once(pthread_once_t *once_control, void (*init_routine)(void))
```

Calls an initialization routine once.

```
pthread_t pthread_self(void)
```

Returns the thread ID of the calling thread.

```
int pthread_setcancelstate(int state, int *oldstate)
```

Sets the current thread's cancelability state.

```
int pthread_setcanceltype(int type, int *oldtype)
```

Sets the current thread's cancelability type.

```
int pthread_set_limit_np(int action, int maxtasks, int maxthreads)
```

Sets the z/OS maximum number of tasks allowed and/or maximum number of threads for the process.

```
void pthread_testcancel(void)
```

Creates a cancellation point in the calling thread.

```
void pthread_yield(void)
```

Allows the scheduler to run another thread instead of the current one.

Attribute Object Routines

```
int pthread_attr_destroy(pthread_attr_t *attr)
```

Destroy a thread attributes object.

```
int pthread_attr_getinheritsched(const pthread_attr_t *attr,
            int *inheritsched)
```

Get the inherit scheduling attribute from a thread attributes object.

```
int pthread_attr_getschedparam(const pthread_attr_t *attr,
            struct sched_param *param)
```

Get the scheduling parameter attribute from a thread attributes object.

```
int pthread_attr_getschedpolicy(const pthread_attr_t *attr, int *policy)
```

Get the scheduling policy attribute from a thread attributes object.

```
int pthread_attr_getscope(const pthread_attr_t *attr, int
            *contentionscope)
```

Get the contention scope attribute from a thread attributes object.

```
int pthread_attr_getstacksize(const pthread_attr_t *attr, size_t
            *stacksize)
```

Get the stack size attribute from a thread attributes object.

```
int pthread_attr_getstackaddr(const pthread_attr_t *attr, void
            **stackaddr)
```

Get the stack address attribute from a thread attributes object.

```
int pthread_attr_getdetachstate(const pthread_attr_t *attr, int
            *detachstate)
```

Get the detach state attribute from a thread attributes object.

```
int pthread_attr_getweight_np(const pthread_attr_t *attr,
              int *weight);
```

Get the thread weight.

```
int pthread_attr_getsynctype_np(const pthread_attr_t *attr, int
                                *synctype);
```

Get the thread sync type.

```
int pthread_attr_init(pthread_attr_t *attr)
```

Initialize a thread attributes object with default values.

```
int pthread_attr_setinheritsched(pthread_attr_t *attr,
                  int inheritsched)
```

Set the inherit scheduling attribute in a thread attributes object.

```
int pthread_attr_setschedparam(pthread_attr_t *attr,
            const struct sched_param *param)
```

Set the scheduling parameter attribute in a thread attributes object.

```
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy)
```

Set the scheduling policy attribute in a thread attributes object.

```
int pthread_attr_setscope(pthread_attr_t *attr,
                          int contentionscope)
```

Set the contention scope attribute in a thread attributes object.

```
int pthread_attr_setstacksize(pthread_attr_t *attr,
                              size_t stacksize)
```

Set the stack size attribute in a thread attributes object.

```
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr)
```

Set the stack address attribute in a thread attributes object.

```
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate)
```

Set the detach state in a thread attributes object.

```
int pthread_attr_setweight_np(pthread_attr_t *attr, int weight)
```

Set the weight in a thread attributes object.

```
int pthread_attr_setsynctype_np(pthread_attr_t *attr, int type)
```

Set the synctype in a thread attributes object.

Mutex Routines

```
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr)
```

Destroy a mutex attributes object.

```
int pthread_mutexattr_getprioceiling(pthread_mutexattr_t *attr, int
            *ceiling)
```

Obtain priority ceiling attribute of mutex attribute object.

```
int pthread_mutexattr_getprotocol(pthread_mutexattr_t *attr, int
            *protocol)
```

Obtain protocol attribute of mutex attribute object.

```
int pthread_mutexattr_gettype(pthread_mutexattr_t *attr, int *type)
```

Obtain the mutex type attribute in the specified mutex attributes object.

```
int pthread_mutexattr_init(pthread_mutexattr_t *attr)
```

Initialize a mutex attributes object with default values.

```
int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr, int
            ceiling)
```

Set priority ceiling attribute of mutex attribute object.

```
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int
            protocol)
```

Set protocol attribute of mutex attribute object.

```
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type)
```

Set the mutex type attribute that is used when a mutex is created.

```
int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

Destroy a mutex.

```
int pthread_mutex_init(pthread_mutex_t *mutex,
            const pthread_mutexattr_t *attr)
```

Initialize a mutex with specified attributes.

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
```

Lock a mutex and block until it becomes available.

```
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
            const struct timespec *abstime)
```

Lock a mutex and block until it becomes available or until the timeout expires.

```
int pthread_mutex_trylock(pthread_mutex_t *mutex)
```

Try to lock a mutex, but do not block if the mutex is locked by another thread, including the current thread.

```
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

Unlock a mutex.

Condition Variable Routines

```
int pthread_condattr_destroy(pthread_condattr_t *attr)
```

Destroy a condition variable attributes object.

```
int pthread_condattr_init(pthread_condattr_t *attr)
```

Initialize a condition variable attributes object with default values.

```
int pthread_cond_broadcast(pthread_cond_t *cond)
```

Unblock all threads currently blocked on the specified condition variable.

```
int pthread_cond_destroy(pthread_cond_t *cond)
```

Destroy a condition variable.

```
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t
            *attr)
```

Initialize a condition variable with specified attributes.

```
int pthread_cond_signal(pthread_cond_t *cond)
```

Unblock at least one of the threads blocked on the specified condition variable.

```
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,
            const struct timespec *abstime)
```

Wait no longer than the specified time for a condition and lock the specified mutex.

```
int pthread_cond_wait(pthread_cond_t *, pthread_mutex_t *mutex)
```

Wait for a condition and lock the specified mutex.

Read/Write Lock Routines

```
int pthread_rwlock_destroy(pthread_rwlock_t *lock)
```

Destroy a read/write lock object.

```
int pthread_rwlock_init(pthread_rwlock_t *lock,
            const pthread_rwlockattr_t *attr)
```

Initialize a read/write lock object.

```
int pthread_rwlock_rdlock(pthread_rwlock_t *lock)
```

Lock a read/write lock for reading, blocking until the lock can be acquired.

```
int pthread_rwlock_tryrdlock(pthread_rwlock_t *lock)
```

Attempt to lock a read/write lock for reading, without blocking if the lock is unavailable.

```
int pthread_rwlock_trywrlock(pthread_rwlock_t *lock)
```

Attempt to lock a read/write lock for writing, without blocking if the lock is unavailable.

```
int pthread_rwlock_unlock(pthread_rwlock_t *lock)
```

Unlock a read/write lock.

```
int pthread_rwlock_wrlock(pthread_rwlock_t *lock)
```

Lock a read/write lock for writing, blocking until the lock can be acquired.

```
int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr)
```

Destroy a read/write lock attribute object.

```
int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *attr,
            int *pshared)
```

Retrieve the process shared setting for the read/write lock attribute object.

```
int pthread_rwlockattr_init(pthread_rwlockattr_t *attr)
```

Initialize a read/write lock attribute object.

```
int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,
            int pshared)
```

Set the process shared setting for the read/write lock attribute object.

Per-Thread Context Routines

```
int pthread_key_create(pthread_key_t *key, void (*routine)(void *))
```

Create a thread-specific data key.

```
int pthread_key_delete(pthread_key_t key)
```

Delete a thread-specific data key.

```
void * pthread_getspecific(pthread_key_t key)
```

Get the thread-specific value for the specified key.

```
int pthread_setspecific(pthread_key_t key, const void *value_ptr)
```

Set the thread-specific value for the specified key.

Cleanup Routines

```
int pthread_atfork(void (*prepare)(void), void (*parent)(void),
            void (*child)(void))
```

Register fork handlers

```
void pthread_cleanup_pop(int execute)
```

Remove the routine at the top of the calling thread's cancellation cleanup stack and optionally invoke it.

```
void pthread_cleanup_push(void (*routine)(void *), void *routine_arg)
```

Push the specified cancellation cleanup handler onto the calling thread's cancellation stack.

IMPLEMENTATION

The Dignus POSIX thread implementation depends on z/OS UNIX Systems Services. Linking a program with the pthread_create(3) forces the use of UNIX Systems Services signal handling. Also, such a program ends with a call to the BPX _exit(2) function and not by returning to the caller.

The implementation defaults to "heavy" weight threads, where a single z/OS TASK per thread is used. The weight of a thread can be set before its creation using **pthread_attr_setweight_np()**. If the thread is set to "medium" then a task can execute the next thread without ending.

IBM recommends that only about 200 tasks be used per process. The **pthread_set_limit_np()** function can be used to indicate how many tasks are allowed per process. If too many tasks are initiated, there can problems with low-memory usage in the LQSA area, which can cause S422 ABENDs or other MVS internal errors. If the operating system reports the error, the Dignus runtime will return an error code of `EMVSERR`, otherwise the task or program may simply be terminated, with appropriate messages on the console log. This situation should be reported to the system managers and or IBM, as it may indicate a problem with z/OS internal functions.

The z/OS object file format and Dignus runtime does not support Thread Local Storage (TLS).

SEE ALSO

pthread_atfork(3), pthread_cleanup_pop(3), pthread_cleanup_push(3), pthread_condattr_destroy(3), pthread_condattr_init(3), pthread_cond_broadcast(3), pthread_cond_destroy(3), pthread_cond_init(3), pthread_cond_signal(3), pthread_cond_timedwait(3), pthread_cond_wait(3), pthread_create(3), pthread_detach(3), pthread_equal(3), pthread_exit(3), pthread_getspecific(3), pthread_join(3), pthread_key_delete(3), pthread_kill(3), pthread_mutexattr_destroy(3), pthread_mutexattr_getprioceiling(3), pthread_mutexattr_getprotocol(3), pthread_mutexattr_gettype(3), pthread_mutexattr_init(3), pthread_mutexattr_setprioceiling(3), pthread_mutexattr_setprotocol(3), pthread_mutexattr_settype(3), pthread_mutex_destroy(3), pthread_mutex_init(3), pthread_mutex_lock(3), pthread_mutex_trylock(3), pthread_mutex_unlock(3), pthread_once(3), pthread_rwlockattr_destroy(3), pthread_rwlockattr_getpshared(3), pthread_rwlockattr_init(3), pthread_rwlock_unlock(3), pthread_rwlock_wrlock(3), pthread_self(3), pthread_setcancelstate(3), pthread_setcanceltype(3), pthread_setspecific(3), pthread_testcancel(3) pthread_set_limit_np(3).

STANDARDS

The functions with the `pthread_` prefix and not `_np` suffix or `pthread_rwlock` prefix conform to ISO/IEC 9945-1:1996 ("POSIX.1").

The functions with the `pthread_` prefix and `_np` suffix are non-portable extensions to POSIX threads.

The functions with the `pthread_rwlock` prefix are extensions created by The Open Group as part of the Version 2 of the Single UNIX Specification ("SUSv2").

PTHREAD_ATFORK(3)

NAME

pthread_atfork – register fork handlers

SYNOPSIS

```
#include <pthread.h>

int
pthread_atfork(void (*prepare)(void), void (*parent)(void),
               void (*child)(void));
```

DESCRIPTION

The **pthread_atfork()** function declares handler functions to be called before and after fork(2), in the context of the thread that called fork(2).

The handler functions registered with **pthread_atfork()** are called at the moments and contexts described below:

prepare — Before fork(2) processing commences in the parent process. If more than one prepare handler is registered they will be called in the opposite order they were registered.

parent — After fork(2) completes in the parent process. If more than one parent handler is registered they will be called in the same order they were registered.

child — After fork(2) processing completes in the child process. If more than one child handler is registered they will be called in the same order they were registered.

If no handling is desired at one or more of these three points, a `NULL` pointer may be passed as the corresponding fork handler function address.

RETURN VALUES

If successful, the **pthread_atfork()** function will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_atfork()** function will fail if:

[ENOMEM] Insufficient memory space exists to record the fork handler address(es).

SEE ALSO

fork(2), pthread(3)

STANDARDS

The **pthread_atfork()** function is expected to conform to IEEE Std 1003.1 ("POSIX.1").

PTHREAD_ATTR(3)

NAME

pthread_attr_init, pthread_attr_destroy, pthread_attr_setstack, pthread_attr_getstack, pthread_attr_setstacksize, pthread_attr_getstacksize, pthread_attr_setguardsize, pthread_attr_getguardsize, pthread_attr_setstackaddr, pthread_attr_getstackaddr, pthread_attr_setdetachstate, pthread_attr_getdetachstate, pthread_attr_setinheritsched, pthread_attr_getinheritsched, pthread_attr_setschedparam, pthread_attr_getschedparam, pthread_attr_setschedpolicy, pthread_attr_getschedpolicy, pthread_attr_setscope, pthread_attr_getscope, pthread_attr_getsynctype_np, pthread_attr_setsynctype_np, pthread_attr_getweight_np, pthread_attr_setweight_np - thread attribute operations

SYNOPSIS

```
#include <pthread.h>

int
pthread_attr_init(pthread_attr_t *attr);

int
pthread_attr_destroy(pthread_attr_t *attr);

int
pthread_attr_setstack(pthread_attr_t *attr,  void *stackaddr,
         size_t stacksize);

int
pthread_attr_getstack(const pthread_attr_t * restrict attr,
         void ** restrict stackaddr, size_t * restrict stacksize);

int
pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);

int
pthread_attr_getstacksize(const pthread_attr_t *attr, size_t *stacksize);

int
pthread_attr_setguardsize(pthread_attr_t *attr, size_t guardsize);

int
pthread_attr_getguardsize(const pthread_attr_t *attr, size_t *guardsize);
```

```
int
pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);

int
pthread_attr_getstackaddr(const pthread_attr_t *attr, void **stackaddr);

int
pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);

int
pthread_attr_getdetachstate(const pthread_attr_t *attr,
         int *detachstate);

int
pthread_attr_setinheritsched(pthread_attr_t *attr, int inheritsched);

int
pthread_attr_getinheritsched(const pthread_attr_t *attr,
         int *inheritsched);

int
pthread_attr_setschedparam(pthread_attr_t *attr,
         const struct sched_param *param);

int
pthread_attr_getschedparam(const pthread_attr_t *attr,
         struct sched_param *param);

int
pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);

int
pthread_attr_getschedpolicy(const pthread_attr_t *attr, int *policy);

int
pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);

int
pthread_attr_getscope(const pthread_attr_t *attr, int *contentionscope);

int
pthread_attr_setweight_np(pthead_attr_t *attr, int weight);

int
pthead_attr_getweight_np(pthread_attr_t *attr, int *weight);

int
```

```
pthread_attr_setsynctype_np(pthead_attr_t *attr, int synctype);

int
pthead_attr_getsynctype_np(pthread_attr_t *attr, int *synctype);
```

DESCRIPTION

Thread attributes are used to specify parameters to **pthread_create()**. One attribute object can be used in multiple calls to **pthread_create()**, with or without modifications between calls.

The **pthread_attr_init()** function initializes *attr* with all the default thread attributes.

The **pthread_attr_destroy()** function destroys *attr*.

The **pthread_attr_set*()** functions set the attribute that corresponds to each function name.

The **pthread_attr_get*()** functions copy the value of the attribute that corresponds to each function name to the location pointed to by the second function parameter.

The **pthread_attr_setweight_np()** and **pthread_attr_getweight_np()** control the weight of the thread about to created. The weight is either `__MEDIUM_WEIGHT` or `__HEAVY_WEIGHT`. A `__MEDIUM_WEIGHT` indicates that a thread can be executed on the same task as a previous thread, which can alleviate task creation time. A `__HEAVY_WEIGHT` thread executes on its own task and when the thread finishes, the task ends. In the Dignus implementation threads are `__HEAVY_WEIGHT` by default.

The **pthread_attr_setsynctype_np()** and **pthread_attr_getsynctype_np()** control the synctype of the thread to be created. The synctype is one of:

`__PTATSYNCHRONOUS` A thread can only be created if a task is available (or the maximum number of threads, if smaller.)

`__PTATASYNCHRONOUS` Threads can be created up to the maximum number of threads.

`__PTATASYNCHRONOUS` threads will be created up to the maximum number of threads. If there are no available tasks, the thread will be queued for execution when another thread ends and a task becomes available. Queued threads can be affected by other pthread functions, but aren't executing until a task becomes available; which complicates thread coordination. The default value for the Dignus runtime is `__PTATSYNCHRONOUS`. The pthread_set_limit_np(3) function can be used to control the number of tasks and threads for the process.

RETURN VALUES

If successful, these functions return 0. Otherwise, an error number is returned to indicate the error.

ERRORS

The **pthread_attr_init()** function will fail if:

`[ENOMEM]` Out of memory.

The **pthread_attr_destroy()** function will fail if:

`[EINVAL]` Invalid value for *attr*.

The **pthread_attr_setstacksize()** and **pthread_attr_setstack()** functions will fail if:

`[EINVAL]` *stacksize* is less than `PTHREAD_STACK_MIN`.

The **pthread_attr_setdetachstate()** function will fail if:

`[EINVAL]` Invalid value for *detachstate*.

The **pthread_attr_setinheritsched()** function will fail if:

`[EINVAL]` Invalid value for *attr*.

The **pthread_attr_setschedparam()** function will fail if:

`[EINVAL]` Invalid value for *attr*.

`[ENOTSUP]` Invalid value for *param*.

The **pthread_attr_setschedpolicy()** function will fail if:

`[EINVAL]` Invalid value for *attr*.

`[ENOTSUP]` Invalid value for *policy*.

The **pthread_attr_setscope()** function will fail if:

[EINVAL] Invalid value for *attr*.

[ENOTSUP] Invalid or unsupported value for *contentionscope*.

The **pthread_attr_setweight_np()** function will fail if:

[EINVAL] Invalid value for *attr*.

The **pthread_attr_setsynctype_np()** function will fail if:

[EINVAL] Invalid value for *attr*.

SEE ALSO

pthread_create(3), pthread_set_limit_np(3).

IMPLEMENTATION

This implementation does not support the process-shared attribute, nor does it support scheduling scope or scheduling attributes.

STANDARDS

pthread_attr_init(), **pthread_attr_destroy()**, **pthread_attr_setstacksize()**, **pthread_attr_getstacksize()**, **pthread_attr_setstackaddr()**, **pthread_attr_getstackaddr()**, **pthread_attr_setdetachstate()**, and **pthread_attr_getdetachstate()** functions conform to ISO/IEC 9945-1:1996 ("POSIX.1")

The **pthread_attr_setinheritsched()**, **pthread_attr_getinheritsched()**, **pthread_attr_setschedparam()**, **pthread_attr_getschedparam()**, **pthread_attr_setschedpolicy()**, **pthread_attr_getschedpolicy()**, **pthread_attr_setscope()**, and **pthread_attr_getscope()** functions conform to Version 2 of the Single UNIX Specification ("SUSv2")

PTHREAD_BARRIER(3)

NAME

pthread_barrier_destroy, pthread_barrier_init, pthread_barrier_wait - destroy, initialize or wait on a barrier object

SYNOPSIS

```
#include <pthread.h>

int
pthread_barrier_destroy(pthread_barrier_t *barrier);

int
pthread_barrier_init(pthread_barrier_t *barrier,
            const pthread_barrierattr_t *attr, unsigned count);

int
pthread_barrier_wait(pthread_barrier_t *barrier);
```

DESCRIPTION

The **pthread_barrier_init()** function will initialize barrier with attributes specified in *attr*, or if it is `NULL`, with default attributes. The number of threads that must call **pthread_barrier_wait()** before any of the waiting threads can be released is specified by *count*. The **pthread_barrier_destroy()** function will destroy *barrier* and release any resources that may have been allocated on its behalf.

The **pthread_barrier_wait()** function will synchronize calling threads at *barrier*. The threads will be blocked from making further progress until a sufficient number of threads calls this function. The number of threads that must call it before any of them will be released is determined by the *count* argument to **pthread_barrier_init()**. Once the threads have been released the barrier will be reset.

RETURN VALUES

If successful, both **pthread_barrier_destroy()** and **pthread_barrier_init()** will return zero. Otherwise, an error number will be returned to indicate the error. If the call to **pthread_barrier_wait()** is successful, all but one of the threads will return zero. That one thread will return `PTHREAD_BARRIER_SERIAL_THREAD`. Otherwise, an error number will be returned to indicate the error.

None of these functions will return `EINTR`.

ERRORS

The **pthread_barrier_destroy()** function will fail if:

`[EBUSY]` An attempt was made to destroy *barrier* while it was in use.

The **pthread_barrier_destroy()** and **pthread_barrier_wait()** functions may fail if:

`[EINVAL]` The value specified by *barrier* is invalid.

The **pthread_barrier_init()** function will fail if:

`[EAGAIN]` The system lacks resources, other than memory, to initialize *barrier*.

`[EINVAL]` The *count* argument is less than 1.

`[ENOMEM]` Insufficient memory to initialize *barrier*.

SEE ALSO

pthread_barrierattr(3)

PTHREAD_BARRIERATTR(3)

NAME

pthread_barrierattr_destroy, pthread_barrierattr_getpshared, pthread_barrierattr_init, pthread_barrierattr_setpshared - manipulate a barrier attribute object

SYNOPSIS

```
#include <pthread.h>

int
pthread_barrierattr_destroy(pthread_barrierattr_t *attr);

int
pthread_barrierattr_getpshared(const pthread_barrierattr_t *attr,
         int *pshared);

int
pthread_barrierattr_init(pthread_barrierattr_t *attr);

int
pthread_barrierattr_setpshared(pthread_barrierattr_t *attr, int pshared);
```

DESCRIPTION

The **pthread_barrierattr_init()** function will initialize *attr* with default attributes. The **pthread_barrierattr_destroy()** function will destroy *attr* and release any resources that may have been allocated on its behalf.

The **pthread_barrierattr_getpshared()** function will put the value of the process-shared attribute from *attr* into the memory area pointed to by *pshared*. The **pthread_barrierattr_setpshared()** function will set the process-shared attribute of *attr* to the value specified in *pshared*. The argument *pshared* may have one of the following values:

`[PTHREAD_PROCESS_PRIVATE]` The barrier object it is attached to may only be accessed by threads in the same process as the one that created the object.

`[PTHREAD_PROCESS_SHARED]` The barrier object it is attached to may be accessed by threads in processes other than the one that created the object.

RETURN VALUES

If successful, all these functions will return zero. Otherwise, an error number will be returned to indicate the error.

None of these functions will return `EINTR`.

ERRORS

The **pthread_barrierattr_destroy()**, **pthread_barrierattr_getpshared()** and **pthread_barrierattr_setpshared()** functions may fail if:

`[EINVAL]` The value specified by *attr* is invalid.

The **pthread_barrierattr_init()** function will fail if:

`[ENOMEM]` Insufficient memory to initialize the barrier attribute object *attr*.

The **pthread_barrierattr_setpshared()** function will fail if:

`[EINVAL]` The value specified in *pshared* is not one of the allowed values.

SEE ALSO

pthread_barrier_destroy(3), pthread_barrier_init(3), pthread_barrier_wait(3)

IMPLEMENTATION

This implementation does not support process-shared barriers.

PTHREAD_CANCEL(3)

NAME

pthread_cancel – cancel execution of a thread

SYNOPSIS

```
#include <pthread.h>

int
pthread_cancel(pthread_t thread);
```

DESCRIPTION

The **pthread_cancel()** function requests that thread be canceled. The target thread's cancelability state and type determines when the cancellation takes effect. When the cancellation is acted on, the cancellation cleanup handlers for thread are called. When the last cancellation cleanup handler returns, the thread-specific data destructor functions will be called for thread. When the last destructor function returns, thread will be terminated.

The cancellation processing in the target thread runs asynchronously with respect to the calling thread returning from **pthread_cancel()**.

A status of `PTHREAD_CANCELED` is made available to any threads joining with the target. The symbolic constant `PTHREAD_CANCELED` expands to a constant expression of type `(void *)`, whose value matches no pointer to an object in memory nor the value `NULL`.

RETURN VALUES

If successful, the **pthread_cancel()** functions will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_cancel()** function will fail if:

`[ESRCH]` No thread could be found corresponding to that specified by the given thread ID.

SEE ALSO

pthread_cleanup_pop(3), pthread_cleanup_push(3), pthread_exit(3), pthread_join(3), pthread_setcancelstate(3), pthread_setcanceltype(3), pthread_testcancel(3)

STANDARDS

The **pthread_cancel()** function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

PTHREAD_CLEANUP_POP(3)

NAME

pthread_cleanup_pop – call the first cleanup routine

SYNOPSIS

```
#include <pthread.h>

void
pthread_cleanup_pop(int execute);
```

DESCRIPTION

The **pthread_cleanup_pop()** function pops the top cleanup routine off of the current threads cleanup routine stack, and, if execute is non-zero, it will execute the function. If there is no cleanup routine then **pthread_cleanup_pop()** does nothing.

The **pthread_cleanup_pop()** function is required to be in the lexical scope as its corresponding **pthread_cleanup_push()**.

RETURN VALUES

The **pthread_cleanup_pop()** function does not return any value.

ERRORS

None

SEE ALSO

pthread_cleanup_push(3), pthread_exit(3)

STANDARDS

The **pthread_cleanup_pop()** function conforms to ISO/IEC 9945-1:1996 (“POSIX.1”).

PTHREAD_CLEANUP_PUSH(3)

NAME

pthread_cleanup_push – add a cleanup function for thread exit

SYNOPSIS

```
#include <pthread.h>

void
pthread_cleanup_push(void (*cleanup_routine)(void *), void *arg);
```

DESCRIPTION

The **pthread_cleanup_push()** function adds *cleanup_routine* to the top of the stack of cleanup handlers that get called when the current thread exits.

When *cleanup_routine* is called, it is passed *arg* as its only argument.

The **pthread_cleanup_push()** function is required to be used in the same lexical scope as its corresponding **pthread_cleanup_pop()**.

RETURN VALUES

The **pthread_cleanup_push()** function does not return any value.

ERRORS

None

SEE ALSO

pthread_cleanup_pop(3), pthread_exit(3)

STANDARDS

The **pthread_cleanup_push()** function conforms to ISO/IEC 9945-1:1996 (“POSIX.1”).

PTHREAD_CONDATTR(3)

NAME

pthread_condattr_init, pthread_condattr_destroy, pthread_condattr_getclock, pthread_condattr_setclock, pthread_condattr_getpshared, pthread_condattr_setpshared – condition attribute operations

SYNOPSIS

```
#include <pthread.h>

int
pthread_condattr_init(pthread_condattr_t *attr);

int
pthread_condattr_destroy(pthread_condattr_t *attr);

int
pthread_condattr_getclock(pthread_condattr_t * restrict attr,
         clock_t * restrict clock_id);

int
pthread_condattr_setclock(pthread_condattr_t *attr, clock_t clock_id);

int
pthread_condattr_getpshared(pthread_condattr_t * restrict attr,
         int * restrict pshared);

int
pthread_condattr_setpshared(pthread_condattr_t *attr, int pshared);
```

DESCRIPTION

Condition attribute objects are used to specify parameters to **pthread_cond_init()**.

The **pthread_condattr_init()** function initializes a condition attribute object with the default attributes.

The **pthread_condattr_destroy()** function destroys a condition attribute object.

The **pthread_condattr_getclock()** function will put the value of the clock attribute from *attr* into the memory area pointed to by *clock_id*. The **pthread_condattr_setclock()** function will set the clock attribute of *attr* to the

value specified in *clock_id*. The clock attribute affects the interpretation of *abstime* in pthread_cond_timedwait(3) and may be set to `CLOCK_REALTIME` (default) or `CLOCK_MONOTONIC`.

The **pthread_condattr_getpshared()** function will put the value of the process-shared attribute from attr into the memory area pointed to by *pshared*. The **pthread_condattr_setpshared()** function will set the process-shared attribute of *attr* to the value specified in *pshared*. The argument *pshared* may have one of the following values:

`PTHREAD_PROCESS_PRIVATE` The condition variable it is attached to may only be accessed by threads in the same process as the one that created the object.

`PTHREAD_PROCESS_SHARED` The condition variable it is attached to may be accessed by threads in processes other than the one that created the object.

This implementation does not support `PTHREAD_PROCESS_SHARED`.

RETURN VALUES

If successful, these functions return 0. Otherwise, an error number is returned to indicate the error.

ERRORS

The **pthread_condattr_init()** function will fail if:

`[ENOMEM]` Out of memory.

The **pthread_condattr_destroy()** function will fail if:

`[EINVAL]` Invalid value for *attr*.

The **pthread_condattr_setclock()** function will fail if:

`[EINVAL]` The value specified in *clock_id* is not one of the allowed values.

The **pthread_condattr_setpshared()** function will fail if:

`[EINVAL]` The value specified in *pshared* is not one of the allowed values.

SEE ALSO

pthread_cond_init(3), pthread_cond_timedwait(3)

STANDARDS

The **pthread_condattr_init()** and **pthread_condattr_destroy()** functions conform to ISO/IEC 9945-1:1996 ("POSIX.1")

PTHREAD_COND_BROADCAST(3)

NAME

pthread_cond_broadcast – unblock all threads waiting for a condition variable

SYNOPSIS

```
#include <pthread.h>

int
pthread_cond_broadcast(pthread_cond_t *cond);
```

DESCRIPTION

The **pthread_cond_broadcast()** function unblocks all threads waiting for the condition variable *cond*.

RETURN VALUES

If successful, the **pthread_cond_broadcast()** function will return zero, otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_cond_broadcast()** function will fail if:

`[EINVAL]` The value specified by *cond* is invalid.

SEE ALSO

pthread_cond_destroy(3), pthread_cond_init(3), pthread_cond_signal(3), pthread_cond_timedwait(3), pthread_cond_wait(3)

STANDARDS

The **pthread_cond_broadcast()** function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

PTHREAD_COND_DESTROY(3)

NAME

pthread_cond_destroy – destroy a condition variable

SYNOPSIS

```
#include <pthread.h>

int
pthread_cond_destroy(pthread_cond_t *cond);
```

DESCRIPTION

The **pthread_cond_destroy()** function frees the resources allocated by the condition variable *cond*.

RETURN VALUES

If successful, the **pthread_cond_destroy()** function will return zero, otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_cond_destroy()** function will fail if:

`[EINVAL]` The value specified by *cond* is invalid.

`[EBUSY]` The variable *cond* is locked by another thread.

SEE ALSO

pthread_cond_broadcast(3), pthread_cond_init(3), pthread_cond_signal(3), pthread_cond_timedwait(3), pthread_cond_wait(3)

STANDARDS

The **pthread_cond_destroy()** function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

PTHREAD_COND_INIT(3)

NAME

pthread_cond_init – create a condition variable

SYNOPSIS

```
#include <pthread.h>

int
pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
```

DESCRIPTION

The **pthread_cond_init()** function creates a new condition variable, with attributes specified with *attr*. If *attr* is `NULL` the default attributes are used.

RETURN VALUES

If successful, the **pthread_cond_init()** function will return zero and put the new condition variable id into *cond*, otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_cond_init()** function will fail if:

`[EINVAL]` The value specified by *attr* is invalid.

`[ENOMEM]` The process cannot allocate enough memory to create another condition variable.

`[EAGAIN]` The system temporarily lacks the resources to create another condition variable.

SEE ALSO

pthread_cond_broadcast(3), pthread_cond_destroy(3), pthread_cond_signal(3), pthread_cond_timedwait(3), pthread_cond_wait(3)

STANDARDS

The **pthread_cond_init()** function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

PTHREAD_COND_SIGNAL(3)

NAME

pthread_cond_signal – unblock a thread waiting for a condition variable

SYNOPSIS

```
#include <pthread.h>

int
pthread_cond_signal(pthread_cond_t *cond);
```

DESCRIPTION

The **pthread_cond_signal()** function unblocks one thread waiting for the condition variable *cond*.

RETURN VALUES

If successful, the **pthread_cond_signal()** function will return zero, otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_cond_signal()** function will fail if:

`[EINVAL]` The value specified by *cond* is invalid.

SEE ALSO

pthread_cond_broadcast(3), pthread_cond_destroy(3), pthread_cond_init(3), pthread_cond_timedwait(3), pthread_cond_wait(3)

STANDARDS

The **pthread_cond_signal()** function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

PTHREAD_COND_TIMEDWAIT(3)

NAME

pthread_cond_timedwait – wait on a condition variable for a specific amount of time

SYNOPSIS

```
#include <pthread.h>

int
pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,
                       const struct timespec *abstime);
```

DESCRIPTION

The **pthread_cond_timedwait()** function atomically blocks the current thread waiting on the condition variable specified by *cond*, and unblocks the mutex specified by *mutex*. The waiting thread unblocks only after another thread calls pthread_cond_signal(3), or pthread_cond_broadcast(3) with the same condition variable, or if the system time reaches the time specified in *abstime*, and the current thread reacquires the lock on *mutex*.

RETURN VALUES

If successful, the **pthread_cond_timedwait()** function will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_cond_timedwait()** function will fail if:

`[EINVAL]` The value specified by *cond*, *mutex* or *abstime* is invalid.

`[ETIMEDOUT]` The system time has reached or exceeded the time specified in *abstime*.

SEE ALSO

pthread_cond_broadcast(3), pthread_cond_destroy(3), pthread_cond_init(3), pthread_cond_signal(3), pthread_cond_wait(3)

STANDARDS

The **pthread_cond_timedwait()** function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

PTHREAD_COND_WAIT(3)

NAME

pthread_cond_wait – wait on a condition variable

SYNOPSIS

```
#include <pthread.h>

int
pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

DESCRIPTION

The **pthread_cond_wait()** function atomically blocks the current thread waiting on the condition variable specified by *cond*, and unblocks the mutex specified by *mutex*. The waiting thread unblocks only after another thread calls pthread_cond_signal(3), or pthread_cond_broadcast(3) with the same condition variable, and the current thread reacquires the lock on *mutex*.

RETURN VALUES

If successful, the **pthread_cond_wait()** function will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_cond_wait()** function will fail if:

`[EINVAL]` The value specified by *cond* or the value specified by *mutex* is invalid.

SEE ALSO

pthread_cond_broadcast(3), pthread_cond_destroy(3), pthread_cond_init(3), pthread_cond_signal(3), pthread_cond_timedwait(3)

STANDARDS

The **pthread_cond_wait()** function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

PTHREAD_CREATE(3)

NAME

pthread_create – create a new thread

SYNOPSIS

```
#include <pthread.h>

int
pthread_create(pthread_t *thread, const pthread_attr_t *attr,
               void *(*start_routine)(void *), void *arg);
```

DESCRIPTION

The **pthread_create()** function is used to create a new thread, with attributes specified by *attr*, within a process. If *attr* is `NULL`, the default attributes are used. If the attributes specified by *attr* are modified later, the thread's attributes are not affected. Upon successful completion **pthread_create()** will store the ID of the created thread in the location specified by *thread*.

The *thread* is created executing *start_routine* with *arg* as its sole argument. If the *start_routine* returns, the effect is as if there was an implicit call to *pthread_exit()* using the return value of *start_routine* as the exit status. Note that the thread in which **main()** was originally invoked differs from this. When it returns from **main()**, the effect is as if there was an implicit call to **exit()** using the return value of **main()** as the exit status.

The signal state of the new thread is initialized as:

- The signal mask is inherited from the creating thread.
- The set of signals pending for the new thread is empty.

RETURN VALUES

If successful, the **pthread_create()** function will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_create()** function will fail if:

`[EAGAIN]` The system lacked the necessary resources to create another thread, or the system-imposed limit on the total number of threads in a process [`PTHREAD_THREADS_MAX`] would be exceeded.

`[EINVAL]` The value specified by *attr* is invalid.

`[EMVSERR]` An internal error occurred in the operation system, this should be reported to IBM.

IMPLEMENTATION

The **pthread_create()** function depends on the use of the Unix Systems Services (POSIX) subsystem and POSIX signals. If **pthread_create()** is linked with your program, then POSIX signals are assumed.

Programs that invoke **pthread_create()** become "threaded". When such programs end they invoke the POSIX _exit(2) function and don't simply return to the caller as other Systems/C programs do.

Unlike stack space for the **main()** program, the stack space for a thread is not expandable. Enough space must be allocated for the thread to use or the program will suffer a Dignus out-of-stack ABEND. The default stack space for both 31-bit and 64-bit programs is 4096 bytes, which can be quite small for some programs.

SEE ALSO

fork(2), pthread_cleanup_pop(3), pthread_cleanup_push(3), pthread_exit(3), pthread_join(3)

STANDARDS

The **pthread_create()** function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

PTHREAD_DETACH(3)

NAME

pthread_detach – detach a thread

SYNOPSIS

```
#include <pthread.h>

int
pthread_detach(pthread_t thread);
```

DESCRIPTION

The **pthread_detach()** function is used to indicate to the implementation that storage for the thread *thread* can be reclaimed when the thread terminates. If *thread* has not terminated, **pthread_detach()** will not cause it to terminate. The effect of multiple **pthread_detach()** calls on the same target thread is unspecified.

RETURN VALUES

If successful, the **pthread_detach()** function will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_detach()** function will fail if:

`[EINVAL]` The implementation has detected that the value specified by *thread* does not refer to a joinable thread.

`[ESRCH]` No thread could be found corresponding to that specified by the given thread ID, *thread*.

SEE ALSO

pthread_join(3)

STANDARDS

The **pthread_detach()** function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

PTHREAD_EQUAL(3)

NAME

pthread_equal – compare thread IDs

SYNOPSIS

```
#include <pthread.h>

int
pthread_equal(pthread_t t1, pthread_t t2);
```

DESCRIPTION

The **pthread_equal()** function compares the thread IDs *t1* and *t2*.

RETURN VALUES

The **pthread_equal()** function will return non-zero if the thread IDs *t1* and *t2* correspond to the same thread, otherwise it will return zero.

ERRORS

None.

SEE ALSO

pthread_create(3), pthread_exit(3)

STANDARDS

The **pthread_equal()** function conforms to ISO/IEC 9945-1:1996 (“POSIX.1”).

PTHREAD_EXIT(3)

NAME

pthread_exit – terminate the calling thread

SYNOPSIS

```
#include <pthread.h>

void
pthread_exit(void *value_ptr);
```

DESCRIPTION

The **pthread_exit()** function terminates the calling thread and makes the value *value_ptr* available to any successful join with the terminating thread. Any cancellation cleanup handlers that have been pushed and are not yet popped are popped in the reverse order that they were pushed and then executed. Thread termination does not release any application visible process resources, including, but not limited to, mutexes and file descriptors, nor does it perform any process level cleanup actions, including, but not limited to, calling **atexit()** routines that may exist.

An implicit call to **pthread_exit()** is made when a thread other than the thread in which **main()** was first invoked returns from the start routine that was used to create it. The function's return value serves as the thread's exit status.

The behavior of **pthread_exit()** is undefined if called from a cancellation handler or destructor function that was invoked as the result of an implicit or explicit call to **pthread_exit()**.

After a thread has terminated, the result of access to local (auto) vari- ables of the thread is undefined. Thus, references to local variables of the exiting thread should not be used for the **pthread_exit()** *value_ptr* parameter value.

The process will exit with an exit status of 0 after the last thread has been terminated. The behavior is as if the implementation called **exit()** with a zero argument at thread termination time.

RETURN VALUES

The **pthread_exit()** function cannot return to its caller.

ERRORS

None.

SEE ALSO

_exit(2), exit(3), pthread_create(3), pthread_join(3)

STANDARDS

The **pthread_exit()** function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

PTHREAD_GETSPECIFIC(3)

NAME

pthread_getspecific – get a thread-specific data value

SYNOPSIS

```
#include <pthread.h>

void *
pthread_getspecific(pthread_key_t key);
```

DESCRIPTION

The **pthread_getspecific()** function returns the value currently bound to the specified *key* on behalf of the calling thread.

The effect of calling **pthread_getspecific()** with a key value not obtained from **pthread_key_create()** or after key has been deleted with **pthread_key_delete()** is undefined.

The **pthread_getspecific()** function may be called from a thread-specific data destructor function.

RETURN VALUES

The **pthread_getspecific()** function will return the thread-specific data value associated with the given *key*. If no thread-specific data value is associated with *key*, then the value `NULL` is returned.

ERRORS

None.

SEE ALSO

pthread_key_create(3), pthread_key_delete(3), pthread_setspecific(3)

STANDARDS

The **pthread_getspecific()** function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

PTHREAD_JOIN(3)

NAME

pthread_join – wait for thread termination

SYNOPSIS

```
#include <pthread.h>

int
pthread_join(pthread_t thread, void **value_ptr);
```

DESCRIPTION

The **pthread_join()** function suspends execution of the calling thread until the target thread terminates unless the target thread has already terminated.

On return from a successful **pthread_join()** call with a non-`NULL` *value_ptr* argument, the value passed to **pthread_exit()** by the terminating thread is stored in the location referenced by *value_ptr*. When a **pthread_join()** returns successfully, the target thread has been terminated. The results of multiple simultaneous calls to **pthread_join()** specifying the same target thread are undefined. If the thread calling **pthread_join()** is cancelled, then the target thread is not detached.

A thread that has exited but remains unjoined counts against [`_POSIX_THREAD_THREADS_MAX`].

RETURN VALUES

If successful, the *pthread_join()* function will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_join()** function will fail if:

`[EINVAL]` The implementation has detected that the value specified by *thread* does not refer to a joinable thread.

`[ESRCH]` No thread could be found corresponding to that specified by the given thread ID, *thread*.

`[EDEADLK]` A deadlock was detected or the value of *thread* specifies the calling thread.

SEE ALSO

wait(2), pthread_create(3)

STANDARDS

The **pthread_join()** function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

PTHREAD_KEY_CREATE(3)

NAME

pthread_key_create – thread-specific data key creation

SYNOPSIS

```
#include <pthread.h>

int
pthread_key_create(pthread_key_t *key, void (*destructor)(void *));
```

DESCRIPTION

The **pthread_key_create()** function creates a thread-specific data key visible to all threads in the process. Key values provided by **pthread_key_create()** are opaque objects used to locate thread-specific data. Although the same key value may be used by different threads, the values bound to the key by **pthread_setspecific()** are maintained on a per-thread basis and persist for the life of the calling thread.

Upon key creation, the value `NULL` is associated with the new key in all active threads. Upon thread creation, the value `NULL` is associated with all defined keys in the new thread.

An optional destructor function may be associated with each key value. At thread exit, if a key value has a non-`NULL` destructor pointer, and the thread has a non-`NULL` value associated with the key, the function pointed to is called with the current associated value as its sole argument. The order of destructor calls is unspecified if more than one destructor exists for a thread when it exits.

If, after all the destructors have been called for all non-`NULL` values with associated destructors, there are still some non-`NULL` values with associated destructors, then the process is repeated. If, after at least [`PTHREAD_DESTRUCTOR_ITERATIONS`] iterations of destructor calls for outstanding non-`NULL` values, there are still some non-`NULL` values with associated destructors, the implementation stops calling destructors.

RETURN VALUES

If successful, the **pthread_key_create()** function will store the newly created key value at the location specified by *key* and returns zero.

Otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_key_create()** function will fail if:

`[EAGAIN]` The system lacked the necessary resources to create another thread-specific data key, or the system-imposed limit on the total number of keys per process [`PTHREAD_KEYS_MAX`] would be exceeded.

`[ENOMEM]` Insufficient memory exists to create the key.

SEE ALSO

pthread_getspecific(3), pthread_key_delete(3), pthread_setspecific(3)

STANDARDS

The **pthread_key_create()** function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

PTHREAD_KEY_DELETE(3)

NAME

pthread_key_delete – delete a thread-specific data key

SYNOPSIS

```
#include <pthread.h>

int
pthread_key_delete(pthread_key_t key);
```

DESCRIPTION

The **pthread_key_delete()** function deletes a thread-specific data key previously returned by **pthread_key_create()**. The thread-specific data values associated with *key* need not be `NULL` at the time that **pthread_key_delete()** is called. It is the responsibility of the application to free any application storage or perform any cleanup actions for data structures related to the deleted key or associated thread-specific data in any threads; this cleanup can be done either before or after **pthread_key_delete()** is called. Any attempt to use key following the call to **pthread_key_delete()** results in undefined behavior.

The **pthread_key_delete()** function is callable from within destructor functions. Destructor functions are not invoked by **pthread_key_delete()**. Any destructor function that may have been associated with *key* will no longer be called upon thread exit.

RETURN VALUES

If successful, the **pthread_key_delete()** function will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_key_delete()** function will fail if:

`[EINVAL]` The *key* value is invalid.

SEE ALSO

pthread_getspecific(3), pthread_key_create(3), pthread_setspecific(3)

STANDARDS

The **pthread_key_delete()** function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

PTHREAD_KILL(3)

NAME

pthread_kill – send a signal to a specified thread

SYNOPSIS

```
#include <pthread.h>
#include <signal.h>

int
pthread_kill(pthread_t thread, int sig);
```

DESCRIPTION

The **pthread_kill()** function sends a signal, specified by *sig*, to a thread, specified by *thread*. If *sig* is 0, error checking is performed, but no signal is actually sent.

RETURN VALUES

If successful, **pthread_kill()** returns 0. Otherwise, an error number is returned.

ERRORS

The **pthread_kill()** function will fail if:

`[ESRCH]` *thread* is an invalid thread ID.

`[EINVAL]` *sig* is an invalid or unsupported signal number.

SEE ALSO

kill(2), pthread_self(3), raise(3)

STANDARDS

The **pthread_kill()** function conforms to ISO/IEC 9945-1:1996 (“POSIX.1”)

PTHREAD_MAIN_NP(3)

NAME

pthread_main_np – identify the initial thread

SYNOPSIS

```
#include <pthread.h>

int
pthread_main_np(void);
```

DESCRIPTION

The **pthread_main_np** function is used identify the initial thread.

RETURN VALUES

The **pthread_main_np** function returns 1 if the calling thread is the initial thread and 0 if the calling thread is not the initial thread.

SEE ALSO

pthread_create(3), pthread_equal(3), pthread_self(3)

PTHREAD_MUTEXATTR(3)

NAME

pthread_mutexattr_init, pthread_mutexattr_destroy, pthread_mutexattr_setprioceiling, pthread_mutexattr_getprioceiling, pthread_mutexattr_setprotocol, pthread_mutexattr_getprotocol, pthread_mutexattr_settype, pthread_mutexattr_gettype – mutex attribute operations

SYNOPSIS

```
#include <pthread.h>

int
pthread_mutexattr_init(pthread_mutexattr_t *attr);

int
pthread_mutexattr_destroy(pthread_mutexattr_t *attr);

int
pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
         int prioceiling);

int
pthread_mutexattr_getprioceiling(pthread_mutexattr_t *attr,
         int *prioceiling);

int
pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol);

int
pthread_mutexattr_getprotocol(pthread_mutexattr_t *attr, int *protocol);

int
pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);

int
pthread_mutexattr_gettype(pthread_mutexattr_t *attr, int *type);
```

DESCRIPTION

Mutex attributes are used to specify parameters to **pthread_mutex_init()**. One attribute object can be used in multiple calls to **pthread_mutex_init()**, with or without modifications between calls.

The **pthread_mutexattr_init()** function initializes *attr* with all the default mutex attributes.

The **pthread_mutexattr_destroy()** function destroys *attr*.

The **pthread_mutexattr_set*()** functions set the attribute that corresponds to each function name.

The **pthread_mutexattr_get*()** functions copy the value of the attribute that corresponds to each function name to the location pointed to by the second function parameter.

RETURN VALUES

If successful, these functions return 0. Otherwise, an error number is returned to indicate the error.

ERRORS

The **pthread_mutexattr_init()** function will fail if:

`[ENOMEM]` Out of memory.

The **pthread_mutexattr_destroy()** function will fail if:

`[EINVAL]` Invalid value for *attr*.

The **pthread_mutexattr_setprioceiling()** function will fail if:

`[EINVAL]` Invalid value for *attr*, or invalid value for *prioceiling*.

The **pthread_mutexattr_getprioceiling()** function will fail if:

`[EINVAL]` Invalid value for *attr*.

The **pthread_mutexattr_setprotocol()** function will fail if:

`[EINVAL]` Invalid value for *attr*, or invalid value for *protocol*.

The **pthread_mutexattr_getprotocol()** function will fail if:

[EINVAL] Invalid value for *attr*.

The **pthread_mutexattr_settype()** function will fail if:

[EINVAL] Invalid value for *attr*, or invalid value for *type*.

The **pthread_mutexattr_gettype()** function will fail if:

[EINVAL] Invalid value for *attr*.

IMPLEMENTATION

This implementation does not support scheduling priority settings.

SEE ALSO

pthread_mutex_init(3)

STANDARDS

The **pthread_mutexattr_init()** and **pthread_mutexattr_destroy()** functions conform to ISO/IEC 9945-1:1996 ("POSIX.1")

The **pthread_mutexattr_setprioceiling()**, **pthread_mutexattr_getprioceiling()**, **pthread_mutexattr_setprotocol()**, **pthread_mutexattr_getprotocol()**, **pthread_mutexattr_settype()**, and **pthread_mutexattr_gettype()** functions conform to Version 2 of the Single UNIX Specification ("SUSv2")

PTHREAD_MUTEX_DESTROY(3)

NAME

pthread_mutex_destroy – free resources allocated for a mutex

SYNOPSIS

```
#include <pthread.h>

int
pthread_mutex_destroy(pthread_mutex_t *mutex);
```

DESCRIPTION

The **pthread_mutex_destroy()** function frees the resources allocated for *mutex*.

RETURN VALUES

If successful, **pthread_mutex_destroy()** will return zero, otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_mutex_destroy()** function will fail if:

`[EINVAL]` The value specified by *mutex* is invalid.

`[EBUSY]` *mutex* is locked by another thread.

SEE ALSO

pthread_mutex_init(3), pthread_mutex_lock(3), pthread_mutex_trylock(3), pthread_mutex_unlock(3)

STANDARDS

The **pthread_mutex_destroy()** function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

PTHREAD_MUTEX_INIT(3)

NAME

pthread_mutex_init – create a mutex

SYNOPSIS

```
#include <pthread.h>

int
pthread_mutex_init(pthread_mutex_t *mutex,
         const pthread_mutexattr_t *attr);
```

DESCRIPTION

The **pthread_mutex_init()** function creates a new mutex, with attributes specified with *attr*. If *attr* is `NULL` the default attributes are used.

RETURN VALUES

If successful, **pthread_mutex_init()** will return zero and put the new mutex id into *mutex*, otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_mutex_init()** function will fail if:

`[EINVAL]` The value specified by *attr* is invalid.

`[ENOMEM]` The process cannot allocate enough memory to create another mutex.

SEE ALSO

pthread_mutex_destroy(3), pthread_mutex_lock(3), pthread_mutex_trylock(3), pthread_mutex_unlock(3)

STANDARDS

The **pthread_mutex_init()** function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

PTHREAD_MUTEX_LOCK(3)

NAME

pthread_mutex_lock – lock a mutex

SYNOPSIS

```
#include <pthread.h>

int
pthread_mutex_lock(pthread_mutex_t *mutex);
```

DESCRIPTION

The **pthread_mutex_lock()** function locks *mutex*. If the *mutex* is already locked, the calling thread will block until the *mutex* becomes available.

RETURN VALUES

If successful, **pthread_mutex_lock()** will return zero, otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_mutex_lock()** function will fail if:

[EINVAL]	The value specified by *mutex* is invalid.
[EDEADLK]	A deadlock would occur if the thread blocked waiting for *mutex*.

SEE ALSO

pthread_mutex_destroy(3), pthread_mutex_init(3), pthread_mutex_trylock(3), pthread_mutex_unlock(3)

STANDARDS

The **pthread_mutex_lock()** function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

PTHREAD_MUTEX_TRYLOCK(3)

NAME

pthread_mutex_trylock – attempt to lock a mutex without blocking

SYNOPSIS

```
#include <pthread.h>

int
pthread_mutex_trylock(pthread_mutex_t *mutex);
```

DESCRIPTION

The **pthread_mutex_trylock()** function locks *mutex*. If the *mutex* is already locked, **pthread_mutex_trylock()** will not block waiting for the *mutex*, but will return an error condition.

RETURN VALUES

If successful, **pthread_mutex_trylock()** will return zero, otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_mutex_trylock()** function will fail if:

`[EINVAL]` The value specified by *mutex* is invalid.

`[EBUSY]` *mutex* is already locked.

SEE ALSO

pthread_mutex_destroy(3), pthread_mutex_init(3), pthread_mutex_lock(3), pthread_mutex_unlock(3)

STANDARDS

The **pthread_mutex_trylock()** function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

PTHREAD_MUTEX_UNLOCK(3)

NAME

pthread_mutex_unlock – unlock a mutex

SYNOPSIS

```
#include <pthread.h>

int
pthread_mutex_unlock(pthread_mutex_t *mutex);
```

DESCRIPTION

If the current thread holds the lock on *mutex*, then the **pthread_mutex_unlock()** function unlocks *mutex*.

RETURN VALUES

If successful, **pthread_mutex_unlock()** will return zero, otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_mutex_unlock()** function will fail if:

`[EINVAL]` The value specified by *mutex* is invalid.

`[EPERM]` The current thread does not hold a lock on *mutex*.

SEE ALSO

pthread_mutex_destroy(3), pthread_mutex_init(3), pthread_mutex_lock(3), pthread_mutex_trylock(3)

STANDARDS

The **pthread_mutex_unlock()** function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

PTHREAD_ONCE(3)

NAME

pthread_once – dynamic package initialization

SYNOPSIS

```
#include <pthread.h>

pthread_once_t once_control = PTHREAD_ONCE_INIT;

int
pthread_once(pthread_once_t *once_control, void (*init_routine)(void));
```

DESCRIPTION

The first call to **pthread_once()** by any thread in a process, with a given *once_control*, will call the *init_routine()* with no arguments. Subsequent calls to **pthread_once()** with the same *once_control* will not call the *init_routine()*. On return from **pthread_once()**, it is guaranteed that *init_routine()* has completed. The *once_control* parameter is used to determine whether the associated initialization routine has been called.

The function **pthread_once()** is not a cancellation point. However, if *init_routine()* is a cancellation point and is cancelled, the effect on *once_control* is as if **pthread_once()** was never called.

The constant `PTHREAD_ONCE_INIT` is defined by header `<pthread.h>`.

The behavior of **pthread_once()** is undefined if *once_control* has automatic storage duration or is not initialized by `PTHREAD_ONCE_INIT`.

RETURN VALUES

If successful, the **pthread_once()** function will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

None.

STANDARDS

The **pthread_once()** function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

PTHREAD_RWLOCKATTR_DESTROY(3)

NAME

pthread_rwlockattr_destroy – destroy a read/write lock

SYNOPSIS

```
#include <pthread.h>

int
pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
```

DESCRIPTION

The **pthread_rwlockattr_destroy()** function is used to destroy a read/write lock attribute object previously created with **pthread_rwlockattr_init()**.

RETURN VALUES

If successful, the **pthread_rwlockattr_destroy()** function will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_rwlockattr_destroy()** function may fail if:

`[EINVAL]` The value specified by *attr* is invalid.

SEE ALSO

pthread_rwlockattr_init(3)

STANDARDS

The **pthread_rwlockattr_destroy()** function is expected to conform to Version 2 of the Single UNIX Specification ("SUSv2").

PTHREAD_RWLOCKATTR_GETPSHARED(3)

NAME

pthread_rwlockattr_getpshared – get the process shared attribute

SYNOPSIS

```
#include <pthread.h>

int
pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *attr,
         int *pshared);
```

DESCRIPTION

The **pthread_rwlockattr_getpshared()** function is used to get the process shared setting of a read/write lock attribute object. The setting is returned via *pshared*, and may be one of two values:

`[PTHREAD_PROCESS_SHARED]` Any thread of any process that has access to the memory where the read/write lock resides can manipulate the lock.

`[PTHREAD_PROCESS_PRIVATE]` Only threads created within the same process as the thread that initialized the read/write lock can manipulate the lock. This is the default value.

RETURN VALUES

If successful, the **pthread_rwlockattr_getpshared()** function will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_rwlockattr_getpshared()** function may fail if:

`[EINVAL]` The value specified by *attr* is invalid.

IMPLEMENTATION

This implementation does not support process shared locks.

SEE ALSO

pthread_rwlockattr_init(3), pthread_rwlockattr_setpshared(3), pthread_rwlock_init(3)

STANDARDS

The **pthread_rwlockattr_getpshared()** function is expected to conform to Version 2 of the Single UNIX Specification ("SUSv2").

PTHREAD_RWLOCKATTR_INIT(3)

NAME

pthread_rwlockattr_init – initialize a read/write lock

SYNOPSIS

```
#include <pthread.h>

int
pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
```

DESCRIPTION

The **pthread_rwlockattr_init()** function is used to initialize a read/write lock attributes object.

RETURN VALUES

If successful, the **pthread_rwlockattr_init()** function will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_rwlockattr_init()** function will fail if:

`[ENOMEM]` Insufficient memory exists to initialize the attribute object.

SEE ALSO

pthread_rwlockattr_destroy(3), pthread_rwlockattr_getpshared(3), pthread_rwlockattr_setpshared(3), pthread_rwlock_init(3)

STANDARDS

The **pthread_rwlockattr_init()** function is expected to conform to Version 2 of the Single UNIX Specification ("SUSv2").

PTHREAD_RWLOCKATTR_SETPSHARED(3)

NAME

pthread_rwlockattr_setpshared – set the process shared attribute

SYNOPSIS

```
#include <pthread.h>

int
pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int pshared);
```

DESCRIPTION

The **pthread_rwlockattr_setpshared()** function sets the process shared attribute of attr to the value referenced by pshared. The *pshared* argument may be one of two values:

`[PTHREAD_PROCESS_SHARED]` Any thread of any process that has access to the memory where the read/write lock resides can manipulate the lock.

`[PTHREAD_PROCESS_PRIVATE]` Only threads created within the same process as the thread that initialized the read/write lock can manipulate the lock. This is the default value.

RETURN VALUES

If successful, the **pthread_rwlockattr_setpshared()** function will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_rwlockattr_setpshared()** function will fail if:

`[EINVAL]` The value specified by *attr* or *pshared* is invalid.

IMPLEMENTATION

This implementation does not support process-sharded locks, the `PTHREAD_PROCESS_SHARED` attribute is not supported.

SEE ALSO

pthread_rwlockattr_getpshared(3), pthread_rwlockattr_init(3), pthread_rwlock_init(3)

STANDARDS

The **pthread_rwlockattr_setpshared()** function is expected to conform to Version 2 of the Single UNIX Specification ("SUSv2").

PTHREAD_RWLOCK_DESTROY(3)

NAME

pthread_rwlock_destroy – destroy a read/write lock

SYNOPSIS

```
#include <pthread.h>

int
pthread_rwlock_destroy(pthread_rwlock_t *lock);
```

DESCRIPTION

The **pthread_rwlock_destroy()** function is used to destroy a read/write lock previously created with **pthread_rwlock_init()**.

RETURN VALUES

If successful, the **pthread_rwlock_destroy()** function will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_rwlock_destroy()** function will fail if:

`[EPERM]` The caller does not have the privilege to perform the operation.

The **pthread_rwlock_destroy()** function may fail if:

`[EBUSY]` The system has detected an attempt to destroy the object referenced by lock while it is locked.

`[EINVAL]` The value specified by *lock* is invalid.

SEE ALSO

pthread_rwlock_init(3)

STANDARDS

The **pthread_rwlock_destroy()** function is expected to conform to Version 2 of the Single UNIX Specification ("SUSv2").

PTHREAD_RWLOCK_INIT(3)

NAME

pthread_rwlock_init – initialize a read/write lock

SYNOPSIS

```
#include <pthread.h>

int
pthread_rwlock_init(pthread_rwlock_t *lock,
         const pthread_rwlockattr_t *attr);
```

DESCRIPTION

The **pthread_rwlock_init()** function is used to initialize a read/write lock, with attributes specified by *attr*. If *attr* is `NULL`, the default read/write lock attributes are used.

The results of calling **pthread_rwlock_init()** with an already initialized lock are undefined.

RETURN VALUES

If successful, the **pthread_rwlock_init()** function will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_rwlock_init()** function will fail if:

`[EAGAIN]` The system lacked the necessary resources (other than memory) to initialize the lock.

`[ENOMEM]` Insufficient memory exists to initialize the lock.

`[EPERM]` The caller does not have sufficient privilege to perform the operation.

The **pthread_rwlock_init()** function may fail if:

[EBUSY] The system has detected an attempt to re-initialize the object referenced by *lock*, a previously initialized but not yet destroyed read/write lock.

[EINVAL] The value specified by *attr* is invalid.

IMPLEMENTATION

The PTHREAD_PROCESS_SHARED attribute is not supported.

SEE ALSO

pthread_rwlockattr_init(3), pthread_rwlockattr_setpshared(3), pthread_rwlock_destroy(3)

STANDARDS

The **pthread_rwlock_init()** function is expected to conform to Version 2 of the Single UNIX Specification ("SUSv2").

PTHREAD_RWLOCK_RDLOCK(3)

NAME

pthread_rwlock_rdlock, pthread_rwlock_tryrdlock – acquire a read/write lock for reading

SYNOPSIS

```
#include <pthread.h>

int
pthread_rwlock_rdlock(pthread_rwlock_t *lock);

int
pthread_rwlock_tryrdlock(pthread_rwlock_t *lock);
```

DESCRIPTION

The **pthread_rwlock_rdlock()** function acquires a read lock on *lock* provided that lock is not presently held for writing and no writer threads are presently blocked on the lock. If the read lock cannot be immediately acquired, the calling thread blocks until it can acquire the lock.

The **pthread_rwlock_tryrdlock()** function performs the same action, but does not block if the lock cannot be immediately obtained (i.e., the lock is held for writing or there are waiting writers).

A thread may hold multiple concurrent read locks. If so, **pthread_rwlock_unlock()** must be called once for each lock obtained.

The results of acquiring a read lock while the calling thread holds a write lock are undefined.

RETURN VALUES

If successful, the **pthread_rwlock_rdlock()** and **pthread_rwlock_tryrdlock()** functions will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_rwlock_tryrdlock()** function will fail if:

[EBUSY] The lock could not be acquired because a writer holds the lock or was blocked on it.

The **pthread_rwlock_rdlock()** and **pthread_rwlock_tryrdlock()** functions may fail if:

[EAGAIN] The lock could not be acquired because the maximum number of read locks against *lock* has been exceeded.

[EDEADLK] The current thread already owns *lock* for writing.

[EINVAL] The value specified by *lock* is invalid.

[ENOMEM] Insufficient memory exists to initialize the lock (applies to statically initialized locks only).

SEE ALSO

pthread_rwlock_init(3), pthread_rwlock_trywrlock(3), pthread_rwlock_unlock(3), pthread_rwlock_wrlock(3)

STANDARDS

The **pthread_rwlock_rdlock()** and **pthread_rwlock_tryrdlock()** functions are expected to conform to Version 2 of the Single UNIX Specification ("SUSv2").

PTHREAD_RWLOCK_UNLOCK(3)

NAME

pthread_rwlock_unlock – release a read/write lock

SYNOPSIS

```
#include <pthread.h>

int
pthread_rwlock_unlock(pthread_rwlock_t *lock);
```

DESCRIPTION

The **pthread_rwlock_unlock()** function is used to release the read/write lock previously obtained by **pthread_rwlock_rdlock()**, **pthread_rwlock_wrlock()**, **pthread_rwlock_tryrdlock()**, or **pthread_rwlock_trywrlock()**.

RETURN VALUES

If successful, the **pthread_rwlock_unlock()** function will return zero. Otherwise an error number will be returned to indicate the error.

The results are undefined if lock is not held by the calling thread.

ERRORS

The **pthread_rwlock_unlock()** function may fail if:

`[EINVAL]` The value specified by *lock* is invalid.

`[EPERM]` The current thread does not own the read/write lock.

SEE ALSO

pthread_rwlock_rdlock(3), pthread_rwlock_wrlock(3)

STANDARDS

The **pthread_rwlock_unlock()** function is expected to conform to Version 2 of the Single UNIX Specification ("SUSv2").

PTHREAD_RWLOCK_WRLOCK(3)

NAME

pthread_rwlock_wrlock, pthread_rwlock_trywrlock – acquire a read/write lock for writing

SYNOPSIS

```
#include <pthread.h>

int
pthread_rwlock_wrlock(pthread_rwlock_t *lock);

int
pthread_rwlock_trywrlock(pthread_rwlock_t *lock);
```

DESCRIPTION

The **pthread_rwlock_wrlock()** function blocks until a write lock can be acquired against *lock*. The **pthread_rwlock_trywrlock()** function performs the same action, but does not block if the lock cannot be immediately obtained.

The results are undefined if the calling thread already holds the lock at the time the call is made.

RETURN VALUES

If successful, the **pthread_rwlock_wrlock()** and **pthread_rwlock_trywrlock()** functions will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_rwlock_trywrlock()** function will fail if:

`[EBUSY]` The calling thread is not able to acquire the lock without blocking.

The **pthread_rwlock_wrlock()** and **pthread_rwlock_trywrlock()** functions may fail if:

[EDEADLK] The calling thread already owns the read/write lock (for reading or writing).

[EINVAL] The value specified by *lock* is invalid.

[ENOMEM] Insufficient memory exists to initialize the lock (applies to statically initialized locks only).

SEE ALSO

pthread_rwlock_init(3), pthread_rwlock_rdlock(3), pthread_rwlock_tryrdlock(3), pthread_rwlock_unlock(3)

STANDARDS

The **pthread_rwlock_wrlock()** and **pthread_rwlock_trywrlock()** functions are expected to conform to Version 2 of the Single UNIX Specification ("SUSv2").

PTHREAD_SELF(3)

NAME

pthread_self – get the calling thread's ID

SYNOPSIS

```
#include <pthread.h>

pthread_t
pthread_self(void);
```

DESCRIPTION

The **pthread_self()** function returns the thread ID of the calling thread.

RETURN VALUES

The **pthread_self()** function returns the thread ID of the calling thread.

ERRORS

None.

SEE ALSO

pthread_create(3), pthread_equal(3)

STANDARDS

The **pthread_self()** function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

PTHREAD_SET_LIMIT_NP(3)

NAME

pthread_set_limit_np – control number of tasks and/or threads allowed in the process

SYNOPSIS

```
#include <pthread.h>

int
pthread_set_limit_np(int which, int maxtasks, int maskthreads);
```

DESCRIPTION

The **pthread_set_limit_np()** function controls the maximum number of tasks and the maximum number of threads allowed for the current process. These values are unique to the process, however the initial settings are inherited from the default settings defined when z/OS is initialized.

The **pthread_set_limit_np()** function can set either the number of z/OS tasks, or the number of threads, or both. The *which* value determines which setting is desired:

`__STL_MAX_TASKS` The maximum number of tasks is set to the value of *maxtasks*.

`__STL_MAX_THREADS` The maximum number of threads is set to the value of *maxthreads*.

`__STL_SET_BOTH` The maximum number of tasks is set to the value of *maxtasks* and the maximum number of threads is set to the value of *maxthreads*.

The z/OS operating system requires memory in the private area below the 16M for each task and thread. The number of threads and tasks depends on how much memory is available in that area. When that area is exhausted, the task or the program may be summarily terminated by the operating system, along with appropriate messages on the system console. IBM recommends that a reasonable limit for threads and threads is 200 to 400.

The `weight` attribute of a thread controls the mapping of threads to tasks. Also the `synctype` attribute controls if a thread can be created without an available task.

RETURN VALUES

If successful, **pthread_set_limit_np()** returns 0, otherwise -1 is returned.

NAME

pthread_setspecific – set a thread-specific data value

SYNOPSIS

```
#include <pthread.h>

int
pthread_setspecific(pthread_key_t key, const void *value);
```

DESCRIPTION

The **pthread_setspecific()** function associates a thread-specific value with a *key* obtained via a previous call to **pthread_key_create()**. Different threads can bind different values to the same *key*. These values are typically pointers to blocks of dynamically allocated memory that have been reserved for use by the calling thread.

The effect of calling **pthread_setspecific()** with a *key* value not obtained from **pthread_key_create()** or after *key* has been deleted with **pthread_key_delete()** is undefined.

The **pthread_setspecific()** function may be called from a thread-specific data destructor function, however this may result in lost storage or infinite loops.

RETURN VALUES

If successful, the **pthread_setspecific()** function will return zero. Otherwise an error number will be returned to indicate the error.

ERRORS

The **pthread_setspecific()** function will fail if:

`[ENOMEM]` Insufficient memory exists to associate the value with the *key*.

`[EINVAL]` The *key* value is invalid.

SEE ALSO

pthread_getspecific(3), pthread_key_create(3), pthread_key_delete(3)

STANDARDS

The **pthread_setspecific()** function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

PTHREAD_SIGMASK(3)

NAME

pthread_sigmask – examine and/or change a thread's signal mask

SYNOPSIS

```
#include <pthread.h>
#include <signal.h>

int
pthread_sigmask(int how, const sigset_t *set, sigset_t *oset);
```

DESCRIPTION

The **pthread_sigmask()** function examines and/or changes the calling thread's signal mask.

If *set* is not `NULL`, it specifies a set of signals to be modified, and *how* specifies what to set the signal mask to:

`[SIG_BLOCK]` Union of the current mask and *set*.

`[SIG_UNBLOCK]` Intersection of the current mask and the complement of *set*.

`[SIG_SETMASK]` *set*.

If *oset* is not `NULL`, the previous signal mask is stored in the location pointed to by *oset*.

`SIGKILL` and `SIGSTOP` cannot be blocked, and will be silently ignored if included in the signal mask.

RETURN VALUES

If successful, **pthread_sigmask()** returns 0. Otherwise, an error is returned.

ERRORS

The **pthread_sigmask()** function will fail if:

`[EINVAL]` *how* is not one of the defined values.

SEE ALSO

sigaction(2), sigpending(2), sigprocmask(2), sigsuspend(2), sigsetops(3)

STANDARDS

The **pthread_sigmask()** function conforms to ISO/IEC 9945-1:1996 ("POSIX.1")

PTHREAD_SPIN_INIT(3)

NAME

pthread_spin_init, pthread_spin_destroy – initialize or destroy a spin lock

SYNOPSIS

```
#include <pthread.h>

int
pthread_spin_init(pthread_spinlock_t *lock, int pshared);

int
pthread_spin_destroy(pthread_spinlock_t *lock);
```

DESCRIPTION

The **pthread_spin_init()** function will initialize *lock* to an unlocked state and allocate any resources necessary to begin using it. If *pshared* is set to `PTHREAD_PROCESS_SHARED`, any thread, whether belonging to the process in which the spinlock was created or not, that has access to the memory area where *lock* resides, can use *lock*. If it is set to `PTHREAD_PROCESS_PRIVATE`, it can only be used by threads within the same process.

The **pthread_spin_destroy()** function will destroy *lock* and release any resources that may have been allocated on its behalf.

RETURN VALUES

If successful, both **pthread_spin_init()** and **pthread_spin_destroy()** will return zero. Otherwise, an error number will be returned to indicate the error.

Neither of these functions will return `EINTR`.

ERRORS

The **pthread_spin_init()** and **pthread_spin_destroy()** functions will fail if:

`[EBUSY]` An attempt to initialize or destroy *lock* while it is in use.

`[EINVAL]` The value specified by *lock* is invalid.

The **pthread_spin_init()** function will fail if:

`[EAGAIN]` Insufficient resources, other than memory, to initialize *lock*.

`[ENOMEM]` Insufficient memory to initialize lock.

SEE ALSO

pthread_spin_lock(3), pthread_spin_unlock(3)

IMPLEMENTATION

This implementation does not support process-shared locks, if any value other than `PTHREAD_PROCESSES_PRIVATE` is specified, **pthread_spin_init()** returns `EINVAL`.

PTHREAD_SPIN_LOCK(3)

NAME

pthread_spin_lock, pthread_spin_trylock, pthread_spin_unlock – lock or unlock a spin lock

SYNOPSIS

```
#include <pthread.h>

int
pthread_spin_lock(pthread_spinlock_t *lock);

int
pthread_spin_trylock(pthread_spinlock_t *lock);

int
pthread_spin_unlock(pthread_spinlock_t *lock);
```

DESCRIPTION

The **pthread_spin_lock()** function will acquire *lock* if it is not currently owned by another thread. If the *lock* cannot be acquired immediately, it will spin attempting to acquire the lock (it will not sleep) until it becomes available.

The **pthread_spin_trylock()** function is the same as **pthread_spin_lock()** except that if it cannot acquire lock immediately it will return with an error.

The **pthread_spin_unlock()** function will release *lock*, which must have been previously locked by a call to **pthread_spin_lock()** or **pthread_spin_trylock()**.

RETURN VALUES

If successful, all these functions will return zero. Otherwise, an error number will be returned to indicate the error.

None of these functions will return `EINTR`.

ERRORS

The **pthread_spin_lock()**, **pthread_spin_trylock()** and **pthread_spin_unlock()** functions will fail if:

[EINVAL] The value specified by *lock* is invalid or is not initialized.

The **pthread_spin_lock()** function may fail if:

[EDEADLK] The calling thread already owns the *lock*.

The **pthread_spin_trylock()** function will fail if:

[EBUSY] Another thread currently holds *lock*.

The **pthread_spin_unlock()** function may fail if:

[EPERM] The calling thread does not own *lock*.

SEE ALSO

pthread_spin_destroy(3), pthread_spin_init(3)

STANDARDS

The implementation of **pthread_spin_lock()**, **pthread_spin_trylock()** and **pthread_spin_unlock()** is expected to conform to IEEE Std 1003.2 ("POSIX.2").

PTHREAD_TESTCANCEL(3)

NAME

pthread_setcancelstate, pthread_setcanceltype, pthread_testcancel – set cancelability state

SYNOPSIS

```
#include <pthread.h>

int
pthread_setcancelstate(int state, int *oldstate);

int
pthread_setcanceltype(int type, int *oldtype);

void
pthread_testcancel(void);
```

DESCRIPTION

The **pthread_setcancelstate()** function atomically both sets the calling thread's cancelability state to the indicated *state* and, if *oldstate* is not `NULL`, returns the previous cancelability state at the location referenced by *oldstate*. Legal values for state are `PTHREAD_CANCEL_ENABLE` and `PTHREAD_CANCEL_DISABLE`.

The **pthread_setcanceltype()** function atomically both sets the calling thread's cancelability type to the indicated *type* and, if *oldtype* is not `NULL`, returns the previous cancelability type at the location referenced by *oldtype*. Legal values for type are `PTHREAD_CANCEL_DEFERRED` and `PTHREAD_CANCEL_ASYNCHRONOUS`.

The cancelability state and type of any newly created threads, including the thread in which **main()** was first invoked, are `PTHREAD_CANCEL_ENABLE` and `PTHREAD_CANCEL_DEFERRED` respectively.

The **pthread_testcancel()** function creates a cancellation point in the calling thread. The **pthread_testcancel()** function has no effect if cancelability is disabled.

Cancelability States

The cancelability state of a thread determines the action taken upon receipt of a cancellation request. The thread may control cancellation in a number of ways.

Each thread maintains its own "cancelability state" which may be encoded in two flags:

Cancelability Enable When cancelability is `PTHREAD_CANCEL_DISABLE`, cancellation requests against the target thread are held pending.

Cancelability Type When cancelability is enabled and the cancelability type is `PTHREAD_CANCEL_ASYNCHRONOUS`, new or pending cancellation requests may be acted upon at any time. When cancelability is enabled and the cancelability type is `PTHREAD_CANCEL_DEFERRED`, cancellation requests are held pending until a cancellation point (see below) is reached. If cancelability is disabled, the setting of the cancelability type has no immediate effect as all cancellation requests are held pending; however, once cancelability is enabled again the new type will be in effect.

Cancellation Points

Cancellation points will occur when a thread is executing at least the following functions: **close()**, **creat()**, **fcntl()**, **fsync()**, **msync()**, **open()**, **pause()**, **pthread_cond_timedwait()**, **pthread_cond_wait()**, **pthread_join()**, **pthread_testcancel()**, **read()**, **sigwaitinfo()**, **sigsuspend()**, **sigwait()**, **sleep()**, **system()**, **tcdrain()**, **wait()**, **waitpid()**, **write()**.

RETURN VALUES

If successful, the **pthread_setcancelstate()** and **pthread_setcanceltype()** functions will return zero. Otherwise, an error number shall be returned to indicate the error.

The **pthread_setcancelstate()** and **pthread_setcanceltype()** functions are used to control the points at which a thread may be asynchronously canceled. For cancellation control to be usable in modular fashion, some rules must be followed.

For purposes of this discussion, consider an object to be a generalization of a procedure. It is a set of procedures and global variables written as a unit and called by clients not known by the object. Objects may depend on other objects.

First, cancelability should only be disabled on entry to an object, never explicitly enabled. On exit from an object, the cancelability state should always be restored to its value on entry to the object.

This follows from a modularity argument: if the client of an object (or the client of an object that uses that object) has disabled cancelability, it is because the client does not want to have to worry about how to clean up if the thread is canceled while executing some sequence of actions. If an object is called in such a state and

it enables cancelability and a cancellation request is pending for that thread, then the thread will be canceled, contrary to the wish of the client that disabled.

Second, the cancelability type may be explicitly set to either deferred or asynchronous upon entry to an object. But as with the cancelability state, on exit from an object that cancelability type should always be restored to its value on entry to the object.

Finally, only functions that are cancel-safe may be called from a thread that is asynchronously cancelable.

ERRORS

The function **pthread_setcancelstate()** may fail with:

`[EINVAL]` The specified *state* is not `PTHREAD_CANCEL_ENABLE` or `PTHREAD_CANCEL_DISABLE`.

The function **pthread_setcanceltype()** may fail with:

`[EINVAL]` The specified state is not `PTHREAD_CANCEL_DEFERRED` or `PTHREAD_CANCEL_ASYNCHRONOUS`.

SEE ALSO

pthread_cancel(3)

STANDARDS

The **pthread_testcancel()** function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

PTHREAD_YIELD(3)

NAME

pthread_yield – yield control of the current thread

SYNOPSIS

```
#include <pthread.h>

void
pthread_yield(void);
```

DESCRIPTION

The **pthread_yield()** forces the running thread to relinquish the processor until it again becomes the head of its thread list.

SEE ALSO

sched_yield(2)

STANDARDS

The **pthread_yield()** is a non-portable (but quite common) extension to IEEE Std 1003.1-2001 (“POSIX.1”).

THRD_CREATE(3)

NAME

call_once, cnd_broadcast, cnd_destroy, cnd_init, cnd_signal, cnd_timedwait, cnd_wait, mtx_destroy, mtx_init, mtx_lock, mtx_timedlock, mtx_trylock, mtx_unlock, thrd_create, thrd_current, thrd_detach, thrd_equal, thrd_exit, thrd_join, thrd_sleep, thrd_yield, tss_create, tss_delete, tss_get, tss_set - C11 threads interface

SYNOPSIS

```
#include <threads.h>

void
call_once(once_flag *flag, void (*func)(void));

int
cnd_broadcast(cnd_t *cond);

void
cnd_destroy(cnd_t *cond);

int
cnd_init(cnd_t *cond);

int
cnd_signal(cnd_t *cond);

int
cnd_timedwait(cnd_t * restrict cond, mtx_t * restrict mtx,
         const struct timespec * restrict ts);

int
cnd_wait(cnd_t *cond, mtx_t *mtx);

void
mtx_destroy(mtx_t *mtx);

int
mtx_init(mtx_t *mtx, int type);

int
mtx_lock(mtx_t *mtx);

int
mtx_timedlock(mtx_t * restrict mtx, const struct timespec * restrict ts);
```

```
int
mtx_trylock(mtx_t *mtx);

int
mtx_unlock(mtx_t *mtx);

int
thrd_create(thrd_t *thr, int (*func)(void *), void *arg);

thrd_t
thrd_current(void);

int
thrd_detach(thrd_t thr);

int
thrd_equal(thrd_t thr0, thrd_t thr1);

_Noreturn void
thrd_exit(int res);

int
thrd_join(thrd_t thr, int *res);

int
thrd_sleep(const struct timespec *duration, struct timespec *remaining);

void
thrd_yield(void);

int
tss_create(tss_t *key, void (*dtor)(void *));

void
tss_delete(tss_t key);

void *
tss_get(tss_t key);

int
tss_set(tss_t key, void *val);
```

DESCRIPTION

As of ISO/IEC 9899:2011 ("ISO C11"), the C standard includes an API for writing multithreaded applications. Since POSIX.1 already includes a threading API that is used by virtually any multithreaded application, the interface provided by the C standard can be considered superfluous.

In this implementation, the threading interface is therefore implemented as a light-weight layer on top of existing interfaces. The functions to which these routines are mapped, are listed in the following table. Please refer to the documentation of the POSIX equivalent functions for more information.

Function	POSIX equivalent
call_once()	pthread_once(3)
cnd_broadcast()	pthread_cond_broadcast(3)
cnd_destroy()	pthread_cond_destroy(3)
cnd_init()	pthread_cond_init(3)
cnd_signal()	pthread_cond_signal(3)
cnd_timedwait()	pthread_cond_timedwait(3)
cnd_wait()	pthread_cond_wait(3)
mtx_destroy()	pthread_mutex_destroy(3)
mtx_init()	pthread_mutex_init(3)
mtx_lock()	pthread_mutex_lock(3)
mtx_timedlock()	pthread_mutex_timedlock(3)
mtx_trylock()	pthread_mutex_trylock(3)
mtx_unlock()	pthread_mutex_unlock(3)
thrd_create()	pthread_create(3)
thrd_current()	pthread_self(3)
thrd_detach()	pthread_detach(3)
thrd_equal()	pthread_equal(3)
thrd_exit()	pthread_exit(3)
thrd_join()	pthread_join(3)
thrd_sleep()	nanosleep(2)

thrd_yield() pthread_yield(3)

tss_create() pthread_key_create(3)

tss_delete() pthread_key_delete(3)

tss_get() pthread_getspecific(3)

tss_set() pthread_setspecific(3)

DIFFERENCES WITH POSIX EQUIVALENTS

The **thrd_exit()** function returns an integer value to the thread calling **thrd_join()**, whereas the **pthread_exit()** function uses a pointer.

The mutex created by **mtx_init()** can be of type `mtx_plain` or `mtx_timed` to distinguish between a mutex that supports **mtx_timedlock()**. This type can be or'd with **mtx_recursive** to create a mutex that allows recursive acquisition. These properties are normally set using **pthread_mutex_init()**'s *attr* parameter.

RETURN VALUES

If successful, the **cnd_broadcast()**, **cnd_init()**, **cnd_signal()**, **cnd_timedwait()**, **cnd_wait()**, **mtx_init()**, **mtx_lock()**, **mtx_timedlock()**, **mtx_trylock()**, **mtx_unlock()**, **thrd_create()**, **thrd_detach()**, **thrd_equal()**, **thrd_join()**, **thrd_sleep()**, **tss_create()** and **tss_set()** functions return `thrd_success`. Otherwise an error code will be returned to indicate the error.

The **thrd_current()** function returns the thread ID of the calling thread.

The **tss_get()** function returns the thread-specific data value associated with the given key. If no thread-specific data value is associated with key, then the value `NULL` is returned.

ERRORS

The **cnd_init()** and **thrd_create()** functions will fail if:

`thrd_nonmem` The system has insufficient memory.

The **cnd_timedwait()** and **mtx_timedlock()** functions will fail if:

`thrd_timedout` The system time has reached or exceeded the time specified in *ts* before the operation could be completed.

The **mtx_trylock()** function will fail if:

`thrd_busy` The mutex is already locked.

In all other cases, these functions may fail by returning general error code `thrd_error`.

SEE ALSO

nanosleep(2), pthread(3)

STANDARDS

These functions are expected to conform to ISO/IEC 9899:2011 ("ISO C11").

CEEPIPI interface

CEEPIPI(3)

NAME

ceepipi - LE Pre-initialization interface services

SYNOPSIS

```
#include <stdlib.h>
#include <machine/ceepipi.h>

int __CEEPIPI_init_main(__CEEPIT *ceexptbl_addr, void *service_rtns,
                                int *token);
int __CEEPIPI_init_main_dp(__CEEPIT *ceexptbl_addr, void *service_rtns,
                                int *token);
int __CEEPIPI_init_sub(__CEEPIT *ceexpttbl_addr, void *service_rtns,
                                char *runtime_opts, int *token);
int __CEEPIPI_init_sub_dp(__CEEPIT *ceexpttbl_addr, void *service_rtns,
                                char *runtime_opts, int *token);
int __CEEPIPI_call_main(int ceexptbl_index, int token,
                           char *runtime_opts, void *parm_list,
                           int *enclave_return_code,
                           int *enclave_reason_code,
                           __CEECTOK *appl_feedback_code);
int __CEEPIPI_call_sub(int ceexptbl_index, int token,
                           void *parm_list,
                           int *sub_ret_code,
                           int *sub_reason_code,
                           __CEECTOK *sub_feedback_code);
int __CEEPIPI_call_sub_addr(void *routine_addr, int token,
                            void *parm_list,
                           int *sub_ret_code,
                           int *sub_reason_code,
                           __CEECTOK *sub_feedback_code);
int __CEEPIPI_end_seq(int token);
int __CEEPIPI_start_seq(int token);
int __CEEPIPI_term(int token, int *env_return_code);
int __CEEPIPI_add_entry( int token, char *routine_name,
                                void **routine_entry,
                                int *ceexptbl_index);
int __CEEPIPI_delete_entry( int token, int ceexptbl_index);
int __CEEPIPI_identify_entry( int token, int ceexptbl_index,
                                int *prog_language);
int __CEEPIPI_identify_environment( int token, int *pipi_environment);
int __CEEPIPI_identify_attributes( int token, int ceexptbl_index,
                                          int *program_attributes);
```

```
int __CEEPIPI_set_user_word( int token, int value);
int __CEEPIPI_get_user_word( int token, int *value);
__CEEPIT *__CEEPIPI_alloc_CEEPIT( int flag, int n, ...);
```

DESCRIPTION

The Systems/C runtime provides an interface to IBM's LE Pre-initialization facility (CEEPIPI). Using this, Systems/C code can invoke LE functions, pass parameters to them and retrieve return values.

For details on the LE Pre-initialization interface, consult the IBM documentation "z/OS Language Environment Programming Guide" (SA22-7561).

LIST OF FUNCTIONS

Name	Appears on Page	Description
__CEEPIPI_init_main	__CEEPIPI_init_main(3)	initialize for main routines
__CEEPIPI_init_main_dp	__CEEPIPI_init_main_dp(3)	initialize for main routines (multiple environment)
__CEEPIPI_init_sub	__CEEPIPI_init_sub(3)	initialize for subroutine
__CEEPIPI_init_sub_dp	__CEEPIPI_init_sub_dp(3)	initialize for subroutine (multiple environment)
__CEEPIPI_call_main	__CEEPIPI_call_main(3)	invocation for main routine
__CEEPIPI_call_sub	__CEEPIPI_call_sub(3)	invocation for subroutines
__CEEPIPI_call_sub_addr	__CEEPIPI_call_sub_addr(3)	invocation for subroutines by address
__CEEPIPI_end_seq	__CEEPIPI_end_seq(3)	end a sequence of calls
__CEEPIPI_start_seq	__CEEPIPI_start_seq(3)	start a sequence of calls
__CEEPIPI_term	__CEEPIPI_term(3)	terminate environment
__CEEPIPI_add_entry	__CEEPIPI_add_entry(3)	add entry to the Preinit table
__CEEPIPI_delete_entry	__CEEPIPI_delete_entry(3)	delete entry from Preinit table
__CEEPIPI_identify_entry	__CEEPIPI_identify_entry(3)	identify an entry in the Preinit table
__CEEPIPI_identify_environment	__CEEPIPI_identify_environment(3)	identify the environment in the Preinit table
__CEEPIPI_identify_attributes	__CEEPIPI_identify_attributes(3)	identify the program attributes in the Preinit table
__CEEPIPI_set_user_word	__CEEPIPI_set_user_word(3)	set value to be used to initialize CAA user word
__CEEPIPI_get_user_word	__CEEPIPI_get_user_word(3)	get value to be used to initialize CAA user word
__CEEPIPI_alloc_CEEPIT	__CEEPIPI_alloc_CEEPIT(3)	create an populate a Preinit table

EXAMPLE

The following C code uses the CEEPIPI functions to invoke an LE C function named "HLLPIPI":

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <machine/ceepipi.h>

main()
{

   int rc;
   int token;
   __CEEPIT *cexptbl;
   int subretc, subrsnc;
   __CEECTOK subfbc;
   int env_rc;

   int parms[1];

   printf("Dignus C caller of LE HLLPIPI function\n");


   /*
    * Allocate and populate an LE Preinitialization table
    * This table has one entry, HLLPIPI
    */
   cexptbl = __CEEPIPI_alloc_CEEPIT(0, 1, "HLLPIPI");
   if(cexptbl == NULL) {
        printf("Out of memory allocating pre-init table");
        exit(12);
   }

   /*
    * Initialize an LE preinitialization subroutine environment
    */
   rc = __CEEPIPI_init_sub(cexptbl, NULL, NULL, &token);
   if(rc != 0) {
        printf("CEEPIPI init_sub failed - rc is %d\n", rc);
        exit(12);
   }

   /*
    * Call the subroutine HLLPIPI(int parm);
    *   The value being passed is "20"
```

```
    */
   parms[0] = 20;
   rc = __CEEPIPI_call_sub(0 /* function #0 */,
                           token, parms, &subretc, &subrsnc, &subfbc);

   printf("call_sub() returns %d\n", rc);
   printf("    subretc=%d, subrsnc=%d\n", subretc, subrsnc);

   if(rc != 0) {
        printf("CEEPIPI_call_sub failed - rc is %d\n", rc);
        exit(12);
   }

   /*
    * Terminate the environment
    */
   rc = __CEEPIPI_term(token, &env_rc);
   if(rc != 0) {
        printf("CEEPIPI_term failed - rc is %d\n", rc);
        exit(12);
   }

   /* free the pre-init table's memory */
   free(cexptbl);
}
```

The function HLLPIPI is written in C and compiled and linked in LE mode to produce a loadable LE module:

```
/*
* HLLPIPI is called by the C program
* using the LE preinitialization program
* subroutine call interface.
**/

#include <stdio.h>
#include <string.h>
#pragma linkage(HLLPIPI, fetchable)

int
HLLPIPI(int parm)
{
  int retval = 100;
  printf(" Enter HLLPIPI\n");
  printf("    This is an LE C program invoked from Dignus C!\n");
  printf("    The parameter passed was: %d\n", parm);
```

```
  printf(" Exit HLLPIPI - returning %d\n", retval+parm);
  retval = retval + parm;
  return retval;
}
```

__CEEPIPI_init_main(3)

NAME

__CEEPIPI_init_main - initialize for main routines

SYNOPSIS

```
#include <stdlib.h>
#include <machine/ceepipi.h>

int __CEEPIPI_init_main(__CEEPIT *ceexptbl_addr, void *service_rtns,
                                  int *token);
```

DESCRIPTION

The **__CEEPIPI_init_main()** function creates and ininitializes a LE runtime environment that allows for multiple executions of the main routine.

ceexptbl_addr pointers to a PreInit table used during initialization of the environment.

token points to a integer to return a value used to identify the newly created environment.

service_rtns contains a pointer to a service routines vector or `NULL`.

RETURN VALUES

On success, **__CEEPIPI_init_main()** returns `0`. Otherwise, it returns the return code specified in the IBM Language Environment Preinitialization Interface documentation.

SEE ALSO

For detailed information about the IBM Preinitialization interface, consult the IBM document "z/OS Language Environment Programming Guide".

__CEEPIPI_init_main_dp(3)

NAME

__CEEPIPI_init_main_dp - initialize for main routines (multiple environments)

SYNOPSIS

```
#include <stdlib.h>
#include <machine/ceepipi.h>

int __CEEPIPI_init_main_dp(__CEEPIT *ceexptbl_addr, void *service_rtns,
int *token);
```

DESCRIPTION

The **__CEEPIPI_init_main_dp()** function creates and ininitializes a Language Environment (LE) runtime environment that allows for multiple executions of the main routine. **__CEEPIPI_init_main_dp()** ensures that the environment tolerates the existence of multiple IBM Language Environment processes or enclaves, thus multiple main environments can be establised using **__CEEPIPI_init_main_dp()** where **__CEEPIPI_init_main()** can only establish one environment.

ceexptbl_addr pointers to a PreInit table used during initialization of the environment.

token points to a integer to return a value used to identify the newly created environment.

service_rtns contains a pointer to a service routines vector or `NULL`.

RETURN VALUES

On success, **__CEEPIPI_init_main_dp()** returns `0`. Otherwise, it returns the return code specified in the IBM Language Environment Preinitialization Interface documentation.

SEE ALSO

For detailed information about the IBM Preinitialization interface, consult the IBM document “z/OS Language Environment Programming Guide”.

__CEEPIPI_init_sub(3)

NAME

__CEEPIPI_init_sub - initialize for subroutines

SYNOPSIS

```
#include <stdlib.h>
#include <machine/ceepipi.h>

int __CEEPIPI_init_sub(__CEEPIT *ceexpttbl_addr, void *service_rtns,
                                 char *runtime_opts, int *token);
```

DESCRIPTION

The **__CEEPIPI_init_sub()** function creates and ininitializes a Language Environment (LE) runtime environment that allows for multiple executions of subroutines.

ceexptbl_addr pointers to a PreInit table used during initialization of the environment.

runtime_opts is a character string containing the LE runtime options. See the IBM "z/OS Language Environment Programming Reference" documentation for a list of available runtime options.

token points to a integer to return a value used to identify the newly created environment.

service_rtns contains a pointer to a service routines vector or `NULL`.

RETURN VALUES

On success, **__CEEPIPI_init_sub()** returns `0`. Otherwise, it returns the return code specified in the IBM Language Environment Preinitialization Interface documentation.

SEE ALSO

For detailed information about the IBM Preinitialization interface, consult the IBM document "z/OS Language Environment Programming Guide".

__CEEPIPI_init_sub_dp(3)

NAME

__CEEPIPI_init_sub_dp - initialize for subroutine (multiple environment)

SYNOPSIS

```
#include <stdlib.h>
#include <machine/ceepipi.h>

int __CEEPIPI_init_sub_dp(__CEEPIT *ceexpttbl_addr, void *service_rtns,
                                 char *runtime_opts, int *token);
```

DESCRIPTION

The **__CEEPIPI_init_sub_dp()** function creates and ininitializes a LE runtime environment that allows for multiple executions of subroutines. Unlike **__CEEPIPI_init_sub()**, **__CEEPIPI_init_sub_dp()** can establish multiple environments.

ceexptbl_addr pointers to a PreInit table used during initialization of the environment.

runtime_opts is a character string containing the LE runtime options. See the IBM "z/OS Language Environment Programming Reference" documentation for a list of available runtime options.

token points to a integer to return a value used to identify the newly created environment.

service_rtns contains a pointer to a service routines vector or `NULL`.

RETURN VALUES

On success, **__CEEPIPI_init_sub_dp()** returns `0`. Otherwise, it returns the return code specified in the IBM Language Environment Preinitialization Interface documentation.

SEE ALSO

For detailed information about the IBM Preinitialization interface, consult the IBM document "z/OS Language Environment Programming Guide".

__CEEPIPI_call_main(3)

NAME

__CEEPIPI_call_main - invocation for main routine

SYNOPSIS

```
#include <stdlib.h>
#include <machine/ceepipi.h>

int __CEEPIPI_call_main(int ceexptbl_index, int token,
                        char *runtime_opts, void *parm_list,
                        int *enclave_return_code,
                        int *enclave_reason_code,
                        __CEECTOK *appl_feedback_code);
```

DESCRIPTION

The **__CEEPIPI_call_main()** function invokes the specified main routine indicated by lvarceexptbl_index in the environment specified by *token*. After the routine returns, the environment becomes dormant.

ceexptbl_index indicates which entry in the environment's PreInit table is to be invoked.

token is the environment's identifier.

runtime_opts is a character string containing the LE runtime options. See the IBM "z/OS Language Environment Programming Reference" documentation for a list of available runtime options.

parm_list is either `NULL` or an array of parameter values passed to the main routine.

enclave_return_code points to an integer where the return code from the enclave is saved after the main routine completes.

enclave_reason_code points to an integer where the reason code from the enclave is saved after the main routine completes.

appl_feedback_code points to 96-bit condition token indicating why the application terminated.

RETURN VALUES

On success, **__CEEPIPI_call_main()** returns `0`. Otherwise, it returns the return code specified in the IBM Language Environment Preinitialization Interface documentation.

SEE ALSO

For detailed information about the IBM Preinitialization interface, consult the IBM document "z/OS Language Environment Programming Guide".

__CEEPIPI_call_sub(3)

NAME

__CEEPIPI_call_sub - invocation for subroutines

SYNOPSIS

```
#include <stdlib.h>
#include <machine/ceepipi.h>

int __CEEPIPI_call_sub(int ceexptbl_index, int token,
                           void *parm_list,
                           int *sub_ret_code,
                           int *sub_reason_code,
                           __CEECTOK *sub_feedback_code);
```

DESCRIPTION

The **__CEEPIPI_call_sub()** function invokes the specified sub routine indicated by lvarceexptbl_index in the environment specified by *token*.

ceexptbl_index indicates which entry in the environment's PreInit table is to be invoked.

token is the environment's identifier.

parm_list is either `NULL` or an array of parameter values passed to the sub routine.

sub_ret_code points to an integer where the return code from the enclave is saved after the sub routine completes.

sub_reason_code points to an integer where the reason code from the enclave is saved after the sub routine completes.

sub_feedback_code points to 96-bit condition token.

RETURN VALUES

On success, **__CEEPIPI_call_sub()** returns `0`. Otherwise, it returns the return code specified in the IBM Language Environment Preinitialization Interface documentation.

SEE ALSO

For detailed information about the IBM Preinitialization interface, consult the IBM document "z/OS Language Environment Programming Guide".

__CEEPIPI_call_sub_addr(3)

NAME

__CEEPIPI_call_sub_addr - invocation for subroutines by address

SYNOPSIS

```
#include <stdlib.h>
#include <machine/ceepipi.h>

int __CEEPIPI_call_sub_addr(void *routine_addr, int token,
                            void *parm_list,
                           int *sub_ret_code,
                           int *sub_reason_code,
                           __CEECTOK *sub_feedback_code);
```

DESCRIPTION

The **__CEEPIPI_call_sub_addr()** function invokes the specified sub routine indicated by lvarroutine_addr in the environment specified by *token*.

routine_addr is a pointer to a structure containing two pointers. The first pointer points to the function's entry point. Initially, the second pointer should be `NULL`. If the environment is XPLINK, the IBM Preinitilization services will provide a new pointer in the second pointer for directly invoking the function on a subsequent call. If the environment is XPLINK and the second pointer is not `NULL`, then the second pointer is used to directly invoke the function and avoid some overhead.

token is the environment's identifier.

parm_list is either `NULL` or an array of parameter values passed to the sub routine.

sub_ret_code points to an integer where the return code from the enclave is saved after the sub routine completes.

sub_reason_code points to an integer where the reason code from the enclave is saved after the sub routine completes.

sub_feedback_code points to 96-bit condition token.

RETURN VALUES

On success, **__CEEPIPI_call_sub_addr()** returns `0`. Otherwise, it returns the return code specified in the IBM Language Environment Preinitialization Interface documentation.

SEE ALSO

For detailed information about the IBM Preinitialization interface, consult the IBM document "z/OS Language Environment Programming Guide".

__CEEPIPI_end_seq(3)

NAME

__CEEPIPI_end_seq - end a sequence of calls

SYNOPSIS

```
#include <stdlib.h>
#include <machine/ceepipi.h>

int __CEEPIPI_end_seq(int token);
```

DESCRIPTION

The **__CEEPIPI_end_seq()** indicates the end of a sequence of uniterrupted subroutine calls for the given environment.

token is the environment's identifier.

RETURN VALUES

On success, **__CEEPIPI_end_seq()** returns `0`. Otherwise, it returns the return code specified in the IBM Language Environment Preinitialization Interface documentation.

SEE ALSO

For detailed information about the IBM Preinitialization interface, consult the IBM document "z/OS Language Environment Programming Guide".

__CEEPIPI_start_seq(3)

NAME

__CEEPIPI_start_seq - start a sequence of calls

SYNOPSIS

```
#include <stdlib.h>
#include <machine/ceepipi.h>

int __CEEPIPI_start_seq(int token);
```

DESCRIPTION

The **__CEEPIPI_start_seq()** function indicates that the program is to begin a sequence of uninterrupted calls into subroutines for the given environment. This minimizes overhead between calls.

token is the environment's identifier.

RETURN VALUES

On success, **__CEEPIPI_start_seq()** returns `0`. Otherwise, it returns the return code specified in the IBM Language Environment Preinitialization Interface documentation.

SEE ALSO

For detailed information about the IBM Preinitialization interface, consult the IBM document "z/OS Language Environment Programming Guide".

__CEEPIPI_term(3)

NAME

__CEEPIPI_term - terminate environment

SYNOPSIS

```
#include <stdlib.h>
#include <machine/ceepipi.h>

int __CEEPIPI_term(int token, int *env_return_code);
```

DESCRIPTION

The **__CEEPIPI_term()** function terminates the environment indicated by *token*.

token is the environment's identifier.

RETURN VALUES

On success, **__CEEPIPI_term()** returns `0`. Otherwise, it returns the return code specified in the IBM Language Environment Preinitialization Interface documentation.

SEE ALSO

For detailed information about the IBM Preinitialization interface, consult the IBM document "z/OS Language Environment Programming Guide".

__CEEPIPI_add_entry(3)

NAME

__CEEPIPI_add_entry - add an entry to the PreInit table

SYNOPSIS

```
#include <stdlib.h>
#include <machine/ceepipi.h>

int __CEEPIPI_add_entry( int token, char *routine_name,
                                 void **routine_entry,
                                 int *ceexptbl_index);
```

DESCRIPTION

The **__CEEPIPI_add_entry()** adds an entry to the environment indicated by *token*. If *routine_entry* is `NULL`, then the routine name given in *routine_name* is used. The index used is returned in *ceexptbl_index*.

__CEEPIPI_add_entry() does not extend the PreInitialization table.

token is the environment's identifier.

routine_name is a left-justified, blank-padded 8-character sequence of characters containing the name of the routine. When not used this field should be all blanks.

routine_entry is `NULL` to indicate that *routine_name* should be used; otherwise it is the address to be added to the table. The high-order bit of this address is used to indicate the AMODE. If the routine is succesfully added, this will be set to the address of the added routine.

On a successful routine, **ceexptl_index* will contain the index into the PreInitialization table of the added routine.

RETURN VALUES

On success, **__CEEPIPI_add_entry()** returns `0`. Otherwise, it returns the return code specified in the IBM Language Environment Preinitialization Interface documentation.

SEE ALSO

For detailed information about the IBM Preinitialization interface, consult the IBM document "z/OS Language Environment Programming Guide".

__CEEPIPI_delete_entry(3)

NAME

__CEEPIPI_delete_entry - delete an entry from the PreInit table

SYNOPSIS

```
#include <stdlib.h>
#include <machine/ceepipi.h>

int __CEEPIPI_delete_entry( int token, int ceexptbl_index);
```

DESCRIPTION

The **__CEEPIPI_delete_entry()** removes the routine specified by *ceexptbl_index* from the environment indicated by *token*.

RETURN VALUES

On success, **__CEEPIPI_delete_entry()** returns `0`. Otherwise, it returns the return code specified in the IBM Language Environment Preinitialization Interface documentation.

SEE ALSO

For detailed information about the IBM Preinitialization interface, consult the IBM document "z/OS Language Environment Programming Guide".

__CEEPIPI_identify_entry(3)

NAME

__CEEPIPI_identify_entry - Identify an entry in the PreInit table

SYNOPSIS

```
#include <stdlib.h>
#include <machine/ceepipi.h>

int __CEEPIPI_identify_entry( int token, int ceexptbl_index,
                              int *prog_language);
```

DESCRIPTION

The **__CEEPIPI_identify_entry()** indicates the the programming language of the function identified by *ceexptbl_index* for the environment indicated by *token*.

**prog_language* will contain the possible language codes, defined as:

`__CEEPIPI_C`	C/C++
`__CEEPIPI_COBOL`	COBOL
`__CEEPIPI_PLI`	PL/I
`__CEEPIPI_EPLI`	Enterprise PL/I for z/OS
`__CEEPIPI_ASM`	Language Environment-enabled assembler
`__CEEPIPI_PLX`	PL/X

RETURN VALUES

On success, **__CEEPIPI_identify_entry()** returns `0`. Otherwise, it returns the return code specified in the IBM Language Environment Preinitialization Interface documentation.

SEE ALSO

For detailed information about the IBM Preinitialization interface, consult the IBM document "z/OS Language Environment Programming Guide".

__CEEPIPI_identify_environment(3)

NAME

__CEEPIPI_identify_environment - identify the environment in the PreInit table

SYNOPSIS

```
#include <stdlib.h>
#include <machine/ceepipi.h>

int __CEEPIPI_identify_environment( int token, int *pipi_environment);
```

DESCRIPTION

The **__CEEPIPI_identify_environment()** indicates the type of environment that was preinitialized.

**prog_language* will contain the possible language codes, defined as:

`__CEEPIPI_ceepipi_main` PreInit main environment is initialized.

`__CEEPIPI_enclave_initialized` PreInit enclave is initialized.

`__CEEPIPI_dp_environment` PreInit sub db environment is intialized.

`__CEEPIPI_seq_of_calls_active` PreInit seq call function is active.

`__CEEPIPI_exits_established` PreInit sub dp exits is set.

`__CEEPIPI_sir_unregistered` PreInit sir is registered.

`__CEEPIPI_sub_environment` PreInit sub environment is initialized.

`__CEEPIPI_XPLINK_environment` PreInit XPLINK environment is initialized.

`__CEEPIPI_init_main_dp_environment` PreInit main dp environment is initialized.

RETURN VALUES

On success, **__CEEPIPI_identify_environment()** returns `0`. Otherwise, it returns the return code specified in the IBM Language Environment Preinitialization Interface documentation.

ERRORS

SEE ALSO

For detailed information about the IBM Preinitialization interface, consult the IBM document "z/OS Language Environment Programming Guide".

__CEEPIPI_identify_attributes(3)

NAME

__CEEPIPI_identify_attributes - identify the program attributes in the PreInit table

SYNOPSIS

```
#include <stdlib.h>
#include <machine/ceepipi.h>

int __CEEPIPI_identify_attributes( int token, int ceexptbl_index,
                                                int *program_attributes);
```

DESCRIPTION

On success, **__CEEPIPI_identify_attributes()** function identified the program attributes of the routine specified by *ceexptbl_index*.

**prog_language* will contain the possible language codes, defined as:

`__CEEPIPI_loaded_by_pipi` The Preinitialization entry was loaded by IBM Language Environment.

`__CEEPIPI_XPLINK_program` The loaded Preinitialization entry is an XPLINK program.

`__CEEPIPI_address_not_resolved` The Preinitialization entry could not be loaded.

RETURN VALUES

On success, **__CEEPIPI_identify_attributes()** returns `0`. Otherwise, it returns the return code specified in the IBM Language Environment Preinitialization Interface documentation.

SEE ALSO

For detailed information about the IBM Preinitialization interface, consult the IBM document "z/OS Language Environment Programming Guide".

__CEEPIPI_set_user_word(3)

NAME

__CEEPIPI_set_user_word - set value to be used to initialize CAA user word

SYNOPSIS

```
#include <stdlib.h>
#include <machine/ceepipi.h>

int __CEEPIPI_set_user_word( int token, int value);
```

DESCRIPTION

The **__CEEPIPI_set_user_word()** saves the given *value* to be used when the initial `CAA` thread is created when a main routine or subroutine is invoked.

RETURN VALUES

On success, **__CEEPIPI_set_user_word()** returns `0`. Otherwise, it returns the return code specified in the IBM Language Environment Preinitialization Interface documentation.

ERRORS

SEE ALSO

For detailed information about the IBM Preinitialization interface, consult the IBM document "z/OS Language Environment Programming Guide".

__CEEPIPI_get_user_word(3)

NAME

__CEEPIPI_get_user_word - get value to be used to initialize CAA user word

SYNOPSIS

```
#include <stdlib.h>
#include <machine/ceepipi.h>

int __CEEPIPI_get_user_word( int token, int *value);
```

DESCRIPTION

The **__CEEPIPI_get_user_word()** retrieves the current user-word value for the environment indicated by *token*.

On return, **value* will contain the current value used to initialize the `CAA` user word.

RETURN VALUES

On success, **__CEEPIPI_get_user_word()** returns `0`. Otherwise, it returns the return code specified in the IBM Language Environment Preinitialization Interface documentation.

SEE ALSO

For detailed information about the IBM Preinitialization interface, consult the IBM document "z/OS Language Environment Programming Guide".

__CEEPIPI_alloc_CEEPIT(3)

NAME

__CEEPIPI_alloc_CEEPIT - allocate and initialize a PreInit table

SYNOPSIS

```
#include <stdlib.h>
#include <machine/ceepipi.h>

__CEEPIT *__CEEPIPI_alloc_CEEPIT( int flag, int n, ...);
```

DESCRIPTION

__CEEPIPI_alloc_CEEPIT() dynamically allocates a PreInitialization table sufficiently large to contain *n* entries. Following *n* should be n parameters that are either `NULL` or a pointer to a character string for the function name.

For example,

```
__CEEPIT *ceexptbl;

ceexptbl = __CEEPIPI_alloc_CEEPIT(0, 2, "FUNC", NULL);
```

allocates a table sufficiently large for 2 entries, the first entry will be initialized with the function name "`FUNC`".

If there is insufficient available memory, **__CEEPIPI_alloc_CEEPIT()** returns `NULL`.

A non-`NULL` address returned by **__CEEPIPI_alloc_CEEPIT()** should be returned using the free(3) function.

RETURN VALUES

On success, **__CEEPIPI_alloc_CEEPIT()** returns a pointer to the allocated memory, or `NULL` if space was not available.

SEE ALSO

For detailed information about the IBM Preinitialization interface, and the format of the PreInitialization table, consult the IBM document "z/OS Language Environment Programming Guide".

Keyed Access (VSAM) I/O

VSAMIO(3)

NAME

vsamio - keyed access (VSAM) input/output library functions

SYNOPSIS

```
#include <fcntl.h>
#include <unistd.h>
#include <machine/vsamio.h>

KFILE * kopen (const char *ddn, const char *parms, int mode);
int kclose (KFILE *k);
int kretrv (void *rec, void *key, int flags, KFILE *k);
int ksearch (const void *key, size_t keylen, int flags, KFILE *k);
int kinsert (const void *rec, size_t length, void *key, KFILE *k);
int kdelete (const void *key, KFILE *k);
int kreplace (const void *rec, size_t length, KFILE *k);
int kgetpos (KFILE *f, kpos_t *pos);
int ksetpos (KFILE *f, const kpos_t *pos);
int kdata (KFILE *k, __KFILE_data *d);
int kerrinfo (KFILE *k, char *RPLRTNCD, char *RPLCMPON, char *RPLERRCD);

ssize_t kread (KFILE *k, void *buf, size_t nbytes):
ssize_t kwrite (KFILE *k, void *buf, size_t nbytes):
off_t kseek (KFILE *k, off_t offset, int whence):
```

DESCRIPTION

The Systems/C runtime supports accessing VSAM files via the VSAMIO functions.

Unlike other mainframe C implementations; this access is not bound with the Standard C input/output functions. The Systems/C runtime supports a more native VSAM facility which more closely matches the z/OS VSAM function.

Opening a VSAM file is accomplished with the kopen(3) function, that returns a pointer to a `KFILE` "handle". A `KFILE` pointer can then be passed to the other VSAMIO functions, described here, to read records, search the file, add records, close the file, etc...

Note that VSAMIO is only supported on z/OS, and only for AMODE 31 programs.

For many of the functions described here, the Systems/C runtime library will map the VSAM logical errors to `errno` values as well as the specific `errno` values provided in the function descriptions:

[ENOSPC] #4 - end of data set

[EEXIST] #8 - duplicate key

[EILSEQ] #12 - an attempt was made to perform sequential or skip-sequential processing against a record whose key/record number does not follow the proper ascending/descending sequential order.

[ENOENT] #16 - record not found.

[ENOLCK] #20 - control Interval exclusive use conflict

[ENOLCK] #21, #22 - for RRS, another LUWID holds the lock

[ENXIO] #24 - record lives on volume that connot be mounted

[ENOSPC] #28 - can't extend dataset because there's no space.

[ENOENT] #32- XRBA specified does not address any record

[EINVAL] #36 - key ranges were specified for the data set when it was defined, but no range was specified that includes the record to be inserted.

[ENOMEM] #40 - no memory.

[ENOMEM] #44 - work area not large enough for the record.

[EINVAL] #48, #52- invalid options, data set attributes or processing conditions.

[EIO] #56 - ACB or LUWID changed underneath the RPL (an RPL reused violation).

[ENOMEM] #64 - no space to add another string.

[EINVAL] #68 - invalid type of processing (i.e. read from OUTPUT)

[EINVAL] #72 - keyed request for ESDS, or GETIX/PUTIX to ESDS or flexed-length RRDS. For RLS - GETIX/PUTIX issued.

[EINVAL] #76 - issued an addressed or CI PUT to add a KSDS or variable-length RRDS, or, issued a constrol interval PUT to a fixed-length RRDS.

[EINVAL] #80 - invalid ERASE request.

[EINVAL] #84 - invalid OPTCD=LOC for PUT.

[ESPIPE] #88 - sequential GET request without proper positioning, or, illegal switch between forward/backward processsing

[EINVAL] #92 - issued PUT for update, or ERASE without previous GET for update, or PUTIX without previous GETIX

[EINVAL] #96 - change the primary key will making an update.

[EINVAL] #100 - changed the length of the record while making update.

[EINVAL] #104 - invalid RPL.

[EINVAL] #108 - invalid RECLEN.

[EINVAL] #112 - KEYLEN too large or is zero.

[EINVAL] #116 - initial loading of empty data set not allow UPDATE (OPTCD=UPD) mode.

[EINVAL] #120 - request is operating under wrong TCB.

[ECANCELED] #124 - request was cancelled by user JRNAD exit.

[ELOOP] #128 - a loop was discovered in index.

[EINVAL] #132 - attempt, in locate mode, to retrieve a spanned record.

[EINVAL] #136 - attemped an addressed GET of a spanned record in a KSDS.

[ENOENT] #144 - invalid pointer (no associated base record) in an alternate index.

[ENOSPC] #148 - maximum number of pointers is alternate index exceeded.

[ENOMEM] #152 - not enough buffers.

[EIO] #156 - invalid control-interval discovred during Keyed processing.

[EBUSY] #160 - buffer multiply marked written.

[EFAULT] #168 - for RLS, the pointer in the RPL is zero.

[EIO] #180 - for RLS, an invalid request for a non-recoverable data set.

[EIO] #184 - for RLS, an ABEND occured during processing.

[ECANCELED] #185 - for RLS, user task cancelled while request was being processed.

[ENOSPC] #186 - End-of-Volume init failed when attempted to extend.

[EIO] #187 - for RLS, error occured with partial EOV processing.

[EIO] #188 - for RLS, the sphere is in lost locks state.

[EINVAL] #192 - invalid relative record number.

[EINVAL] #196 - addressed request for fixed- or variable-length RRDS.

[EINVAL] #200 - attempted addressed or control-interval access through a path.

[EINVAL] #204 - PUT insert requests (or for RLS, IDALKADD requests) are not allowed in backward mode.

[EBUSY] #208 - ENDREQ macro isseud against an RPL that has an outstanding WAIT against its associated ECB.

[ENOSPC] #212 - During split processing, an existing condition prevents the split - Index or data control interval size may need to be increased.

[EIO] #218 - unrecognized return code.

[EINVAL] #224 - MRKBFR OUT was issued for a buffer with invalid contents.

[EPERM] #228 - cross-memory issues.

[ENOSPC] #229 - record length changed during decompression.

[EIO] #230 - processing environment changed by the user of UPAP exit.

[EPERM] #232 - UPAD error; ECB was not posted by user in cross-memory.

[EINVAL] #236 - validity check of errof re SHAROPTIONS 3 or 4.

[EIO] #237, #238, #239, #241, #242, #243 - reserved.

[EAGAIN] #240 - for shared resources, one of the following is being performed: (1) an attempt is being made to obtain a buffer in exclusived contro, (2) a buffer is being invalidated, or (3) the buffer use chain is chaking. For more detailed feedback, reissue the request.

[EFAULT] #244 - register 14 stack size is not large enough.

[EIO] #246 - severe error returned by decompression management.

[EIO] #250 - no valid dictionary token exists for the data set. VSAM is unable to decompress the data record.

[EINVAL] #252 - record mode processing not allowed for LDS.

[EINVAL] #253 - VERIFY is not valid for LDS.

[EBUSY] #254 - I/O activity on the data set not quiesced before WRTBTF TYPE=DS issued.

LINEAR DATA SETS

Unlike other keyed-access files, a Linear Data Set represents a stream of bytes, processed in 4K blocks with no other structure. There is no record format, and no key (beyond a simple offset value) for accessing particular sections of data. Linear Data Sets are often used in data base environments and to implement z/OS USS file systems.

The kopen(3) function can be used to open a Linear Data Set. In that case, the Data In Virtual (DIV) macros are used to perform I/O to the file, not the normal VSAM

related system macros. (For more information about DIV see the IBM manual "z/OS Application Development Guide.")

DIV access to the Linear Data set provides a mechanism to map sections of the file into memory via "windows". Each window represents bytes from the file beginning at a certain offset for a particular number of bytes. The `bufsize` and `bufmax` parms of the kopen(3) function specify the size and number of DIV windows the system will manage to access the Linear Data Set.

The kread(3), kwrite(3) and kseek(3) functions are used with Linear Data Sets, in lieu of the other keyed access functions. These provide mechanisms for reading and writing a stream of bytes or positioning to a given offset. Besides kopen(3) and kclose(3), these are the only functions that can be used for I/O with Linear Data Sets.

When kopen(3) opens a Linear Data set, the system allocates the first window (if no space can be allocated, then the kopen(3) will fail.) After that, windows of the given size are allocated until the total count reaches the specified limit, or allocations fail due to unavailable memory. When no more DIV windows can be used, the system reuses existing windows in a least-recently-used fashion.

The kwrite(3) function does not directly cause output to the Linear Data Set, the actual writing to the file system occurs when the DIV window is released, either on reuse or when the kclose(3) function closes the Linear Data Set.

LIST OF FUNCTIONS

Name	Appears on Page	Description
kclose	kclose(3)	close a keyed-access, or Linear Data Set file
kdata	kdata(3)	provide information about an open keyed-access file
kdelete	kdelete(3)	delete the previously retrieved record
kgetpos	kgetpos(3)	return the position of the current record
kerrinfo	kerrinfo(3)	provide information about last logical error
kinsert	kinsert(3)	insert a record into a keyed-access file
kopen	kopen(3)	open a keyed-access or Linear Data Set file
kreplace	kreplace(3)	replace the previously retrieved record
kretrv	kretrv(3)	retrieve the next record
ksearch	ksearch(3)	search for a record with a specified key
ksetpos	ksetpos(3)	set the position to a previous saved value

Name	Appears on Page	Description
kread	kread(3)	read data from a Linear Data Set (LDS)
kseek	kseek(3)	position to a particular byte offset in a Linear Data Set (LDS)
kwrite	kwrite(3)	write data to a Linear Data Set (LDS)

KCLOSE(3)

NAME

kclose - close a keyed-access or Linear Data Set file.

SYNOPSIS

```
#include <fcntl.h>
#include <machine/vsamio.h>

int kclose (KFILE *k);
```

DESCRIPTION

The **kclose()** function closes a previously opened keyed-access or Linear Data Set file.

RETURN VALUES

If there is an error or an invalid KFILE was specified, **kopen()** returns `-1`; otherwise a `0` value is returned, indicating success.

ERRORS

When a failure is returned, `errno` is set to one of the following values:

`[EIO]`	The VSAM `CLOSE` macro indicated a non-zero return code.
`[EFAULT]`	The specified KFILE *k* was an invalid pointer.
`[EBADF]`	The specified KFILE *k* was not open.

SEE ALSO

kopen(3)

KDATA(3)

NAME

kdata - return information about an open keyed-access file

SYNOPSIS

```
#include <fcntl.h>
#include <machine/vsamio.h>

int kdata (KFILE *k, __KFILE_data *d)
```

DESCRIPTION

The **kdata()** function populates the __KFILE_data structure addressed by *d* from the KFILE pointer *k*.

The __KFILE_data structure (shown below) is defined in <machine/vsamio.h>.

```
typedef struct __KFILEdata {
                unsigned int access:2;
#define _KFILE_ACCESS_UNKNOWN (0)
#define _KFILE_ACCESS_SEQUENTIAL (1)
#define _KFILE_ACCESS_DIRECT (2)
#define _KFILE_ACCESS_SKIP_SEQUENTIAL (3)
                unsigned int vsam_org:5;
#define _KFILE_VSAM_ORG_UNKNOWN (0)
#define _KFILE_VSAM_ORG_KSDS (1)
#define _KFILE_VSAM_ORG_RRDS (2)
#define _KFILE_VSAM_ORG_ESDS (3)
#define _KFILE_VSAM_ORG_LDS  (4)
                unsigned int mode:4;
#define _KFILE_VSAM_MODE_UNKNOWN (0)
#define _KFILE_VSAM_MODE_INPUT (1)
#define _KFILE_VSAM_MODE_OUTPUT (2)
#define _KFILE_VSAM_MODE_UPDATE (3)
                unsigned char *ddname;
                unsigned int vsam_keylen;
                unsigned int vsam_keyoff;
                unsigned int reclen;
                unsigned int last_read_reclen;
} __KFILE_data;
```

The fields of the __KFILE_data structure are:

access	How the VSAM file may be accessed. The access field will be set to one of _KFILE_ACCESS_UNKNOWN, _KFILE_ACCESS_SEQUENTIAL, _KFILE_ACCESS_DIRECT, or _KFILE_ACCESS_SKIP_SEQUENTIAL,
vsam_org	VSAM file organization. The vsam_org field will be set to one of _KFILE_VSAM_ORG_UNKNOWN, _KFILE_VSAM_ORG_KSDS, _KFILE_VSAM_ORG_RRDS, _KFILE_VSAM_ORG_ESDS, or _KFILE_VSAM_ORG_LDS.
mode	I/O mode. The mode field will be set to one of _VSAM_MODE_UNKNOWN, _VSAM_MODE_INPUT, _VSAM_MODE_OUTPUT or _VSAM_MODE_UPDATE,
ddname	Pointer to the DDNAME used to open the file. This space will be overwritten on subsequent invocations of **kdata()**.
vsam_keylen	Key length specified at open or 0.
vsam_keyoff	Offset of key specified at open or 0.
reclen	Record length specified at open.
last_read_reclen	Length of the last record read.

RETURN VALUES

On success, **kdata()** returns 1. If there was an error, **kdata()** returns 0 and sets the errno value.

ERRORS

When a failure is returned, errno is set to one of the following values.

[EFAULT]	The specified KFILE *k* or __KFILE_data *d* was an invalid pointer.

SEE ALSO

kopen(3)

KDELETE(3)

NAME

kdelete - delete the last record retrieved

SYNOPSIS

```
#include <fcntl.h>
#include <machine/vsamio.h>

int kdelete (const void *key, KFILE *k);
```

DESCRIPTION

The **kdelete()** function removes the previously retrieved record from the keyed-access file *k*.

The previously retrieved record must be accessed with the kretrv(3) function.

The *key* parameter is currently ignored.

RETURN VALUES

On success, **kdelete()** returns `0`. If there was an error, **kdelete()** returns `-1` and sets the `errno` value.

ERRORS

When a failure is returned, `errno` is set to one of the following values, as well as the generic mapping of VSAM logical errors to `errno` values.

`[EFAULT]`	The specified KFILE *k* was an invalid pointer.
`[EBADF]`	The specified KFILE *k* was not open.
`[EIO]`	There was no previously retrieved record.
`[EIO]`	The return value from the `ERASE` macro was greater than 8.
`[ENOSYS]`	The RPL for the VSAM file could not be accessed or modified.

SEE ALSO

kretrv(3), kreplace(3)

KERRINFO(3)

NAME

kerrinfo - provide information about last logical error

SYNOPSIS

```
#include <fcntl.h>
#include <machine/vsamio.h>

int kerrinfo (KFILE *k, char *RPLRTNCD, char *RPLCMPON, char *RPLERRCD);
```

DESCRIPTION

When a VSAM logical error occurs, the VSAM library executes a VSAM `SHOWCB FIELDS=FDBK` macro to discern the reasons for the logical error and set the `errno` value appropriately.

The **kerrinfo()** function returns the values set by that `SHOWCB` invocation.

If the keyed-access file function returns an error indication, the **kerrinfo()** function can be used to learn more details about the error condition.

These values are set to zero before every VSAM operation, so if the values are zero, there is no more information available, or the error condition was unrelated to VSAM functions.

RETURN VALUES

On success, **kerrinfo()** returns `1`. If there was an error, **kerrinfo()** returns `0` and sets the `errno` value.

ERRORS

When a failure is returned, `errno` is set to one of the following values.

`[EFAULT]` The specified `KFILE` *k* was an invalid pointer.

`[EBADF]` The specified `KFILE` *k* was not open.

`[EIO]` The internal VSAM error information could not be located.

KGETPOS(3)

NAME

kgetpos - return the position of the current record

SYNOPSIS

```
#include <fcntl.h>
#include <machine/vsamio.h>

int kgetpos (KFILE *k, kpos_t *pos);
```

DESCRIPTION

The **kgetpos()** function determines the current record location and places that into the `kpos_t` pointer *pos*.

The value in *pos* can subsequently be used by the ksetpos(3) function to "seek" to this location.

The `kpos_t` is purposefully defined in an opaque manner to allow flexibility for changes to the positioning mechanisms in the future. There should be no assumption about the type or values of `kpos_t` values.

RETURN VALUES

On success, **kgetpos()** returns `0`. If there was an error, **kgetpos()** returns `-1` and sets the `errno` value.

ERRORS

When a failure is returned, `errno` is set to one of the following values, as well as the generic mapping of VSAM logical errors to `errno` values.

`[EIO]`	The VSAM `SHOWCB` macro failed.
`[EFAULT]`	The specified `KFILE` *k* or `kpos_t` *pos* was an invalid pointer.
`[EBADF]`	The specified `KFILE` *k* was not open.
`[ENOSYS]`	The RPL for the VSAM file could not be accessed or modified.

SEE ALSO

ksetpos(3)

KINSERT(3)

NAME

kinsert - Insert a record into a key-access file.

SYNOPSIS

```
#include <fcntl.h>
#include <machine/vsamio.h>

int kinsert (void *rec, size_t length, void *key, KFILE *k);
```

DESCRIPTION

The **kinsert()** function inserts a record into the keyed-access file *k*.

The *rec* parameter is a pointer to the record to insert, of *length* bytes. Note that if the file is an ESDS or RRDS file, then *length* must include the 4-byte key prefix.

The *key* parameter is the key to use for insertion. If *key* is `NULL`, then the insertion key is taken from the given record using the values specified when the file was **kopen**'d. For ESDS files, *key* points to the area to provide the new VSAM-assigned key.

RETURN VALUES

On success, **kinsert()** returns `0`. If there was an error, **kinsert()** returns `-1` and sets the `errno` value.

ERRORS

When a failure is returned, `errno` is set to one of the following values, as well as the generic mapping of VSAM logical errors to `errno` values.

`[EEXIST]`	The key is a duplicate and duplicate keys are not allowed.
`[EIO]`	The VSAM `PUT` or `SHOWCB` macros failed.
`[EFAULT]`	The specified KFILE *k* was an invalid pointer.
`[EBADF]`	The specified KFILE *k* was not open.

[EBADF] The given *keylen* was not zero and the VSAM organization was not KSDS.

[ENOSYS] The RPL for the VSAM file could not be accessed or modified.

KOPEN(3)

NAME

kopen - open a keyed-access or Linear Data Set file.

SYNOPSIS

```
#include <fcntl.h>
#include <machine/vsamio.h>

KFILE * kopen (const char *ddn, const char *parms, int mode);
```

DESCRIPTION

The **kopen()** function opens a keyed-access or Linear Data Set file for input, output or update.

The file to open is specified via the `DD` name *ddn*. The name should be a simple DD name, with no Systems/C prefix. It must be a name which is associated with the VSAM file. The *parms* parameter is a pointer to a string that defines the type of file to open, and several options that may apply. The *mode* parameter describes the direction of the operation; either `O_RDONLY` for read access, `O_WRONLY` for write access or `O_RDWR` for update access.

kopen() is roughly analagous to open(2).

The `KFILE` pointer returned is a handle which can be passed to other keyed-access I/O functions.

The *parms* parameter is a pointer to a constant character string that contains a comma separate list of parameters. Each parameter is of the form *NAME=VALUE* pairs. The following names and values are current supported:

`recfm`	specifies the record format, values are `v`, `f` and `u`. VSAM RRDS and LDS are RECFM=F, all other VSAM files are RECFM=V.
`reclen`	specifies the record format, values are an integer value indicating the record length, or the special symbol `x` for RECFM=V files. VSAM LDS is always reclen=4096.
`blksize`	specifies the block size, values are an integer value indicating the requested size.

`bufsize` specifies the size, in bytes, of the windows for a Linear Data Set (LDS). The integer value specified will be rounded up to a multiple of 4K. The default value is 262144 (256K).

`bufmax` specifies the maximum number of DIV windows for a Linear Data Set (LDS) - default is 4.

`keyoff` specifies the offset of the key embedded in the VSAM record. The value can be zero for non-key files. For ESDS and RRDS files, the value must be zero. If the value is not specified for KSDS files, the value found in the system catalogue will be used.

`keyoff` specifies the offset of the key embedded in the VSAM record. The value can be zero for non-key files. For ESDS and RRDS files, the value must be zero. If the value is not specified for KSDS files, the value found in the system catalogue will be used.

`keylen` specifies the length of the key embedded in the VSAM record. If the value is not specified for KSDS files, the value found in the system catalogue will be used.

`org` specifies the VSAM organization. The value is a two character field, where "`ks`" indicates KSDS, "`es`" indicates ESDS, "`rr`" indicates RRDS and "`ls`" indicates LDS.

`org` specifies the VSAM organization. The value is a two character field, where "`ks`" indicates KSDS, "`es`" indicates ESDS, "`rr`" indicates RRDS and "`ls`" indicates LDS.

`access` specifies the kind of VSAM access is to be used. The value is a three character field, where "`seq`" indicates sequential access, "`skp`" indicates skip access, and "`dir`" indicates direct access.

`unsafeshare` indicates that the system should allow sharing kopen() of an LDS file without `SHAREOPTIONS(1,3)` specified in the file allocation.

Other values specified in the *parms* parameter are silently ignored.

RETURN VALUES

If the **kopen()** was successful, a pointer to the allocated `KFILE` handle is returned; otherwise the `NULL` pointer is returned and `errno` is set.

ERRORS

Access to the file is denied if:

`[EACCESS]` The DIV ACCESS macro indicated access was denied when opening an LDS file.

`[EBUSY]` The DIV ACCESS macro indicated the file is being opened by multiple tasks without `SHAREOPTIONS(1,3)` being specified in the allocation. Use "unsafeshare" to ignore this error.

`[ENOMEM]` Insufficient memory was available.

`[ENOSYS]` A DFSMS macro or facility indicated failure.

`[EIO]` A DFSMS or DIV macro or facility indicated failure after VSAM file was successfully opened.

`[EFAULT]` The given *ddn* or *parms* value was `NULL`.

`[EFTYPE]` A kopen() of an LDS file with the `O_RDONLY` mode was specified, but the source file is empty. An empty LDS file must be opened with O_RDWR or O_WRONLY.

`[EINVAL]` An invalid *ddn*, *parms* or *mode* paramter was passed. Or, the specified `"org="`, `"keylen="` or `"keyoff="` value in the *parms* did not match the information returned by the system Catalog Search Interface. Or, the *mode* value was invalid for the specified `"access="` value.

`[EFTYPE]` An invalid `"org="` value was specified in *parms*.

`[ENOENT]` The specified *ddn* isn't allocated, or the allocation points to an incorrect `DSN` that was not found in the system catalog. `ENOENT` can also be set if the `OPEN` system service succeeded with a return code of 4, in which case the file is closed with the `CLOSE` system service and `NULL` is returned.

`[ENOATTR]` The Catalog Search Interface could not locate the information for the file.

SEE ALSO

kopen(3)

KREAD(3)

NAME

kread - read input from a Linear Data Set

SYNOPSIS

```
size_t
kread(KFILE *k, void *buf, size_t nbytes)
```

DESCRIPTION

kread() attempts to read *nbytes* of data from the Linear Data Set referenced by the KFILE pointer *k* into the buffer pointed to by *buf*.

The **kread()** starts at a position given by the pointer associated with *k* (see kseek(2)). Upon return from **kread()**, the pointer is incremented by the number of bytes actually read.

Upon successful completion, **kread()** returns the number of bytes actually read and placed in the buffer.

IMPLEMENTATION NOTES

kread() only does "binary" reading, no attempt is made to restructure the data into records.

kread() uses the `DIV` macro to process "windows" into the Linear Data Set. The *bufsize* and *bufmax* parms on the associated kopen(3) can adjust the size and number of those windows which can affect performance. When the number of windows is exhausted, the least recently used window will be re-used.

RETURN VALUES

If successful, the number of bytes actually read is returned. Upon exhausing the input data (end-of-file), zero is returned. Otherwise, a -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

kread() will succeed unless:

`[EBADF]`	*k* is not a valid `KFILE` pointer open for reading.
`[EINVAL]`	The KFILE pointed to by *k* is not a Linear Data Set.
`[EFAULT]`	*buf* points outside the allocated address space.
`[EIO]`	An low-level error occurred while processing DIV windows.
`[ENOMEM]`	Inadequate memory was available to set up various internal data structures.

SEE ALSO

kopen(3), kseek(3)

KREPLACE(3)

NAME

kreplace - replace the last record retrieved

SYNOPSIS

```
#include <fcntl.h>
#include <machine/vsamio.h>

int kreplace (void *rec, size_t length, KFILE *k);
```

DESCRIPTION

The **kreplace()** function replaces, in place, the previous record retrieved via kretrv(3) with the record specified by *rec* of *length* bytes.

If the file is `ESDS` or `RSDS` then *length* includes the 4-byte prefix.

RETURN VALUES

On success, **kreplace()** returns `0`. If there was an error, **kreplace()** returns `-1` and sets the `errno` value.

ERRORS

When a failure is returned, `errno` is set to one of the following values, as well as the generic mapping of VSAM logical errors to `errno` values.

`[EFAULT]`	The specified KFILE *k* was an invalid pointer.
`[EBADF]`	The specified KFILE *k* was not open.
`[EIO]`	There was no previously retrieved record.
`[EIO]`	The return value from the `PUT` macro was greater than 8.
`[ENOSYS]`	The RPL for the VSAM file could not be accessed or modified.

SEE ALSO

kretrv(3), kdelete(3)

KRETRV(3)

NAME

kretrv - Retrieve the next record from the keyed-acccess file.

SYNOPSIS

```
#include <fcntl.h>
#include <machine/vsamio.h>

int kretrv (void *rec, void *key, int flags, KFILE *k);
```

DESCRIPTION

The **kretrv()** function returns the next record from the specified `KFILE` *k*. The record is stored in the area addressed by the *rec* parameter.

If the *key* pointer is non-`NULL`, then **kretrv()** will save the record's key in the memory addressed by *key*.

Note that no length parameters are specified, the program must ensure that the memory addressed by *rec* and *key* is sufficient to contain the largest record and key from the file.

The *flags* value can be one of `K_backwards` indicating the previous record in the file should be returned, or `K_noupdate` indicating that the program does not intend to alter the records in the file. These values can be logically OR'd together. If `K_backwards` isn't specified, then the current record is returned and for sequential access, the position advances to the next record.

RETURN VALUES

The **kretrv()** function returns the length of the retrieved record. This length includes a 4-byte key prefix for ESDS and RRDS files. If the end-of-file (or begining if `K_backwards` is specified) is reached, **kretrv()** returns `0`. **kretrv()** returns `-1` on error.

ERRORS

When a failure is returned, `errno` is set to one of the following values:

`[EIO]`	The VSAM `GET` or `SHOWCB` macros failed.
`[EFAULT]`	The specified KFILE *k* was an invalid pointer.
`[EBADF]`	The specified KFILE *k* was not open.
`[ENOSYS]`	The RPL for the VSAM file could not be accessed or modified.

SEE ALSO

kopen(3), ksearch(3), kdelete(3)

KSEARCH(3)

NAME

ksearch - search a keyed-access file based on key.

SYNOPSIS

```
#include <fcntl.h>
#include <machine/vsamio.h>

int ksearch (const void *key, size_t keylen, int flags, KFILE *k);
```

DESCRIPTION

The **ksearch()** function searches for records in the specified `KFILE` *k*, returning `1` if the record was found, `0` or `-1` otherwise. If the record is found, the file position is set to that record, so it can be retrieved with the **kretrv** function.

The *key* parameter is a pointer to the bytes to be used for the search key, up to *keylen* bytes in length.

The *flags* value indicates how the search should proceed. The specified value in *flags* can be `K_exact`, `K_backwards`, `K_noupdate` and can be logically OR'd together:

`K_exact`	Indicates that the given record must match exactly, up to *keylen* bytes for a generic search. If `K_exact` is not specified, the first record found with a key that is greater than or equal (less than or equal for a backwards searches). `K_exact` must be specified for ESDS files.
`K_backwards`	The search is performed in descending key order.
`K_noupdate`	The program does not intend to replace or delete records, thus a subsequent **kretr()** is needed.

RETURN VALUES

The **ksearch()** function returns `1` if a matching record is located, `0` if no record is located, and `-1` if there is an error.

ERRORS

When a failure is returned, `errno` is set to one of the following values:

`[EIO]`	The VSAM `POINT`, `SHOWCB` macros failed, or the returned value from VSAM could not be interpreted.
`[EINVAL]`	A *keylen* was specified by the *key* pointer is `NULL`.
`[EFAULT]`	The specified KFILE *k* was an invalid pointer.
`[EBADF]`	The specified KFILE *k* was not open.
`[EBADF]`	The given *keylen* was not zero and the VSAM organization was not KSDS.
`[ENOSYS]`	The RPL for the VSAM file could not be accessed or modified.

SEE ALSO

kopen(3), kretrv(3)

KSEEK(2)

NAME

kseek - reposition read/write file offset of Linear Data Set

SYNOPSIS

```
#include <unistd.h>
#include <machine/vsamio.h>

off_t
kseek(KFILE *k, off_t offset, int whence)
```

DESCRIPTION

The **kseek()** function repositions the offset of the Linear Data Set specified in the KFILE pointer *k* to the argument offset according to the directive *whence*. The argument *k* must be an open KFILE. **kseek()** repositions the file position pointer associated with the KFILE pointer *k* as follows:

- If whence is `SEEK_SET`, the offset is set to offset bytes.
- If whence is `SEEK_CUR`, the offset is set to its current location plus offset bytes.
- If whence is `SEEK_END`, the offset is set to the size of the file plus offset bytes.

If the KFILE was opened for output (`O_WRONLY`) then the **kseek()** function allows the file offset to be set beyond the end of the existing end-of-file of the file. If data is later written at this point, subsequent reads of the data in the gap return bytes of zeros (until data is actually written into the gap).

RETURN VALUES

Upon successful completion, **kseek()** returns the resulting offset location as measured in bytes from the beginning of the file. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

ERRORS

kseek() will fail and the file position pointer will remain unchanged if:

[EBADF]	*k* is not an open file KFILE pointer, or *k* does not specify an LDS file.
[EINVAL]	*Whence* is not a proper value.

SEE ALSO

dup(2), open(2)

KSETPOS(3)

NAME

ksetpos - set the position of the keyed-access file

SYNOPSIS

```
#include <fcntl.h>
#include <machine/vsamio.h>

int ksetpos (KFILE *k, const kpos_t *pos);
```

DESCRIPTION

The **ksetpos()** function repositions the keyed-access file *k* to the position addressed by the point *pos*.

The value specified via *pos* was previously obtained in a kgetpos(3) function call.

The `kpos_t` is purposefully defined in an opaque manner to allow flexibility for changes to the positioning mechanisms in the future. There should be no assumption about the type or values of `kpos_t` values.

RETURN VALUES

On success, **ksetpos()** returns `0`. If there was an error, **ksetpos()** returns `-1` and sets the `errno` value.

ERRORS

When a failure is returned, `errno` is set to one of the following values, as well as the generic mapping of VSAM logical errors to `errno` values.

`[EIO]`	The VSAM `POINT` macro failed.
`[EFAULT]`	The specified `KFILE` *k* or `kpos_t` *pos* was an invalid pointer.
`[EBADF]`	The specified `KFILE` *k* was not open.
`[ENOSYS]`	The RPL for the VSAM file could not be accessed or modified.

SEE ALSO

kgetpos(3)

KWRITE(3)

NAME

kwrite - write output to a Linear Data Set

SYNOPSIS

```
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
#include <machine/vsamio.h>

size_t
kwrite(FILE *k, const void *buf, size_t nbytes)
```

DESCRIPTION

kwrite() attempts to write *nbytes* of data to the Linear Data Set referenced by the KFILE pointer *k* from the buffer pointed to by *buf*.

kwrite() starts at a position given by the pointer associated with *k*, see kseek(3). Upon return from **kwrite()**, the pointer is incremented by the number of bytes which were written. **kwrite()** writes the data in "binary" mode, no attempt is made to place record boundaries onto the data.

kwrite() may write fewer bytes than requested; the return value must be noted, and the remainder of the operation should be retried when possible.

IMPLEMENTATION NOTES

kwrite() is implemented using the DIV (Data In Virtual) macros. Bytes are copied to the managed DIV windows which map into offsets of the Linear Data Set. The bytes are not actually written onto the Linear Data Set until the DIV window is UNMAP'd. A DIV window will be UNMAP'd when the file is closed with the kclose(3) function, or when the number of available DIV windows has been exhausted and the window needs to be re-used.

A Linear Data Set is processed in terms of 4K data blocks, so a **kwrite()** the output file will have a size that is a multiple of 4K after the file has been closed via kclose(3), even those less than that number of bytes may have been processed via **kwrite()**.

RETURN VALUES

Upon successful completion the number of bytes which were written is returned. Otherwise a -1 is returned and the global variable `errno` is set to indicate the error.

ERRORS

kwrite() will fail and the file pointer will remain unchanged if:

`[EBADF]`	The pointer associated with *k* was not open for OUTPUT or did not represent a Linear Data Set.
`[EINVAL]`	The pointer associated with *k* did not represent a Linear Data Set.
`[EIO]`	An error occurred while saving or mapping a DIV window to the file system.

SEE ALSO

kopen(3), kseek(2)

ASCII/EBCDIC Translation Table

The Systems/C compiler and utilities use the following tables to translate characters between ASCII and EBCDIC. These tables represent the mapping of the IBM Code Page 1047 to ISO LATIN-1.

ASCII to EBCDIC

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	01	02	03	37	2D	2E	2F	16	05	15	0B	0C	0D	0E	0F
1	10	11	12	13	3C	3D	32	26	18	19	3F	27	1C	1D	1E	1F
2	40	5A	7F	7B	5B	6C	50	7D	4D	5D	5C	4E	6B	60	4B	61
3	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	7A	5E	4C	7E	6E	6F
4	7C	C1	C2	C3	C4	C5	C6	C7	C8	C9	D1	D2	D3	D4	D5	D6
5	D7	D8	D9	E2	E3	E4	E5	E6	E7	E8	E9	AD	E0	BD	5F	6D
6	79	81	82	83	84	85	86	87	88	89	91	92	93	94	95	96
7	97	98	99	A2	A3	A4	A5	A6	A7	A8	A9	C0	4F	D0	A1	07
8	20	21	22	23	24	25	06	17	28	29	2A	2B	2C	09	0A	1B
9	30	31	1A	33	34	35	36	08	38	39	3A	3B	04	14	3E	FF
A	41	AA	4A	B1	9F	B2	6A	B5	BB	B4	9A	8A	B0	CA	AF	BC
B	90	8F	EA	FA	BE	A0	B6	B3	9D	DA	9B	8B	B7	B8	B9	AB
C	64	65	62	66	63	67	9E	68	74	71	72	73	78	75	76	77
D	AC	69	ED	EE	EB	EF	EC	BF	80	FD	FE	FB	FC	BA	AE	59
E	44	45	42	46	43	47	9C	48	54	51	52	53	58	55	56	57
F	8C	49	CD	CE	CB	CF	CC	E1	70	DD	DE	DB	DC	8D	8E	DF

EBCDIC to ASCII

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	01	02	03	9C	09	86	7F	97	8D	8E	0B	0C	0D	0E	0F
1	10	11	12	13	9D	0A	08	87	18	19	92	8F	1C	1D	1E	1F
2	80	81	82	83	84	85	17	1B	88	89	8A	8B	8C	05	06	07
3	90	91	16	93	94	95	96	04	98	99	9A	9B	14	15	9E	1A
4	20	A0	E2	E4	E0	E1	E3	E5	E7	F1	A2	2E	3C	28	2B	7C
5	26	E9	EA	EB	E8	ED	EE	EF	EC	DF	21	24	2A	29	3B	5E
6	2D	2F	C2	C4	C0	C1	C3	C5	C7	D1	A6	2C	25	5F	3E	3F
7	F8	C9	CA	CB	C8	CD	CE	CF	CC	60	3A	23	40	27	3D	22
8	D8	61	62	63	64	65	66	67	68	69	AB	BB	F0	FD	FE	B1
9	B0	6A	6B	6C	6D	6E	6F	70	71	72	AA	BA	E6	B8	C6	A4
A	B5	7E	73	74	75	76	77	78	79	7A	A1	BF	D0	5B	DE	AE
B	AC	A3	A5	B7	A9	A7	B6	BC	BD	BE	DD	A8	AF	5D	B4	D7
C	7B	41	42	43	44	45	46	47	48	49	AD	F4	F6	F2	F3	F5
D	7D	4A	4B	4C	4D	4E	4F	50	51	52	B9	FB	FC	F9	FA	FF
E	5C	F7	53	54	55	56	57	58	59	5A	B2	D4	D6	D2	D3	D5
F	30	31	32	33	34	35	36	37	38	39	B3	DB	DC	D9	DA	9F

SIGABND example to catch ABEND 978 (out-of-stack)

The following example demonstrates how to set up a signal handler on an alternate stack to catch an ABEND and report the ABEND and REASON codes, then re-issue the ABEND to have it go thru normal percolation.

In this example, the `SIGABND` handler is provided an alternate execution stack, so that ABEND 978 (out-of-stack) can be handled. If an alternate stack is not provided, then an ABEND 978 will automatically be percolated.

Also note that the `SA_RESETHAND` flag is used when establishing the signal handler (via sigaction(2)) in order to avoid looping into the signal handler when the ABEND is re-issued by the return of the signal handler.

```
/*
* Demonstrates "catching" an ABEND, displaying
* some information about it, then returning to
* re-issue the ABEND and have it percolate.
*
**/
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>

int __bpxsig = 1; /* force the use of BPX signals which */
  /* enables TRAP(ON) */

/*
*  The abend_handler() will be invoked for the SIGABND
* signal.  Because SA_RESETHAND is set, it will only
* be invoked once.  On return from the handler, the library
* will restore the processor state at the point of the
* ABEND, and the ABEND will be re-issued and will percolate
* thru normal ABEND processing.
*
*  If SA_RESETHAND is not set (or the signal handler does
```

```
* not set the SIGABND handler to SIG_DFL) then a return
* from the signal handler will restore the processor state
* to the instruction that issued the ABEND and re-issue the
* ABEND, causing the signal handler to be re-invoked in a loop.
*
**/
void
abend_handler(int signum, siginfo_t *info, void *ctx)
{
  int a = __abendcode();
  int r = __rsncode();

  printf("in abend_handler:\n");
  printf("   __abendcode() is 0x%08x\n", a);
/* user completion is the last 12 bits */
  printf("       user completion code is %d\n", a & 0xfff);
  printf("   __rsncode() is %d\n", __rsncode());
  fflush(stdout);

  /* A SIGABEND can't return to the point of the interrupt */
  /* (if the signal handler remains installed, then we     */
  /* return to the point of interrupt, which causes the    */
  /* ABEND to be re-issued which brings us right back here.*/
  /* This is handled by the SA_RESETHAND flag on the       */
  /* sigaction setting.  If SA_RESETHAND is on, then the   */
  /* signal handler is set back to SIG_DFL before invoking */
  /* this routine.  Thus, if we return, we redo the ABEND  */
  /* but with the signal being SIG_DFL, normal ABEND       */
  /* processing happens. */

  /* So - an ABEND catcher can do a few things:  */
  /*  Return to loop forever                     */
  /*  longjump() or setcontext() to a previous   */
  /*    program state                            */
  /*  exit() the program.                        */

  /* In this example, we are returning to allow  */
  /* normal ABEND processing to take over.       */
  return;
}

int counter;

/* Cause an out-of-stack situation by recursively
 * consuming memory until we run out.  When this
 * occurs, the C library will issue ABEND 978.
 */
```

```
int recurse()
{
  int big_array[4096*16];
  counter++;
  return big_array[counter] + recurse();
}

main()
{
   stack_t sigstk;
   struct sigaction act;

   printf("CATCH ABEND 978 (out-of-stack)\n");

   /* Allocate memory for the alternate stack and */
   /* set it up. */
   if ((sigstk.ss_sp = malloc(SIGSTKSZ+4096)) == NULL) {
  fprintf(stderr,"No mem");
                  /* error return */
exit(12);
   }

   /* Define the alternate signal execution stack */
   sigstk.ss_size = SIGSTKSZ+4096;
   sigstk.ss_flags = 0;
   if (sigaltstack(&sigstk,0) < 0) {
      perror("sigaltstack");
   }

   /* Define an ABEND signal handler to use the */
   /* alternate stack, and to reset the handler */
   /* back to SIG_DFL at the time signal handler*/
   /* function is invoked. */
   act.sa_sigaction = abend_handler;
   act.sa_flags = (SA_SIGINFO|SA_ONSTACK|SA_RESETHAND);
   sigemptyset(&act.sa_mask);
   sigaction(SIGABND, &act, NULL);

   /* Invoke the function to eventually consume */
   /* all space for the stack and thus cause a  */
   /* user ABEND 978. */
   return recurse();
}
```

DCALL example

The following example demonstrates how to use the Direct CALL facility to package together a set of functions and make these available to any 31-bit mainframe environment.

The functions here define an initialization function, named INIT(), two support functions, SUP1() and SUP2() and a termination function named END(). The user of this package would first call the INIT() function, then could make use of the support functions, and terminate everything with the END() function.

Each of these functions accepts as the first parameter the address of a 4-byte area. This area holds the environment "handle" returned by the **__dcall_env()** function. It is set by the INIT() function and retrieved by the

```
@@FNDENV
```

assembly code as needed by the other functions.

```
/*
* Direct CALL packaging example
*
*   This example demonstrates how to use Direct CALL
*  in the situation where you want to package functions
*  together to be linked (or dynamically loaded) from
*  any mainframe environment.
*
*   The package provides
* 1) An initialization function, name "INIT".
*    This must be the first function invoked
*    to initialize the package.
*
* 2) Two support functions "SUP1" and "SUP2".
*    Which re-use the extent environment and
*    perform any interesting functions.
*    These also use the FINDENV DCALL facility
```

```
*    to locate the extent environment.
*
* 3) A destruction function, named "END", that
*          ends the environment.
*
*
*   Each of these is called with a first parameter
*   that is a by-reference parm which will contain
*   the DCALL environment handle.  When the "INIT"
*   function is invoked, the Systems/C direct-call
*   environment pointer is saved in the first parm.
*   The other functions retrieve the pointer from
*   there using the direct-call FINDENV facility.
*
*   In C, these functions would be prototyped as:
* INIT(void **env);
* SUP1(void **env, int p1);
* SUP2(void **env, int p1);
* END(void **env);
*
*  Also note that the functions are upper-case, to
*  make linking in other environments easier.
*
*  When calling these in an IBM C/C++ environment, it
*  is important to declare the functions with #pragma
*  OS linkage, as in:
*           #pragma linkage(INIT, OS)
*           #pragma linkage(SUP1, OS)
*           #pragma linkage(SUP2, OS)
*           #pragma linkage(END, OS)
*
*  When pre-linking this package, it is a good idea
*  to use the PLINK features which obscure the C runtime
*  by renaming everything.  We want to rename everything
*  except the INIT, SUP1, SUP2 and END functions; so
*  these PLINK options would be appropriate:

* plink -renameall -prefix=@SX -except=INIT,END,SUP1,SUP2 ...

*  This obfuscates all of the external names consistently
*  and avoids potential clashes with other runtime environments.
*
**/

#include <machine/dcall.h>

/*
```

```
* allocate function
*
* Sets the void * pointer passed as
* a parameter to the direct-call environment
*       handle.  Subsequent calls to the functions
*       below must pass that same pointer as the
*       first parameter.
**/

#pragma prolkey(INIT,"DCALL=ALLOCATE")
void
INIT(void **env)

*env = __dcall_env();




/*
*  supplied function
*    Retrieves the environment pointer from the 1st parm,
*    and executes the function.
*/
#pragma prolkey(SUP1, "DCALL=SUPPLIED,FINDENV=@@FNDENV")
void
SUP1(void **env, int parm1)




/*
*  supplied function
*    Retrieves the environment pointer from the 1st parm,
*    and executes the function.
*/
#pragma prolkey(SUP2, "DCALL=SUPPLIED,FINDENV=@@FNDENV")
void
SUP2(void **env, int parm1)



/*
```

```
*  end function
*    Retrieves the environment pointer from the 1st parm,
*    and destroys the environment.
*/
#pragma prolkey(END, "DCALL=DESTROY,FINDENV=@@FNDENV")
void
END(void **env)



/*
* @@FNDENV assembly function
*
* Invoked by the DCALL=SUPPLIED,FINDENV=@@FNDENV
* functions above.  Loads the first parm and
* copies that as the environment pointer into R0.
*
* It's assumed that R1 points to a parm
* block of the form:
*               +--------------+
*   R1 -> | ptr to env.  | --> +--------------+
*               +--------------+     | env handle  |
*                   ...              +--------------+
*               +--------------+
*
* which would be the typical pass-by-refernce
*       form from a COBOL module, or a #pragma linkage OS
* IBM C/C++ function call.
*
*  NB: @@FNDENV will be an externally visible CSECT, and thus
*      should only be defined in one source file.
*
**/
__asm
@@FNDENV CSECT
@@FNDENV AMODE ANY
@@FNDENV RMODE ANY
     USING @@FNDENV,15
     L 2,0(0,1)
     L 0,0(0,2)     Get ENV ptr into R0
     L 15,=V(CRT9A)
     BR 15
     LTORG
```