FORTRAN 77 Reference Manual Pages

IRIS-4D Series

SiliconGraphics
Computer Systems

Document number: 007-0621-030

FORTRAN 77 Reference Manual Pages

Document Version 3.0

Document Number 007-0621-030

9/90

FORTRAN 77 Reference Manual Pages
Document Version 3.0
Document Number 007-0621-030

Silicon Graphics, Inc.
Mountain View, California

Preface

Here are your FORTRAN 77 Reference Manual Pages. You may place them behind your *FORTRAN 77 Programmer's Guide* or *FORTRAN 77 Reference Manual* or put them in the binder labelled *IRIS-4D Optional Manual Pages*. You received this binder with your IRIS-4D Series Reference Manuals.

NAME

asa – interpret ASA carriage control characters

SYNOPSIS

asa [files]

DESCRIPTION

Asa interprets the output of FORTRAN programs that utilize ASA carriage control characters. It processes either the *files* whose names are given as arguments or the standard input if no file names are supplied. The first character of each line is assumed to be a control character; their meanings are:

' ' (blank) single new line before printing

0 double new line before printing

1 new page before printing

\+ overprint previous line.

Lines beginning with other than the above characters are treated as if they began with ' '. The first character of a line is *not* printed. If any such lines appear, an appropriate diagnostic will appear on standard error. This program forces the first line of each input file to start on a new page.

EXAMPLE

To correctly view the output of FORTRAN programs which use ASA carriage control characters, *asa* could be used as a filter thusly:

```
a.out | asa | lpr
```

and the output, properly formatted and paginated, would be directed to the line printer. FORTRAN output sent to a file could be viewed by:

```
asa file
```

SEE ALSO

fsplit(1)

ORIGIN

AT&T V.3

NAME

f77 – MIPS FORTRAN 77 compiler

SYNOPSIS

f77 [option] ... file ...

DESCRIPTION

f77, the MIPS *ucode* FORTRAN 77 compiler, produces files in the following formats: MIPS object code in MIPS extended *coff* format (the normal result), binary or symbolic *ucode*, *ucode* object files and binary or symbolic assembly language.

f77 accepts several types of *file* arguments. *file* argument types are indicated by suffixes. Intermediate files and results files are usually placed in files whose names are generated from *file* by removing leading directories from *file* and substituting a different suffix. The suffixes accepted and generated by *f77* are the following:

Suffix	File Type
.f	FORTRAN source file
.i	C macro preprocessor output
.l	PFA listing file
.m	PFA intermediate file
.o	object file
.p	M4 preprocessor output
.r	RATFOR source file
.s	symbolic assembly language source
.u	*ucode* object file
.B	binary *ucode* produced by the front end
.F	FORTRAN source file
.G	binary assembly language produced by the code generator and the symbolic to binary assembler
.M	binary *ucode* produced by the *ucode* merger
.O	binary *ucode* produced by the optimizer
.S	binary *ucode* produced by the *ucode* object file splitter
.T	symbol table for binary *ucode*, symbolic *ucode*, or binary assembly language
.U	symbolic *ucode*

file arguments whose suffixes are *.f* or *.F* are compiled, and each resulting object program is placed in a *.o* file. The *.o* file is deleted when a single source program is compiled and loaded all at once.

RATFOR source programs, *.r files*, are first transformed by *ratfor*(1) and then compiled by *f77* producing *.o* files.

f77 always defines the C preprocessor macros **sgi, mips, host_mips, unix, SVR3, SYSTEM_SYSV,** and **MIPSEB** to the C macro preprocessor. *f77* automatically calls the C preprocessor *cpp*(1), and defines the C preprocessor macro **LANGUAGE_FORTRAN** when a *.f* or *.r* file is being compiled.

If the highest level of optimization is specified (with the **–O3** flag) or only *ucode* object files are to be produced (with the **–j** flag) each FORTRAN 77 or RATFOR source file is compiled into a *ucode* object file *(.u)*.

Symbolic assembly language source programs, *.s files*, are assembled and produce a *.o* file. *f77* will define the C preprocessor macro **LANGUAGE_ASSEMBLY** when a *.s* file is being compiled.

file arguments whose names end with *.B*, *.O*, *.S*, *.M*, and *.T* primarily aid compiler development and are not generally used.

If the environment variable **TMPDIR** is set, the value is used as the directory to place any temporary files rather than the default */tmp*.

The following *options* are interpreted by *f77*. See *ld*(1) for load-time *options*.

–c Suppress the loading phase of the compilation and force an object file to be produced even if only one program is compiled.

–g0 Have the compiler produce no symbol table information for symbolic debugging. This is the default.

–g1 Have the compiler produce symbol table information for accurate but limited symbolic debugging of partially optimized code. This option overrides the optimization options (**–O, –O1, –O2, –O3**).

–g or **–g2**
Have the compiler produce additional symbol table information for full symbolic debugging and not do optimizations that limit full symbolic debugging. These options override the optimization options (**–O, –O1, –O2, –O3**).

–g3 Have the compiler produce additional symbol table information for full symbolic debugging for fully optimized code. This option makes the debugger inaccurate. This option can be used with the optimization options (**–O, –O1, –O2, –O3**).

–w Suppress warning messages.

–p0 Do not permit any profiling. If loading happens, the standard runtime startup routine (*crt1.o*) is used; no profiling library is loaded.

–p or **–p1**
Set up for profiling by periodically sampling the value of the program counter. This option only effects the loading. When loading happens, this option replaces the standard runtime startup routine with the profiling runtime startup routine (*mcrt1.o*) and searches the level 1 profiling library (*libprof1.a*) When profiling happens, the startup routine calls *monstartup*(3) and produces a file *mon.out* that contains execution-profiling data for use with the postprocessor *prof*(1).

–O0 Turn off all optimizations.

–O1 Turn on all optimizations that can be done quickly. This is the default.

–O or **–O2**
Invoke the global *ucode* optimizer.

–O3 Do all optimizations, including global register allocation. With this option, a *ucode* object file is created for each FORTRAN 77 or RATFOR source file and left in a *.u* file. The newly created *ucode* object files, the *ucode* object files specified on the command line, the runtime startup routine, and all of the runtime libraries are *ucode* linked. Optimization is done on the resulting *ucode* linked file and then it is linked as normal producing an *a.out* file. No resulting *.o* file is left from the *ucode* linked result as in previous releases. **–c** cannot be specified with **–O3**.

–feedback file
Used with the **–cord** option to specify *file* to be used as a feedback file. This *file* is produced by *prof*(1) with its **–feedback** option from an execution of the program produced by *pixie*(1). Multiple feedback files may be provided as *–feedback*file1 *–feedback*file2... *–feedback*filen.

–cord Run the procedure rearranger, *cord*(1), on the resulting file after linking. The rearrangement is done to improve the caching and paging performance of the program's text. The output of *cord*(1) is left in the file specified by the **–o** *output* option or 'a.out' by default. At least one **–feedback** *file* must be specified.

–j Compile the specified source programs, and leave the *ucode* object file output in *.u* files. Please note that this switch is nonstandard and may not be supported across product lines.

–ko *output*

Name the output file created by the *ucode* loader as *output*. This file is not removed. If this file is compiled, the object file is left in a file whose name consists of *output* with the suffix changed to a *.o*. If *output* has no suffix, a *.o* suffix is appended to *output*. Please note that this switch is non-standard and may not be supported across product lines.

–k Pass options that start with a **–k** to the *ucode* loader. This option is used to specify *ucode* libraries (with **–klx**) and other *ucode* loader options. Please note that this switch is non-standard and may not be supported across product lines.

–S Compile the specified source programs and leave the symbolic assembly language output in corresponding files suffixed with *.s*. If the **–O3** option is used, then a single file, *u.out.s*, is produced.

–P Run only the C macro preprocessor and put the result for each source file (i.e., *.f*, *.r*, or *.s* file) in a corresponding *.i* file after being preprocessed by the appropriate preprocessors. The *.i* file has no "#" lines in it.

–E Run only the C macro preprocessor on the files (regardless of any suffix or not), and send the result to the standard output. This is also done for *.f* and *.r* files after being processed by appropriate preprocessors.

–o *output*

Name the final output file *output*. If this option is used, the file *a.out* is undisturbed.

–D*name*=*def*

–D*name* Define the *name* to the C macro preprocessor, as if by "#define". If no definition, *def*, is given, the name is defined as "1".

–U*name* Remove any initial definition of *name*.

–I*dir* "#include" files whose names do not begin with "/" are always sought first in the directory of the *file* argument, then in directories specified in **–I** options, and finally in the standard directory (*/usr/include*).

–I This option will cause "#include" files never to be searched for in the standard directory (*/usr/include*).

–G *num* Specify the maximum size, in bytes, of a data item that is to be accessed from the global pointer. *num* is assumed to be a decimal number. If *num* is zero, no data is accessed from the global pointer. The default value for *num* is 8 bytes. Data stored off of the global pointer can be accessed by the program quickly, but

this space is limited. Large programs may overflow the space accessed by the global pointer at load time. If the loader gives the error message `Bad -G num value`, recompile with a smaller **-G** *num* value (less than 8). Please note that this switch is non-standard and may not be supported across product lines.

-v Print the passes as they execute with their arguments and their input and output files.

-V Print the version of the driver and the versions of all passes. This is done with the *what*(1) command. Please note that this switch is non-standard and may not be supported across product lines.

-cpp Run the C macro preprocessor on the files before compiling. This is the default.

-nocpp Do not run the C macro preprocessor on C and assembly source files before compiling.

-Olimit *num*
Specify the maximum size, in basic blocks, of a routine that will be optimized by the global optimizer. If a routine has more than this number of basic blocks it will not be optimized and a message will be printed. An option specifying that the global optimizer is to be run (**-O, -O2, -O3**) must also be specified. *num* is assumed to be a decimal number. The default value for *num* is 500 basic blocks.

The following options are specific for *f77*:

-mp Enable the multiprocessing directives.

-mp_keep
Keep the compiler generated temporary file and generate correct line numbers for debugging multiprocessed *DO* loops. This switch should be used with either the **-mp** or the **-pfa** switch. The saved file name has the form: $TMPDIR/P<user_subroutine_name><machine_name><pid>. If the **TMPDIR** environment variable is not set, then the file can be found in /tmp.

-pfa Run the *pfa*(1) preprocessor to automatically discover parallelism in the source code. This also enables the multiprocessing directives. There are two optional arguments: **-pfa list** will run *pfa*, and also produce a listing file with suffix *.l* explaining which loops were parallelized, and if not, why not. **-pfa keep** runs *pfa*, produces the listing file, and also keeps the transformed multiprocessed FORTRAN intermediate file in a file with suffix *.m*.

–pfaprepass

This option permits source code to be passed through *pfa* multiple times. *pfa* is run using the options found on the **–pfaprepass** option, except that no parallel compiler directives are generated. The output from this pre-pass is then fed back into *pfa*, using the normal options. This is occasionally useful on a few programs. In the vast majority of cases, multiple passes have no effect. This option should only be used when it has already determined that there is a good reason for doing so. Options to *pfa* appear on the **–pfaprepass** option exactly as in the **–WK** option. Multiple **–pfaprepass** options may be used; they are executed in left to right order.

–mp_schedtype=type

Has the same effect as putting a *C$MP_SCHEDTYPE*=type directive at the beginning of the file. The supported *type*s are *simple*, *interleave*, *dynamic*, *gss*, and *runtime*. See the *FORTRAN 77 Programmer's Guide* for more details.

–chunk=integer

Has the same effect as putting a *C$CHUNK*=integer directive at the beginning of the file. See the *FORTRAN 77 Programmer's Guide* for more details.

–i2 Make the default integer constants and variables short (2 bytes). All logical quantities will be short.

–i4 Make the default integer constants and variables long (4 bytes). All logical quantities will be long. This is the default.

–onetrip or **–1**

Compile DO loops that execute at least once if reached. (FORTRAN 77 DO loops are not executed if the upper limit is smaller than the lower limit.)

–66 Suppress extensions that enhance FORTRAN 66 compatibility.

–check_bounds

–C Generate code for runtime subscript range checking. The default suppresses range checking.

–U Do not "fold" cases. *f77* is normally a case-insensitive language (for example **a** is equivalent to **A**). The **–U** option causes *f77* to treat uppercase and lowercase separately. Note that the compiler only recognizes keywords in lowercase when this flag is used.

–u Make the default type of a variable *undefined*, rather than using the default FORTRAN rules.

–w1 Suppress the warning message for unused variables (but permit other warnings unless **–w** is specified. Please note that this switch is non-standard and may not be supported across product lines. This is the default.

–w0 Do not suppress the warning message for unused variables. Please note that this switch is non-standard and may not be supported across product lines.

–w66 Suppress only FORTRAN 66 compatibility warnings messages.

–F Apply the RATFOR preprocessor to relevant files and put the result in files whose names have their suffix changed to *.f*. (No *.o* files are created.)

–m Apply the M4 preprocessor, *m4*(1), to each RATFOR source file before transforming it with the *ratfor*(1) preprocessor. The temporary file used as the output of *m4*(1), is a *.p* file. This temporary file is removed unless the **–K** option is specified.

–R Use any remaining characters in the argument as RATFOR options whenever processing a *.r* file. The temporary file used as the output of the RATFOR preprocessor is that of the last component of the source file with a *.f* substituted for the *.r*. This temporary file is removed unless the **–K** option is specified.

–automatic
Place local variables on the runtime stack. The same restrictions apply for this option as they do for the automatic keyword. This is the default.

–static Cause all local variables to be statically allocated.

–noextend_source
Cause the compiler is to restrict the range of FORTRAN source text from column 1 through column 72.

–extend_source
Pad each source line if necessary to make it 132 bytes long and give a warning if it exceeds 132 bytes.

–d_lines
The d_lines option specifies that lines with a D in column 1 are to be compiled and not to be treated as comment lines. The default is to treat lines with a D in column 1 as comment lines.

–col72 Sets the source statement format to the following:

Column	Contents
1-5	Statement label
6	Continuation indicator
7-72	Statement body
73-end	Ignored

–col120

Sets the source statement format to the following:

Column	Contents
1-5	Statement label
6	Continuation indicator
7-120	Statement body
121-end	Ignored

–old_rl Use pre-4D1-3.2 release record length specification for unformatted direct access file i.e. the record length specifier is interpreted as the number of bytes instead of number of words.

–vms_cc

Use VMS FORTRAN carriage control interpretation on unit 6.

–vms_stdin

Allow rereading from stdin after EOF has been encountered.

–vms_endfile

Write a VMS endfile record to the output file when ENDFILE statement is executed and allow subsequent reading from an input file after an endfile record is encountered.

–N[qxscnl]*nnn*

Make static tables in the compiler bigger. The compiler will complain if it overflows its tables and suggest you apply one or more of these flags. These flags have the following meanings:

q Maximum number of equivalenced variables. Default is 150.

x Maximum number of external names (common block names, subroutine and function names). Default is 200.

s Maximum number of statement numbers. Default is 401.

c Maximum depth of nesting for control statements (e.g. DO loops). Default is 20.

n Maximum number of identifiers. Default is 1009.

l Maximum number of labels. Default is 125.

–ZG Load the program with IRIS-4D Series compatible FORTRAN library routines *getarg* and *iargc*. **NOTE:** This option is maintained for backward compatibility with older IRIS systems only and should not be used. Instead, the FORTRAN code should be modified to use the IRIS-4D Series default library routines *getarg*(3f) and *iargc*(3f).

–W*c*,*arg1*[,*arg2*]...
Pass the argument[s] *argi* to the compiler pass *c*. The *c* is one of [**pfjusmocablKyz**]. The *c* selects the compiler pass in the same way as the **–t** option (see **–t** below). Please note that this switch is not standard and may not be supported across product lines.

When using either the Graphics Library (**–lgl**), the Shared Graphics Library (**–lgl_s**), or the Distributed Graphics Library (**–ldgl**) you must also specify the Fortran Graphics Library Interface (**–lfgl**) on the link line. For example:

```
f77 file.f -lfgl -lgl_s
```

Use the Shared Graphics Library (**–lgl_s**) to ensure portability across the IRIS-4D product line.

The following three options when used at compile time generate various degrees of misaligned data in common blocks, and the code to deal with the misalignment. You must include these *option*s to *f77* in the compilation of all modules that reference or define common blocks with misaligned data. Failure to do so could cause core dumps (if the trap handler is not used), or mismatched common blocks.

To load the system libraries capable of handling misaligned data, use the **–L/usr/lib/align** switch at load time. The trap handler may be needed to handle misaligned data passed to system libraries not included in the */usr/lib/align* directory (see *fixade*(3f) and *unaligned*(3x)).

–align8 Permits objects larger than 8 bits to be aligned on 8-bit boundaries. Using this option will have the largest impact on performance.

–align16
Permits objects larger than 16 bits to be aligned on 16-bit boundaries; 16-bit objects must still be aligned on 16-bit boundaries (MC68000-like alignment rules).

–align32

Permits objects larger than 32 bits to be aligned on 32-bit boundaries; 16-bit objects must still be aligned on 16-bit boundaries, and 32-bit objects must still be aligned on 32-bit boundaries.

The options described below primarily aid compiler development and are not generally used:

–H*c* Halt compiling after the pass specified by the character *c*, producing an intermediate file for the next pass. The *c* can be [**fjusmoca**] (see **–t** below for definitions). It selects the compiler pass in the same way as the **–t** option. If this option is used, the symbol table file produced and used by the passes, is the last component of the source file with the suffix changed to *.T* and is not removed. Please note that this switch is non-standard and may not be supported across product lines.

–K Instead of putting intermediate files in */tmp* or TMPDIR, use the standard algorithm for generating file names with the conventional suffix for the type of file (for example *.B* file for binary *ucode* produced by the front end). These intermediate files are never removed even when a pass encounters a fatal error. When *ucode* linking is performed and the **–K** option is specified the basename of the files created after the *ucode* link is *u.out* by default. If **–ko** *output* is specified, the basename of the object file is *output* with the appropriate suffix appended at the end if *output* has no suffix. Please note that this switch is non-standard and may not be supported across product lines.

The options **–t[hpfjusmocablrFIUMKnyz]**, **–h***path*, and **–B***string* select a name to use for a particular pass, startup routine, or standard library. These arguments are processed from left to right so their order is significant. When the **–B** option is encountered, the selection of names takes place using the last **–h** and **–t** options.

Therefore, the **–B** option is always required when using **–h** or **–t**. Sets of these options can be used to select any combination of names.

Any of the **–p[01]** options and any of the **–g[0123]** options must precede all **–B** options because they can affect the location of runtimes and what runtimes are used.

If no **–t** argument has been processed before the **–B** then a **–B***string* is passed to the loader to use with its **–l***x* arguments.

–t[hpfjusmocablrFIUMKnyz]

Select the names. The names selected are those designated by the characters following the **–t** option according to the following table:

Name	Character
include	**h** (see note below)
cpp	**p**
pfa	**K**
fcom	**f**
ujoin	**j**
uld	**u**
usplit	**s**
umerge	**m**
uopt	**o**
ugen	**c**
as0	**a**
as1	**b**
ld	**l**
[m]crt[1n].o	**r**
libF77.a	**F**
libI77.a	**I**
libU77.a	**U**
libisam.a	**S**
libm.a	**M**
libprof1.a	**n**
ftoc	**y**
cord	**z**

Note: although *f77* may be used to compile source files in such languages as *C* and *Pascal*, only the name used for the front-end of the *FORTRAN* compiler is selected by the **–tf** option.

–h*path* Use *path* rather than the directory where the name is normally found. Please note that this switch is non-standard and may not be supported across product lines.

–B*string*

Append *string* to all names specified by the **–t** option. If no **–t** option has been processed before the **–B**, the **–t** option is assumed to be "hpfjusmocablrFIUMKnyz". This list designates all names. Invoking the compiler with a name of the form **f77***string* has the same effect as using a **–B***string* option on the command line.

Other arguments are assumed to be either loader options or FORTRAN 77 compatible object files, typically produced by an earlier *f77* run, or perhaps libraries of FORTRAN 77 compatible routines. These files, together with the results of any compilations specified, are loaded in the order given,

producing an executable program with the default name *a.out*.

FILES

/tmp/ctm*	temporary
/usr/lib/cpp	C macro preprocessor
/usr/lib/pfa	PFA preprocessor
/usr/lib/fcom	FORTRAN 77 front end
/usr/lib/ujoin	binary ucode and symbol table joiner
/usr/bin/uld	ucode loader
/usr/lib/usplit	binary ucode and symbol table splitter
/usr/lib/umerge	procedure intergrator
/usr/lib/uopt	optional global ucode optimizer
/usr/lib/ugen	code generator
/usr/lib/as0	symbolic to binary assembly language translator
/usr/lib/as1	binary assembly language assembler and reorganizer
/usr/lib/cord	procedure rearranger
/usr/lib/ftoc	feedback file to reorder list translator
/usr/lib/crt1.o	runtime startup
/usr/lib/crtn.o	runtime startup
/usr/lib/mcrt1.o	startup for profiling
/usr/lib/libc.a	standard library, see *intro*(3)
/usr/lib/libfgl.a	FORTRAN graphics library interface
/usr/lib/libfpe.a	floating point exception handler library, see *fsigfpe*(3f)
/usr/lib/libgl.a	graphics library
/usr/lib/libgl_s.a	shared graphics library
/usr/lib/libprof1.a	level 1 profiling library
/usr/lib/libF77.a	FORTRAN intrinsic function library
/usr/lib/libI77_mp.a	Multi-processing routines
/usr/lib/libI77.a	FORTRAN I/O library
/usr/lib/libU77.a	FORTRAN UNIX interface library
/usr/lib/libm.a	math library
/usr/lib/libisam.a	indexed sequential access method library
/usr/include	standard directory for ''#include'' files
/usr/bin/ld	MIPS loader
/usr/bin/ratfor	rational FORTRAN dialect preprocessor
mon.out	file produced for analysis by *prof*(1)

BUGS

The compiler attempts to continue after finding semantic errors. These errors may result in compiler internal errors.

SEE ALSO

as(1), asa(1), cc(1), cord(1), cpp(1), dbx(1), edge(1), fsplit(1), ftoc(1), ld(1), m4(1), pfa(1), pixie(1), prof(1), ratfor(1), uconv(1), what(1), fixade(3f), fsigfpe(3f), monstartup(3c), mp(3f), unaligned(3x).

IRIS-4D Series Compiler Guide
FORTRAN 77 Programmer's Guide
FORTRAN 77 Language Reference Manual
POWER FORTRAN Accelerator User's Guide

DIAGNOSTICS

The diagnostics produced by *f77* are intended to be self-explanatory. Occasional messages can be produced by the assembler or loader.

NOTES

The standard library, */usr/lib/libc.a*, is loaded by using the **–lc** loader option and not a full path name. The wrong one could be loaded if there are files with the name libc.a*string* in the directories specified with the **–L** loader option or in the default directories searched by the loader.

The handling of *include* directories and *libc.a* is confusing.

Since *cpp* does not recognize FORTRAN comments, a FORTRAN comment containing the character sequence "/*" will result in deleting the rest of the FORTRAN program. The FORTRAN error message will usually refer to the last source line which can be very confusing. When this happens, try the **–nocpp** option.

ORIGIN

MIPS Computer Systems

NAME

fsplit – split FORTRAN or RATFOR files

SYNOPSIS

fsplit options files

DESCRIPTION

Fsplit splits the named *file(s)* into separate files, with one procedure per file. A procedure includes *blockdata*, *function*, *main*, *program*, and *subroutine* program segments. Procedure *X* is put in file *X*.**f**, *X*.**r**, or *X*.**e** depending on the language option chosen, with the following exceptions: *main* is put in the file *MAIN*.[**efr**] and unnamed *blockdata* segments in the files *blockdataN*.[**efr**] where *N* is a unique integer value for each file.

The following *options* pertain:

–f (default) Input files are *FORTRAN*.

–r Input files are *RATFOR*.

–s Strip *FORTRAN* input lines to 72 or fewer characters with trailing blanks removed.

NOTES

The characters dot (.), underbar (_), and dollar ($) are allowed as name characters in MIPS FORTRAN. If used in subroutine or function names they are also used in the file names of the created files.

If more than one main program is encountered in the list of *file(s)*, *fsplit* will write those after the first in files named *MAINn*.[**efr**] where the "n" is 1 for the second main, 2 for the third, etc.

Comment lines after an *end* and before the next non-comment line are discarded. Comment lines before the first non-comment line are discarded.

SEE ALSO

csplit(1), split(1).

ORIGIN

AT&T V.3

NAME

ratfor – rational FORTRAN dialect

SYNOPSIS

ratfor [option ...] [filename ...]

DESCRIPTION

Ratfor converts a rational dialect of FORTRAN into ordinary irrational FORTRAN. *Ratfor* provides control flow constructs essentially identical to those in C:

statement grouping:

```
{ statement; statement; statement }
```

decision-making:

```
if (condition) statement [ else statement ]
switch (integer value) {
        case integer:   statement
        ...
        [ default: ]    statement
}
```

loops:

```
while (condition) statement
for (expression; condition; expression) statement
do limits statement
repeat statement [ until (condition) ]
break
next
```

and some syntactic sugar to make programs easier to read and write:

free form input:

multiple statements/line; automatic continuation

comments:

this is a comment

translation of relationals:

>, >=, etc., become .GT., .GE., etc.

return (expression)

returns expression to caller from function

define: define name replacement

include: include filename

Ratfor is best used with *f77*(1).

SEE ALSO

f77(1)

B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, 1976.

ORIGIN

AT&T V.3

NAME

uconv – convert FORTRAN unformatted file

SYNOPSIS

uconv [**–ic**] [file1] [**–r** num] [**–o** file2]

DESCRIPTION

uconv converts a FORTRAN unformatted data file either from IRIS Series 2000 or IRIS Series 3000 FORTRAN form to IRIS-4D Series FORTRAN form, or vice versa. *uconv* allows FORTRAN users to port their otherwise non-portable data files opened as FORM="UNFORMATTED".

The *uconv* command has the following options:

–i Identifies the input file as an IRIS Series 2000 or IRIS Series 3000 FORTRAN unformatted data file. This is the default. Note: the **–c** option may not be specified with this option.

–c Identifies the input file as an IRIS-4D Series FORTRAN unformatted data file. Note: the **–i** option may not be specified with this option.

–r *num* Identifies the input and output files as FORTRAN unformatted direct access data files, with record length *num*. Absence of this switch identifies the files as FORTRAN unformatted sequential access data files.

file1 The input file to be converted. Default is stdin.

–o *file2* Specifies the output file to be created. *file2* must not exist. If this option is absent, output is printed to stdout.

AUTHOR

Deborah Ryan

NOTES

uconv can not convert IRIS Series 2000 or IRIS Series 3000 FORTRAN data files opened as FORM="BINARY". *uconv* can not convert IRIS Series 2000 or IRIS Series 3000 FORTRAN data files opened as FORM="UNFORMATTED" with the $BINARY option.

SEE ALSO

Porting FORTRAN Code to IRIS-4D Series Workstations.

ORIGIN

Silicon Graphics, Inc.

NAME

access – determine accessibility of a file

FORTRAN SYNOPSIS

integer function access (name, mode)
character*(*) name, mode

DESCRIPTION

Path points to a path name naming a file. *access* checks the named file for accessibility according to *mode* using the real user ID in place of the effective user ID and the group access list (including the real group ID) in place of the effective group ID. The variable *mode* may include in any order and in any combination one or more of:

r test for read permission

w test for write permission

x test for execute permission

(blank) test for existence

access will fail if one or more of the following are true:

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

[ENOENT] Read, write, or execute (search) permission is requested for a null path name.

[ENOENT] The named file does not exist.

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

[ENAMETOOLONG] The length of *path* exceeds *{PATH_MAX}*, or a pathname component is longer than *{NAME_MAX}*.

[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 that is being executed.

[EACCES] Permission bits of the file mode do not permit the requested access.

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

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 permissions checked with respect to the "group" mode bits, and all others have permissions checked with respect to the "other" mode bits.

SEE ALSO

chmod(2), stat(2).

DIAGNOSTICS

If the requested access is permitted, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

acct – enable or disable process accounting

FORTRAN SYNOPSIS

integer *4 function acct (path)
character *(*) path

DESCRIPTION

acct is used to enable or disable the system process accounting routine. If the routine is enabled, an accounting record will be written on an accounting file for each process that terminates. Termination can be caused by one of two things: an *exit* call or a signal [see *exit*(2) and *signal*(2)]. The effective user ID of the calling process must be superuser to use this call.

path points to a pathname naming the accounting file. The accounting file format is given in *acct*(4).

The accounting routine is enabled if *path* is non-zero and no errors occur during the system call. It is disabled if *path* is zero and no errors occur during the system call.

acct will fail if one or more of the following are true:

[EPERM]	The effective user of the calling process is not superuser.
[EBUSY]	An attempt is being made to enable accounting when it is already enabled.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	One or more components of the accounting file pathname do not exist.
[EACCES]	The file named by *path* is not an ordinary file.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*path* points to an illegal address.

SEE ALSO

exit(2), signal(2), acct(4).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

blockproc, unblockproc, setblockproccnt, blockprocall, unblockprocall, setblockproccntall – routines to block/unblock processes

FORTRAN SYNOPSIS

integer*4 function blockproc (pid)
integer*4 pid;

integer*4 function unblockproc (pid)
integer*4 pid;

integer*4 function setblockproccnt (pid, count)
integer*4 pid;
integer*4 count;

integer*4 function blockprocall (pid)
integer*4 pid;

integer*4 function unblockprocall (pid)
integer*4 pid;

integer*4 function setblockproccntall (pid, count)
integer*4 pid;
integer*4 count;

DESCRIPTION

These routines provide a complete set of blocking/unblocking capabilities for processes. Blocking is implemented with a counting semaphore in the kernel. Each call to *blockproc* decrements the count. When the count becomes negative, the process is suspended. When *unblockproc* is called, the count is incremented. If the count becomes non-negative (>= 0), the process is restarted. This provides both a simple, race free synchronization ability between two processes and a much more powerful capability to synchronize multiple processes.

In order to guarantee a known starting place, the *setblockproccnt* function may be called, which will force the semaphore count to the value given by *count*. New processes have their semaphore zeroed. Normally, *count* should be set to 0. If the resulting block count is greater than or equal to zero and the process is currently blocked, it will be unblocked. If the resulting block count is less than zero, the process will be blocked. Using this, a simple rendezvous mechanism can be set up. If one process wants to wait for *n* other processes to complete, it could set its block count to *-n*. This would immediately force the process to block. Then as each process finishes, it unblocks the waiting process. When the *n'th* process finishes the waiting process will be awakened.

The *blockprocall*, *unblockprocall*, and *setblockproccntall* system calls perform the same actions as *blockproc*, *unblockproc*, and *setblockproccnt*, respectively, but act on all processes in the given process' share group. A share group is a group of processes created with the *sproc(2)* system call. If a process does not belong to a share group, the effect of the plural form of a call will be the same as that of the singular form.

A process may block another provided that standard UNIX permissions are satisfied.

A process may determine whether another is blocked by using the *prctl(2)* system call. It should be noted that since other processes may unblock the subject process at any time, the answer should be interpreted as a snapshot only.

These routines will fail and no operation will be performed if one or more of the following are true:

[ESRCH] The *pid* specified does not exist.

[EPERM] The caller is not operating on itself, its effective user ID is not super-user, and its real or effective user ID does not match the real or effective user ID of the target process.

[EINVAL] The count value that would result from the requested *blockproc*, *unblockproc* or *setblockproccnt* is less than **PR_MINBLOCKCNT** or greater than **PR_MAXBLOCKCNT** as defined in *sys/prctl.h*.

SEE ALSO

sproc(2), prctl(2).

DIAGNOSTICS

Upon successful completion, 0 is returned. Otherwise, a value of −1 is returned to the calling process, and *errno* is set to indicate the error. When using the *blockprocall*, *unblockprocall*, and *setblockproccntall* calls, an error may occur on any of the processes in the share group. These calls will attempt to perform the given action on each process in the share group despite earlier errors, and set *errno* to indicate the error of the last failure to occur.

NAME

brk, sbrk – change data segment space allocation

FORTRAN SYNOPSIS

integer *4 function brk (endds)
character * (*) endds

character * 4096 function sbrk (incr)
integer *4 incr

DESCRIPTION

brk and *sbrk* are used to change dynamically the amount of space allocated for the calling process's data segment [see *exec*(2)]. The change is made by resetting the process's break value and allocating the appropriate amount of space. The break value is the address of the first location beyond the end of the data segment. The amount of allocated space increases as the break value increases. Newly allocated space is set to zero. If, however, the same memory space is reallocated to the same process its contents are undefined.

brk sets the break value to *endds* and changes the allocated space accordingly.

Sbrk adds *incr* bytes to the break value and changes the allocated space accordingly. *Incr* can be negative, in which case the amount of allocated space is decreased.

brk and *sbrk* will fail without making any change in the allocated space if one or more of the following are true:

[ENOMEM] Such a change would result in more space being allocated than is allowed by the system-imposed maximum process size *{PROCSIZE_MAX}*. [see *intro*(2)].

[EAGAIN] There is insufficient amount of operating system memory to hold the data structures needed to describe the requested space. This is likely a temporary failure.

SEE ALSO

exec(2), intro(2), shmop(2), getrlimit(2), ulimit(2), end(3C).

DIAGNOSTICS

Upon successful completion, *brk* returns a value of 0 and *sbrk* returns the old break value. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

chdir – change working directory

FORTRAN SYNOPSIS

integer function chdir (path)
character*(*) path

DESCRIPTION

Path points to the path name of a directory. *chdir* causes the named directory to become the current working directory, the starting point for path searches for path names not beginning with /.

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 name is not a directory.
[ENOENT]	The named directory does not exist.
[EACCES]	Search permission is denied for any component of the path name.
[EFAULT]	*Path* points outside the allocated address space of the process.
[ELOOP]	A path name lookup involved too many symbolic links.
[ENAMETOOLONG]	The length of *path* exceeds *{PATH_MAX}*, or a pathname component is longer than *{NAME_MAX}*.

SEE ALSO

chroot(2), getwd(3C).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

chown, fchown – change owner and group of a file (System V and 4.3BSD)

FORTRAN SYNOPSIS

integer *4 function chown (path, owner, group)
character * (*) path
integer *4 owner, group

integer *4 function fchown (fd, owner, group)
integer *4 fd, owner, group

DESCRIPTION

Path points to a path name naming a file, and *fd* refers to the file descriptor associated with a file. The owner ID and group ID of the named file are set to the numeric values contained in *owner* and *group* respectively.

Only processes with effective user ID equal to the file owner or super-user may change the ownership of a file.

If *chown* is invoked by other than the super-user, the set-user-ID and set-group-ID bits of the file mode, 04000 and 02000 respectively, will be cleared. Note that this has the side-effect of disabling mandatory file/record locking.

The only difference between the System V and 4.3BSD versions is that the 4.3BSD versions allow either the owner or group ID to be left unchanged by specifying it as a –1.

chown will fail and the owner and group of the named file will remain unchanged if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied on a component of the path prefix.
[EPERM]	The effective user ID does not match the owner of the file and the effective user ID is not super-user.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*Path* points outside the allocated address space of the process.
[ELOOP]	A path name lookup involved too many symbolic links.
[ENAMETOOLONG]	The length of *path* exceeds *{PATH_MAX}*, or a path-name component is longer than *{NAME_MAX}*.

fchown will fail if:

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

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

SEE ALSO

chmod(2), fchmod(2).
chown(1) in the *User's Reference Manual*.

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

chroot – change root directory

FORTRAN SYNOPSIS

integer *4 function chroot (path)
character *(*) path

DESCRIPTION

Path points to a path name naming a directory. *chroot* causes the named directory to become the root directory, the starting point for path searches for path names beginning with /. The user's working directory is unaffected by the *chroot* system call.

The effective user ID of the process must be super-user to change the root directory.

The .. entry in the root directory is interpreted to mean the root directory itself. Thus, .. cannot be used to access files outside the subtree rooted at the root directory.

chroot will fail and the root directory will remain unchanged if one or more of the following are true:

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

[ENOENT] The named directory does not exist.

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

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

[EINTR] A signal was caught during the *chroot* system call.

[ENOLINK] *Path* points to a remote machine and the link to that machine is no longer active.

[EMULTIHOP] Components of *path* require hopping to multiple remote machines.

[ELOOP] A path name lookup involved too many symbolic links.

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

SEE ALSO

chdir(2).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

close – close a file descriptor

FORTRAN SYNOPSIS

integer*4 function close (fildes)
integer*4 fildes

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call. *close* closes the file descriptor indicated by *fildes*. All outstanding record locks owned by the process (on the file indicated by *fildes*) are removed.

If a STREAMS [see *intro*(2)] file is closed, and the calling process had previously registered to receive a SIGPOLL signal [see *signal*(2) and *sigset*(2)] for events associated with that file [see I_SETSIG in *streamio*(7)], the calling process will be unregistered for events associated with the file. The last *close* for a *stream* causes the *stream* associated with *fildes* to be dismantled. If O_NDELAY is not set and there have been no signals posted for the *stream*, *close* waits up to 15 seconds, for each module and driver, for any output to drain before dismantling the *stream*. If the O_NDELAY flag is set or if there are any pending signals, *close* does not wait for output to drain, and dismantles the *stream* immediately.

The named file is closed unless one or more of the following are true:

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

[EINTR] A signal was caught during the *close* system call.

SEE ALSO

creat(2), dup(2), exec(2), fcntl(2), intro(2), open(2), pipe(2), signal(2), sigset(2).
streamio(7) in the *System Administrator's Reference Manual.*

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

creat – create a new file or rewrite an existing one

FORTRAN SYNOPSIS

integer *4 function creat (path, mode)
character *(*) path
integer *4 mode

DESCRIPTION

creat creates a new ordinary file or prepares to rewrite an existing file named by the path name pointed to by *path*.

If the file exists, the length is truncated to 0 and the mode and owner are unchanged. Otherwise, the file's owner ID is set to the effective user ID, of the process the group ID is set to the effective group ID, of the process or to the group ID of the directory in which the file is being created. This is determined as follows:

> If the underlying filesystem was mounted with the BSD file creation semantics flag [see *fstab*(4)] or the S_ISGID bit is set [see *chmod*(2)] on the parent directory, then the group ID of the new file is set to the group ID of the parent directory, otherwise it is set to the effective group ID of the calling process.

The low-order 12 bits of the file mode are set to the value of *mode* modified as follows:

> All bits set in the process's file mode creation mask are cleared [see *umask*(2)].
>
> The "sticky bit" of the mode is cleared [see *chmod*(2)].

Upon successful completion, a write-only file descriptor is returned and the file is open for writing, even if the mode does not permit writing. A new file may be created with a mode that forbids writing.

The file pointer used to mark the current position within the file is set to the beginning of the file.

The new file descriptor is set to remain open across *execve*(2) system calls [see *fcntl*(2)].

There is a system enforced limit on the number of open file descriptors per process *{OPEN_MAX}*, whose value is returned by the *getdtablesize*(2) function.

creat fails if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	A component of the path prefix does not exist.
[EACCES]	Search permission is denied on a component of the path prefix.
[ENOENT]	The path name is null.
[EACCES]	The file does not exist and the directory in which the file is to be created does not permit writing.
[EROFS]	The named file resides or would reside on a read-only file system.
[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed.
[EACCES]	The file exists and write permission is denied.
[EISDIR]	The named file is an existing directory.
[EMFILE]	The system imposed limit for open file descriptors per process *{OPEN_MAX}* has already been reached.
[EFAULT]	*Path* points outside the allocated address space of the process.
[ENFILE]	The system file table has exceeded *{NFILE_MAX}* concurrently open files.
[EAGAIN]	The file exists, mandatory file/record locking is set, and there are outstanding record locks on the file [see *chmod*(2)].
[ENOSPC]	The file system is out of inodes.
[ENAMETOOLONG]	The length of *path* exceeds *{PATH_MAX}*, or a path-name component is longer than *{NAME_MAX}*.
[EOPNOTSUPP]	An attempt was made to open a socket (not currently supported).

SEE ALSO

chmod(2), close(2), dup(2), fcntl(2), lseek(2), open(2), read(2), umask(2), write(2), fstab(4).

DIAGNOSTICS

Upon successful completion, a non-negative integer, namely the file descriptor, is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

dup – duplicate an open file descriptor

FORTRAN SYNOPSIS

integer *4 function dup (fildes)
integer *4 fildes

DESCRIPTION

Fildes is a descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, *pipe*, *socket*, or *socketpair* system call. *dup* returns a new descriptor having the following in common with the original:

Refers to same object as the original descriptor.

Same file pointer (i.e., both file descriptors share one file pointer).

Same access mode (read, write or read/write).

Same descriptor status flags (i.e., both descriptors share the same status flags).

Shares any file locks.

The new descriptor is set to remain open across *exec* system calls [see *fcntl*(2)].

The descriptor returned is the lowest one available.

dup will fail if one or more of the following are true:

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

[EMFILE] The system imposed limit for open file descriptors per process *{OPEN_MAX}* has already been reached.

SEE ALSO

close(2), creat(2), exec(2), fcntl(2), intro(2), open(2), pipe(2), lockf(3C).

DIAGNOSTICS

Upon successful completion a non-negative integer, namely the file descriptor, is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

exit, _exit – terminate process

FORTRAN SYNOPSIS

integer status
call exit (status)

DESCRIPTION

exit terminates the calling process with the following consequences:

All of the file descriptors open in the calling process are closed. If the process is sharing file descriptors via an *sproc*, other members of the share group do NOT have their file descriptors closed.

If the parent process of the calling process is executing a *wait*, it is notified of the calling process's termination and the low order eight bits (i.e., bits 0377) of *status* are made available to it [see *wait*(2)].

If the parent process of the calling process is not executing a *wait*, and the parent process is not ignoring **SIGCLD**, the calling process is transformed into a zombie process. A *zombie process* is a process that only occupies a slot in the process table. It has no other space allocated either in user or kernel space. The process table slot that it occupies contains the time accounting information [see **<sys/proc.h>**] to be used by *times*.

The parent process ID of all of the calling processes' existing child processes and zombie processes is set to 1. This means the initialization process [see *intro*(2)] inherits each of these processes.

If the process belongs to a share group, it is removed from that group. Its stack segment is deallocated and removed from the share group's virtual space. All other virtual space that was shared with the share group is left untouched. If the *prctl* (**PR_SETEXITSIG**) option has been enabled for the share group, than the specified signal is sent to all remaining share group members.

Each attached shared memory segment is detached and the value of **shm_nattach** in the data structure associated with its shared memory identifier is decremented by 1.

For each semaphore for which the calling process has set a semadj value [see *semop*(2)], that semadj value is added to the semval of the specified semaphore.

If the process has a process, text, or data lock, an *unlock* is performed [see *plock*(2)]. If the process has any pages locked, they are unlocked [see *mpin*(2)].

An accounting record is written on the accounting file if the system's accounting routine is enabled [see *acct*(2)].

If the calling process is a process group leader (its process ID matches its process group ID), and it became a process group leader by invoking the *setpgrp*(2) function, and it is a terminal group leader (has an associated controlling terminal), then the **SIGHUP** signal is sent to each process that has a process group ID equal to that of the calling process.

If the calling process is a process group leader (its process ID matches its process group ID), and it became a process group leader by invoking the *setpgid*(2) function, and it is a session leader [see *setsid*(2)], and it has an associated controlling terminal, then the **SIGHUP** signal is sent to each process in the foreground process group of the controlling terminal belonging to calling process.

If the calling process is a process group leader (its process ID matches its process group ID), and it became a process group leader by invoking the *setpgrp*(2) function, no signal is sent.

In all cases, if the calling process is a process group leader and has an associated controlling terminal, the controlling terminal is disassociated from the process allowing it to be acquired by another process group leader.

Any mapped files are closed and any written pages flushed to disk.

A death of child signal is sent to the parent.

The C function *exit* causes all file streams to be closed unless one has done an *sproc* which causes the file streams to simply be flushed. The function *_exit* circumvents all cleanup.

SEE ALSO

acct(2), intro(2), mmap(2), mpin(2), plock(2), prctl(2), semop(2), setpgid(2), setpgrp(2), signal(2), sigset(2), sigaction(2), sigprocmask(2), sigvec(3B), sigblock(3B), sigsetmask(3B), sproc(2), wait(2).

WARNING

See *WARNING* in *signal*(2).

DIAGNOSTICS

None. There can be no return from an *exit* system call.

NAME

fcntl – file and descriptor control

FORTRAN SYNOPSIS

integer *4 function fcntl (fildes, cmd, arg)
integer *4 fildes, cmd, arg

DESCRIPTION

fcntl provides for control over open descriptors. *Fildes* is an open descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, *pipe*, *socket*, or *socketpair* system call.

The commands available are:

F_DUPFD Return a new descriptor as follows:

Lowest numbered available descriptor greater than or equal to the third argument, *arg*, taken as an integer of type int.

Refers to the same object as the original descriptor.

Same file pointer as the original file (i.e., both file descriptors share one file pointer).

Same access mode (read, write or read/write).

Same descriptor status flags (i.e., both descriptors share the same status flags).

Shares any file locks.

The close-on-exec flag (**FD_CLOEXEC**) associated with the new descriptor is cleared to keep the file open across calls to the *exec*(2) family of functions.

F_GETFD Get the file descriptor flags associated with the descriptor *fildes*. If the **FD_CLOEXEC** flag is **0** the descriptor will remain open across *exec*, otherwise the descriptor will be closed upon execution of *exec*.

F_SETFD Set the file descriptor flags for *fildes*. Currently the only flag implemented is **FD_CLOEXEC**. Note: this flag is a per-process and per-descriptor flag; setting or clearing it for a particular descriptor will not affect the flag on descriptors copied from it by a *dup*(2) or **F_DUPFD** operation, nor will it affect the flag on other processes instances of that descriptor.

F_GETFL Get *file* status flags and file access modes. The file access modes may be extracted from the return value using the mask **O_ACCMODE**.

F_SETFL Set *file* status flags to the third argument, *arg*, taken as type int. Only the following flags can be set [see *fcntl*(5)]: **FAPPEND, FSYNC, FNDELAY, FNONBLOCK,** and **FASYNC**. **FAPPEND** is equivalent to **O_APPEND**; **FSYNC** is equivalent to **O_SYNC**; **FNDELAY** is equivalent to **O_NDELAY**; and **FNONBLOCK** is equivalent to **O_NONBLOCK**. **FASYNC** is equivalent to calling *ioctl* with the **FIOASYNC** command. This enables the **SIGIO** facilities and is currently supported only on sockets.

Since the descriptor status flags are shared with descriptors copied from a given descriptor by a *dup*(2) or **F_DUPFD** operation, and by other processes instances of that descriptor a **F_SETFL** operation will affect those other descriptors and other instances of the given descriptors as well. For example, setting or clearing the **FNDELAY** flag will logically cause an **FIONBIO** *ioctl*(2) to be performed on the object referred to by that descriptor. Thus all descriptors referring to that object will be affected.

Flags not understood for a particular descriptor are silently ignored.

F_GETLK Get the first lock which blocks the lock description given by the variable of type *struct flock* pointed to by *arg*. The information retrieved overwrites the information passed to *fcntl* in the *flock* structure. If no lock is found that would prevent this lock from being created, then the structure is passed back unchanged except for the lock type which will be set to **F_UNLCK**.

F_SETLK Set or clear a file segment lock according to the variable of type *struct flock* pointed to by *arg* [see *fcntl*(5)]. The *cmd* **F_SETLK** is used to establish read (F_RDLCK) and write (F_WRLCK) locks, as well as remove either type of lock (F_UNLCK). If a read or write lock cannot be set *fcntl* will return immediately with an error value of –1.

F_SETLKW This *cmd* is the same as **F_SETLK** except that if a read or write lock is blocked by other locks, the process will sleep until the segment is free to be locked.

F_CHKFL This flag is used internally by **F_SETFL** to check the legality of file flag changes.

F_GETOWN Used by sockets: get the process ID or process group currently receiving **SIGIO** and **SIGURG** signals; process groups are returned as negative values.

F_SETOWN Used by sockets: set the process or process group to receive SIGIO and SIGURG signals; process groups are specified by supplying *arg* as negative, otherwise *arg* is interpreted as a process ID.

A read lock prevents any process from write locking the protected area. More than one read lock may exist for a given segment of a file at a given time. The file descriptor on which a read lock is being placed must have been opened with read access.

A write lock prevents any process from read locking or write locking the protected area. Only one write lock may exist for a given segment of a file at a given time. The file descriptor on which a write lock is being placed must have been opened with write access.

The structure *flock* describes the type (*l_type*), starting offset (*l_whence*), relative offset (*l_start*), size (*l_len*), process id (*l_pid*), and RFS system id (*l_sysid*) of the segment of the file to be affected. The process id and system id fields are used only with the **F_GETLK** *cmd* to return the values for a blocking lock. Locks may start and extend beyond the current end of a file, but may not be negative relative to the beginning of the file. A lock may be set to always extend to the end of file by setting *l_len* to zero (0). If such a lock also has *l_whence* and *l_start* set to zero (0), the whole file will be locked. Changing or unlocking a segment from the middle of a larger locked segment leaves two smaller segments for either end. Locking a segment that is already locked by the calling process causes the old lock type to be removed and the new lock type to take effect. All locks associated with a file for a given process are removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process in a *fork*(2) system call.

When mandatory file and record locking is active on a file, [see *chmod*(2)], *read* and *write* system calls issued on the file will be affected by the record locks in effect.

fcntl will fail if one or more of the following are true:

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

[EINVAL] *Cmd* is F_DUPFD. *arg* is either negative, or greater than or equal to the maximum number of open file descriptors allowed each user [see *getdtablesize*(2)].

[EMFILE] *Cmd* is F_DUPFD and *{OPEN_MAX}* file descriptors are currently in use by this process, or no file descriptors greater than or equal to *arg* are available.

[EINVAL] *Cmd* is F_GETLK, F_SETLK, or SETLKW and *arg* or the data it points to is not valid.

[EAGAIN] *Cmd* is F_SETLK the type of lock (*l_type*) is a read (F_RDLCK) lock and the segment of a file to be locked is already write locked by another process or the type is a write (F_WRLCK) lock and the segment of a file to be locked is already read or write locked by another process.

[ENOLCK] *Cmd* is F_SETLK or F_SETLKW, the type of lock is a read or write lock, and there are no more record locks available (too many file segments locked) because the system maximum *{FLOCK_MAX}* [see *intro*(2)], has been exceeded.

[EINTR] *Cmd* is F_SETLKW and a signal interrupted the process while it was waiting for the lock to be granted.

[EDEADLK] *Cmd* is F_SETLKW, the lock is blocked by some lock from another process, and putting the calling-process to sleep, waiting for that lock to become free, would cause a deadlock.

[EFAULT] *Cmd* is F_SETLK, *arg* points outside the program address space.

[ESRCH] *Cmd* is F_SETOWN and no process can be found corresponding to that specified by *arg*.

SEE ALSO

close(2), creat(2), dup(2), exec(2), fork(2), getdtablesize(2), intro(2), open(2), pipe(2), fcntl(5).

DIAGNOSTICS

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

F_DUPFD	A new file descriptor.
F_GETFD	Value of flag (only the low-order bit is defined).
F_SETFD	Value other than −1.
F_GETFL	Value of file flags.
F_SETFL	Value other than −1.
F_GETLK	Value other than −1.
F_SETLK	Value other than −1.
F_SETLKW	Value other than −1.
F_GETOWN	*Pid* of socket owner.
F_SETOWN	Value other than −1.

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

NAME

fork – create a new process

FORTRAN SYNOPSIS

integer function fork()

DESCRIPTION

fork causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process). This means the child process inherits the following attributes from the parent process:

environment
close-on-exec flag [see *exec*(2)]
signal handling settings (i.e., **SIG_DFL**, **SIG_IGN**, **SIG_HOLD**, function addresses and signal masks)
set-user-ID mode bit
set-group-ID mode bit
profiling on/off status
debugger trailing status
nice value [see *nice*(2)]
all attached shared memory segments [see *shmop*(2)]
all mapped files [see *mmap*(2)]
non-degrading priority [see *schedctl*(2)]
process group ID
tty group ID [see *exit*(2)]
current working directory
root directory
file mode creation mask [see *umask*(2)]
file size limit [see *ulimit*(2)]

The child process differs from the parent process in the following ways:

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 file descriptors. Each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent.

File locks previously set by the parent are not inherited by the child [see *fcntl*(2)].

All semadj values are cleared [see *semop*(2)].

Process locks, text locks and data locks are not inherited by the child [see *plock*(2)].

The set of signals pending to the parent is not inherited by the child.

Page locks are not inherited [see *mpin*(2)].

The child process's *utime*, *stime*, *cutime*, and *cstime* are set to 0. The time left until an alarm clock signal is reset to 0.

The time left until an itimer signal is reset to 0.

The child will not inherit the ability to make graphics calls. The child must establish itself as a graphics process by invoking the *winopen*(3G) (or *ginit*(3G)) call. Otherwise the child process may receive a segmentation fault upon attempting to make a graphics call.

The share mask is set to 0 [see *sproc*(2)].

fork will fail and no child process will be created if one or more of the following are true:

[EAGAIN] The system-imposed limit on the total number of processes under execution, *{NPROC}* [see *intro*(2)], would be exceeded.

[EAGAIN] The system-imposed limit on the total number of processes under execution by a single user *{CHILD_MAX}* [see *intro*(2)], would be exceeded.

[EAGAIN] Amount of system memory required is temporarily unavailable.

SEE ALSO

exec(2), intro(2), mmap(2), nice(2), pcreate(3C), plock(2), ptrace(2), schedctl(2), semop(2), shmop(2), signal(2), sigset(2), sigaction(2), signal(3B), sigvec(3B), sproc(2), times(2), ulimit(2), umask(2), wait(2).

DIAGNOSTICS

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 *errno* is set to indicate the error.

NAME

getdents – read directory entries and put in a file system independent format

FORTRAN SYNOPSIS

integer *4 function getdents (fildes, buf, nbyte)
integer *4 fildes
character * (*) buf
integer *4 nbyte

DESCRIPTION

Fildes is a file descriptor obtained from an *open*(2) or *dup*(2) system call.

getdents attempts to read *nbyte* bytes from the directory associated with *fildes* and to format them as file system independent directory entries in the buffer pointed to by *buf*. Since the file system independent directory entries are of variable length, in most cases the actual number of bytes returned will be strictly less than *nbyte*.

The file system independent directory entry is specified by the *dirent* structure. For a description of this see *dirent*(4).

On devices capable of seeking, *getdents* starts at a position in the file given by the file pointer associated with *fildes*. Upon return from *getdents*, the file pointer is incremented to point to the next directory entry.

This system call was developed in order to implement the *readdir*(3C) routine [for a description see *directory*(3C)], and should not be used for other purposes.

getdents will fail if one or more of the following are true:

[EBADF]	*Fildes* is not a valid file descriptor open for reading.
[EFAULT]	*Buf* points outside the allocated address space.
[EINVAL]	*nbyte* is not large enough for one directory entry.
[ENOENT]	The current file pointer for the directory is not located at a valid entry.
[ENOLINK]	*Fildes* points to a remote machine and the link to that machine is no longer active.
[ENOTDIR]	*Fildes* is not a directory.
[EIO]	An I/O error occurred while accessing the file system.

SEE ALSO

directory(3C), dirent(4).

DIAGNOSTICS

Upon successful completion a non-negative integer is returned indicating the number of bytes actually read. A value of 0 indicates the end of the directory has been reached. If the system call failed, a −1 is returned and *errno* is set to indicate the error.

NAME

gethostid, sethostid – get/set unique identifier of current host

FORTRAN SYNOPSIS

integer *4 function gethostid ()

subroutine sethostid ()
integer *4 hostid

DESCRIPTION

Sethostid establishes a 32-bit identifier for the current processor that is intended to be unique among all UNIX systems in existence. This is normally a DARPA Internet address for the local machine. This call is allowed only to the super-user and is normally performed at boot time.

Gethostid returns the 32-bit identifier for the current processor.

SEE ALSO

hostid(1), gethostname(2)

BUGS

32 bits for the identifier is too small.

NAME

gethostname, sethostname – get/set name of current host

FORTRAN SYNOPSIS

integer function gethostname (name, namelen)
character *1 name (namelen)
integer namelen

DESCRIPTION

Gethostname returns the standard host name for the current processor, as previously set by *sethostname*. The parameter *namelen* specifies the size of the *name* array. The returned name is null-terminated unless insufficient space is provided.

Sethostname sets the name of the host machine to be *name*, which has length *namelen*. This call is restricted to the super-user and is normally used only when the system is bootstrapped.

RETURN VALUE

If the call succeeds a value of 0 is returned. If the call fails, then 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 these calls:

[EFAULT]	The *name* or *namelen* parameter gave an invalid address.
[EPERM]	The caller tried to set the hostname and was not the super-user.
[EINVAL]	The *namelen* parameter was too large.

SEE ALSO

gethostid(2)

BUGS

Host names are limited to MAXHOSTNAMELEN (from <*sys/param.h*>) characters, currently 64.

NAME

getpid, getpgrp, getppid – get process, process group, and parent process IDs

FORTRAN SYNOPSIS

integer function getpid()
integer function getpgrp()
integer function getppid()

DESCRIPTION

getpid returns the process ID of the calling process.

getpgrp returns the process group ID of the calling process.

getppid returns the parent process ID of the calling process.

SEE ALSO

exec(2), fork(2), intro(2), setpgrp(2), signal(2).

NAME

getsockopt, setsockopt – get and set options on sockets

FORTRAN SYNOPSIS

subroutine getsockopt (s, level, optname, optval, optlen)
integer *4 s, level, optname
character * (*) optval
integer *4 optlen

subroutine setsockopt (s, level, optname, optval, optlen)
integer *4 s, level, optname
character * (*) optval
integer *4 optlen

DESCRIPTION

Getsockopt and *setsockopt* manipulate *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*(3N).

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 supplied as 0.

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; consult the appropriate entries in section (4P).

Most socket-level options take an *int* parameter for *optval*. For *setsockopt*, the parameter should 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).

The following options are recognized at the socket level. Except as noted, each may be examined with *getsockopt* and set with *setsockopt*.

SO_DEBUG	toggle recording of debugging information
SO_REUSEADDR	toggle local address reuse
SO_KEEPALIVE	toggle keep connections alive
SO_DONTROUTE	toggle routing bypass for outgoing messages
SO_LINGER	linger on close if data present
SO_BROADCAST	toggle permission to transmit broadcast messages
SO_OOBINLINE	toggle reception of out-of-band data in band
SO_SNDBUF	set buffer size for output
SO_RCVBUF	set buffer size 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_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. 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* 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 the *setsockopt* call when SO_LINGER is requested). If SO_LINGER is disabled and a *close* 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 the system. 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* or *read* calls without the MSG_OOB flag. 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. The system places an absolute limit on these values. Finally, SO_TYPE and SO_ERROR are options used only with *setsockopt*.

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.

RETURN VALUE

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(3N)

BUGS

Several of the socket options should be handled at lower levels of the system.

NAME

getuid, geteuid, getgid, getegid – get real user, effective user, real group, and effective group IDs

FORTRAN SYNOPSIS

integer function getuid()
integer*2 function geteuid()
integer function getgid()
integer*2 function getegid()

DESCRIPTION

getuid returns the real user ID of the calling process.

geteuid returns the effective user ID of the calling process.

getgid returns the real group ID of the calling process.

getegid returns the effective group ID of the calling process.

SEE ALSO

intro(2), setuid(2).

NAME

ioctl – control device

FORTRAN SYNOPSIS

integer *4 function ioctl (fildes, request, arg)
integer*4 fildes, request, arg

DESCRIPTION

ioctl performs a variety of control functions on devices and STREAMS. For non-STREAMS files, the functions performed by this call are *device-specific* control functions. The *request* and optional third argument are passed to the file designated by *fildes* and are interpreted by the device driver. For a given device, the *requests* that are understood are documented in the section 7 manual page for that device. This control is infrequently used on non-STREAMS devices, with the basic input/output functions performed through the *read*(2) and *write*(2) system calls.

For STREAMS files, specific functions are performed by the *ioctl* call as described in *streamio*(7).

Fildes is an open file descriptor that refers to a device. *Request* selects the control function to be performed and will depend on the device being addressed. The optional third argument represents additional information that is needed by this specific device to perform the requested function. The data type of the third argument depends upon the particular control request, but it is either an integer or a pointer to a device-specific data structure.

In addition to device-specific and STREAMS functions, generic functions are provided by more than one device driver, for example, the general terminal interface [see *termio*(7)].

ioctl will fail for any type of file if one or more of the following are true:

[EACCES] Future error.

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

[ENOTTY] *Fildes* is not associated with a device driver that accepts control functions.

[EINTR] A signal was caught during the *ioctl* system call.

ioctl will also fail if the device driver detects an error. In this case, the error is passed through *ioctl* without change to the caller. A particular driver might not have all of the following error cases. Other requests to device drivers will fail if one or more of the following are true:

[EFAULT] *Request* requires a data transfer to or from a buffer pointed to by the third argument but some part of the buffer is outside the process's allocated space.

[EINVAL] *Request* or the third argument is not valid for this device.

[EIO] Some physical I/O error has occurred.

[ENXIO] The *request* and the third argument are valid for this device driver, but the service requested can not be performed on this particular subdevice.

STREAMS errors are described in *streamio*(7).

SEE ALSO

streamio(7), termio(7) in the *System Administrator's Reference Manual.*

DIAGNOSTICS

Upon successful completion, the value returned depends upon the device control function, but must be a non-negative integer. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

kill – send a signal to a process or a group of processes

FORTRAN SYNOPSIS

integer function kill (pid, sig)

DESCRIPTION

kill sends a signal to a process or a group of processes. The process or group of processes to which the signal is to be sent is specified by *pid*. The signal that is to be sent is specified by *sig* and is either one from the list given in *signal*(2), or 0. If *sig* is 0 (the null signal), error checking is performed but no signal is actually sent. This can be used to check the validity of *pid*.

The real or effective user ID of the sending process must match the real, saved, or effective user ID of the receiving process, unless the effective user ID of the sending process is super-user. An exception to this is the signal SIGCONT, which may be sent to any descendant, or any process in the same session (having the same session ID) as the current process.

The processes with a process ID of 0 and a process ID of 1 are special processes [see *intro*(2)] and will be referred to below as *proc0* and *proc1*, respectively.

If *pid* is greater than zero, *sig* will be sent to the process whose process ID is equal to *pid*. *Pid* may equal 1.

If *pid* is 0, *sig* will be sent to all processes excluding *proc0* and *proc1* whose process group ID is equal to the process group ID of the sender.

If *pid* is −1 and the effective user ID of the sender is not super-user, *sig* will be sent to all processes excluding *proc0* and *proc1* whose real user ID is equal to the effective user ID of the sender.

If *pid* is −1 and the effective user ID of the sender is super-user, *sig* will be sent to all processes excluding *proc0* and *proc1*.

If *pid* is negative but not −1, *sig* will be sent to all processes whose process group ID is equal to the absolute value of *pid*.

kill will fail and no signal will be sent if one or more of the following are true:

[EINVAL] *Sig* is not a valid signal number.

[EINVAL] *Sig* is SIGKILL and *pid* is 1 (proc1).

[ESRCH] No process can be found corresponding to that specified by *pid*.

[ESRCH] The process group was given as 0 but the sending process does not have a process group.

[EPERM] The user ID of the sending process is not super-user, and its real or effective user ID does not match the real, saved, or effective user ID of the receiving process.

SEE ALSO

exec(2), getpid(2), setpgrp(2), setsid(2), signal(2), sigset(2), sigaction(2), sigprocmask(2), sigvec(3B), sigblock(3B), sigsetmask(3B), killpg(3B). kill(1) in the *User's Reference Manual.*

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

link – link to a file

FORTRAN SYNOPSIS

integer function link (path1, path2)
character*(*) path1, path2

DESCRIPTION

Path1 points to a path name naming an existing file. *Path2* points to a path name naming the new directory entry to be created. *link* creates a new link (directory entry) for the existing file.

link will fail and no link will be created if one or more of the following are true:

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

[ENOENT] A component of either path prefix does not exist.

[EACCES] A component of either path prefix denies search permission.

[ENOENT] The file named by *path1* does not exist.

[EEXIST] The link named by *path2* exists.

[ENAMETOOLONG] The length of the *path1* or *path2* argument exceeds *{PATH_MAX}*, or a pathname component is longer than *{NAME_MAX}*.

[EPERM] The file named by *path1* is a directory and the effective user ID is not super-user.

[EXDEV] The link named by *path2* and the file named by *path1* are on different logical devices (file systems).

[ENOENT] *Path2* points to a null path name.

[EACCES] The requested link requires writing in a directory with a mode that denies write permission.

[EROFS] The requested link requires writing in a directory on a read-only file system.

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

[EMLINK] The maximum number of links to a file would be exceeded.

SEE ALSO

unlink(2).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

lseek – move read/write file pointer (System V and 4.3BSD)

FORTRAN SYNOPSIS

integer*4 function lseek (fildes, offset, whence)
integer*4 fildes, offset, whence

DESCRIPTION

Fildes is a file descriptor returned from a *creat*, *open*, *dup*, or *fcntl* system call. *lseek* sets the file pointer associated with *fildes* as follows:

If *whence* is **SEEK_SET (L_SET)**, the pointer is set to *offset* bytes.

If *whence* is **SEEK_CUR (L_INCR)**, the pointer is set to its current location plus *offset*.

If *whence* is **SEEK_END (L_XTND)**, the pointer is set to the size of the file plus *offset*.

Upon successful completion, the resulting pointer location, as measured in bytes from the beginning of the file, is returned. Note that if *fildes* is a remote file descriptor and *offset* is negative, *lseek* will return the file pointer even if it is negative.

lseek allows the file offset to be set beyond the end of existing data in the file. If data is later written at that point, subsequent reads of the data in the gap return bytes with the value zero until data is actually written into the gap.

lseek will fail and the file pointer will remain unchanged if one or more of the following are true:

[EBADF] *Fildes* is not an open file descriptor.

[ESPIPE] *Fildes* is associated with a pipe, socket, or fifo.

[EINVAL and SIGSYS signal]
Whence is not a proper value.

[EINVAL] *Fildes* is not a remote file descriptor, and the resulting file pointer would be negative.

Some devices are incapable of seeking. The value of the file pointer associated with such a device is undefined.

SEE ALSO

creat(2), dup(2), fcntl(2), open(2).

DIAGNOSTICS

Upon successful completion, a non-negative integer indicating the file pointer value is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

mkdir – make a directory

FORTRAN SYNOPSIS

integer *4 function mkdir (path, mode)
character * (*) path
integer *4 mode

DESCRIPTION

The routine *mkdir* creates a new directory with the name *path*. The mode of the new directory is initialized from the *mode*. The protection part of the *mode* argument is modified by the process's mode mask [see *umask*(2)].

The directory's owner ID is set to the process's effective user ID. The directory's group ID is set to the process's effective group ID or the group ID of the directory in which the directory is being created. This is determined as follows:

> If the underlying filesystem was mounted with the BSD file creation semantics flag [see *fstab*(4)] or the S_ISGID bit is set [see *chmod*(2)] on the parent directory, then the group ID of the new file is set to the group ID of the parent directory, otherwise it is set to the effective group ID of the calling process.

The newly created directory is empty with the possible exception of entries for "." and "..".

mkdir will fail and no directory will be created if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	A component of the path prefix does not exist.
[ENAMETOOLONG]	The length of the *path* argument exceeds *{PATH_MAX}*, or a pathname component is longer than *{NAME_MAX}*.
[EACCES]	Either a component of the path prefix denies search permission or write permission is denied on the parent directory of the directory to be created.
[ENOENT]	The path is longer than the maximum allowed.
[EEXIST]	The named file already exists.
[EROFS]	The path prefix resides on a read-only file system.
[EFAULT]	*Path* points outside the allocated address space of the process.

[EMLINK]	The maximum number of links to the parent directory would exceed *{LINK_MAX}*.
[ENOSPC]	No space is available.
[EIO]	An I/O error has occurred while accessing the file system.

SEE ALSO

mkdir(1), chmod(2), mknod(2), unlink(2), fstab(4).

DIAGNOSTICS

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

NAME

mknod – make a directory, or a special or ordinary file

FORTRAN SYNOPSIS

integer *4 function mknod (path, mode, dev)
character * (*) path
integer *4 mode, dev

DESCRIPTION

mknod creates a new file named by the path name pointed to by *path*. The mode of the new file (including file type bits) is initialized from *mode*. The value of the file type bits which are permitted are:

S_IFIFO fifo special
S_IFCHR character special
S_IFBLK block special
S_IFREG ordinary file

All other mode bits are interpreted as described in *chown*(2).

The owner ID of the file is set to the effective user ID of the process. The group ID of the file is set to the effective group ID of the process or the group ID of the directory in which the file is being created. This is determined as follows:

> If the underlying filesystem was mounted with the BSD file creation semantics flag [see *fstab*(4)] or the **S_ISGID** bit is set [see *chmod*(2)] on the parent directory, then the group ID of the new file is set to the group ID of the parent directory, otherwise it is set to the effective group ID of the calling process.

Values of *mode* other than those above are undefined and should not be used. The low-order 9 bits of *mode* are modified by the process's file mode creation mask: all bits set in the process's file mode creation mask are cleared [see *umask*(2)]. If *mode* indicates a block or character special file, *dev* is a configuration-dependent specification of a character or block I/O device. If *mode* does not indicate a block special or character special device, *dev* is ignored.

mknod may be invoked only by the super-user for file types other than FIFO special.

mknod will fail and the new file will not be created if one or more of the following are true:

[EPERM]	The effective user ID of the process is not super-user.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	A component of the path prefix does not exist.
[EROFS]	The directory in which the file is to be created is located on a read-only file system.
[EEXIST]	The named file exists.
[EFAULT]	*Path* points outside the allocated address space of the process.
[ENAMETOOLONG]	The length of the *path* argument exceeds *{PATH_MAX}*, or a pathname component is longer than *{NAME_MAX}*.
[ENOSPC]	No space is available.
[EINVAL]	If you create files of the type fifo special, character special, or block special on an NFS-mounted file system.

SEE ALSO

chmod(2), exec(2), mkdir(2), umask(2), fstab(4).
mkdir(1) in the *User's Reference Manual.*

DIAGNOSTICS

Upon successful completion a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

mount – mount a file system

FORTRAN SYNOPSIS

integer *4 function mount (spec, dir, [mflag, fstyp, [data, datalen]])
character * (*) spec, dir
integer *4 mflag, fstyp
integer *4 data, datalen

DESCRIPTION

mount attaches a file system to a directory. After a successful return, references to directory *dir* will refer to the root directory on the newly mounted file system. *Dir* is a pointer to a pathname of a existent directory. Its old contents are inaccessible while the filesystem is mounted. *Spec* and *dir* are pointers to path names. *Fstyp* is the file system type number. The *sysfs*(2) system call can be used to determine the file system type number. Note that if the MS_FSS flag bit of *mflag* is off, the file system type will default to the root file system type. Only if the bit is on will *fstyp* be used to indicate the file system type.

The low-order bit of *mflag* is used to control write permission on the mounted file system; if **1**, writing is forbidden, otherwise writing is permitted according to individual file accessibility.

mount may be invoked only by the super-user. It is intended for use only by the *mount*(1M) utility.

mount will fail if one or more of the following are true:

[EPERM]	The effective user ID is not super-user.
[ENOENT]	Any of the named files does not exist.
[ENOTDIR]	A component of a path prefix is not a directory.
[ENOTBLK]	*Spec* is not a block special device.
[ENXIO]	The device associated with *spec* does not exist.
[ENOTDIR]	*Dir* is not a directory.
[EFAULT]	*Spec* or *dir* points outside the allocated address space of the process.
[EBUSY]	*Dir* is currently mounted on, is someone's current working directory, or is otherwise busy.
[EBUSY]	The device associated with *spec* is currently mounted.
[EBUSY]	There are no more mount table entries.

[EROFS] *Spec* is write protected and *mflag* requests write permission.

[ENOSPC] The file system state in the super-block is not FsOKAY and *mflag* requests write permission.

[EINVAL] The super block has an invalid magic number or the *fstyp* is invalid or *mflag* is not valid.

SEE ALSO

sysfs(2), umount(2), fs(4).
mount(1M) in the *System Administrator's Reference Manual.*

DIAGNOSTICS

Upon successful completion a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

nice – change priority of a process

FORTRAN SYNOPSIS

integer *4 function nice (incr)
integer *4 incr

DESCRIPTION

nice adds the value of *incr* to the nice value of the calling process. A process's *nice value* is a non-negative number for which a more positive value results in lower CPU priority.

A maximum nice value of 39 and a minimum nice value of 0 are imposed by the system. (The default nice value is 20.) Requests for values above or below these limits result in the nice value being set to the corresponding limit.

[EPERM] *nice* will fail and not change the nice value if *incr* is negative or greater than 39 and the effective user ID of the calling process is not super-user.

SEE ALSO

exec(2), setpriority(2), schedctl(2).
nice(1), csh(1), sh(1) in the *User's Reference Manual.*

DIAGNOSTICS

Upon successful completion, *nice* returns the new nice value minus 20. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NOTES

The *csh(1)* has a version of *nice* as a builtin command which alas, has slightly different syntax and semantics.

NAME

open – open for reading or writing

FORTRAN SYNOPSIS

integer *4 function open (path, oflag, mode)
character * (*) path
integer *4 oflag, mode

DESCRIPTION

Path points to a path name naming a file. *open* opens a file descriptor for the named file and sets the file status flags according to the value of *oflag*. For non-STREAMS [see *intro*(2)] files, *oflag* values are constructed by or-ing flags from the following list (only one of the first three flags below may be used):

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

O_NOCTTY If set, and *path* identifies a terminal device, the *open* function shall not cause the terminal device to become the controlling terminal for the process.

O_NDELAY This flag may affect subsequent reads and writes [see *read*(2) and *write*(2)].

When opening a FIFO with O_RDONLY or O_WRONLY set:

If O_NDELAY is set:

An *open* for reading-only will return without delay. An *open* for writing-only will return an error if no process currently has the file open for reading.

If O_NDELAY is clear:

An *open* for reading-only will block until a process opens the file for writing. An *open* for writing-only will block until a process opens the file for reading.

When opening a file associated with a communication line:

If O_NDELAY is set:

The open will return without waiting for carrier.

If O_NDELAY is clear:

The open will block until carrier is present.

O_NONBLOCK
This flag functions identically to O_NDELAY with regard to the *open* function. [See *read*(2) and *write*(2))].

O_APPEND If set, the file pointer will be set to the end of the file prior to each write.

O_SYNC When opening a regular file, this flag affects subsequent writes. If set, each *write*(2) will wait for both the file data and file status to be physically updated.

O_CREAT If the file exists, this flag has no effect. Otherwise, the owner ID of the file is set to the effective user ID of the process, the group ID of the file is set to the effective group ID of the process or to the group ID of the directory in which the file is being created. This is determined as follows:

If the underlying filesystem was mounted with the BSD file creation semantics flag [see *fstab*(4)] or the **S_ISGID** bit is set [see *chmod*(2)] on the parent directory, then the group ID of the new file is set to the group ID of the parent directory, otherwise it is set to the effective group ID of the calling process.

The low-order 12 bits of the file mode are set to the value of *mode* modified as follows [see *creat*(2)]:

All bits set in the file mode creation mask of the process are cleared [see *umask*(2)].

The "sticky bit" of the mode is cleared [see *chmod*(2)].

O_TRUNC If the file exists, its length is truncated to 0 and the mode and owner are unchanged.

O_EXCL If O_EXCL and O_CREAT are set, *open* will fail if the file exists.

When opening a STREAMS file, *oflag* may be constructed from O_NDELAY or-ed with either O_RDONLY, O_WRONLY or O_RDWR. Other flag values are not applicable to STREAMS devices and have no effect on them. The value of O_NDELAY affects the operation of STREAMS drivers and certain system calls [see *read*(2), *getmsg*(2), *putmsg*(2) and *write*(2)]. For drivers, the implementation of O_NDELAY is device-specific. Each STREAMS device driver may treat this option differently.

Certain flag values can be set following *open* as described in *fcntl*(2).

The file pointer used to mark the current position within the file is set to the beginning of the file.

The new file descriptor is set to remain open across *execve*(2) system calls [see *fcntl*(2)].

There is a system enforced limit on the number of open file descriptors per process *{OPEN_MAX}*, whose value is returned by the *getdtablesize*(2) function.

The named file is opened unless one or more of the following are true:

[EACCES] A component of the path prefix denies search permission.

[ENAMETOOLONG] The length of *path* exceeds *{PATH_MAX}*, or a pathname component is longer than *{NAME_MAX}*.

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

[EACCES] *oflag* permission is denied for the named file.

[EAGAIN] The file exists, mandatory file/record locking is set, and there are outstanding record locks on the file [see *chmod* (2)].

[EEXIST] O_CREAT and O_EXCL are set, and the named file exists.

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

[EINTR] A signal was caught during the *open* system call.

[EIO] A hangup or error occurred during a STREAMS *open*.

[EISDIR] The named file is a directory and *oflag* is write or read/write.

[EMFILE] The system imposed limit for open file descriptors per process *{OPEN_MAX}* has already been reached.

[ENFILE] The system file table has exceeded *{NFILE_MAX}* concurrently open files.

[ENOENT] O_CREAT is not set and the named file does not exist.

[ENOMEM] The system is unable to allocate a send descriptor.

[ENOSPC]	O_CREAT and O_EXCL are set, and the file system is out of inodes.
[ENOSR]	Unable to allocate a *stream*.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENXIO]	The named file is a character special or block special file, and the device associated with this special file does not exist.
[ENXIO]	O_NDELAY is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading.
[ENXIO]	A STREAMS module or driver open routine failed.
[EROFS]	The named file resides on a read-only file system and *oflag* is write or read/write.
[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed and *oflag* is write or read/write.
[EOPNOTSUPP]	An attempt was made to open a socket (not currently supported).

SEE ALSO

chmod(2), close(2), creat(2), dup(2), fcntl(2), getdtablesize(2), intro(2), lseek(2), read(2), getmsg(2), putmsg(2), umask(2), write(2).

DIAGNOSTICS

Upon successful completion, the file descriptor is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

pause – suspend process until signal

FORTRAN SYNOPSIS

integer *4 function pause()

DESCRIPTION

pause suspends the calling process until it receives a signal. The signal must be one that is not currently set to be ignored by the calling process.

If the signal causes termination of the calling process, *pause* will not return.

If the signal is *caught* by the calling process and control is returned from the signal-catching function [see *signal*(2)], the calling process resumes execution from the point of suspension; with a return value of –1 from *pause* and *errno* set to **EINTR.**

SEE ALSO

alarm(2), kill(2), signal(2), sigpause(2), wait(2), sigaction(2), sigpending(2), sigprocmask(2), sigsuspend(2), sigvec(3B), signal(3B), sigblock(3B), sigpause(3B), sigsetmask(3B).

NAME

pipe – create an interprocess channel

FORTRAN SYNOPSIS

integer *4 function pipe (fildes)
integer *4 fildes (2)

DESCRIPTION

pipe creates an I/O mechanism called a pipe and returns two file descriptors, *fildes*[0] and *fildes*[1]. *Fildes*[0] is opened for reading and *fildes*[1] is opened for writing.

Up to **PIPE_BUF** (defined in **limits.h**) are guaranteed to be written atomically. *Up to* **PIPE_MAX** (defined in **limits.h**) bytes of data are buffered by the pipe before the writing process is blocked. A read only file descriptor *fildes*[0] accesses the data written to *fildes*[1] on a first-in-first-out (FIFO) basis.

pipe will fail if:

[EMFILE] more than *{OPEN_MAX}-2* file descriptors are currently open.

[ENFILE] The system file table has exceeded *{NFILE_MAX}* concurrently open files.

SEE ALSO

read(2), write(2).
sh(1) in the *User's Reference Manual.*

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

plock – lock process, text, or data in memory

FORTRAN SYNOPSIS

integer *4 function plock (op)
integer *4 op

DESCRIPTION

plock allows the calling process to lock its text segment (text lock), its data and stack segments (data lock), or its text, data, and stack segments (process lock) into memory. Locked segments are immune to all routine swapping. *plock* also allows these segments to be unlocked. The effective user ID of the calling process must be super-user to use this call. *Op* specifies the following:

PROCLOCK – lock text and data segments into memory (process lock)

TXTLOCK – lock text segment into memory (text lock)

DATLOCK – lock data and stack segments into memory (data lock)

UNLOCK – remove locks

plock will fail and not perform the requested operation if one or more of the following are true:

[EPERM] The effective user ID of the calling process is not super-user.

[EINVAL] *Op* is equal to **PROCLOCK** and a process lock, a text lock, or a data lock already exists on the calling process.

[EINVAL] *Op* is equal to **TXTLOCK** and a text lock or a process lock already exists on the calling process.

[EINVAL] *Op* is equal to **DATLOCK** and a data lock or a process lock already exists on the calling process.

[EINVAL] *Op* is equal to **UNLOCK** and no type of lock exists on the calling process.

[EAGAIN] There was insufficient lockable memory to lock the requested segment. This may occur even though the amount requested was less than the system-imposed maximum number of locked pages.

[ENOMEM] The caller was not super-user and the number of pages to be locked exceeded the per process limit *{PLOCK_MAX}*.

[ENOMEM] The total number of pages locked by the caller would exceed the maximum resident size for the process [see *setrlimit*(2)].

SEE ALSO

intro(2), exec(2), exit(2), getrlimit(2), mpin(2), plock(2), shmctl(2), ulimit(2).

WARNING

If a locked data segment is grown, the newly-allocated pages are not locked into memory.

DIAGNOSTICS

Upon successful completion, a value of 0 is returned to the calling process. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

BUGS

Shared library text and data segments and mapped files are not currently affected by calls to *plock*.

NAME

prctl – operations on a process

FORTRAN SYNOPSIS

integer*4 prctl (option, [value, [value2]])
integer*4 option, value, value2

DESCRIPTION

prctl provides information about processes and the ability to control certain of their attributes. *Option* specifies one of the following actions:

PR_MAXPROCS returns the system imposed limit on the number of processes per user.

PR_MAXPPROCS returns the maximum number of processes the system is willing to run in parallel.

PR_ISBLOCKED returns 1 if the specified process is currently blocked. *value* specifies the pid. Since other processes could have subsequently unblocked the subject process, the result should be considered as a snapshot only.

PR_SETSTACKSIZE sets the maximum stack size for the current process. This affects future stack growths and forks only. The new value, suitably rounded, is returned. The value is expressed in terms of bytes. This option and the **RLIMIT_STACK** option of *setrlimit*(2) act on the same value.

PR_GETSTACKSIZE returns the current process's maximum stack size in bytes. This size is an upper limit on the size of the current process's stack.

PR_UNBLKONEXEC sets a flag so that when the calling process subsequently calls *exec*(2), the process whose pid is specified by *value* is unblocked. This can be used in conjunction with the **PR_BLOCK** option of *sproc*(2) to provide race-free process creation.

PR_SETEXITSIG controls whether all members of a share group will be signaled if any one of them terminates. If *value* is 0, then normal **IRIX** process termination rules apply, namely that the parent is sent a **SIGCLD** upon death of child, but no indication of death of parent is given. If *value* is a valid signal number [see *signal*(2)] then if any member of a share group terminates, that signal is sent to ALL surviving members of the share group.

PR_RESIDENT makes the process immune to process swapout.

PR_ATTACHADDR attaches the virtual segment containing the address *value2* in the process whose pid is *value* to the calling process. Both processes must be members of the same share group. The address of where the virtual segment was attached is returned. This address has the same logical offset into the virtual space as the passed in address.

prctl will fail if one or more of the following are true:

[EINVAL] *option* is not valid.

[ESRCH] The *value* passed with the **PR_ISBLOCKED** or **PR_UNBLKONEXEC** option doesn't match the *pid* of any process.

[EINVAL] The value given for the new maximum stack size is negative or exceeds the maximum process size allowed.

[EINVAL] The value given for the **PR_SETEXITSIG** option is not a valid signal number.

[EINVAL] The calling process already has specified a process (or the specified process is the caller itself) to be unblocked on exec via the **PR_UNBLKONEXEC** option.

[EPERM] The caller does not have permission to unblock the process specified by the *value* passed for the **PR_UNBLKONEXEC** option.

SEE ALSO

blockproc(2), signal(2), setrlimit(2), sproc(2).

DIAGNOSTICS

Upon successful completion, *prctl* returns the requested information. Otherwise, a value of −1 is returned to the calling process, and *errno* is set to indicate the error.

NAME

profil – execution time profile

FORTRAN SYNOPSIS

integer *4 function profil (buff, buffsiz, offset, scale)
integer *2 (*) buff
integer *4 bufsiz, offset, scale

DESCRIPTION

Buff points to an area of core whose length (in bytes) is given by *bufsiz*. After this call, the user's program counter (pc) is examined each clock tick (10 milliseconds); *offset* is subtracted from it, and the result multiplied by *scale*. If the resulting number corresponds to a word inside *buff*, that word is incremented.

The scale is interpreted as an unsigned, fixed-point fraction with 16 bits of fraction: 0x10000 gives a 1-1 mapping of pc's to words in *buff*; 0x8000 maps each pair of instruction words together.

Since each bucket is only 16 bits, it is concievable for it to overflow. No indication that this has occurred is given.

Profiling is turned off by giving a *scale* of 0 or 1. It is rendered ineffective by giving a *bufsiz* of 0. Profiling is turned off when an *execve* is executed, but remains on in child and parent both after a *fork* or *sproc*. Profiling is turned off if an update in *buff* would cause a memory fault.

SEE ALSO

fork(2), sproc(2), monitor(3X).

DIAGNOSTICS

A 0, indicating success, is always returned.

NAME

ptrace – process trace

FORTRAN SYNOPSIS

integer *4 function ptrace (request, pid, addr, data)
integer *4 request, pid, addr, data

DESCRIPTION

Ptrace provides a way for a process to control the execution of another process and to examine and change that process's core image. *Ptrace* is used primarily to implement breakpoint debugging.

There are four arguments whose interpretation depends on a *request* argument. Generally, *pid* is the process ID of the traced process. A process being traced behaves normally until it encounters some signal, whether internally generated (for example, "illegal instruction")) or externally generated (for example, "interrupt"). See *signal*(2) for the list.

When the traced process encounters a signal, it enters a stopped state. The process tracing it is notified by *wait*(2). If the the traced process stops with a SIGTRAP, the process might have stopped for many reasons. Two status words, which are addressable as registers, in the traced process's uarea qualify SIGTRAPs: TRAPCAUSE, which contains the cause of the trap, and TRAPINFO, which contains extra information about the trap.

When the traced process is in the stopped state, its core image can be examined and modified using *ptrace*. Another *ptrace* request can cause the traced process either to terminate or to continue, possibly ignoring the signal.

The value of the *request* argument determines the precise action of the call:

0 This request is the only one that can be used by a child process. Request 0 can declare that the child process is to be traced by its parent. All other arguments are ignored. Peculiar results happen when the parent does not expect to trace the child.

1,2 The word in the traced process's address space at *addr* is returned. If I and D space are separated (for example, historically on a PDP-11), Request 1 specifies I space and Request 2 specifies D space. *Addr* must be 4-byte aligned. The traced process must be stopped. The input *data* is ignored.

3 The word of the system's per-process data area that corresponds to *addr* is returned. *Addr* is a constant defined in ptrace.h. This space contains the registers and other information about the process. The constants correspond to fields in the system's *user* structure.

4,5 The specified *data* is written at the word in the process's address space corresponds to *addr*. *Addr* must be 4-byte aligned. Upon successful completion, the value written into the address space of the child is returned to the parent. If I and D space are separated, Request 4 specifies I space and Request 5 specifies D space. Attempts to write in pure procedure fail when another process is executing the same file.

6 The process's system data is written as it is read with Request 3. Only a few locations can be written this way: the general registers, the floating point status and registers, and certain bits of the processor status word. The old value at the address is returned.

7 The *data* argument is taken as a signal number and the traced process's execution continues at location *addr* as if it had incurred that signal. The signal number can be 0, indicating the signal that caused the stop should be ignored, or the signal can be the value fetched from the process's image, indicating what signal caused the stop. If *addr* is (int *)1, execution continues from where it stopped.

8 The traced process terminates. The *addr* argument is ignored and the *data* argument is the signal specified in Request 7.

9 Execution continues as in Request 7; however, as soon as possible after execution of at least one instruction, execution stops again. The signal number from the stop is SIGTRAP. TRAPCAUSE contains CAUSESINGLE. This part of *ptrace* is used to implement breakpoints. The *addr* and *data* arguments are defined in Request 7.

Requests 20-29 have not been fully defined or implemented.

20 This request is the same as Request 0, except it is executed by the tracing process and the pid field is non-zero. That pid's process pid stops execution. On a signal, it becomes a traced process that returns control to the tracing process rather than to the parent. The tracing process must have the same user-id (uid) as the traced process.

21,22
These requests return MAXREG general registers or MAXFREG floating registers, respectively. Their values are copied to the locations starting at the address in the tracing process specified by the *addr* argument.

24,25
These requests are the same as Requests 20 and 21, except that they write the registers instead of reading them.

26 This request specifies a watchpoint in the data or stack segment of the traced process. If any byte address (starting at the *addr* argument and continuing for the number of bytes specified by the *data* argument) is accessed in an instruction, the traced process stops execution with a SIGTRAP. TRAPCAUSE contains CAUSEWATCH; TRAPINFO contains the address causing the trap. *Ptrace* returns a wid (watchpoint identifier). MAXWIDS specifies the maximum number of watchpoints per process.

27 This request's data argument specifies a wid to delete.

28 This request turns off tracing for the traced process that has the specified pid.

29 This request returns an open file descriptor for the file attached to pid. This is useful for accessing the symbol table of an execed process.

These calls (except for Requests 0 and 20) can be used only when the subject process has terminated. The *wait* call determines when a process terminates. Then, the "termination" status returned by *wait* has the value 0177 to show stoppage rather than termination. If multiple processes are being traced, **wait** can be called multiple times and returns the status for the next stopped child, terminated child, or traced process.

To prevent fraud, *ptrace* inhibits the set-user-id and set-group-id facilities on later *execve*(2) calls. If a traced process calls *execve*, the process terminates before executing the first instruction of the new image showing signal SIGTRAP. TRAPCAUSE contains CAUSEEXEC; TRAPINFO does not contain anything interesting. If a traced process execs again, the same thing happens.

If a traced process forks, both parent and child are traced, and the breakpoints from the parent are copied into the child. At the time of the fork, the child stops with a SIGTRAP. The tracing process can end the trace, if desired. TRAPCAUSE contains CAUSEFORK; TRAPINFO contains the its parent's pid.

RETURN VALUE

If the call succeeds, a 0 value is returned. If the call fails, then a −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

[EINVAL]	The request code is invalid.
[EINVAL]	The specified process does not exist.
[EINVAL]	The given signal number is invalid.

[EFAULT] The specified address is out of bounds.

[EPERM] The specified process cannot be traced.

SEE ALSO

wait(2), sigvec(2).

BUGS

There is file system called /debug where each "file" is actually an active process. The process' file name is /debug/processid where processid is the process number. *open*(2), *read*(2), etc can be used to acess the (running) process. Use *fcntl*(2) to control the process. See *<sys/fs/dbfcntl.h>* for a list of the control functions available. The /debug facility solves the problems with *ptrace* mentioned below.

Ptrace is unique and arcane; it should be replaced with a special file that can be opened, read, and written. The control functions could be implemented with *ioctl*(2) calls on this file. This would be easier to understand and have much higher performance.

The Request 0 call should specify signals that are to be treated normally and should not cause a termination. Then, programs with simulated floating point (which use "illegal instruction" signals at a high rate) could be efficiently debugged.

The error indication −1 is a legitimate function value *errno*. See *intro*(2) to disambiguate.

It should be possible to stop a process on occurrence of a system call. In this way, a completely controlled environment could be provided.

NAME

read – read from file

FORTRAN SYNOPSIS

integer *4 function read (fildes, buf, nbyte)
integer *4 fildes
character * (*) buf
integer *4 byte

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*(2), *open*(2), *dup*(2), *fcntl*(2), *socket*(2), *socketpair*(2), or *pipe*(2) system call.

read attempts to read *nbyte* bytes from the file associated with *fildes* into the buffer pointed to by *buf*.

On devices capable of seeking, the *read* starts at a position in the file given by the file pointer associated with *fildes*. Upon return from *read*, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. The value of a file pointer associated with such a file is undefined.

Upon successful completion, *read* returns the number of bytes actually read and placed in the buffer; this number may be less than *nbyte* if the file is associated with a communication line [see *ioctl*(2) and *termio*(7)], or a socket [see *socket*(2)], or if the number of bytes left in the file is less than *nbyte* bytes. A value of 0 is returned when an end-of-file has been reached.

A *read* from a STREAMS [see *intro*(2)] file can operate in three different modes: "byte-stream" mode, "message-nondiscard" mode, and "message-discard" mode. The default is byte-stream mode. This can be changed using the I_SRDOPT *ioctl* request [see *streamio*(7)], and can be tested with the I_GRDOPT *ioctl*. In byte-stream mode, *read* will retrieve data from the *stream* until it has retrieved *nbyte* bytes, or until there is no more data to be retrieved. Byte-stream mode ignores message boundaries.

In STREAMS message-nondiscard mode, *read* retrieves data until it has read *nbyte* bytes, or until it reaches a message boundary. If the *read* does not retrieve all the data in a message, the remaining data are replaced on the *stream*, and can be retrieved by the next *read* or *getmsg*(2) call. Message-discard mode also retrieves data until it has retrieved *nbyte* bytes, or it reaches a message boundary. However, unread data remaining in a message after the *read* returns are discarded, and are not available for a subsequent *read* or *getmsg*.

When attempting to read from a regular file with mandatory file/record locking set [see *chmod*(2)], and there is a blocking (i.e. owned by another process) write lock on the segment of the file to be read:

If O_NDELAY or O_NONBLOCK is set, the read will return a -1 and set errno to EAGAIN.

If O_NDELAY and O_NONBLOCK are clear, the read will sleep until the blocking record lock is removed.

When attempting to read from an empty pipe (or FIFO):

If O_NDELAY is set, the read will return a 0.

If O_NONBLOCK is set, the read will return a -1 and set errno to EAGAIN.

If O_NDELAY and O_NONBLOCK are clear, the read will block until data is written to the file or the file is no longer open for writing.

When attempting to read a file associated with a tty that has no data currently available:

If O_NDELAY is set, the read will return a 0.

If O_NONBLOCK is set, the read will return a -1 and set errno to EAGAIN.

If O_NDELAY and O_NONBLOCK are clear, the read will block until data becomes available.

When attempting to read a file associated with a *stream* that has no data currently available:

If O_NDELAY or O_NONBLOCK is set, the read will return a -1 and set errno to EAGAIN.

If O_NDELAY and O_NONBLOCK are clear, the read will block until data becomes available.

Due to the different semantics of O_NDELAY and O_NONBLOCK in two of the above 4 cases, these flags *must* not be used simultaneously.

When reading from a STREAMS file, handling of zero-byte messages is determined by the current read mode setting. In byte-stream mode, *read* accepts data until it has read *nbyte* bytes, or until there is no more data to read, or until a zero-byte message block is encountered. *read* then returns the number of bytes read, and places the zero-byte message back on the *stream* to be retrieved by the next *read* or *getmsg*. In the two other modes, a zero-byte message returns a value of 0 and the message is removed from the *stream*. When a zero-byte message is read as the first message on a *stream*, a value of 0 is returned regardless of the read mode.

A *read* from a STREAMS file can only process data messages. It cannot process any type of protocol message and will fail if a protocol message is encountered at the *stream head.*

read will fail if one or more of the following are true:

[EAGAIN] Mandatory file/record locking was set, O_NDELAY was set, and there was a blocking record lock.

[ENOMEM] Insufficient amount of system virtual memory is available with which to map the user pages when reading via raw IO.

[EAGAIN] No message waiting to be read on a *stream* and O_NDELAY flag set.

[EBADF] *Fildes* is not a valid file descriptor open for reading.

[EBADMSG] Message waiting to be read on a *stream* is not a data message.

[EDEADLK] The read was going to go to sleep and cause a deadlock situation to occur.

[EFAULT] *Buf* points outside the allocated address space.

[EINTR] A signal was caught during the *read* system call.

[EINVAL] Attempted to read from a *stream* linked to a multiplexor.

[ENOLCK] The system record lock table was full, so the read could not go to sleep until the blocking record lock was removed.

A *read* from a STREAMS file will also fail if an error message is received at the *stream head.* In this case, *errno* is set to the value returned in the error message. If a hangup occurs on the *stream* being read, *read* will continue to operate normally until the *stream head* read queue is empty. Thereafter, it will return 0.

SEE ALSO

creat(2), dup(2), fcntl(2), ioctl(2), intro(2), open(2), pipe(2), getmsg(2), socket(2).

streamio(7), termio(7) in the *System Administrator's Reference Manual.*

DIAGNOSTICS

Upon successful completion a non-negative integer is returned indicating the number of bytes actually read. Otherwise, a −1 is returned and *errno* is set to indicate the error.

NAME

readlink – read value of a symbolic link

FORTRAN SYNOPSIS

integer *4 function readlink (path, buf, bufsize)
character * (*) path, buf
integer *4 bufsize

DESCRIPTION

Readlink places the contents of the symbolic link *path* in the buffer *buf* which has size *bufsiz*. The contents of the link are not null terminated when returned.

RETURN VALUE

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 and the file mode will be unchanged if:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[ENXIO]	The named file is not a symbolic link.
[EACCES]	Search permission is denied on a component of the path prefix.
[EFAULT]	*Buf* extends outside the process's allocated address space.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.

SEE ALSO

stat(2), symlink(2)

NAME

rmdir – remove a directory

FORTRAN SYNOPSIS

integer *4 function rmdir (path)
character * (*) path

DESCRIPTION

rmdir removes the directory named by the path name pointed to by *path*. The directory must not have any entries other than "." and "..".

The named directory is removed unless one or more of the following are true:

[EINVAL]	The current directory may not be removed.
[EINVAL]	The "." entry of a directory may not be removed.
[EEXIST]	The directory contains entries other than those for "." and "..".
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named directory 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 directory to be removed.
[EACCES]	The parent directory of the directory to be removed has the sticky bit set and the parent directory is not owned by the user and the directory to be removed is not owned by the user and the directory to be removed is not writable by the user and the user is not superuser.
[ENAMETOOLONG]	The length of *path* exceeds *{PATH_MAX}*, or a path-name component is longer than *{NAME_MAX}*.
[EBUSY]	The directory to be removed is the mount point for a mounted file system.
[EROFS]	The directory entry to be removed is part of a read-only file system.
[EFAULT]	*Path* points outside the process's allocated address space.

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

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

mkdir(2).

rmdir(1), rm(1), and mkdir(1) in the *User's Reference Manual.*

NAME

send, sendto, sendmsg – send a message from a socket

FORTRAN SYNOPSIS

integer *4 function send (s, msg, len, flags)
integer *4 s
character * (*) msg
integer *4 len, 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, then the error EMSGSIZE is returned, and the message is not transmitted.

No indication of failure to deliver is implicit in a *send*. Return values of –1 indicate some locally detected errors.

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_DONTROUTE    0x4    /* bypass routing,
                                    use direct interface */
```

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_DONTROUTE is usually used only by diagnostic or routing programs.

See *recv*(2) for a description of the *msghdr* structure.

RETURN VALUE

The call returns the number of characters sent, or –1 if an error occurred.

ERRORS

[EBADF]	An invalid descriptor was specified.
[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.
[EWOULDBLOCK]	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.

SEE ALSO

fcntl(2), recv(2), select(2), getsockopt(2), socket(2), write(2)

NAME

setpgrp, BSDsetpgrp – set process group ID (System V and 4.3BSD)

FORTRAN SYNOPSIS

integer *4 function setpgrp ()

DESCRIPTION

The System V version of *setpgrp* sets the process group ID of the calling process to the process ID of the calling process and returns the new process group ID.

The BSD version of *setpgrp* set 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.

ERRORS: BSD VERSION ONLY

setpgrp and *BSDsetpgrp* will fail and the process group will not be altered if one of the following occur:

[ESRCH] The requested process does not exist.

[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.

SEE ALSO

exec(2), fork(2), getpgrp(2), getpid(2), intro(2), kill(2), setpgid(2), signal(2).

DIAGNOSTICS

The System V version of *setpgrp* returns the value of the new process group ID with no possibility of error. The BSD version also returns the new process group ID if the operation was successful. If the request failed, –1 is returned and the global variable *errno* indicates the reason.

NAME

setuid, setgid – set user and group IDs

FORTRAN SYNOPSIS

integer *4 function setuid (uid)
integer *4 uid

integer *4 function setgid (gid)
integer *4 gid

DESCRIPTION

setuid (*setgid*) is used to set the real user (group) ID and effective user (group) ID of the calling process.

If the effective user ID of the calling process is super-user, the real user (group) ID and effective user (group) ID are set to *uid* (*gid*).

If the effective user ID of the calling process is not super-user, but its real user (group) ID is equal to *uid* (*gid*), the effective user (group) ID is set to *uid* (*gid*).

If the effective user ID of the calling process is not super-user, but the saved set-user (group) ID from *exec*(2) is equal to *uid* (*gid*), the effective user (group) ID is set to *uid* (*gid*).

setuid or *setgid* will fail if one or more of the following are true:

[EPERM] *setuid* (*setgid*) will fail if the real user (group) ID of the calling process is not equal to *uid* (*gid*) and its effective user ID is not super-user.

[EINVAL] The *uid (gid)* is out of range.

SEE ALSO

getuid(2), intro(2).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

sginap – timed sleep and processor yield function

FORTRAN SYNOPSIS

subroutine sginap (ticks)
integer *4 ticks

DESCRIPTION

The *sginap* system call provides two functions. With an argument of 0, it yields the processor to any higher or equal priority processes immediately, thus potentially allowing another process to run. Note that because normally the user has no direct control over the exact priority of a given process, this does not guarantee that another process will run.

With an argument which is non-zero, *sginap* will suspend the process for **ticks** clock ticks. The length of a clock tick is defined by **CLK_TCK** in the include file *<limits.h>*. This is the same for all *IRIS-4D* products.

SEE ALSO

sleep(3), alarm(2), pause(2), schedctl(2), setitimer(2).

NAME

shmop: shmat, shmdt – shared memory operations

FORTRAN SYNOPSIS

integer *4 function shmdt (shmaddr)
character * (*) shmaddr

DESCRIPTION

shmat attaches the shared memory segment associated with the shared memory identifier specified by *shmid* to the data segment of the calling process. The segment is attached at the address specified by one of the following criteria:

If *shmaddr* is equal to zero, the segment is attached at the first available address as selected by the system.

If *shmaddr* is not equal to zero and (*shmflg* & **SHM_RND**) is "true", the segment is attached at the address given by (*shmaddr* - (*shmaddr* modulus **SHMLBA**)).

If *shmaddr* is not equal to zero and (*shmflg* & **SHM_RND**) is "false", the segment is attached at the address given by *shmaddr*.

shmdt detaches from the calling process's data segment the shared memory segment located at the address specified by *shmaddr*.

The segment is attached for reading if (*shmflg* & **SHM_RDONLY**) is "true" {READ}, otherwise it is attached for reading and writing {READ/WRITE}.

shmat will fail and not attach the shared memory segment if one or more of the following are true:

[EINVAL] *Shmid* is not a valid shared memory identifier.

[EACCES] Operation permission is denied to the calling process [see *intro*(2)].

[ENOMEM] The available virtual space of the caller (either total size *{PROCSIZE_MAX}* or a large enough gap between other previously allocated virtual spaces) cannot accommodate the shared memory segment.

[EINVAL] *Shmaddr* is not equal to zero, and the value of (*shmaddr* - (*shmaddr* modulus **SHMLBA**)) is an illegal address.

[EINVAL] *Shmaddr* is not equal to zero, (*shmflg* & **SHM_RND**) is "false", and the value of *shmaddr* is an illegal address. Attach addresses must be a multiple of **SHMLBA** [see **<sys/shm.h>**].

[EMFILE] The number of shared memory segments attached to the calling process would exceed the system-imposed limit *{SHMAT_MAX}* [see *intro*(2)].

shmdt will fail and not detach the shared memory segment if one or more of the following are true:

[EBUSY] The shared memory segment is in use by another member of the calling process's share group [see *sproc*(2)].

[EINVAL] *Shmaddr* is not the start address of a shared memory segment.

SEE ALSO

exec(2), exit(2), fork(2), intro(2), shmctl(2), shmget(2), sproc(2).

DIAGNOSTICS

Upon successful completion, the return value is as follows:

shmat returns the data segment start address of the attached shared memory segment.

shmdt returns a value of 0.

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

NOTES

The user must explicitly remove shared memory segments after the last reference to them has been removed.

NAME

signal – software signal facilities (System V)

FORTRAN SYNOPSIS

integer function signal (sig, func, flag)
integer sig, func, flag
external func

DESCRIPTION

signal allows the calling process to choose one of three ways in which it is possible to handle the receipt of a specific signal. *Sig* specifies the signal and *func* specifies the choice.

FORTRAN interface routine signal takes an extra argument *flag*. if *flag* is a negative value *func* must be an external FORTRAN procedure name. Otherwise, *func* is ignored and *flag* can contain **SIG_DFL, SIG_IGN,** or the address of a C signal-handling routine. In this case, *flag* will be passed to the system call *signal* as *func*. *flag* may be the value returned from a previous call to *signal* and, thus, can be used to restore a previous action definition. Note that *flag* can only be an integer variable containing the address of a C function and not the C function itself. *Sig* is the signal to be caught, and must be in the range

(0 < sig < NSIG).

Sig can be assigned any one of the following except **SIGKILL** or **SIGSTOP:**

SIGHUP	01	hangup
SIGINT	02	interrupt
SIGQUIT	03[1]	quit
SIGILL	04[1]	illegal instruction (not reset when caught)
SIGTRAP	05[1][5]	trace trap (not reset when caught)
SIGABRT	06[1]	abort
SIGEMT	07[1][4]	EMT instruction
SIGFPE	08[1]	floating point exception
SIGKILL	09	kill (cannot be caught or ignored)
SIGBUS	10[1]	bus error
SIGSEGV	11[1]	segmentation violation
SIGSYS	12[1]	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGUSR1	16	user-defined signal 1
SIGUSR2	17	user-defined signal 2
SIGCLD	18[2]	death of a child

SIGPWR	19[2]	power fail (not reset when caught)
SIGSTOP	20[6]	stop (cannot be caught or ignored)
SIGTSTP	21[6]	stop signal generated from keyboard
SIGPOLL	22[3]	selectable event pending
SIGIO	23[2]	input/output possible
SIGURG	24[2]	urgent condition on IO channel
SIGWINCH	25[2]	window size changes
SIGVTALRM	26	virtual time alarm
SIGPROF	27	profiling alarm
SIGCONT	28[6]	continue after stop (cannot be ignored)
SIGTTIN	29[6]	background read from control terminal
SIGTTOU	30[6]	background write to control terminal
SIGXCPU	31	cpu time limit exceeded [see *setrlimit*(2)]
SIGXFSZ	32	file size limit exceeded [see *setrlimit*(2)]

Func is assigned one of three values: **SIG_DFL** or **SIG_IGN**, which are macros (defined in <*sys/signal.h*>) that expand to constant expressions, or a *function address*.

The actions prescribed by its value are as follows:

SIG_DFL – terminate process upon receipt of a signal

Upon receipt of the signal *sig*, the receiving process is to be terminated with all of the consequences outlined in *exit*(2). See SIGNAL NOTES [1] below.

SIG_IGN – ignore signal

The signal *sig* is to be ignored.

Note: the signals **SIGKILL, SIGSTOP** and **SIGCONT** cannot be ignored.

function address – catch signal

Upon receipt of the signal *sig*, the receiving process is to execute the signal-catching function whose address is specified via this parameter. The function will be invoked as follows:

handler (int sig, int code, struct sigcontext *sc);

Where *handler* is the specified function-name. *code* is valid only in the following cases:

Condition	Signal	Code
User breakpoint	SIGTRAP	BRK_USERBP
User breakpoint	SIGTRAP	BRK_SSTEPBP
Integer overflow	SIGTRAP	BRK_OVERFLOW
Divide by zero	SIGTRAP	BRK_DIVZERO
Multiply overflow	SIGTRAP	BRK_MULOVF

Invalid virtual address	SIGSEGV	EFAULT
Read-only address	SIGSEGV	EACCESS
Read beyond mapped object	SIGSEGV	ENXIO

The third argument *sc* is a pointer to a *struct sigcontext (defined in <sys/signal.h>)* that contains the processor context at the time of the signal. The FORTRAN arguments are defined in the same way except for the last argument which can be defined either as an array of integers or as a record.

Upon return from the signal-catching function, the receiving process will resume execution at the point it was interrupted.

Before entering the signal-catching function, the value of *func* for the caught signal will be set to **SIG_DFL** unless the signal is **SIGILL, SIGTRAP,** or **SIGPWR**. This means that before exiting the handler, a *signal* call is necessary to again set the disposition to catch the signal.

When a signal that is to be caught occurs during a *read*(2), a *write*(2), an *open*(2), or an *ioctl*(2) system call on a slow device (like a terminal; but not a file), during a *pause*(2) system call, or during a *wait*(2) system call that does not return immediately due to the existence of a previously stopped or zombie process, the signal catching function will be executed and then the interrupted system call may return a −1 to the calling process with *errno* set to **EINTR**.

Note: The signals **SIGKILL** and **SIGSTOP** cannot be caught.

SIGNAL NOTES

[1] If **SIG_DFL** is assigned for these signals, in addition to the process being terminated, a "core image" will be constructed in the current working directory of the process, if the following conditions are met:

The effective user ID and the real user ID of the receiving process are equal.

An ordinary file named **core** exists and is writable or can be created. If the file must be created, it will have the following properties:

a mode of 0666 modified by the file creation mask [see *umask*(2)]

a file owner ID that is the same as the effective user ID of the receiving process.

a file group ID that is the same as the effective group ID of the receiving process

NOTE: The core file may be truncated if the resultant file size would exceed either *ulimit* [see *ulimit*(2)] or the process's maximum core file size [see *setrlimit*(2)].

[2] For the signals **SIGCLD, SIGWINCH, SIGPWR, SIGURG,** and **SIGIO,** the handler parameter is assigned one of three values: **SIG_DFL, SIG_IGN,** or a *function address*. The actions prescribed by these values are:

SIG_DFL – ignore signal
The signal is to be ignored.

SIG_IGN – ignore signal
The signal is to be ignored. Also, if *sig* is **SIGCLD,** the calling process's child processes will not create zombie processes when they terminate [see *exit*(2)].

function address – catch signal
If the signal is **SIGPWR, SIGURG, SIGIO,** or **SIGWINCH,** the action to be taken is the same as that described above for a handler parameter equal to *function address*. The same is true if the signal is **SIGCLD** with one exception: while the process is executing the signal-catching function, all terminating child processes will be queued. The *wait* system call removes the first entry of the queue. If the *signal* system call is used to catch **SIGCLD,** the signal handler must be re-attached when exiting the handler, and at that time--if the queue is not empty--**SIGCLD** is re-raised before *signal* returns. See *wait*(2).

In addition, **SIGCLD** affects the *wait* and *exit* system calls as follows:

wait If the handler parameter of **SIGCLD** is set to **SIG_IGN** and a *wait* is executed, the *wait* will block until all of the calling process's child processes terminate; it will then return a value of –1 with *errno* set to **ECHILD.**

exit If in the exiting process's parent process the handler parameter of **SIGCLD** is set to **SIG_IGN,** the exiting process will not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the proceeding processes. A process that may be piped into in this manner (and thus become the parent of other processes) should take care not to set **SIGCLD** to be caught.

[3] **SIGPOLL** is issued when a file descriptor corresponding to a STREAMS [see *intro*(2)] file has a "selectable" event pending. A process must specifically request that this signal be sent using the **I_SETSIG** *ioctl* call. Otherwise, the process will never receive **SIGPOLL**.

[4] **SIGEMT** is never generated on an **IRIS-4D** system.

[5] **SIGTRAP** is generated for breakpoint instructions, overflows, divide by zeros, range errors, and multiply overflows. The second argument *code* gives specific details of the cause of the signal. Possible values are described in <*sys/signal.h*>.

[6] The signals **SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU** and **SIGCONT** are used by command interpreters like the C shell [see *csh*(1)] to provide job control. The first four signals listed will cause the receiving process to be stopped, unless the signal is caught or ignored. **SIGCONT** causes a stopped process to be resumed. **SIGTSTP** is sent from the terminal driver in response to the SWTCH character being entered from the keyboard [see *termio*(7)]. **SIGTTIN** is sent from the terminal driver when a background process attempts to read from its controlling terminal. If **SIGTTIN** is ignored by the process, then the read will return **EIO**. **SIGTTOU** is sent from the terminal driver when a background process attempts to write to its controlling terminal when the terminal is in **TOSTOP** mode. If **SIGTTOU** is ignored by the process, then the write will succeed regardless of the state of the controlling terminal.

EXAMPLES

This is an example in FORTRAN:

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

EXTERNAL FPROC
INTEGER SIGNAL
INTEGER CPADDR

I = SIGNAL(SIGTERM, FPROC, -1)
J = SIGNAL(SIGINT, 0, CPADDR)
```

The first call to *signal* sets up the FORTRAN function *fproc* as the signal-handling routine for **SIGTERM**. The second call sets up a C function whose address is in the variable *cpaddr* as the signal-handling routine for **SIGINT**.

NOTES

signal will not catch an invalid function argument, *func*, and results are undefined when an attempt is made to execute the function at the bad address.

SIGKILL will immediately terminate a process, regardless of its state. Processes which are stopped via job control (typically <Ctrl>-Z) will not act upon any delivered signals other than SIGKILL until the job is restarted. Processes which are blocked via a *blockproc* system call will unblock if they receive a signal which is fatal (i.e., a non-job-control signal which the are NOT catching), but will still be stopped if the job of which they are a part is stopped. Only upon restart will they die. Any non-fatal signals received by a blocked process will NOT cause the process to be unblocked (an *unblockproc*(2) or *unblockprocall*(2) system call is necessary).

A call to *signal* cancels a pending signal *sig* except for a pending SIGKILL signal.

[EINVAL] *signal* will fail if *sig* is an illegal signal number, including SIGKILL and SIGSTOP.

[EINVAL] *signal* will fail if an illegal operation is requested (for example, ignoring SIGCONT, which is ignored by default).

After a *fork*(2) the child inherits all handlers and signal masks, but not the set of the pending signals.

The *exec*(2) routines reset all caught signals to the default action; ignored signals remain ignored; the blocked signal mask is unchanged and pending signals remain pending.

SEE ALSO

intro(2), blockproc(2), kill(2), pause(2), ptrace(2), sigaction(2), sigset(2), wait(2), setjmp(3C), sigvec(3B).
kill(1) in the *User's Reference Manual.*

DIAGNOSTICS

Upon successful completion, *signal* returns the previous value of *func* for the specified signal *sig*. Otherwise, a value of SIG_ERR is returned and *errno* is set to indicate the error. SIG_ERR is defined in the header file *<sys/signal.h>*.

WARNINGS

Signals raised by the instruction stream, SIGILL, SIGEMT, SIGBUS, SIGSEGV will cause infinite loops if their handler returns, or the action is set to SIG_IGN.

WARNING

The POSIX signal routines (*sigaction*(2), *sigpending*(2), *sigprocmask*(2), *sigsuspend*(2), *sigsetjmp*(3)), and the 4.3BSD signal routines (*sigvec*(3B), *signal*(3B), *sigblock*(3B), *sigpause*(3B), *sigsetmask*(3B)) must NEVER be used with *signal*(2) or *sigset*(2).

Before entering the signal-catching function, the value of *func* for the caught signal will be set to **SIG_DFL** unless the signal is **SIGILL**, **SIGTRAP**, or **SIGPWR**. This means that before exiting the handler, a *signal* call is necessary to again set the disposition to catch the signal.

Note that handlers installed by *signal* execute with *no* signals blocked, not even the one that invoked the handler.

NAME

sigset, sighold, sigrelse, sigignore, sigpause – signal management (System V)

FORTRAN SYNOPSIS

integer *4 function sighold (sig)
integer *4 sig

integer *4 function sigrelse (sig)
integer *4 sig

integer *4 function sigignore (sig)
integer *4 sig

integer *4 function sigpause (sig)
integer *4 sig

DESCRIPTION

These functions provide signal management for application processes. *sigset* specifies the system signal action to be taken upon receipt of signal *sig*. This action is either calling a process signal-catching handler *func* or performing a system-defined action.

sighold and *sigrelse* are used to establish critical regions of code. *sighold* is analogous to raising the priority level and deferring or holding a signal until the priority is lowered by *sigrelse*. *sigrelse* restores the system signal action to that specified previously by *sigset*.

sigignore sets the action for signal *sig* to SIG_IGN (see below).

sigpause suspends the calling process until it receives a signal, the same as *pause*(2). However, if the signal *sig* had been received and held, it is released and the system signal action taken. This system call is useful for testing variables that are changed on the occurrence of a signal. The correct usage is to use *sighold* to block the signal first, then test the variables. If they have not changed, then call *sigpause* to wait for the signal.

Sig can be assigned any one of the following values except SIGKILL and SIGSTOP:

SIGHUP	01	hangup
SIGINT	02	interrupt
SIGQUIT	03[1]	quit
SIGILL	04[1]	illegal instruction (not reset when caught)
SIGTRAP	05[1][5]	trace trap (not reset when caught)
SIGABRT	06[1]	abort
SIGEMT	07[1][4]	EMT instruction
SIGFPE	08[1]	floating point exception

SIGKILL	09	kill (cannot be caught or ignored)
SIGBUS	10[1]	bus error
SIGSEGV	11[1]	segmentation violation
SIGSYS	12[1]	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGUSR1	16	user-defined signal 1
SIGUSR2	17	user-defined signal 2
SIGCLD	18[2]	death of a child
SIGPWR	19[2]	power fail (not reset when caught)
SIGSTOP	20[6]	stop (cannot be caught or ignored)
SIGTSTP	21[6]	stop signal generated from keyboard
SIGPOLL	22[3]	selectable event pending
SIGIO	23[2]	input/output possible
SIGURG	24[2]	urgent condition on IO channel
SIGWINCH	25[2]	window size changes
SIGVTALRM	26	virtual time alarm
SIGPROF	27	profiling alarm
SIGCONT	28[6]	continue after stop (cannot be ignored)
SIGTTIN	29[6]	background read from control terminal
SIGTTOU	30[6]	background write to control terminal
SIGXCPU	31	cpu time limit exceeded [see *setrlimit*(2)]
SIGXFSZ	32	file size limit exceeded [see *setrlimit*(2)]

Func is assigned one of four values: **SIG_DFL**, **SIG_IGN**, or **SIG_HOLD**, which are macros (defined in <*sys/signal.h*>) that expand to constant expressions, or a *function address*.

The actions prescribed by its value are as follows:

SIG_DFL – terminate process upon receipt of a signal
Upon receipt of the signal *sig*, the receiving process is to be terminated with all of the consequences outlined in *exit*(2). See SIGNAL NOTES [1] below.

SIG_IGN – ignore signal
The signal *sig* is to be ignored.

Note: the signals **SIGKILL**, **SIGSTOP** and **SIGCONT** cannot be ignored.

SIG_HOLD – hold signal
The signal *sig* is to be held upon receipt. Any pending signal of this type remains held. Only one signal of each type is held.

function address – catch signal

Upon receipt of the signal *sig*, the receiving process is to execute the signal-catching function whose address is specified via this parameter. The function will be invoked as follows:

handler (int sig, int code, struct sigcontext *sc);

Where *handler* is the specified function-name. *code* is valid only in the following cases:

Condition	Signal	Code
User breakpoint	SIGTRAP	BRK_USERBP
User breakpoint	SIGTRAP	BRK_SSTEPBP
Integer overflow	SIGTRAP	BRK_OVERFLOW
Divide by zero	SIGTRAP	BRK_DIVZERO
Multiply overflow	SIGTRAP	BRK_MULOVF
Invalid virtual address	SIGSEGV	EFAULT
Read-only address	SIGSEGV	EACCESS
Read beyond mapped object	SIGSEGV	ENXIO

The third argument *sc* is a pointer to a *struct sigcontext* (defined in <*sys/signal.h*>) that contains the processor context at the time of the signal. The FORTRAN arguments are defined in the same way except for the last argument which can be defined either as an array of integers or as a record.

Before the handler is invoked the signal action will be changed to **SIG_HOLD**.

The signal-catching function remains installed after it is invoked. During normal return from the signal-catching handler, the system signal action is restored to *func* and any held signal of this type released. If a non-local goto (*longjmp*) is taken, then *sigrelse* must be called to restore the system signal action and release any held signal of this type.

Upon return from the signal-catching function, the receiving process will resume execution at the point it was interrupted. See WARNINGS below.

When a signal that is to be caught occurs during a *read*(2), a *write*(2), an *open*(2), or an *ioctl*(2) system call on a slow device (like a terminal; but not a file), during a *pause*(2) system call, or during a *wait*(2) system call that does not return immediately due to the existence of a previously stopped or zombie process, the signal catching function will be executed and then the interrupted system call may return a –1 to the

calling process with *errno* set to **EINTR**.

Note: The signals **SIGKILL** and **SIGSTOP** cannot be caught.

SIGNAL NOTES

[1] If **SIG_DFL** is assigned for these signals, in addition to the process being terminated, a "core image" will be constructed in the current working directory of the process, if the following conditions are met:

The effective user ID and the real user ID of the receiving process are equal.

An ordinary file named **core** exists and is writable or can be created. If the file must be created, it will have the following properties:

a mode of 0666 modified by the file creation mask [see *umask*(2)]

a file owner ID that is the same as the effective user ID of the receiving process.

a file group ID that is the same as the effective group ID of the receiving process

NOTE: The core file may be truncated if the resultant file size would exceed either *ulimit* [see *ulimit*(2)] or the process's maximum core file size [see *setrlimit*(2)].

[2] For the signals **SIGCLD, SIGWINCH, SIGPWR, SIGURG,** and **SIGIO,** the handler parameter is assigned one of three values: **SIG_DFL, SIG_IGN,** or a *function address*. The actions prescribed by these values are:

SIG_DFL - ignore signal
The signal is to be ignored.

SIG_IGN - ignore signal
The signal is to be ignored. Also, if *sig* is **SIGCLD**, the calling process's child processes will not create zombie processes when they terminate [see *exit*(2)].

function address - catch signal
If the signal is **SIGPWR, SIGWINCH, SIGURG,** or **SIGIO,** the action to be taken is the same as that described above for a handler parameter equal to *function address*. The same is true if the signal is **SIGCLD** with one exception: while the process is executing the signal-catching function, all terminating child processes will be queued. The *wait* system call removes the first entry of the queue. To ensure that no **SIGCLD**'s are missed while executing in a

SIGCLD handler, it is necessary to call *sigset* to re-attach the handler before exiting from it, and at that time--if the queue is not empty--SIGCLD is re-raised before *sigset* returns. See *wait*(2). If the signal handler is simply exited from, then SIGCLD will NOT be re-raised automatically.

In addition, SIGCLD affects the *wait* and *exit* system calls as follows:

wait If the handler parameter of SIGCLD is set to SIG_IGN and a *wait* is executed, the *wait* will block until all of the calling process's child processes terminate; it will then return a value of −1 with *errno* set to ECHILD.

exit If in the exiting process's parent process the handler parameter of SIGCLD is set to SIG_IGN, the exiting process will not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the proceeding processes. A process that may be piped into in this manner (and thus become the parent of other processes) should take care not to set SIGCLD to be caught.

[3] SIGPOLL is issued when a file descriptor corresponding to a STREAMS [see *intro*(2)] file has a "selectable" event pending. A process must specifically request that this signal be sent using the I_SETSIG *ioctl* call. Otherwise, the process will never receive SIGPOLL.

[4] SIGEMT is never generated on an **IRIS-4D** system.

[5] SIGTRAP is generated for breakpoint instructions, overflows, divide by zeros, range errors, and multiply overflows. The second argument *code* gives specific details of the cause of the signal. Possible values are described in <*sys/signal.h*>.

[6] The signals SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU and SIGCONT are used by command interpreters like the C shell [see *csh*(1)] to provide job control. The first four signals listed will cause the receiving process to be stopped, unless the signal is caught or ignored. SIGCONT causes a stopped process to be resumed. SIGTSTP is sent from the terminal driver in response to the SWTCH character being entered from the keyboard [see *termio*(7)]. SIGTTIN is sent from the terminal driver when a background process attempts to read from its controlling terminal. If SIGTTIN is ignored by the process, then the read will return EIO. SIGTTOU is sent from the terminal driver when a background process attempts to write to its controlling terminal when the terminal is in TOSTOP mode. If SIGTTOU is ignored by the

process, then the write will succeed regardless of the state of the controlling terminal.

EXAMPLES

This is an example in FORTRAN:

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

EXTERNAL FPROC
INTEGER SIGNAL
INTEGER CPADDR

I = SIGNAL(SIGTERM, FPROC, -1)
J = SIGNAL(SIGINT, 0, CPADDR)
```

The first call to *signal* sets up the FORTRAN function *fproc* as the signal-handling routine for **SIGTERM**. The second call sets up a C function whose address is in the variable *cpaddr* as the signal-handling routine for **SIGINT**.

NOTES

SIGKILL will immediately terminate a process, regardless of its state. Processes which are stopped via job control (<ctrl>z) will not act upon any delivered signals other than **SIGKILL** until the job is restarted. Processes which are blocked via a *blockproc* system call will unblock if they receive a signal which is fatal (i.e. a non-job-control signal which the are NOT catching), but will still be stopped if the job of which they are a part is stopped. Only upon restart will they die. Any non-fatal signals received by a blocked process will NOT cause the process to be unblocked (an *unblockproc* or *unblockprocall* system call is necessary).

After a *fork*(2) the child inherits all handlers and signal masks, but not the set of pending signals.

The *exec*(2) routines reset all caught signals to the default action; ignored signals remain ignored, the blocked signal mask is unchanged and pending signals remain pending.

sigset will fail if one or more of the following are true:

[EINVAL] *Sig* is an illegal signal number (including **SIGKILL** and **SIGSTOP**) or the default handling of *sig* cannot be changed.

[EINVAL] The requested action is illegal (e.g. ignoring **SIGCONT**, which is ignored by default).

[EINTR] A signal was caught during the system call *sigpause*.

DIAGNOSTICS

Upon successful completion, *sigset* returns the previous value of the system signal action for the specified signal *sig*. Otherwise, a value of **SIG_ERR** is returned and *errno* is set to indicate the error. **SIG_ERR** is defined in *<sys/signal.h>*.

For the other functions, upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

csh(1), kill(2), pause(2), setrlimit(2), signal(2), ulimit(2), wait(2), sigaction(2), setjmp(3C), sigvec(3B), blockproc(2).

WARNINGS

Signals raised by the instruction stream, **SIGILL, SIGEMT, SIGBUS, SIGSEGV** will cause infinite loops if their handler returns, or the action is set to **SIG_IGN.**

WARNING

The POSIX signal routines (*sigaction*(2), *sigpending*(2), *sigprocmask*(2), *sigsuspend*(2), *sigsetjmp*(3)), and the 4.3BSD signal routines (*sigvec*(3B), *signal*(3B), *sigblock*(3B), *sigpause*(3B), *sigsetmask*(3B)) must NEVER be used with *signal*(2) or *sigset*(2).

NAME

socket – create an endpoint for communication

FORTRAN SYNOPSIS

integer *4 function socket (domain, type, protocol)
integer *4 domain, type, 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. The protocol family generally is the same as the address family for the addresses supplied in later operations on the socket. These families are defined in the include file *<sys/socket.h>*. The currently understood formats are:

PF_INET	(DARPA Internet protocols)
PF_RAW	(Link-level protocols)
PF_UNIX	(4.3BSD UNIX internal protocols)

The following are defined but currently unimplemented:

PF_NS	(Xerox Network Systems protocols), and
PF_IMPLINK	(IMP "host at IMP" link layer).

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

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). SOCK_RAW sockets, which are available only to the super-user, provide access to internal network protocols and interfaces. The types SOCK_SEQPACKET and SOCK_RDM are currently unimplemented.

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; see *getprotoent*(3N).

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. Note that for the *read* and *recv*-style calls, the number of bytes actually read may be less than the number requested. 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_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.

An *fcntl*(2) call can be used to specify a process group to receive a SIGURG signal when the out-of-band data arrives. The FIONBIO i/o control (see *ioctl*(2)) or the FNDELAY fcntl (see *fcntl*(2)) enable non-blocking I/O and asynchronous notification of I/O events via SIGIO.

The operation of sockets is controlled by socket level *options*. These options are defined in the file *<sys/socket.h>*. *setsockopt*(2) and *getsockopt*(2) are used to set and get options, respectively.

RETURN VALUE

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.
[EACCESS]	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.

SEE ALSO

accept(2), bind(2), connect(2), fcntl(2), getsockname(2), getsockopt(2), ioctl(2), listen(2), read(2), recv(2), select(2), send(2), socketpair(2), write(2)

Network Programming chapter in the *Network Communications Guide*.

NAME

sproc – create a new share group process

FORTRAN SYNOPSIS

integer*4 function sproc (entry, inh, arg)
external entry
integer*4 inh
integer*4 arg

DESCRIPTION

The *sproc* system call is a variant of the standard *fork*(2) call. Like *fork*, *sproc* creates a new process that is a clone of the process that called *sproc*. The difference is that after an *sproc*, the new child process shares the virtual address space of the parent process (assuming that this sharing option is selected, as described below), rather than simply being a copy of the parent. The parent and the child each have their own program counter value and stack pointer, but all the text and data space is visible to both processes. This provides one of the basic mechanisms upon which parallel programs can be built.

A group of processes created by *sproc* calls from a common ancestor is referred to as a *share group* or *shared process group*. A share group is initially formed when a process first executes an *sproc* call. All subsequent *sproc* calls by either the parent or other children in his share group will add another process to the share group. In addition to virtual address space, members of a share group can share other attributes such as file tables, current working directories, effective userids and others described below.

The new child process resulting from *sproc*(2) differs from a normally forked process in the following ways:

The child's stack is set to a virtual address that doesn't overlap the stack of the parent process. There is a maximum stack size different from the maximum allowable amount of virtual space per process. This value may be read and set using *prctl*(2) or *setrlimit*(2).

If the **PR_SADDR** bit is set in *inh* then the new process will share ALL the virtual space of the parent, except the PRDA (see below). During a normal *fork*(2), the writable portions of the process's address space are marked copy-on-write. If either process writes into a given page, then a copy is made of the page and given to the process. Thus writes by one process will not be visible to the other forks. With the **PR_SADDR** option of *sproc(2)*, however, all the processes have read/write privileges to the entire virtual space.

The new process can reference the parent's stack.

The new process has its own *process data area* (PRDA) which contains, among other things, the *process id*. Part of the PRDA is used by the system, part by system libraries, and part is available to the application program [see **<sys/prctl.h>**]. The PRDA is at a fixed virtual address in each process which is given by the constant **PRDA** defined in **prctl.h**.

The machine state (general/floating point registers) is not duplicated with the exception of the floating point control register. This means that if a process has enabled floating point traps, these will be enabled in the child process.

The new process will be invoked as follows:

entry(arg)

In addition to the attributes inherited during the *sproc* call itself, the *inh* flag to *sproc* can request that the new process have future changes in any member of the share group be applied to itself. A process can only request that a child process share attributes that it itself is sharing. The creator of a share group is effectively sharing everything. These persisting attributes are selectable via the *inh* flag:

PR_SADDR All virtual space attributes (shared memory, mapped files, data space) are shared. If one process in a share group attaches to a shared memory segment, all processes in the group can access that segment.

PR_SFDS The open file table is kept synchronized. If one member of the share group opens a file, the open file descriptor will appear in the file tables of all members of the share group. Note that there is only one file pointer for each file descriptor shared within a shared process group.

PR_SDIR The current and root directories are kept synchronized. If one member of the group issues a *chdir*(2) or *chroot*(2) call, the current working directory or root directory will be changed for all members of the share group.

PR_SUMASK The file creation mask, *umask* is kept synchronized.

PR_SULIMIT The limit on maximum file size is kept synchronized.

PR_SID The real and effective user and group ids are kept synchronized.

To take advantage of sharing all possible attributes, the constant **PR_SALL** may be used.

In addition to specifying shared attributes, the *inh* flag can be used to pass flags that govern certain operations within the *sproc* call itself. Currently one flag is supported, **PR_BLOCK**, which causes the calling process to be blocked [see *blockproc*(2)] before returning from a successful call. This can be used to allow the child process access to the parent's stack without the possibility of collision.

No scheduling synchronization is implied between shared processes: they are free to run on any processor in any sequence. Any required synchronization must be provided by the application using locks and semaphores [see *usinit*(3P)] or other mechanisms.

If one member of a share group exits or otherwise dies, its stack is removed from the virtual space of the share group. In addition, if the **PR_SETEXITSIG** option [see *prctl*(2)] has been enabled then all remaining members of the share group will be signaled.

There are two versions of *sproc*, one in **libc.a** and one in **libmpc.a**. Users linking with the semaphored version of libc, **libmpc.a**, by using the **-lmpc** flag to the compiler, will have standard routines such as *printf* and *malloc* function properly even though two or more shared processes access them simultaneously. To accomplish this, a special arena is set up [see *usinit*(3P)] to hold the locks and semaphores required. Each process in the share group needs access to this arena and requires a single file lock [see *fcntl*(2)]. This may require more file locks to be configured into the system than the default system configuration provides.

sproc will fail and no new process will be created if one or more of the following are true:

[ENOMEM] If there is not enough virtual space to allocate a new stack. The default stack size is settable via *prctl*(2), or *setrlimit*(2).

[EAGAIN] The system-imposed limit on the total number of processes under execution, *{NPROC}* [see *intro*(2)], would be exceeded.

[EAGAIN] The system-imposed limit on the total number of processes under execution by a single user *{CHILD_MAX}* [see *intro*(2)], would be exceeded.

[EAGAIN] Amount of system memory required is temporarily unavailable.

When linked with **libmpc.a**, in addition to the above errors *sproc* will fail and no new process will be created if one or more of the following are true:

[ENOSPC] If the size of the share group exceeds the number of users specified via *usconfig*(3P) (8 by default). Any changes via *usconfig*(3P) must be done BEFORE the first *sproc* is performed.

[ENOLCK] There are not enough file locks in the system.

New share group member pid # could not join I/O arena. error:<..>
if the new share group member could not properly join the semaphored libc arena. The new process exits with a −1.

See also the possible errors from *usinit(3P)*.

NOTES

This manual entry has described ways in which processes created by *sproc* differ from those created by *fork*. Attributes and behavior not mentioned as different should be assumed to work the same way for *sproc* processes as for processes created by *fork*. Here are some respects in which the two types of processes are the same:

The parent and child after an *sproc* each have a unique process id (*pid*), but are in the same process group.

A signal sent to a specific *pid* in a share group [see *kill*(2)] will be received by only the process to which it was sent. Other members of the share group will not be affected. A signal sent to an entire process group will be received by all the members of the process group, regardless of share group affiliations [see *killpg*(3B)]. See *prctl*(2) for ways to alter this behavior.

If the child process resulting from an *sproc* dies or calls *exit*(2), the parent process receives the SIGCLD signal [see *sigset*(2), *sigaction*(2), and *sigvec*(3B)].

CAVEATS

Removing virtual space (e.g. unmapping a file) is an expensive operation and effectively forces all processes in the share group to single thread.

Note that the global variable *errno* is shared by all processes in an *sproc* share group in which address space is a shared attribute. This means that if multiple processes in the group make system calls, the value of *errno* is no longer useful, since it may be overwritten at any time by a system call in another process in the share group. In order to allow a process in a share group to determine the value of *errno* reliably, the system call modules in **libmpc.a** store the error return code in a location in the PRDA that is private to each process in the share group, in addition to storing it in the

global variable *errno*. A library routine *oserror*(3C) is provided in both **libc.a** and **libmpc.a** that returns the current value *errno* for the process making the call.

SEE ALSO

blockproc(2), fcntl(2), fork(2), prctl(2), setrlimit(2), oserror(3C), pcreate(3C), usconfig(3P), usinit(3P).

DIAGNOSTICS

Upon successful completion, *sproc* returns the process id of the new process. Otherwise, a value of −1 is returned to the calling process, and *errno* is set to indicate the error.

NAME

stat, lstat, fstat – get file status

FORTRAN SYNOPSIS

integer function stat (path, statb)
character *(*) path
integer statb (12)

integer function lstat (path, statb)
character *(*) path
integer statb (12)

integer function fstat (lunit, statb)
integer lunit, statb (12)

DESCRIPTION

Path points to a path name naming a file. 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. *stat* obtains information about the named file. The order and meaning of the information returned in array *statb* is identical to that returned in *stat*.

lstat is like *stat* except in the case where the named file is a symbolic link, in which case *lstat* returns the information about the link, while *stat* returns the information about the file the link references.

Similarly, *fstat* obtains information about an open file known by the file descriptor *fildes*, obtained from a successful *open*, *creat*, *dup*, *fcntl*, or *pipe* system call.

Buf is a pointer to a *stat* structure into which information is placed concerning the file.

The contents of the structure pointed to by *buf* include the following members:

```
struct stat {
        dev_t   st_dev;     /* ID of device containing */
                            /* a directory entry for this file */
        ino_t   st_ino;     /* Inode number */
        mode_t  st_mode;    /* File mode; see mknod(2) */
        short   st_nlink;   /* Number of links */
        ushort  st_uid;     /* User ID of the file's owner */
        ushort  st_gid;     /* Group ID of the file's group */
        dev_t   st_rdev;    /* ID of device */
                            /* This entry is defined only for */
                            /* character special or block special files */
        off_t   st_size;    /* File size in bytes */
                            /* or, for fstat on block devices, */
```

```
                                /* device size in 512-byte blocks */
        time_t  st_atime;       /* Time of last access */
        time_t  st_mtime;       /* Time of last data modification */
        time_t  st_ctime;       /* Time of last file status change */
                                /* Times measured in seconds since */
                                /* 00:00:00 GMT, Jan. 1, 1970 */
};
```

st_atime
Time when file data was last accessed. Changed by the following system calls: *creat*(2), *mknod*(2), *pipe*(2), *utime*(2), and *read*(2).

st_mtime
Time when data was last modified. Changed by the following system calls: *creat*(2), *mknod*(2), *pipe*(2), *utime*(2), and *write*(2).

st_ctime
Time when file status was last changed. Changed by the following system calls: *chmod*(2), *chown*(2), *creat*(2), *link*(2), *mknod*(2), *pipe*(2), *unlink*(2), *utime*(2), and *write*(2).

Note: the st_size field is set for block devices only by *fstat* and not by *stat*. It is set only for block device files which are associated with a real disk device.

stat and *lstat* will fail if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[ENAMETOOLONG]	The length of *path* exceeds *{PATH_MAX}*, or a pathname component is longer than *{NAME_MAX}*.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EFAULT]	*Buf* or *path* points to an invalid address.

fstat will fail if one or more of the following are true:

[EBADF]	*Fildes* is not a valid open file descriptor.
[EFAULT]	*Buf* points to an invalid address.

SEE ALSO

chmod(2), chown(2), creat(2), link(2), mknod(2), time(2), truncate(2), unlink(2), utime(2), utimes(3B).

DIAGNOSTICS

Upon successful completion a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

stime – set time

FORTRAN SYNOPSIS

integer *4 function stime (tp)
integer *4 tp

DESCRIPTION

stime sets the system's idea of the time and date. *Tp* points to the value of time as measured in seconds from 00:00:00 GMT January 1, 1970.

[EPERM] *stime* will fail if the effective user ID of the calling process is not super-user.

SEE ALSO

time(2), gettimeofday(3B), ctime(3C).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

symlink – make symbolic link to a file

FORTRAN SYNOPSIS

integer *4 function symlink (name1, name2)
character * (*) name1, 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 arbitrary path name; the files need not be on the same file system.

RETURN VALUE

Upon successful completion, a zero value is returned. If an error occurs, the error code is stored in *errno* and a –1 value is returned.

ERRORS

The symbolic link is made unless on or more of the following are true:

[ENOTDIR]	A component of the *name2* prefix is not a directory.
[ENOENT]	A component of the *name2* prefix does not exist.
[EEXIST]	*Name2* already exists.
[EACCES]	A component of the *name2* path prefix denies search permission.
[EROFS]	The file *name2* would reside on a read-only file system.
[EFAULT]	*Name1* or *name2* points outside the process's allocated address space.
[ELOOP]	Too may symbolic links were encountered in translating the pathname.

SEE ALSO

link(2), ln(1), readlink(2), unlink(2)

NAME

sync – update super block

FORTRAN SYNOPSIS

subroutine sync ()

DESCRIPTION

sync causes all information in memory that should be on disk to be written out. This includes modified super blocks, modified i-nodes, and delayed block I/O.

It should be used by programs which examine a file system, for example *fsck*, *df*, etc. It is mandatory before a re-boot.

The writing, although scheduled, is not necessarily complete upon return from *sync*.

NAME

sysmp – multiprocessing control

FORTRAN SYNOPSIS

integer *4 function sysmp (cmd, arg1, arg2, arg3, arg4)
integer *4 cmd, arg1, arg2, arg3, arg4

DESCRIPTION

sysmp provides control/information for miscellaneous system services. This system call is usually used by system programs and is not intended for general use. The arguments *arg1*, *arg2*, *arg3*, *arg4* are provided for command-dependent use.

As specified by *cmd*, the following commands are available:

MP_PGSIZE The page size of the system is returned [see *getpagesize*(2)].

MP_SCHED Interface for the *schedctl*(2) system call.

MP_NPROCS Returns the number of processors physically configured.

MP_NAPROCS Returns the number of processors that are available to schedule unrestricted processes.

MP_STAT The processor ids and status flag bits of the physically configured processors are copied into an array of *pda_stat* structures to which *arg1* points. The array must be large enough to hold as many *pda_stat* structures as the number of processors returned by the **MP_NPROCS** *sysmp* command. The *pda_stat* structure and the various status bits are defined in <*sys/pda.h*>.

MP_EMPOWER Empowers processor numbered *arg1* to run any unrestricted processes. This is the default system configuration for all processors. This command requires superuser authority.

MP_RESTRICT Restricts processor numbered *arg1* from running any processes except those assigned to it by a **MP_MUSTRUN** command, a *runon(1)* command or because of hardware necessity. This command requires superuser authority.

MP_CLOCK Moves the operating system software clock handling to the processor numbered *arg1*. This command requires superuser authority.

MP_MUSTRUN Assigns the calling process to run only on the processor numbered *arg1*, except as required for communications with hardware devices.

MP_RUNANYWHERE
Frees the calling process to run on whatever processor the system deems suitable.

MP_KERNADDR Returns the address of various kernel data structures. The structure returned is selected by *arg1*. The list of available structures is detailed in *<sys/sysmp.h>*. This option is used by many system programs to avoid having to look in **/unix** for the location of the data structures.

MP_SASZ Returns the size of various system accounting structures. As above, the structure returned is governed by *arg1*.

MP_SAGET1 Returns the contents of various system accounting structures. The information is only for the processor specified by *arg4*. As above, the structure returned is governed by *arg1*. *arg2* points to a buffer in the address space of the calling process and *arg3* specifies the maximum number to bytes to transfer.

MP_SAGET Returns the contents of various system accounting structures. The information is summed across all processors before it is returned. As above, the structure returned is governed by *arg1*. *arg2* points to a buffer in the address space of the calling process and *arg3* specifies the maximum number to bytes to transfer.

Possible errors from *sysmp* are:

[EPERM] The effective user ID is not superuser. Many of the commands require superuser privilege.

[EINVAL] The processor named by a MP_EMPOWER, MP_RESTRICT, MP_CLOCK or MP_SAGET1 command does not exist.

[EINVAL] The *cmd* argument is invalid.

[EINVAL] The *arg1* argument to a MP_KERNADDR command is invalid.

[EBUSY] An attempt was made to restrict the only unrestricted processor or to restrict the master processor.

[EFAULT] An invalid buffer address has been supplied by the calling process.

SEE ALSO

mpdmin(1), runon(1), getpagesize(2), schedctl(2).

DIAGNOSTICS

Upon successful completion, the *cmd* dependent data is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

uadmin – administrative control

FORTRAN SYNOPSIS

integer *4 function uadmin (cmd, fcn, mdep)
integer *4 cmd, fcn, mdep

DESCRIPTION

uadmin provides control for basic administrative functions. This system call is tightly coupled to the system administrative procedures and is not intended for general use. The argument *mdep* is provided for machine-dependent use and is not defined here.

As specified by *cmd*, the following commands are available:

A_SHUTDOWN The system is shutdown. All user processes are killed, the buffer cache is flushed, and the root file system is unmounted. The action to be taken after the system has been shut down is specified by *fcn*. The functions are generic; the hardware capabilities vary on specific machines.

- AD_HALT Halt the processor and turn off the power.
- AD_BOOT Reboot the system, using /unix.
- AD_IBOOT Interactive reboot; user is prompted for system name. Not supported; it is treated the same as AD_HALT.

A_REBOOT The system stops immediately without any further processing. The action to be taken next is specified by *fcn* as above.

A_REMOUNT The root file system is mounted again after having been fixed. This should be used only during the startup process.

A_KILLALL All processes are killed except those belonging to the process group specified by *fcn*. They are sent the signal specified by *mdep*.

uadmin fails if any of the following are true:

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

DIAGNOSTICS

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

A_SHUTDOWN	Never returns.
A_REBOOT	Never returns.
A_REMOUNT	0
A_KILLALL	0

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

NAME

ulimit – get and set user limits

FORTRAN SYNOPSIS

integer *4 function ulimit (cmd, newlimit)
integer *4 cmd, newlimit

DESCRIPTION

This function provides for control over process limits. The *cmd* values available are:

1 Get the regular file size limit of the process. The limit is in units of 512-byte blocks and is inherited by child processes. Files of any size can be read.

2 Set the regular file size limit of the process to the value of *newlimit*. Any process may decrease this limit, but only a process with an effective user ID of super-user may increase the limit. *ulimit* fails and the limit is unchanged if a process with an effective user ID other than super-user attempts to increase its regular file size limit. [EPERM]

3 Get the maximum possible break value [see *brk*(2)].

4 Get the current value of the maximum number of open files per process configured in the system.

SEE ALSO

brk(2), setrlimit(2), write(2).

WARNING

ulimit is effective in limiting the growth of regular files. Pipes are currently limited to 10240 bytes.

DIAGNOSTICS

Upon successful completion, a non-negative value is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

umask – set and get file creation mask

FORTRAN SYNOPSIS

integer *4 function umask (cmask)
integer *4 cmask

DESCRIPTION

umask sets the process's file mode creation mask to *cmask* and returns the previous value of the mask. Only the low-order 9 bits of *cmask* and the file mode creation mask are used.

SEE ALSO

chmod(2), creat(2), mknod(2), open(2).
mkdir(1), sh(1) in the *User's Reference Manual.*

DIAGNOSTICS

The previous value of the file mode creation mask is returned.

NAME

umount – unmount a file system

FORTRAN SYNOPSIS

integer *4 function umount (file)
character * (*) file

DESCRIPTION

umount requests that a previously mounted file system contained on the block special device or directory identified by *file* be unmounted. *File* is a pointer to a path name. After unmounting the file system, the directory upon which the file system was mounted reverts to its ordinary interpretation.

umount may be invoked only by the super-user.

umount will fail if one or more of the following are true:

[EPERM]	The process's effective user ID is not super-user.
[EINVAL]	*File* does not exist.
[ENOTBLK]	*File* is not a block special device.
[EINVAL]	*File* is not mounted.
[EBUSY]	A file on *file* is busy.
[EFAULT]	*File* points to an illegal address.

SEE ALSO

mount(2).

DIAGNOSTICS

Upon successful completion a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

unlink – remove directory entry

FORTRAN SYNOPSIS

integer function unlink (path)
character *(*) path

DESCRIPTION

unlink removes the directory entry named by the path name pointed to by *path*.

The named file is unlinked unless one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[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.
[EACCES]	The parent directory has the sticky bit set and the file is not writable by the user and the user does not own the parent directory and the user does not own the file and the user is not superuser.
[EPERM]	The named file is a directory and the effective user ID of the process is not super-user.
[ENAMETOOLONG]	The length of *path* exceeds *{PATH_MAX}*, or a pathname component is longer than *{NAME_MAX}*.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EBUSY]	The entry to be unlinked is the mount point for a mounted file system.
[EROFS]	The directory entry to be unlinked is part of a read-only file system.
[EFAULT]	*Path* points outside the process's allocated address space.

When all links to a file have been removed and no process has the file open, the space occupied by the file is freed and the file ceases to exist. If one or more processes have the file open when the last link is removed, the removal is postponed until all references to the file have been closed.

SEE ALSO

close(2), link(2), open(2), rename(2), rmdir(2).
rm(1) in the *User's Reference Manual.*

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

wait, waitpid, wait3 – wait for child processes to stop or terminate

FORTRAN SYNOPSIS

integer function wait (status) integer status

DESCRIPTION

Wait functions: The *wait* functions suspend the calling process until one of the immediate children terminate, or until a child that is being traced stops because it has hit a break point. These system calls will return prematurely if a signal is received, and if a child process stopped or terminated prior to the call then return is immediate. If the call is successful, the process ID of a child is returned. The two versions differ in the *type* of their input parameter *(statptr)*, but the information conveyed is identical if the macros in **<sys/wait.h>** are used (see below description in the PARAMETERS section).

Wait3: *Wait3* is BSD's extension of *wait*. It provides an alternate interface for programs that must not block when collecting the status of child processes.

Waitpid: The *waitpid* function is POSIX's extension of *wait*. The *pid* argument specifies a set of child processes for which status is requested. The *waitpid* function only returns the status of a child process from this set.

PARAMETERS

Statptr (all functions): If *Statptr* is non-zero, 16 bits of information called *status* are stored in the low-order 16 bits of the location pointed to by *statptr*. *Status* can be used to differentiate between stopped and terminated child processes. If the child process terminated, *status* identifies the cause of termination and passes useful information to the parent. A more precise definition of the *status* structure is given in *<sys/wait.h>*. *Status* is interpreted as follows:

If the child process stopped, the predicate **WIFSTOPPED(**statptr*)** will evaluate to non-zero and **WSTOPSIG(**statptr*)** will return the signal number that caused the process to stop. (The high-order 8 bits of status will contain the signal number and the low-order 8 bits are set equal to 0177.)

If the child process terminated due to an *exit* call, the predicate **WIFEXITED(**statptr*)** will evaluate to non-zero, and **WEXITSTATUS(**statptr*)** will return the argument that the child process passed to *_exit* or *exit*, or the value the child process returned from *main* [see *exit*(2)]. (The low-order 8 bits of status will be zero and the high-order 8 bits will contain the low-order 8 bits of the exiting argument.)

If the child process terminated due to a signal, the predicate **WIFSIGNALED(*statptr*)** will evaluate to non-zero, and **WTERMSIG(*statptr*)** will return the signal number that caused the termination. (The high-order 8 bits of status will be zero and the low-order 8 bits will contain the number of the signal.) In addition, if the low-order seventh bit (i.e., bit 0200) is set, a "core image" will have been produced [see *signal*(2)].

Rusage (wait3): If *wait3's rusage* parameter is non-zero, a summary of the resources used by the terminated process and all its children is returned (this information is currently not available for stopped processes).

Pid (waitpid):

1) If *pid* is equal to −1, status is requested for any child process. In this respect, *waitpid* is then equivalent to *wait*.

2) If *pid* is greater than zero, it specifies the process ID of a single child process for which status is requested.

3) If *pid* is equal to zero, status is requested for any child process whose process group ID is equal to that of the calling process.

4) If *pid* is less than −1, status is requested for any child process whose process group ID is equal to the absolute value of *pid*.

Options (waitpid and wait3): The *options argument* is constructed from the bitwise inclusive OR of zero or more of the following flags, defined in the header <*sys/wait.h*>:

WNOHANG — The function will not suspend execution of the calling process if status is not immediately available for one of the child processes.

WUNTRACED — The status of child processes that are stopped due to a **SIGTTIN**, **SIGTTOU**, **SIGTSTP**, or **SIGSTOP** signal, and whose status has not yet been reported since they stopped, are reported to the requesting process.

If a parent process terminates without waiting for its child processes to terminate, the parent process ID of each child process is set to 1. This means the initialization process inherits the child processes [see *intro*(2)].

SIGCLD HANDLING

IRIX has three distinct version of signal routines: System V (*signal*(2) and *sigset*(2)), 4.3BSD (*signal*(3B) and *sigvec*(3B)), and POSIX (*sigaction*(2)). Each version has a method by which a parent can be certain that it waits on all of its children even if they are executing concurrently. In each version, the parent installs a signal handler for **SIGCLD** to wait for its children, but the specific code differs in subtle, albeit vital, ways. Sample programs

below are used to illustrate each of the three methods.

Note that System V refers to this signal as **SIGCLD**, whereas BSD calls it **SIGCHLD**. For compatibility with both systems they are defined to be the same signal number, and may therefore be used interchangeably.

System V: System V's **SIGCLD** mechanism guarantees that no **SIGCLD** signals will be lost. It accomplishes this by forcing the process to reinstall the handler (via *signal* or *sigset* calls) when leaving the handler. Note that whereas *signal*(2) sets the signal disposition back to **SIG_DFL** each time the handler is called, *sigset*(2) keeps it installed, so **SIGCLD** is the only signal that demands this reinstallation, and that only because the installation call allows the kernel to check for additional instances of the signal that occurred while the process was executing in the handler. The code below is the System V example. Note that the *sigpause*(2) creates a window during which **SIGCLD** is not blocked, allowing the parent to enter its handler.

```
/*
 * System V example of wait-in-SIGCLD-handler usage
 */
#include <signal.h>
#include <stdio.h>
#include <sys/wait.h>

static void handler(int);

#define NUMKIDS   4
volatile int kids = NUMKIDS;

main()
{
    int i, pid;

    sigset(SIGCLD, handler);
    sighold(SIGCLD);
    for (i = 0; i < NUMKIDS; i++) {
        if (fork() == 0) {
            printf("Child %d\n", getpid());
            exit(0);
        }
    }
    while (kids > 0) {
        sigpause(SIGCLD);
        sighold(SIGCLD);
    }
```

```
}

static void
handler(int sig)
{
    int pid, status;

    printf("Parent (%d) in handler, ", getpid());
    pid = wait(&status);
    kids--;
    printf("child %d, now %d left\n", pid, kids);
    /*
     * Now reinstall handler & cause SIGCLD to be re-raised
     * if any more children exited while we were in here.
     */
    sigset(SIGCLD, handler);
}
```

BSD: 4.3BSD solved this problem differently: instead of guaranteeing that no **SIGCHLD** signals are lost, it provides a **WNOHANG** option to *wait3* that allows parent processes to do non-blocking waits in loops, until no more stopped or zombied children exist. Note that the handler must be able to deal with the case in which no applicable children exist; if one or more children exit while the parent is in the handler, all may get reaped, yet if one or more **SIGCHLD** signals arrived while the parent was in its handler, the signal will remain pending, the parent will reenter the handler, and the *wait3* call will return 0. Note that it is not necessary to call *sigvec* upon exit from the handler.

```
/*
 * BSD example of wait3-in-SIGCHLD handler usage
 */

#define _BSD_SIGNALS
#include <signal.h>
#include <stdio.h>
#include <sys/wait.h>

static int handler(int);

#define NUMKIDS   4
volatile int kids = NUMKIDS;

main()
{
```

```
    int i, pid;
    struct sigvec vec;

    vec.sv_handler = handler;
    vec.sv_mask = sigmask(SIGCHLD);
    vec.sv_flags = 0;

    sigvec(SIGCHLD, &vec, NULL);
    sigsetmask(sigmask(SIGCHLD));
    for (i = 0; i < NUMKIDS; i++) {
        if (fork() == 0) {
            printf("Child %d\n", getpid());
            exit(0);
        }
    }
    while (kids > 0) {
        sigpause(0);
    }
}

static int
handler(int sig)
{
    int pid;
    union wait status;

    printf("Parent (%d) in handler, ", getpid());
    while ((pid = wait3(&status, WNOHANG, NULL)) > 0) {
        kids--;
        printf("child %d, now %d left\n", pid, kids);
    }
}
```

POSIX: POSIX improved on the BSD method by providing *waitpid*, that allows a parent to wait on a particular child process if desired. In addition, the IRIX implementation of *sigaction*(2) checks for zombied children upon exit from the system call if the specified signal was **SIGCLD** and the disposition of the signal handling was changed. If zombied children exist, another **SIGCLD** is raised. This solves the problem that occurs when a parent creates children, but a module that it links with (typically a libc routine such as *system*(3)) creates and waits on its own children.

Two problems have classically arisen in such a scheme: 1) until the advent of *waitpid*, the called routine could not specify which children to wait on; it therefore looped, waiting and discarding children until the one (or ones) it had created terminated, and 2) if the called routine changed the disposition of **SIGCLD** and then restored the previous handler upon exit, children of the parent (calling) process that had terminated while the called routine executed would be missed in the parent, because the called routine's **SIGCLD** handler would reap and discard those children. The addition of *waitpid* and the IRIX implementation of *sigaction* solves both of these problems. Note that neither the BSD nor the System V signal routines on IRIX have these properties, in the interests of compatibility.

WARNING: programs that install **SIGCLD** handlers that set flags instead of executing *waitpid*s and then attempt to restore the previous signal handler (via *sigaction*) upon return from the handler will create infinite loops.

```
/*
* POSIX example of waitpid-in-SIGCHLD handler usage
*/

#include <signal.h>
#include <stdio.h>
#include <sys/wait.h>

static void handler(int);

#define NUMKIDS   4
volatile int kids = NUMKIDS;

/*
* If waitpid's 1st argument is -1, it waits for any child.
*/
#define ANYKID -1

main()
{
    int i;
    pid_t pid;
    struct sigaction act;
    sigset_t set, emptyset;

    act.sa_handler = handler;
    act.sa_mask = sigmask(SIGCHLD);
    act.sa_flags = 0;
```

```
    sigaction(SIGCHLD, &act, NULL);
    sigemptyset(&set);
    sigemptyset(&emptyset);
    sigaddset(&set, SIGCHLD);
    sigprocmask(SIG_BLOCK, &set, NULL);
    setbuf(stdout, NULL);

    for (i = 0; i < NUMKIDS; i++) {
        if (fork() == 0) {
            printf("Child %d\n", getpid());
            exit(0);
        }
    }
    while (kids > 0) {
        sigsuspend(&emptyset);
    }
}

static void
handler(int sig)
{
    pid_t pid;
    int status;

    printf("Parent (%d) in handler, ", getpid());
    pid = waitpid(ANYKID, &status, WNOHANG);
    while (pid > 0) {
        kids--;
        printf("child %d, now %d left\n", pid, kids);
        pid = waitpid(ANYKID, &status, WNOHANG);
    }
}
```

DIAGNOSTICS

Wait fails and its actions are undefined if *statptr* points to an invalid address. If *wait*, *wait3*, or *waitpid* return due to a stopped or terminated child process, the process ID of the child is returned to the calling process. *Wait3* and *waitpid* return 0 if **WNOHANG** is specified and there are currently no stopped or exited children (although children DO exist). Otherwise, a value of −1 is returned and *errno* is set to indicate the error:

[EINTR] **wait, wait3, waitpid:** The calling process received a signal.

[ECHILD] **wait, wait3, waitpid:** The calling process has no existing unwaited-for child processes. **waitpid:** The process or process group specified by *pid* does not exist or is not a child of the calling process.

[EFAULT] **wait3, waitpid:** The *rusage* or *statptr* arguments (where applicable) point to illegal addresses.

[EINVAL] **waitpid:** The value of the *options* argument is not valid.

SEE ALSO

exec(2), exit(2), fork(2), intro(2), pause(2), ptrace(2), signal(2), sigset(2), sigpause(2), sigaction(2), sigsuspend(2), sigprocmask(2), signal(3B), sigvec(3B), sigpause(3B).

NOTE

Currently, *wait3* returns only the user and system time in *rusage*.

NAME

write – write on a file

FORTRAN SYNOPSIS

integer *4 function write (fildes, buf, nbyte)
integer *4 fildes
character * (*) buf
integer *4 nbyte

DESCRIPTION

fildes is a file descriptor obtained from a *creat*(2), *open*(2), *dup*(2), *fcntl*(2), *pipe*(2), *socket*(2), or *socketpair*(2) system call.

write attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the *fildes*.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from *write*, the file pointer is incremented by the number of bytes actually written.

On devices incapable of seeking, writing always takes place starting at the current position. The value of a file pointer associated with such a device is undefined.

If the O_APPEND flag of the file status flags is set, the file pointer will be set to the end of the file prior to each write.

For regular files, if the O_SYNC flag of the file status flags is set, *write* will not return until both the file data and file status have been physically updated. This function is for special applications that require extra reliability at the cost of performance. For block special files, if O_SYNC is set, the write will not return until the data has been physically updated.

A write to a regular file will be blocked if mandatory file/record locking is set [see *chmod*(2)], and there is a record lock owned by another process on the segment of the file to be written. If neither O_NDELAY or O_NONBLOCK are set, the write will sleep until the blocking record lock is removed, otherwise (either flag set) *write* returns −1 and *errno* is set to EAGAIN.

For STREAMS [see *intro*(2)] files, the operation of *write* is determined by the values of the minimum and maximum *nbyte* range ("packet size") accepted by the *stream*. These values are contained in the topmost *stream* module. Unless the user pushes [see I_PUSH in *streamio*(7)] the topmost module, these values can not be set or tested from user level. If *nbyte* falls within the packet size range, *nbyte* bytes will be written. If *nbyte* does not fall within the range and the minimum packet size value is zero, *write* will break the buffer into maximum packet size segments prior to sending the data downstream (the last segment may contain less than the maximum

packet size). If *nbyte* does not fall within the range and the minimum value is non-zero, *write* will fail with *errno* set to ERANGE. Writing a zero-length buffer (*nbyte* is zero) sends zero bytes with zero returned.

For STREAMS files, if O_NDELAY and O_NONBLOCK are not set and the *stream* can not accept data (the *stream* write queue is full due to internal flow control conditions), *write* will block until data can be accepted. O_NDELAY or O_NONBLOCK will prevent a process from blocking due to flow control conditions. If O_NDELAY or O_NONBLOCK is set and the *stream* can not accept data, *write* will fail, returning −1 and setting *errno* to EAGAIN. If O_NDELAY or O_NONBLOCK is set and part of the buffer has been written when a condition in which the *stream* can not accept additional data occurs, *write* will terminate and return the number of bytes written.

write will fail and the file pointer will remain unchanged if one or more of the following are true:

[EAGAIN] Mandatory file/record locking was set, O_NDELAY or O_NONBLOCK was set, and there was a blocking record lock.

[EAGAIN] Total amount of system memory available when reading via raw IO is temporarily insufficient.

[EAGAIN] Attempt to write to a *stream* that can not accept data with the O_NDELAY or O_NONBLOCK flag set.

[EBADF] *fildes* is not a valid file descriptor open for writing.

[EDEADLK] The write was going to go to sleep and cause a deadlock situation to occur.

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

[EFBIG] An attempt was made to write a file that exceeds the process's file size limit or the maximum file size [see *ulimit*(2)].

[EINTR] A signal was caught during the *write* system call.

[EINVAL] Attempt to write to a *stream* linked below a multiplexor.

[ENOLCK] The system record lock table was full, so the write could not go to sleep until the blocking record lock was removed.

[ENOSPC] During a *write* to an ordinary file, there is no free space left on the device.

[ENXIO] A hangup occurred on the *stream* being written to.

[EPIPE and SIGPIPE signal]
An attempt is made to write to a pipe that is not open for reading by any process.

[ERANGE] Attempt to write to a *stream* with *nbyte* outside specified minimum and maximum write range, and the minimum value is non-zero.

If a *write* requests that more bytes be written than there is room for (e.g., the *ulimit* [see *ulimit*(2) and *setrlimit*(2)] or the physical end of a medium), only as many bytes as there is room for will be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512-bytes will return 20. The next write of a non-zero number of bytes will give a failure return (except as noted below).

If the file being written is a pipe (or FIFO) and the O_NDELAY flag of the file flag word is set, then write to a full pipe (or FIFO) will return a count of 0. If the file being written is a pipe (or FIFO) and the O_NONBLOCK flag of the file flag word is set, then write to a full pipe (or FIFO) will return −1 and set *errno* to EAGAIN. Otherwise (O_NDELAY and O_NONBLOCK clear), writes to a full pipe (or FIFO) will block until space becomes available.

A write to a STREAMS file can fail if an error message has been received at the stream head. In this case, *errno* is set to the value included in the error message.

CAVEATS

Due to the different semantics of O_NDELAY and O_NONBLOCK in the case of pipes or FIFOs, these flags *must* not be used simultaneously.

SEE ALSO

creat(2), dup(2), fcntl(2), intro(2), lseek(2), open(2), pipe(2), setrlimit(2), ulimit(2).

DIAGNOSTICS

Upon successful completion the number of bytes actually written is returned. Otherwise, −1 is returned and *errno* is set to indicate the error.

NAME

abort – terminate Fortran program

SYNOPSIS

call abort ( )

DESCRIPTION

abort terminates the program which calls it, closing all open files truncated to the current position of the file pointer. The abort usually results in a core dump.

DIAGNOSTICS

When invoked, *abort* prints "Fortran abort routine called" on the standard error output. The shell prints the message "abort - core dumped" if a core dump results.

SEE ALSO

abort(3C).
sh(1) in the *User's Reference Manual.*

ORIGIN

AT&T V.3

NAME

abs, iabs, dabs, cabs, zabs, iiabs, jiabs – FORTRAN absolute value

SYNOPSIS

```
integer i1, i2
real r1, r2
double precision dp1, dp2
complex cx1, cx2
double complex dx1, dx2
integer*2 ii1, ii2
integer*4 ji1, ji2

r2 = abs(r1)

i2 = iabs(i1)
i2 = abs(i1)

dp2 = dabs(dp1)
dp2 = abs(dp1)

cx2 = cabs(cx1)
cx2 = abs(cx1)

dx2 = zabs(dx1)
dx2 = abs(dx1)

ii2 = iiabs(ii1)
ii2 = abs(ii1)

ji2 = jiabs(ji1)
ji2 = abs(ji1)
```

DESCRIPTION

abs is the family of absolute value functions. *iabs* returns the integer absolute value of its integer argument. It accepts either integer*2 or integer*4 arguments and the result is the same type. *dabs* returns the double-precision absolute value of its double-precision argument. *cabs* returns the complex absolute value of its complex argument. *zabs* returns the double-complex absolute value of its double-complex argument. *iiabs* returns the integer*2 absolute value of its integer*2 argument. *jiabs* returns the integer*4 absolute value of its integer*4 argument. The generic form *abs* returns the type of its argument.

SEE ALSO

floor(3M).

CAVEAT

In two's-complement integer (integer*2 or integer*4) representation the absolute value of the negative integer with largest magnitude is undefined. Some implementations trap this error, but others simply ignore it.

NAME

acos, dacos, acosd, dacosd – FORTRAN arccosine intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2
real*4 r3, r4
real*8 dp3, dp4

r2 = acos(r1)

dp2 = dacos(dp1)
dp2 = acos(dp1)

r4 = acosd(r3)

dp4 = dacosd(dp3)
dp4 = acosd(dp3)
```

DESCRIPTION

acos returns the real arccosine of its real argument. *dacos* returns the double-precision arccosine of its double-precision argument. The absolute value of the argument for these routines must be less than or equal to one. The result is in radians and the range is less than or equal to one. The generic form *acos* may be used with impunity as its argument will determine the type of the returned value.

acosd returns the real*4 arccosine of its real*4 argument. *dacosd* returns the real*8 arccosine of its real88 argument. The absolute value of the argument for these routines must be less than or equal to one and the result is in degrees. The generic form *acosd* may be used with impunity for *acosd* *dacosd* as its argument will determine the type of the returned value.

SEE ALSO

trig(3M).

ORIGIN

MIPS Computer Systems

NAME

aint, dint, iint, jint, iidint, jidint – FORTRAN integer part intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2
real*4 r3
real*8 dp3
integer*2 ii
integer*4 ji

r2 = aint(r1)

dp2 = dint(dp1)

ii = iint(r3)

ji = jint(r3)

ii = iidint(dp3)

ji = jidint(dp3)
```

DESCRIPTION

aint returns the truncated value of its real argument in a real. *dint* returns the truncated value of its double-precision argument as a double-precision value. *iint* returns the truncated value of its real*4 argument in a integer*2. *jint* returns the truncated value of its real*4 argument in a integer*4. *iidint* returns the truncated value of its real*8 argument in a integer*2. *jidint* returns the truncated value of its real*8 argument in a integer*4.

ORIGIN

MIPS Computer Systems

NAME

alarm – execute a subroutine after a specified time

SYNOPSIS

integer function alarm (time, proc)
integer time
external proc

DESCRIPTION

This routine arranges for subroutine *proc* to be called after *time* seconds. If *time* is "0", the alarm is turned off and no routine will be called. The returned value will be the time remaining on the last alarm.

FILES

/usr/lib/libU77.a

SEE ALSO

sleep(3F), signal(3F)

BUGS

Alarm and *sleep* interact. If *sleep* is called after *alarm*, the *alarm* process will never be called. SIGALRM will occur at the lesser of the remaining *alarm* time or the *sleep* time.

ORIGIN

MIPS Computer Systems

NAME

asin, dasin, asind, dasind – FORTRAN arcsine intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2
real*4 r3, r4
real*8 dp3, dp4

r2 = asin(r1)

dp2 = dasin(dp1)
dp2 = asin(dp1)

r4 = asind(r3)

dp4 = dasind(dp3)
dp4 = asind(dp3)
```

DESCRIPTION

asin returns the real arcsine of its real argument. *dasin* returns the double-precision arcsine of its double-precision argument. The absolute value of the arguments for *asin* and *dasin* must be less than or equal to one. The result is in radians and is in the range -p/2 < result < p/2. The generic form *asin* may be used with impunity as it derives its type from that of its argument.

asind returns the real*4 arcsine of its real*4 argument. *dasind* returns the real*8 arcsine of its real*8 argument. The absolute value of the arguments for *asind* and *dasind* must be less than or equal to one. The result is in degrees. The generic form *asind* may be used with impunity as it derives its type from that of its argument.

SEE ALSO

trig(3M).

ORIGIN

MIPS Computer Systems

NAME

atan, datan, atand, datand – FORTRAN arctangent intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2
real*4 r3, r4
real*8 dp3, dp4

r2 = atan(r1)

dp2 = datan(dp1)
dp2 = atan(dp1)

r4 = atand(r3)

dp4 = datand(dp3)
dp4 = atand(dp3)

DESCRIPTION

atan returns the real arctangent of its real argument. *datan* returns the double-precision arctangent of its double-precision argument. The generic form *atan* may be used with a double-precision argument returning a double-precision value. The result of *atan* and *datan* is in radians.

atand returns the real*4 arctangent of its real*4 argument. *datand* returns the real*8 arctangent of its real*8 argument. The generic form *atand* may be used with a real*8 argument returning a real*8 value. The result of *atand* and *datand* is in degrees.

SEE ALSO

trig(3M).

ORIGIN

MIPS Computer Systems

NAME

atan2, datan2, atan2d, datan2d – FORTRAN arctangent intrinsic function

SYNOPSIS

real r1, r2, r3
double precision dp1, dp2, dp3
real*4 r4, r5, r6
real*8 dp4, dp5, dp6

r3 = atan2(r1, r2)

dp3 = datan2(dp1, dp2)
dp3 = atan2(dp1, dp2)

r6 = atan2d(r4, r5)

dp6 = datan2d(dp4, dp5)
dp6 = atan2d(dp4, dp5)

DESCRIPTION

atan2 returns the arctangent of *arg1/arg2* as a real value. *datan2* returns the double-precision arctangent of its double-precision arguments. The generic form *atan2* may be used with impunity with double-precision arguments. If the value of the first argument of *atan2* or *datan2* is positive, the result is positive. When the value of the first argument is zero, the result is zero if the second argument is positive and P if the second argument is negative. If the value of the first argument is negative, the result is negative. If the value of the second argument is zero, the absolute value of the result is P/2. Both arguments must not have the value zero. The result of *atan2* and *datan2* is in radians.

atan2d returns the arctangent of *arg1/arg2* as a real*4 value. *datan2d* returns the real*8 arctangent of its real*8 arguments. The generic form *atan2d* may be used with impunity with real*8 arguments. If the value of the first argument of *atan2d* or *datan2d* is positive, the result is positive. When the value of the first argument is zero, the result is zero if the second argument is positive and P if the second argument is negative. If the value of the first argument is negative, the result is negative. If the value of the second argument is zero, the absolute value of the result is P/2. Both arguments must not have the value zero. The result of *atan2d* and *datan2d* is in the range: -180 degrees < result < 180 degrees.

SEE ALSO

trig(3M).

ORIGIN

MIPS Computer Systems

NAME

bool: iand, and, iior, ior, or, jior, inot, jnot, not, iieor, jieor, ieor, xor, iishft, jishft, ishft, lshift, rshift, iishftc, jishftc, ishftc, iibits, jibits, ibits, iibset, jibset, ibset, bitest, bjtest, btest, iibclr, jibclr, ibclr, mvbits – FORTRAN bitwise boolean functions

SYNOPSIS

integer i, k, l, m, n, len
integer*2 ii1, ii2, ii3
logical b
logical*2 c

i = iand(m, n)
i = and(m, n)

ii3 = iior(ii1, ii2)
i = ior(m, n)
i = or(m, n)
i = jior(m, n)

ii3 = inot(ii1)
i = jnot(m)
i = not(m)

ii3 = iieor(ii1, ii2)
i = jieor(m, n)
i = ieor(m, n)
i = xor(m, n)

ii3 = iishft(ii1, ii2)
i = jishft(m, k)
i = ishft(m, k)
i = lshift(m, k)
i = rshift(m, k)

ii3 = iishftc(ii1, ii2, len)
i = jishftc(m, k, len)
i = ishftc(m, k, len)

ii3 = iibits(ii1, ii2, len)
i = jibits(m, k, len)
i = ibits(m, k, len)

ii3 = iibset(ii1, ii2)
i = jibset(n, k)
i = ibset(n, k)

c = bitest(ii1, ii2)
b = bjtest(n, k)
b = btest(n, k)

ii3 = iibclr(ii1, ii2)
i = jibclr(n, k)
i = ibclr(n, k)

call mvbits(m, k, len, n, l)

DESCRIPTION

bool is the general name for the bit field manipulation intrinsic functions and subroutines from the FORTRAN Military Standard (MIL-STD-1753).

and, or and *xor* return the value of the binary operations on their arguments. *not* is a unary operator returning the one's complement of its argument. *ior, iand, not, ieor* – return the same results as *and, or, not,* and *xor.*

lshift and *rshift* return the value of the first argument shifted left or right, respectively, the number of times specified by the second (integer) argument.

ishft, ishftc – **m** specifies the integer to be shifted. **k** specifies the shift count. **k** > 0 indicates a left shift. **k** = 0 indicates no shift. **k** < 0 indicates a right shift. In *ishft*, zeros are shifted in. In *ishftc*, the rightmost **len** bits are shifted circularly **k** bits. If **k** is greater than the machine word-size, *ishftc* will not shift.

iand, ior, not, ieor, and *ishft* accept either integer*2 or integer*4 arguments and the result is the same type. When one of these intrinsics is specified as an *argument* in a subroutine call or function reference, the compiler supplies either an integer*2 or integer*4 function depending on the **-i2** command line option.

Bit fields are numbered from right to left and the rightmost bit position is zero. The length of the **len** field must be greater than zero.

ibits – extract a subfield of **len** bits from **m** starting with bit position **k** and extending left for **len** bits. The result field is right justified and the remaining bits are set to zero.

btest – The **k**th bit of argument **n** is tested. The value of the function is **.TRUE.** if the bit is a 1 and **.FALSE.** if the bit is 0.

ibset – the result is the value of **n** with the **k**th bit set to 1.

ibclr – the result is the value of **n** with the **k**th bit set to 0.

mvbits – **len** bits are moved beginning at position **k** of argument **m** to position **l** of argument **n**.

ORIGIN

MIPS Computer Systems

NAME

chmod – change mode of a file

SYNOPSIS

integer function chmod (name, mode)
character*(*) name, mode

DESCRIPTION

This function changes the filesystem *mode* of file *name*. *Mode* can be any specification recognized by *chmod*(1). *Name* must be a single pathname.

The normal returned value is 0. Any other value will be a system error number.

FILES

/usr/lib/libU77.a
/bin/chmod exec'ed to change the mode.

SEE ALSO

chmod(1)

BUGS

Pathnames can be no longer than MAXPATHLEN as defined in *<sys/param.h>*.

ORIGIN

MIPS Computer Systems

NAME

conjg, dconjg – FORTRAN complex conjugate intrinsic function

SYNOPSIS

complex cx1, cx2
double complex dx1, dx2

cx2 = conjg(cx1)

dx2 = dconjg(dx1)

DESCRIPTION

conjg returns the complex conjugate of its complex argument. *dconjg* returns the double-complex conjugate of its double-complex argument.

ORIGIN

MIPS Computer Systems

NAME

cos, dcos, ccos, zcos, cosd, dcosd – FORTRAN cosine intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2
complex cx1, cx2
complex*16 cd1, cd2
real*4 r3, r4
real*8 dp3, dp4

r2 = cos(r1)

dp2 = dcos(dp1)
dp2 = cos(dp1)

cx2 = ccos(cx1)
cx2 = cos(cx1)

dp4 = zcos(dp3)
dp4 = cos(dp3)

r4 = cosd(r3)

dp4 = dcosd(dp3)
dp4 = cosd(dp3)
```

DESCRIPTION

cos returns the real cosine of its real argument. *dcos* returns the double-precision cosine of its double-precision argument. *ccos* returns the complex cosine of its complex argument. *zcos* returns the complex*16 cosine of its complex*16 argument. The arguments for these routines must be in radians and is treated modulo 2P. The generic form *cos* may be used with impunity as its returned type is determined by that of its argument.

cosd returns the real*4 cosine of its real*4 argument. The argument for *cosd* must be in degrees and is treated as modulo 360. *dcosd* returns the real*8 cosine of its real*8 argument. The generic form *cosd* may be used with impunity as its returned type is determined by that of its argument.

SEE ALSO

trig(3M).

ORIGIN

MIPS Computer Systems

NAME

cosh, dcosh – FORTRAN hyperbolic cosine intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2

r2 = cosh(r1)

dp2 = dcosh(dp1)
dp2 = cosh(dp1)

DESCRIPTION

cosh returns the real hyperbolic cosine of its real argument. *dcosh* returns the double-precision hyperbolic cosine of its double-precision argument. The generic form *cosh* may be used to return the hyperbolic cosine in the type of its argument.

SEE ALSO

sinh(3M).

ORIGIN

MIPS Computer Systems

NAME

dim, ddim, idim, iidim, jidim – FORTRAN positive difference intrinsic functions

SYNOPSIS

real r1, r2, r3 double precision dp1, dp2, dp3 integer i1, i2, i3 integer*2 ii1, ii2, ii3 integer*4 ji1, ji2, ji3

r3 = dim(r1, r2)

dp3 = ddim(dp1, dp2) dp3 = dim(dp1, dp2)

i3 = idim(i1, i2) i3 = dim(i1, i2)

ii3 = iidim(ii1, ii2) ii3 = idim(ii1, ii2) ii3 = dim(ii1, ii2)

ji3 = jidim(ji1, ji2) ji3 = idim(ji1, ji2) ji3 = dim(ji1, ji2)

DESCRIPTION

These functions return:

arg1–arg2	if arg1 > arg2
0	if arg1 <= arg2

ORIGIN

MIPS Computer Systems

NAME

dprod – FORTRAN double precision product intrinsic function

SYNOPSIS

real a1, a2
double precision a3

a3 = dprod(a1, a2)

DESCRIPTION

dprod returns the double precision product of its real arguments.

ORIGIN

MIPS Computer Systems

NAME

etime, dtime – return elapsed execution time

SYNOPSIS

function etime (tarray)
real tarray(2)

function dtime (tarray)
real tarray(2)

DESCRIPTION

These two routines return elapsed runtime in seconds for the calling process. *Dtime* returns the elapsed time since the last call to *dtime*, or the start of execution on the first call.

The argument array returns user time in the first element and system time in the second element. The function value is the sum of user and system time.

The resolution of all timing is 1/HZ. See the system include file *param.h* in */usr/include/sys* for the value of HZ.

FILES

/usr/lib/libU77.a

SEE ALSO

times(2)

ORIGIN

MIPS Computer Systems

NAME

exp, dexp, cexp, zexp – FORTRAN exponential intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2
complex cx1, cx2
complex*16 cd1, cd2

r2 = exp(r1)

dp2 = dexp(dp1)
dp2 = exp(dp1)

cx2 = cexp(cx1)
cx2 = exp(cx1)

cd2 = zexp(cd1)
cd2 = exp(cd1)
```

DESCRIPTION

exp returns the real exponential function e^x of its real argument. *dexp* returns the double-precision exponential function of its double-precision argument. *cexp* returns the complex exponential function of its complex argument. *zexp* returns the complex*16 exponential function of its complex*16 argument. The generic function *exp* becomes a call to *dexp*, *cexp* or *zexp* as required, depending on the type of its argument.

SEE ALSO

exp(3M).

ORIGIN

MIPS Computer Systems

NAME

fdate – return date and time in an ASCII string

SYNOPSIS

subroutine fdate (string)
character*(*) string

character*(*) function fdate()

DESCRIPTION

Fdate returns the current date and time as a 24 character string in the format described under *ctime*(3). Neither 'newline' nor NULL will be included.

Fdate can be called either as a function or as a subroutine. If called as a function, the calling routine must define its type and length. For example:

```
character*24  fdate
external      fdate

write(*,*) fdate()
```

FILES

/usr/lib/libU77.a

SEE ALSO

ctime(3), time(3F), itime(3F), idate(3F), ltime(3F)

ORIGIN

MIPS Computer Systems

NAME

fixade – FORTRAN misaligned data bus error handler and report generator

SYNOPSIS

subroutine handle_unaligned_traps

subroutine list_by_addr

subroutine summary_listing

subroutine print_unaligned_summary

DESCRIPTION

Fixade is a FORTRAN bus error handler which fields, corrects, and reports bus errors arising due to misaligned data in FORTRAN programs. The MIPS architecture, for performance reasons, is very restrictive on the alignment of data which can be used with its standard instruction set. Usually, the compilers can guarantee this alignment. In FORTRAN, however, some situations exist in which this guarantee cannot be made. These misalignments may be necessary to satisfy equivalence statements, due to mismatched formal/actual parameter types, or due to user-instructed suppression of common block padding (via use of the *-align* switches, see *f77*(1)). Unless the bus error arising due to a load or store from a misaligned address is caught, it will cause unexpected program failure. Routines in the *fixade* package provide a bus error handler to catch these errors, correct them, and allow the program to continue execution. They also provide a reporting facility so that the causes of these errors can be located and remedied.

> **NOTE: the use of this trap handler is intended for diagnostic purposes only. Program efficiency may be severly impacted by its use.**

None of the routines of *fixade* have arguments. The routine **handle_unaligned_traps** *must* be called to initialize the handler. If a misaligned reference is encountered prior to calling this initialization routine, the reference will produce a core dump. No other routines of the trap handler may be called prior to calling this initialization routine.

No other routines of the trap handler need to be called unless a report of misaligned references is desired. A report of misaligned references consists of two portions: a summary of the types of misaligned instructions, their counts and relative frequency. (e.g., 'half aligned load-word occurred fifteen times, and accounted for 2% of all misaligned references'); and a listing based either on the instruction addresses at which the faults occurred, *or* the data addresses producing the faults.

This listing is either an *exhaustive* listing (default), or a *summary* listing. The summary listing will list the address (either instruction or data, as opted) associated with the fault, and its absolute and relative frequency, as a percentage. The exhaustive listing will list *all* instruction/data address pairs producing a fault. This listing will be sorted by the address on which the listing is based (i.e., by instruction address or data address). By default, the listing is exhaustive. If only a summary of misalignment errors is desired, the routine **summary_listing** must be called immediately after the initialization routine.

Also by default, the listing is based on instruction addresses. If it is desired to base the listing on data addresses, the routine **list_by_addr** must be invoked during initialization.

Prior to program exit, the routine **print_unaligned_summary** may be called to print the listing of bus error events, in either *summary* or *exhaustive* format, as described previously. This listing will go to the standard output. A sample line of this listing in summary format might be

```
0x0042445c  1536  33%  67%
```

where 0x0042445c is the address associated with 1536 faults (33% of the total). The final percentage is cumulative. Whether the address is of the data causing the fault or the instruction at which it occurred is indicated in a printed heading.

New options have been added to *f77*(1) to generate (much slower) code which tolerates misalignments (see *f77*(1)). As discussed previously, use of these options will suppress the padding of common usually done by the fortran compiler to align elements. They will also generate code which uses pessimistic code sequences to avoid bus errors due to misalignment. No bus errors due to misaligned data will occur in modules compiled with these new options.

Users desiring to find and repair instances of misaligned data may use either instruction addresses to decide which modules need to be specially compiled (see *f77*(1)), or data addresses to find misalignments. In either case, a symbol table listing produced by *nm*(1), using the -Bgn options, will be necessary to map the addresses to routine (or common block) names.

FILES

/usr/lib/fixade.o

AUTHOR

Larry Weber
Greg Boyd

SEE ALSO

f77(1)

DIAGNOSTICS

When making an exhaustive listing, the trap handler's tables may overflow. If this occurs, the message

number events not listed due to insufficient table size.

will be printed at the end of the listing.

ORIGIN

Silicon Graphics, Inc.

NAME

fseek, ftell – reposition a file on a logical unit

SYNOPSIS

integer function fseek (lunit, offset, from)
integer offset, from

integer function ftell (lunit)

DESCRIPTION

lunit must refer to an open logical unit. *offset* is an offset in bytes relative to the position specified by *from*. Valid values for *from* are:

0 meaning 'beginning of the file'
1 meaning 'the current position'
2 meaning 'the end of the file'

The value returned by *fseek* will be 0 if successful, a system error code otherwise. (See *perror*(3F))

Ftell returns the current position of the file associated with the specified logical unit. The value is an offset, in bytes, from the beginning of the file. If the value returned is negative, it indicates an error and will be the negation of the system error code. (See *perror*(3F))

FILES

/usr/lib/libU77.a

SEE ALSO

fseek(3S), perror(3F)

ORIGIN

MIPS Computer Systems

NAME

handle_sigfpes – floating-point exception handler package

SYNOPSIS

#include <fsigfpe.h>

subroutine
handle_sigfpes(onoff,en_mask,user_routine,abort_action,abort_routine)
integer *4 onoff, en_mask, abort_action
integer *4 abort_routine, user_routine
external abort_routine, user_routine

structure /sigfpe_template/
integer * 4 repls
integer * 4 count
integer * 4 trace
integer * 4 abort
integer * 4 exit
end structure
record /sigfpe_template/ fsigfpe (0:FPE_N_EXCEPTION_TYPES)
common / sigfpe / fsigfpe (0:FPE_N_EXCEPTION_TYPES)

integer * 4 results(0:FPE_N_INVALIDOP_RESULTS)
common / invalidop_results / results

integer * 4 invop(0:FPE_N_INVALIDOP_OPERANDS)
common / invalidop_operands / invop

DESCRIPTION

The MIPS floating-point accelerator may raise floating-point exceptions due to five conditions: **FPE_OVERFL**(*overflow*), **FPE_UNDERFL**(*underflow*), **FPE_DIVZERO**(*divide-by-zero*), **FPE_INEXACT**(*inexact result*), or **FPE_INVALID**(*invalid operand*, e.g., infinity). Usually these conditions are masked, and do not cause a floating-point exception. Instead, a default value is substituted for the result of the operation, and the program continues silently. This event may be intercepted by causing an exception to be raised. Once an exception is raised, the specific conditions which caused the exception may be determined, and more appropriate action taken.

The library **libfpe.a** privides two methods to unmask and handle these conditions: the subrouitne **handle_sigfpes**, and the environment variable **TRAP_FPE**. Both methods provide a mechanism for unmasking each condition except **FPE_INEXACT**, for *handling* and classifying exceptions arising from them, and for substituting either a default value or a chosen one. They also provide mechnisms to count, trace, exit or abort on enabled exceptions. The subroutine **handle_sigfpes** will always override options set by the environment variable **TRAP_FPE**. **TRAP_FPE** is supported for

Fortran, C and Pascal. **Handle_sigfpes** is supported for C and Fortran.

Arguments to **handle_sigfpes** have the following interpretation:

onoff is a flag indicating whether handling is being turned on *(onoff == FPE_ON)* or off *(onoff == FPE_OFF)*. Information from the sigfpe structure will be printed if *(onoff == FPE_DEBUG)*. (defined in *fsigfpe.h*).

en_mask indicates which of the four conditions should be unmasked, enabling them to raise floating-point exceptions. *en_mask* is only valid if onoff == FPE_ON, and is the sum of the constants FPE_EN_UNDERFL, FPE_EN_OVERFL, FPE_EN_DIVZERO, and FPE_EN_INVALID (defined in *fsigfpe.h*).

user_routine: **handle_sigfpes** provides a mechanism for setting the result of the operation to any one of a set of well-known values. If full control over the value of selected operations is desired for one or more exception conditions, a subroutine *user_routine* must be provided. For these selected exception conditions, *user_routine* will be called to set the value resulting from the operation.

abort_action: If the handler encounters an unexpected condition, an inconsistency, or begins looping, the flag *abort_action* indicates what action should be taken. Legal values are:

FPE_TURN_OFF_HANDLER_ON_ERROR	instruct the floating-point-accelerator to cease causing exceptions and continue. (i.e., disable handling)
FPE_ABORT_ON_ERROR	kill the process after giving an error message and possibly calling a user-supplied cleanup routine.
FPE_REPLACE_HANDLER_ON_ERROR	install the indicated user routine as the handler when such an error is encountered. Future floating-point exceptions will branch to the user-routine. (see signal(2))

abort_routine: When a fatal error (i.e., one described under *abort_action* above) is encountered, *abort_routine* is used as the address of a user subroutine. If abort_action is FPE_ABORT_ON_ERROR, and *abort_routine* is valid, it is called before aborting, and passed a pointer to the address of the instruction causing the exception as its single argument (*see below under* DIAGNOSTICS).

If *abort_action* is FPE_REPLACE_HANDLER_ON_ERROR, and *abort_routine* is valid, it will be installed as the new handler. In this case, the instruction which caused the unexpected exception will be re-executed, causing a new exception, and *abort_routine* entered. (see signal(2) for the correct interface for exception handlers)

When an exception is encountered, the handler examines the instruction causing the exception, the state of the floating-point accelerator and the sigfpe structure to determine the correct action to take, and the program is continued. In the cases of **FPE_UNDERFL, FPE_OVERFL, FPE_DIVZERO,** and some instances of **FPE_INVALID,** an appropriate value is substituted for the result of the operation, and the instruction which caused the exception is skipped. For most exceptions arising due to an invalid operand (**FPE_INVALID** exceptions), more meaningful behavior may be obtained by replacing an erroneous operand. For these conditions, the operand is replaced, and the instruction re-issued.

sigfpe: For each enabled exception, the sigfpe structure contains the fields: repls, count, trace, exit and abort. For each enabled exception <p>, and each non-zero entry <n> in the sigfpe structure, the trap handler will take the following actions:
count: A count of all enabled traps will be printed to stderr at the end of execution of the program , and every at <n>th exception <p>.
trace: A dbx stack trace will be printed to stderr every execption <p>, up to <n> times.
abort: Core dump and abort program upon encountering the <n>th exception <p>. The abort option takes precedence over the exit option.
exit: Exit program upon encountering the <n>th exception <p>. **repls:** Each of the exceptions **_UNDERFL**, **_OVERFL**, and **_DIVZERO** has an associated default value which is used as the result of the operation causing the exception. These default values may be overridden by initializing this integer value. This value is interpreted as an integer code used to select one of a set of replacement values, or to indicate that the routine *user_routine* is responsible for setting the value.

These integer codes are listed below:

FPE_ZERO	use zero as the replacement value
FPE_MIN	use the appropriately-typed minimum value as the replacement. (i.e., the smallest number which is representable in that format *without* denormalizing)
FPE_MAX	use the appropriately-typed maximum value as the replacement
FPE_INF	use the appropriately-typed value for infinity as the replacement
FPE_NAN	use the appropriately-typed value for not-a-number as the replacement. (A *quiet* not-a-number is used.)
FPE_APPROPRIATE	use a handler-supplied appropriate value as the replacement. These are different from the default values: FPE_ZERO for **FPE_UNDERFL**, FPE_MAX for **FPE_OVERFL**, FPE_INF for **FPE_DIVZERO**. Values for **FPE_INVALID** are handled on a case-by-case basis.
FPE_USER_DETERMINED	invoke the routine *user_routine* (see note) to set the value of the operation. If this is the code used for **FPE_INVALID** exceptions, all such exceptions will defer to *user_routine* to set their value. In this case, *invalidop_results_* and *invalidop_operands_* will be ignored.

The default values used as the results of floating-point exceptions are:

values for *fsigfpe().repls*			
#	element mnemonic	exception condition	default value
0	(none)	(ignored)	
1	FPE_UNDERFL	underflow	FPE_MIN
2	FPE_OVERFL	overflow	FPE_MAX
3	FPE_DIVZERO	divide-by-zero	FPE_MAX
4	FPE_INVALID	invalid operand	FPE_APPROPRIATE

For **FPE_INVALID** exceptions, the correct action may be either to set the result and skip the instruction, or to replace an operand and retry the instruction. There are four cases in which the result is set. The integer array constituting the named common *invalidop_results* is consulted for replacement codes for these cases:

array in common block *invalidop_results*			
#	element mnemonic	exception condition	default value
0	(none)	(ignored)	
1	FPE_MAGNITUDE_INF_SUBTRACTION	$\infty - \infty$	FPE_INF
2	FPE_ZERO_TIMES_INF	$0 * \infty$	FPE_ZERO
3	FPE_ZERO_DIV_ZERO	0/0	FPE_ZERO
4	FPE_INF_DIV_INF	∞ / ∞	FPE_INF

There are six cases in which an offending operand is replaced. An integer array constituting the named common *invalidop_operands* is consulted for user-initialized codes for these cases.

Each element governs the following cases:

array in common block *invalidop_operands*			
#	element mnemonic	exception condition	default value
0	(none)	(ignored)	(none)
1	FPE_SQRT_NEG_X	sqrt(-x)	(currently not supported)
2	FPE_CVT_OVERFL	conversion to real caused target to overflow	FPE_MAX
3	FPE_TRUNK_OVERFL	conversion to integer caused target to overflow	FPE_MAX
4	FPE_CVT_NAN	conversion of NaN	FPE_MAX
5	FPE_CVT_INF	conversion of ∞	FPE_MAX
6	FPE_UNORDERED_CMP	comparison to NaN	FPE_MAX
7	FPE_SNAN_OP	operand was Signaling Nan	FPE_MAX

NOTE

Use of *user_routine* to set values

If the integer code defining the replacement value for a particular exception condition is FPE_USER_DEFINED, the user-supplied routine *user_routine* is called:

call user_routine(exception_parameters, value)

value is an *integer * 4* array of length two into which *user_routine* should store the replacement value.

exception_parameters is a zero-based *integer * 4* array of length five which describes the exception condition:

array *exception_parameters*		
element #	mnemonic	description
0	FPE_EXCEPTION_TYPE	the exception type (FPE_DIVZERO, etc).
1	FPE_INVALID_ACTION	value = FPE_SET_RESULT if result is being set. Otherwise, an operand is being replaced. This element is meaningful only if the exception type is **FPE_INVALID.**
2	FPE_INVALID_TYPE	This element is meaningful only if the exception type is **FPE_INVALID** It is the index corresponding to the particular conditions giving rise to the exception. In conjunction with element 1, this value uniquely determines the exception condition. (e.g., if FPE_INVALID_ACTION is FPE_SET_RESULT and FPE_INVALID_TYPE is 2, the **FPE_INVALID** exception is due to FPE_ZERO_TIMES_INF.)
3	FPE_VALUE_TYPE	the type of the replacement value - either FPE_SINGLE, FPE_DOUBLE or FPE_WORD.
4	FPE_VALUE_SIGN	the suggested sign *user_routine* should use for the replacement value - either FPE_POSITIVE or FPE_NEGATIVE.

The environment variable **TRAP_FPE:**

If the code has been compiled with **libfpe.a**, the runtime startup routine will check for the environment variable "**TRAP_FPE**". The string read as the value of **TRAP_FPE** will be interpreted and **handle_sigfpes** will be called with the resulting values. If the program contains an explicit call to handle_sigfpes, that call will override all actions defined by **TRAP_FPE.**

TRAP_FPE is read in upper case letters only. The string assigned to **TRAP_FPE** may be in upper case or lower case. **TRAP_FPE** can take one of two forms: either a global value, or a list of individual items.

global values:

"" or OFF	Execute the program with no trap handling enabled. Same as TRAP_FPE undefined. Same as linking without libfpe.a
ON	Same as TRAP_FPE="ALL=DEFAULT".

Alternately, replacement values and actions may be specified for each of the possible trap types individually. This is accomplished by setting the environment variable as follows:

setenv **TRAP_FPE** "item;item;item...."

an item can be one of the following:

traptype=statuslist	Where traptype defines the specific floating point exception to enable, and statuslist defines the list of actions upon encountering the trap.
DEBUG	Confirm the parsing of the environment variable, and the trap actions.

Traptype can be one of the following literal strings:

UNDERFL	underflow
OVERFL	overflow
DIVZERO	divide by zero
INVALID	invalid operand
ALL	all of the above

Statuslist is a list seperated by commas. It contains an optional symbolic replacement value, and an optional list of actions.

symbolic replacement values:

DEFAULT	Do not override the predefined default values.
IEEE	Maps to integer code _APPROPRIATE.
ZERO	Maps to integer code _ZERO.
MIN	Maps to integer code _MIN.
MAX	Maps to integer code _MAX.
INF	Maps to integer code _INF.
NAN	Maps to integer code _NAN.

All actions take an optional integer in parentheses:

Note: for any traps that have an action and no specified replacement value, the DEFAULT replacement value will be used.

COUNT(n)	A count of the trap type will be printed to stderr every nth trap, and at the end of the program. Default is MAXINT.
ABORT(n)	Core dump and abort the program upon encountering the nth trap. Default id 1.
EXIT(n)	Exit program upon encountering the nth trap. Default id 1.
TRACE(n)	If a trap is encountered, Print a stack trace to stderr up to n times. Default is 10.

EXAMPLE

setenv **TRAP_FPE** "ALL=COUNT; UNDERFL=ZERO; OVERFL=IEEE,TRACE(5), ABORT(100); DIVZERO=ABORT"

Count all traps, trace the first five overflows, abort on the first divide by zero, or the 100th overflow. Replace zero for underflows, the "appropriate" value for overflows, and the default values for divide by zero, and invalid operands.

SEE ALSO

signal(3c), sigfpe(3c)

DIAGNOSTICS

If the handler encounters an unexpected condition, an inconsistency, or begins looping, the flag *abort_action* and subroutine address *abort_routine* (parameters to **handle_sigfpes**) indicate what action should be taken. If abort_action is FPE_ABORT_ON_ERROR, the handler will be removed leaving the exceptions enabled, an error message printed, and the instruction causing the fault re-issued, giving a core dump. Prior to this, if *abort_routine* is valid, it is invoked as

call abort_routine(ptr_to_pc)

where *ptr_to_pc* is an integer * 4 parameter whose value is the address of the instruction which caused the exception.

If *abort_action* is FPE_REPLACE_HANDLER_ON_ERROR, and *abort_routine* is valid, **handle_sigfpes** removes its handler and installs *abort_routine* as the new handler. The instruction which caused the exception will be re-executed, causing a new exception, and *abort_routine* entered. (see signal(2))

If *abort_action* is FPE_TURN_OFF_HANDLER_ON_ERROR **handle_sigfpes** will mask (disable) floating-point exceptions and remove its handler. The instruction which caused the fault will then be re-issued, continuing the program as if floating-point exceptions had never been enabled.

Any other combination of the two parameters *abort_action* and *abort_routine* will cause **handle_sigfpes** to remove its handler, generate an error message, and re-issue the instruction causing the exception, producing a core dump.

NAME

ftype: int, ifix, iifix, jifix, idint, real, float, floati, floatj, sngl, dble, dfloti, dflotj, dfloat, cmplx, dcmplx, ichar, char – explicit FORTRAN type conversion

SYNOPSIS

```
integer i, j
real r, s
double precision dp, dq
complex cx, cy, cz
double complex dcx, dcy, dcz
character*1 ch
integer*2 ii
integer*4 ji
real*4 r1
real*8 dp1

i = int(j)
i = int(r)
i = int(dp)
i = int(cx)
i = ifix(r)
ii = iifix(r1)
ji = jifix(r1)
i = idint(dp)
i = idint(cx)

r = real(i)
r = real(dp)
r = real(cx)
r = real(s)
r = float(i)
r1 = floati(ii)
r1 = floatj(ji)
r = sngl(dp)
r = sngl(cx)
r = sngl(s)

dp = dble(i)
dp = dble(r)
dp = dble(dq)
dp = dble(cx)
dp = dfloat(r)
dp = dfloat(dp)
dp = dfloat(cx)
dp1 = dfloti(ii)
```

dp1 = dflotj(ji)

cx = cmplx(i)
cx = cmplx(i, j)
cx = cmplx(r)
cx = cmplx(r, s)
cx = cmplx(dp)
cx = cmplx(dp, dq)
cx = cmplx(cy)
cx = cmplx(cy, cz)
cx = cmplx(dcx)
cx = cmplx(dcx, dcy)

dcx = dcmplx(i)
dcx = dcmplx(i, j)
dcx = dcmplx(r)
dcx = dcmplx(r, s)
dcx = dcmplx(dp)
dcx = dcmplx(dp, dq)
dcx = dcmplx(cx)
dcx = dcmplx(cx, cy)
dcx = dcmplx(dcy)
dcx = dcmplx(dcy, dcz)

i = ichar(ch)
ch = char(i)

DESCRIPTION

These functions perform conversion from one data type to another.

The function **int** converts to *integer* form its *real*, *integer*, *real*4*, *double precision*, or *complex* argument. If the argument is *real*, *integer*, *real*4*, or *double precision*, **int** returns the integer whose magnitude is the largest integer that does not exceed the magnitude of the argument and whose sign is the same as the sign of the argument (i.e. truncation). For *complex* the above rule is applied to the real part. **ifix** converts only *real* arguments. **int** and **ifix** return result type *integer*2* if the **-i2** option is in effect; otherwise, the result type is *integer*4*. **iifix** and **jifix** convert only *real*4* to *integer*2* and *integer*4*, respectively. **idint** converts *double precision* and *complex* arguments only.

The function **real** converts to *real* form an *integer*, *integer*2*, *integer*4*, *real*, *double precision*, or *complex* argument. If the argument is *double precision*, as much precision is kept as is possible. If the argument is *complex*, the real part is returned. **float** converts *integer* arguments only. **floati** and **floatj** convert *integer*2* and *integer*4* arguments respectively to *real*4*. **sngl** converts *double*, *complex* and *real* arguments to *real*.

The function **dble** converts any *integer*, *real*, *double*, *complex*, *integer*2* or *integer*4* argument to *double precision* form. If the argument is *complex*, the real part is returned. **dfloat** converts *real*, *double*, and *complex* to *double*. **dfloti** and **dflotj** convert *integer*2* and *integer*4* to *real*8*.

The function **cmplx** converts its *integer*, *real*, *double precision*, or *double complex* argument(s) to *complex* form.

The function **dcmplx** converts to *double complex* form its *integer*, *real*, *double precision*, or *complex* argument(s).

Either one or two arguments may be supplied to **cmplx** and **dcmplx** . If there is only one argument, it is taken as the real part of the complex type and an imaginary part of zero is supplied. If two arguments are supplied, the first is taken as the real part and the second as the imaginary part.

The function **ichar** converts from a character to an integer depending on the character's position in the collating sequence. **ichar** returns the result type *integer*2* if the **-i2** compile option is in effect; otherwise the result type is *integer*4*.

The function **char** returns the character in the *i*th position in the processor collating sequence where *i* is the supplied argument.

ORIGIN

MIPS Computer Systems

NAME

getarg, iargc – return Fortran command-line argument

SYNOPSIS

```
character*N c
integer i, j
integer function iargc

call getarg(i, c)
j = iargc()
```

DESCRIPTION

getarg returns the *i*-th command-line argument of the current process.

iargc returns the index of the last argument.

```
foo arg1 arg2 arg3
```

getarg(2, c) would return the string "arg2" in the character variable *c*.

iargc would return 3 as the value of the function call.

SEE ALSO

getopt(3C).

NOTES

The compiler expects the existence of a Fortran MAIN_ program when these functions are used.

ORIGIN

AT&T V.3

NAME

getc, fgetc – get a character from a logical unit

SYNOPSIS

integer function getc (char)
character char

integer function fgetc (lunit, char)
character char

DESCRIPTION

These routines return the next character from a file associated with a fortran logical unit, bypassing normal fortran I/O. *Getc* reads from logical unit 5, normally connected to the control terminal input.

The value of each function is a system status code. Zero indicates no error occurred on the read; −1 indicates end of file was detected. A positive value will be either a UNIX system error code or an f77 I/O error code. See perror(3F).

BUGS

fgetc(3f) does not work for FORTRAN unit numbers other than 5.

FILES

/usr/lib/libU77.a

SEE ALSO

getc(3S), intro(2), perror(3F)

ORIGIN

MIPS Computer Systems

NAME

getenv – get value of environment variable

SYNOPSIS

subroutine getenv(ename, evalue)

character *(*) ename, evalue

DESCRIPTION

getenv returns the character-string value of the environment variable represented by its first argument into the character variable of its second argument. If no such environment variable exists, all blanks will be returned.

SEE ALSO

getenv(3C), environ(5).

ORIGIN

AT&T V.3

NAME

getlog – get user's login name

SYNOPSIS

subroutine getlog (name)
character*(*) name

character*(*) function getlog()

DESCRIPTION

Getlog will return the user's login name or all blanks if the process is running detached from a terminal.

FILES

/usr/lib/libU77.a

SEE ALSO

getlogin(3)

ORIGIN

MIPS Computer Systems

NAME

idate, itime – return date or time in numerical form

SYNOPSIS

subroutine idate (imon,iday,iyear)
integer imon,iday,iyear

subroutine itime (iarray)
integer iarray(3)

DESCRIPTION

Idate returns the current date in the variables *imon*, *iday*, and *iyear*. The order is: mon, day, year. Month will be in the range 1-12. Year will be returned as the last two digits.

Itime returns the current time in *iarray*. The order is: hour, minute, second.

FILES

/usr/lib/libU77.a

SEE ALSO

ctime(3F), fdate(3F)

ORIGIN

MIPS Computer Systems

NAME

imag, aimag, dimag – FORTRAN imaginary part of complex argument

SYNOPSIS

```
real r
complex cxr
double precision dp
double complex cxd

r = aimag(cxr)
r = imag(cxr)

dp = dimag(cxd)
dp = imag(cxd)
```

DESCRIPTION

aimag returns the imaginary part of its single-precision complex argument. *dimag* returns the double-precision imaginary part of its double-complex argument. The generic form *imag* may be used with impunity as its argument will determine the type of the returned value.

ORIGIN

MIPS Computer Systems

NAME

index – return location of FORTRAN substring

SYNOPSIS

```
character*N1 ch1
character*N2 ch2
integer i

i = index(ch1, ch2)
```

DESCRIPTION

The result of *index* is an integer value indicating the position in the first argument of the first substring which is identical to the second argument. The result of *index('ABCDEF','CD')*, for example, would be 3. If no substring of the first argument matches the second argument, the result is zero. *index* returns the result type integer*2 if the **-i2** compile option is in effect; otherwise, the result type is integer*4.

ORIGIN

MIPS Computer Systems

NAME

len – return length of Fortran string

SYNOPSIS

```
character*N ch
integer i

i = len(ch)
```

DESCRIPTION

len returns the length of string *ch*.

ORIGIN

MIPS Computer Systems

NAME

loc – return the address of an object

SYNOPSIS

function loc(arg)

DESCRIPTION

The returned value will be the address of *arg*.

FILES

/usr/lib/libU77.a

ORIGIN

MIPS Computer Systems

NAME

log, alog, dlog, clog, zlog – FORTRAN natural logarithm intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2
complex cx1, cx2
complex*16 cd1, cd2

r2 = alog(r1)
r2 = log(r1)

dp2 = dlog(dp1)
dp2 = log(dp1)

cx2 = clog(cx1)
cx2 = log(cx1)

cd2 = zlog(cd1)
cd2 = log(cd1)
```

DESCRIPTION

alog returns the real natural logarithm of its real argument. *dlog* returns the double-precision natural logarithm of its double-precision argument. The argument of *alog* and *dlog* must be greater than zero. *clog* returns the complex logarithm of its complex argument. The argument of *clog* must not be (0.,0.). The range of the imaginary part of *clog* is: -p < imaginary part <= p. *zlog* returns the complex*16 logarithm of its complex*16 argument. The generic function *log* becomes a call to *alog*, *dlog*, *clog*, or *zlog* depending on the type of its argument.

SEE ALSO

exp(3M).

ORIGIN

MIPS Computer Systems

NAME

log10, alog10, dlog10 – FORTRAN common logarithm intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2

r2 = alog10(r1)
r2 = log10(r1)

dp2 = dlog10(dp1)
dp2 = log10(dp1)

DESCRIPTION

alog10 returns the real common logarithm of its real argument. *dlog10* returns the double-precision common logarithm of its double-precision argument. The absolute value of the argument for *alog10* and *dlog10* must be greater than zero. The generic function *log10* becomes a call to *alog10* or *dlog10* depending on the type of its argument.

SEE ALSO

exp(3M).

ORIGIN

MIPS Computer Systems

NAME

malloc, free – main memory allocator

SYNOPSIS

pointer ptr

ptr = malloc(nbytes)

call free(ptr)

DESCRIPTION

malloc and *free* provide a simple general-purpose memory allocation package. *malloc* returns a pointer to a block of at least *nbytes* bytes suitably aligned for any use.

The argument to *free* is a pointer to a block previously allocated by *malloc*; after *free* is performed this space is made available for further allocation, but its contents are left undisturbed.

Undefined results will occur if the space assigned by *malloc* is overrun or if some random number is handed to *free*.

malloc allocates the first big enough contiguous reach of free space found in a circular search from the last block allocated or freed, coalescing adjacent free blocks as it searches. It calls *sbrk* [see *brk*(2)] to get more memory from the system when there is no suitable space already free.

DIAGNOSTICS

malloc, returns a NULL pointer if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block. When this happens the block pointed to by *ptr* may be destroyed.

NOTES

Search time increases when many objects have been allocated; that is, if a program allocates but never frees, then each successive allocation takes longer.

NAME

max, max0, imax0, jmax0, amax0, max1, amax1, dmax1, imax1, jmax1, aimax0, ajmax0 – FORTRAN maximum-value functions

SYNOPSIS

```
integer i, j, k, l
integer*2 ii1, ii2, ii3, ii4
integer*4 ji1, ji2, ji3, ji4
real a, b, c, d
real*4 r1, r2, r3, r4
double precision dp1, dp2, dp3

k = max0(i, j)
l = max(i, j, k)

ii4 = imax0(ii1, ii2, ii3)
ii4 = max(ii1, ii2, ii3)
ii3 = max0(ii1, ii2)

ji4 = jmax0(ji1, ji2, ji3)
ji3 = max(ji1, ji2)
ji4 = max0(ji1, ji2, ji3)

a = amax0(i, j, k)
a = max(i, j, k)

i = max1(a, b)
i = max(a, b, c)

d = amax1(a, b, c)
c = max(a, b)

dp3 = dmax1(dp1, dp2)
dp4 = max(dp1, dp2, dp3)

ii1 = imax1(r1, r2)
ii1 = max1(r1, r2, r3)

ji1 = jmax1(r1, r2)
ji1 = max1(r1, r2, r3)

r1 = aimax0(ii1, ii2, ii3)
r1 = amax0(ii1, ii2, ii3)

r1 = aimax0(ji1, ji2, ji3)
r1 = amax0(ji1, ji2, ji3)
```

DESCRIPTION

The maximum-value functions return the largest of their arguments. There may be any number of arguments, but they must all be of the same type. *max0* returns the integer form of the maximum value of its integer

arguments; *amax0*, the real form of its integer arguments; *max1*, the integer form of its real arguments; *amax1*, the real form of its real arguments; *dmax1*, the double-precision form of its double-precision arguments; *imax1*, the integer*2 form of its real*4 arguments; *jmax1*, the integer*4 form of its real*4 arguments; *aimax0*, the real*4 form of its integer*2 arguments; and *ajmax0*, the real*4 form of its integer*4 arguments. *max*, *max0*, *max1*, and *amax0* are the generic forms which can be used as indicated above.

SEE ALSO

min(3F).

ORIGIN

MIPS Computer Systems

NAME

mclock – return Fortran time accounting

SYNOPSIS

integer i

i = mclock( )

DESCRIPTION

mclock returns time accounting information about the current process and its child processes. The value returned is the sum of the current process's user time and the user and system times of all child processes.

SEE ALSO

times(2), clock(3C), system(3F).

ORIGIN

AT&T V.3

NAME

min, min0, imin0, jmin0, amin0, min1, amin1, dmin1, imin1, jmin1, aimin0, ajmin0 – FORTRAN minimum-value functions

SYNOPSIS

```
integer i, j, k, l
integer*2 ii1, ii2, ii3, ii4
integer*4 ji1, ji2, ji3, ji4
real a, b, c, d
real*4 r1, r2, r3, r4
double precision dp1, dp2, dp3

k = min0(i, j)
l = min(i, j, k)

ii4 = imin0(ii1, ii2, ii3)
ii4 = min(ii1, ii2, ii3)
ii3 = min0(ii1, ii2)

ji4 = jmin0(ji1, ji2, ji3)
ji3 = min(ji1, ji2)
ji4 = min0(ji1, ji2, ji3)

a = amin0(i, j, k)
a = min(i, j, k)

i = min1(a, b)
i = min(a, b, c)

d = amin1(a, b, c)
c = min(a, b)

dp3 = dmin1(dp1, dp2)
dp4 = min(dp1, dp2, dp3)

ii1 = imin1(r1, r2)
ii1 = min1(r1, r2, r3)

ji1 = jmin1(r1, r2)
ji1 = min1(r1, r2, r3)

r1 = aimin0(ii1, ii2, ii3)
r1 = amin0(ii1, ii2, ii3)

r1 = aimin0(ji1, ji2, ji3)
r1 = amin0(ji1, ji2, ji3)
```

DESCRIPTION

The minimum-value functions return the minimum of their arguments. There may be any number of arguments, but they must all be of the same type. *min0* returns the integer form of the minimum value of its integer

arguments; *amin0*, the real form of its integer arguments; *min1*, the integer form of its real arguments; *amin1*, the real form of its real arguments; *dmin1*, the double-precision form of its double-precision arguments; *imin1*, the integer*2 form of its real*4 arguments; *jmin1*, the integer*4 form of its real*4 arguments; *aimin0*, the real*4 form of its integer*2 arguments; and *ajmin0*, the real*4 form of its integer*4 arguments. *min*, *min0*, *min1*, and *amin0* are the generic forms which can be used as indicated above.

SEE ALSO

max(3F).

ORIGIN

MIPS Computer Systems

NAME

mod, imod, jmod, amod, dmod – FORTRAN remaindering intrinsic functions

SYNOPSIS

```
integer i, j, k
integer*2 ii1, ii2, ii3
integer*4 ji1, ji2, ji3
real r1, r2, r3
double precision dp1, dp2, dp3

k = mod(i, j)

ii3 = imod(ii1, ii2)
ii3 = mod(ii1, ii2)

ji3 = jmod(ji1, ji2)
ji3 = mod(ji1, ji2)

r3 = amod(r1, r2)
r3 = mod(r1, r2)

dp3 = dmod(dp1, dp2)
dp3 = mod(dp1, dp2)
```

DESCRIPTION

mod returns the integer remainder of its first argument divided by its second argument. *imod* returns the integer*2 remainder of its two integer*2 arguments. *jmod* returns the integer*4 remainder of its two integer*4 arguments. *amod* and *dmod* return, respectively, the real and double-precision whole number remainder of the integer division of their two arguments. The generic version *mod* will return the data type of its arguments. The result of these intrinsics is undefined when the value of the second argument is zero.

ORIGIN

MIPS Computer Systems

NAME

mp: mp_block, mp_blocktime, mp_create, mp_destroy, mp_my_threadnum, mp_numthreads, mp_set_numthreads, mp_setup, mp_unblock – FORTRAN multiprocessing utility routines

SYNOPSIS

subroutine mp_block()

subroutine mp_unblock()

subroutine mp_blocktime(iters)
integer iters

subroutine mp_setup()

subroutine mp_create(num)
integer num

subroutine mp_destroy()

integer function mp_numthreads()

subroutine mp_set_numthreads(num)
integer num

integer function mp_my_threadnum()

DESCRIPTION

These routines give some measure of control over the parallelism used in FORTRAN jobs. They should not be needed by most users, but will help to tune specific applications.

mp_block puts all slave threads to sleep via *blockproc*(2). This frees the processors for use by other jobs. This is useful if it is known that the slaves will not be needed for some time, and the machine is being shared by several users. Calls to *mp_block* may not be nested; a warning is issued if an attempt to do so is made.

mp_unblock wakes up the slave threads that were previously blocked via *mp_block*. It is an error to unblock threads that are not currently blocked; a warning is issued if an attempt is made to do so.

It is not necessary to explicitly call *mp_unblock*. When a FORTRAN parallel region is entered, a check is made, and if the slaves are currently blocked, a call is made to *mp_unblock* automatically.

mp_blocktime controls the amount of time a slave thread waits for work before giving up. When enough time has elapsed, the slave thread blocks itself. This automatic blocking is independent of the user level blocking provided by the *mp_block/mp_unblock* calls. Slave threads that have blocked themselves will be automatically unblocked upon entering a parallel region. The argument to *mp_blocktime* is the number of times to spin in the wait loop. By default, it is set to 10,000,000. This takes about 3 seconds on a 16MHz processor. As a special case, an argument of 0 disables the automatic blocking, and the slaves will spin wait without limit. The environment variable *MP_BLOCKTIME* may be set to an integer value. It acts like an implicit call to *mp_blocktime* during program startup.

mp_destroy deletes the slave threads. They are stopped by forcing them to call *exit*(2). In general, doing this is discouraged. *mp_block* can be used in most cases.

mp_create creates and initializes threads. It creates enough threads so that the total number is equal to the argument. Since the calling thread already counts as one, *mp_create* will create one less than its argument in new slave threads.

mp_setup also creates and initializes threads. It takes no arguments. It simply calls *mp_create* using the current default number of threads. Normally the default number is equal to the number of cpu's currently on the machine. If the user has not called either of the thread creation routines already, then *mp_setup* is invoked automatically when the first parallel region is entered. If the environment variable *MP_SETUP* is set, then *mp_setup* is called during FORTRAN initialization, before any user code is executed.

mp_numthreads returns the number of threads that would participate in an immediately following parallel region. If the threads have already been created, then it returns the current number of threads. If the threads have not been created, then it returns the current default number of threads. Knowing this can be useful in optimizing certain kinds of parallel loops by hand.

mp_set_numthreads sets the current default number of threads to the specified value. Note that this call does not directly create the threads, it only specifies the number that a subsequent *mp_setup* call should use. If the environment variable *MP_SET_NUMTHREADS* is set to an integer value, it acts like an implicit call to *mp_set_numthreads* during program startup. For compatibility with earlier releases, *NUM_THREADS* is supported as a synonym for *MP_SET_NUMTHREADS*.

mp_my_threadnum returns an integer between 0 and *n*-1 where *n* is the value returned by *mp_numthreads*. The master process is always thread 0. This is occasionally useful for optimizing certain kinds of loops by hand.

SEE ALSO

FORTRAN 77 Programmer's Guide

ORIGIN

Silicon Graphics, Inc.

NAME

perror, gerror, ierrno – get system error messages

SYNOPSIS

subroutine perror (string)
character*(*) string

subroutine gerror (string)
character*(*) string

character*(*) function gerror()

function ierrno()

DESCRIPTION

Perror will write a message to fortran logical unit 0 appropriate to the last detected system error. *String* will be written preceding the standard error message.

Gerror returns the system error message in character variable *string*. *Gerror* may be called either as a subroutine or as a function.

Ierrno will return the error number of the last detected system error. This number is updated only when an error actually occurs. Most routines and I/O statements that might generate such errors return an error code after the call; that value is a more reliable indicator of what caused the error condition.

FILES

/usr/lib/libU77.a

SEE ALSO

intro(2), perror(3)
D. L. Wasley, *Introduction to the f77 I/O Library*

BUGS

String in the call to *perror* can be no longer than 127 characters.

The length of the string returned by *gerror* is determined by the calling program.

NOTES

UNIX system error codes are described in *intro*(2). The f77 I/O error codes and their meanings are:

100	"error in format"
101	"illegal unit number"
102	"formatted i/o not allowed"

```
103   "unformatted i/o not allowed"
104   "direct i/o not allowed"
105   "sequential i/o not allowed"
106   "can't backspace file"
107   "off beginning of record"
108   "can't stat file"
109   "no * after repeat count"
110   "off end of record"
111   "truncation failed"
112   "incomprehensible list input"
113   "out of free space"
114   "unit not connected"
115   "invalid data for integer format term"
116   "invalid data for logical format term"
117   "'new' file exists"
118   "can't find 'old' file"
119   "opening too many files or unknown system error"
120   "requires seek ability"
121   "illegal argument"
122   "negative repeat count"
123   "illegal operation for unit"
124   "off beginning of record"
125   "no * after repeat count"
126   "'new' file exists"
127   "can't find 'old' file"
128   "unknown system error"
129   "requires seek ability"
130   "illegal argument"
131   "duplicate key value on write"
132   "indexed file not open"
133   "bad isam argument"
134   "bad key description"
135   "too many open indexed files"
136   "corrupted isam file"
137   "isam file not opened for exclusive access"
138   "record locked"
139   "key already exists"
140   "cannot delete primary key"
141   "beginning or end of file reached"
142   "cannot find requested record"
143   "current record not defined"
144   "isam file is exclusively locked"
145   "filename too long"
```

146 "cannot create lock file"
147 "record too long"
148 "key structure does not match file structure"
149 "direct access on an indexed file not allowed"
150 "keyed access on a f77sequential file not allowed"
151 "keyed access on a relative file not allowed"
152 "append access on an indexed file not allowed"
153 "must specify record length"
154 "key field value type does not match key type"
155 "character key field value length too long"
156 "fixed record on f77sequential file not allowed"
157 "variable records allowed only on unformatted f77sequential file"
158 "stream records allowed only on f77formatted f77sequential file"
159 "maximum number of records in direct access file exceeded"
160 "attempt to write to a readonly file"
161 "must specify key descriptions"
162 "carriage control not allowed for unformatted units"
163 "indexed files only"
164 "cannot use on indexed file"
165 "cannot use on indexed or append file"

ORIGIN

MIPS Computer Systems

NAME

putc, fputc – write a character to a fortran logical unit

SYNOPSIS

integer function putc (char)
character char

integer function fputc (lunit, char)
character char

DESCRIPTION

These funtions write a character to the file associated with a fortran logical unit bypassing normal fortran I/O. *Putc* writes to logical unit 6, normally connected to the control terminal output.

The value of each function will be zero unless some error occurred; a system error code otherwise. See *perror*(3F).

FILES

/usr/lib/libU77.a

SEE ALSO

putc(3S), intro(2), perror(3F)

ORIGIN

MIPS Computer Systems

NAME

putenv – change or add Fortran environment variable

SYNOPSIS

integer function putenv (string)
character *(*) string

DESCRIPTION

String contains a character string in the form *name=value*. *Putenv* makes the value of the environment variable *name* equal to *value* by altering or creating an environment variable.

FILES

/usr/lib/libU77.a

SEE ALSO

putenv(3C)

ORIGIN

AT&T V.3

NAME

qsort – quick sort

SYNOPSIS

subroutine qsort (array, len, isize, compar)
external compar
integer*2 compar

DESCRIPTION

One dimensional *array* contains the elements to be sorted. *len* is the number of elements in the array. *isize* is the size of an element, typically -

4 for **integer** and **real**
8 for **double precision** or **complex**
16 for **double complex**
(length of character object) for **character** arrays

Compar is the name of a user supplied integer*2 function that will determine the sorting order. This function will be called with 2 arguments that will be elements of *array*. The function must return -

negative if arg 1 is considered to precede arg 2
zero if arg 1 is equivalent to arg 2
positive if arg 1 is considered to follow arg 2

On return, the elements of *array* will be sorted.

FILES

/usr/lib/libU77.a

SEE ALSO

qsort(3)

ORIGIN

MIPS Computer Systems

NAME

rand, irand, srand – random number generator

SYNOPSIS

```
integer iseed, i, irand
double precision x, rand

call srand(iseed)

i = irand( )

x = rand( )
```

DESCRIPTION

Irand generates successive pseudo-random integers in the range from 0 to 2**15–1. *rand* generates pseudo-random numbers distributed in [0, 1.0]. *Srand* uses its integer argument to re-initialize the seed for successive invocations of *irand* and *rand*.

SEE ALSO

rand(3C).

ORIGIN

AT&T V.3

NAME

rename – rename a file

SYNOPSIS

integer function rename (from, to)
character*(*) from, to

DESCRIPTION

From must be the pathname of an existing file. *To* will become the new pathname for the file. If *to* exists, then both *from* and *to* must be the same type of file, and must reside on the same filesystem. If *to* exists, it will be removed first.

The returned value will be 0 if successful; a system error code otherwise.

FILES

/usr/lib/libU77.a

SEE ALSO

rename(2), perror(3F)

BUGS

Pathnames can be no longer than MAXPATHLEN as defined in *<sys/param.h>*.

ORIGIN

MIPS Computer Systems

NAME

round: anint, dnint, nint, inint, jnint, idnint, iidnnt, jidnnt – FORTRAN nearest integer functions

SYNOPSIS

```
integer i
integer*2 ii
integer*4 ji
real r1, r2
real*4 r3
double precision dp1, dp2
real*8 dp3

r2 = anint(r1)

dp2 = dnint(dp1)
dp2 = anint(dp1)

i = nint(dp1)

ii = inint(r3)
ii = nint(r3)

ji = jnint(r3)
ji = nint(r3)

i = idnint(dp1)
i = nint(dp1)

ii = iidnnt(dp3)
ji = jidnnt(dp3)
```

DESCRIPTION

anint returns the nearest whole real number to its real argument (i.e., int(a+0.5) if a ≥ 0, int(a–0.5) otherwise). *dnint* does the same for its double-precision argument. *anint* is the generic form of *anint* and *dnint*, performing the same operation and returning the data type of its argument. *nint* returns the nearest integer to its real argument. *inint* returns the nearest integer*2 to its real*4 argument. *jnint* returns the nearest integer*4 to its real*4 argument. *idnint* returns the nearest integer to its double precision argument. *nint* is the generic form of *inint*, *jnint* and *idnint*. *idnint* is also the generic form for *iidnnt*, which returns the nearest integer*2 to its real*8 argument, and *jidnnt*, which returns the nearest integer*4 to its real*8 argument. When *nint* or *idnint* is specified as an *argument* in a subroutine call or function reference, the compiler supplies either an integer*2 or integer*4 function depending on the **-i2** command line option.

NAME

sign, isign, iisign, jisign, dsign – FORTRAN transfer-of-sign intrinsic function

SYNOPSIS

```
integer i, j, k
integer*2 ii1, ii2, ii3
integer*4 ji1, ji2, ji3
real r1, r2, r3
double precision dp1, dp2, dp3

k = isign(i, j)
k = sign(i, j)

ii3 = iisign(ii1, ii2)
ii3 = sign(ii1, ii2)

ji3 = jisign(ji1, ji2)
ji3 = sign(ji1, ji2)

r3 = sign(r1, r2)

dp3 = dsign(dp1, dp2)
dp3 = sign(dp1, dp2)
```

DESCRIPTION

isign returns the magnitude of its first argument with the sign of its second argument. It accepts either integer*2 or integer*4 arguments and the result is the same type. *iisign* and *jisign* take integer*2 and integer*4 arguments, respectively. *sign* and *dsign* are *isign*'s real and double-precision counterparts, respectively. If the value of the first argument of *isign*, *sign*, or *dsign* is zero, the result is zero. The generic version is *sign* and will devolve to the appropriate type depending on its arguments.

ORIGIN

MIPS Computer Systems

NAME

sin, dsin, csin, zsin, sind, dsind – FORTRAN sine intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2
complex cx1, cx2
complex*16 cd1, cd2
real*4 r3, r4
real*8 dp3, dp4

r2 = sin(r1)

dp2 = dsin(dp1)
dp2 = sin(dp1)

cx2 = csin(cx1)
cx2 = sin(cx1)

cd2 = zsin(cd1)
cd2 = sin(cd1)

r4 = sind(r3)

dp4 = dsind(dp3)
dp4 = sind(dp3)
```

DESCRIPTION

sin returns the real sine of its real argument. *dsin* returns the double-precision sine of its double-precision argument. *csin* returns the complex sine of its complex argument. *zsin* returns the complex*16 sine of its complex*16 argument. The argument for these functions must be in radians and is treated modulo 2P. The generic *sin* function becomes *dsin*, *csin*, or *zsin* as required by argument type.

sind returns the real*4 sine of its real*4 argument or the real*8 sine of its real*8 argument. *dsind* returns the real*8 sine of its real*8 argument. The argument for *sind* and *dsind* must be in degrees and is treated as modulo 360.

SEE ALSO

trig(3M).

ORIGIN

MIPS Computer Systems

NAME

sinh, dsinh – FORTRAN hyperbolic sine intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2

r2 = sinh(r1)

dp2 = dsinh(dp1)
dp2 = sinh(dp1)

DESCRIPTION

sinh returns the real hyperbolic sine of its real argument. *dsinh* returns the double-precision hyperbolic sine of its double-precision argument. The generic form *sinh* may be used to return a double-precision value when given a double-precision argument.

SEE ALSO

sinh(3M).

ORIGIN

MIPS Computer Systems

NAME

sleep – suspend execution for an interval

SYNOPSIS

subroutine sleep (itime)

DESCRIPTION

Sleep causes the calling process to be suspended for *itime* seconds. The actual time can be up to 1 second less than *itime* due to granularity in system timekeeping.

FILES

/usr/lib/libU77.a

SEE ALSO

sleep(3)

ORIGIN

MIPS Computer Systems

NAME

sqrt, dsqrt, csqrt, zsqrt – FORTRAN square root intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2
complex cx1, cx2
complex*16 cd1, cd2

r2 = sqrt(r1)

dp2 = dsqrt(dp1)
dp2 = sqrt(dp1)

cx2 = csqrt(cx1)
cx2 = sqrt(cx1)

cd2 = zsqrt(cd1)
cd2 = sqrt(cd1)
```

DESCRIPTION

sqrt returns the real square root of its real argument. *dsqrt* returns the double-precision square root of its double-precision argument. The value of the argument of *sqrt* and *dsqrt* must be greater than or equal to zero.

csqrt returns the complex square root of its complex argument. The result of *csqrt* is the principle value with the real part greater than or equal to zero. When the real part is zero, the imaginary part is greater than or equal to zero.

zsqrt returns the complex*16 square root of its complex*16 argument.

sqrt, the generic form, will become *dsqrt*, *csqrt*, or *zsqrt* as required by its argument type.

SEE ALSO

exp(3M).

ORIGIN

MIPS Computer Systems

NAME

strcmp: lge, lgt, lle, llt – FORTRAN string comparison intrinsic functions

SYNOPSIS

character*N a1, a2
logical l

l = lge(a1, a2)
l = lgt(a1, a2)
l = lle(a1, a2)
l = llt(a1, a2)

DESCRIPTION

These functions return .TRUE. if the inequality holds and .FALSE. otherwise. They return the result type logical*2 if the **$log2** compile option is in effect; otherwise, the result type is logical*4.

ORIGIN

MIPS Computer Systems

NAME

system – issue a shell command from Fortran

SYNOPSIS

character*N c

call system(c)

DESCRIPTION

system causes its character argument to be given to *sh*(1) as input, as if the string had been typed at a terminal. The current process waits until the shell has completed.

SEE ALSO

exec(2), system(3S).
sh(1) in the *User's Reference Manual.*

ORIGIN

AT&T V.3

NAME

tan, dtan, tand, dtand – FORTRAN tangent intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2
real*4 r3, r4
real*8 dp3, dp4

r2 = tan(r1)

dp2 = dtan(dp1)
dp2 = tan(dp1)

r4 = tand(r3)

dp4 = dtand(dp3)
dp4 = tand(dp3)
```

DESCRIPTION

tan returns the real tangent of its real argument. *dtan* returns the double-precision tangent of its double-precision argument. The argument for *tan* and *dtan* must be in radians and is treated modulo 2P. The generic *tan* function becomes *dtan* as required with a double-precision argument.

tand returns the real*4 tangent of its real*4 argument. The argument for *tand* must be in degrees and is treated as modulo 360. *dtand* returns the real*8 tangent of its real*8 argument. The generic *tand* function becomes *dtand* as required with a real*8 argument.

SEE ALSO

trig(3M).

ORIGIN

MIPS Computer Systems

NAME

tanh, dtanh – FORTRAN hyperbolic tangent intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2

r2 = tanh(r1)

dp2 = dtanh(dp1)
dp2 = tanh(dp1)
```

DESCRIPTION

tanh returns the real hyperbolic tangent of its real argument. *dtanh* returns the double-precision hyperbolic tangent of its double-precision argument. The generic form *tanh* may be used to return a double-precision value given a double-precision argument.

SEE ALSO

sinh(3M).

ORIGIN

MIPS Computer Systems

NAME

time, ctime, ltime, gmtime – return system time

SYNOPSIS

integer function time()

character*(*) function ctime (stime)
integer stime

subroutine ltime (stime, tarray)
integer stime, tarray(9)

subroutine gmtime (stime, tarray)
integer stime, tarray(9)

DESCRIPTION

Time returns the time since 00:00:00 GMT, Jan. 1, 1970, measured in seconds. This is the value of the UNIX system clock.

Ctime converts a system time to a 24 character ASCII string. The format is described under *ctime*(3). No 'newline' or NULL will be included.

Ltime and *gmtime* disect a UNIX time into month, day, etc., either for the local time zone or as GMT. The order and meaning of each element returned in *tarray* is described under *ctime*(3).

FILES

/usr/lib/libU77.a

SEE ALSO

ctime(3), itime(3F), idate(3F), fdate(3F)

ORIGIN

MIPS Computer Systems

NAME

ttynam, isatty – find name of a terminal port

SYNOPSIS

character*(*) function ttynam (lunit)

logical function isatty (lunit)

DESCRIPTION

Ttynam returns a blank padded path name of the terminal device associated with logical unit *lunit*.

Isatty returns **.true.** if *lunit* is associated with a terminal device, **.false.** otherwise.

FILES

/dev/*
/usr/lib/libU77.a

DIAGNOSTICS

Ttynam returns an empty string (all blanks) if *lunit* is not associated with a terminal device in directory '/dev'.

ORIGIN

MIPS Computer Systems

Silicon Graphics, Inc.

Date _____

Your name _____

Title _____

Department _____

Company _____

Address _____

_____

_____

Phone _____

COMMENTS

Manual title and version _____

_____

Please list any errors, inaccuracies, or omissions you have found in this manual

_____

_____

_____

_____

Please list any suggestions you may have for improving this manual

_____

_____

_____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 45 MOUNTAIN VIEW, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Silicon Graphics, Inc.
Attention: Technical Publications
2011 N. Shoreline Boulevard
Mountain View, CA 94039-7311