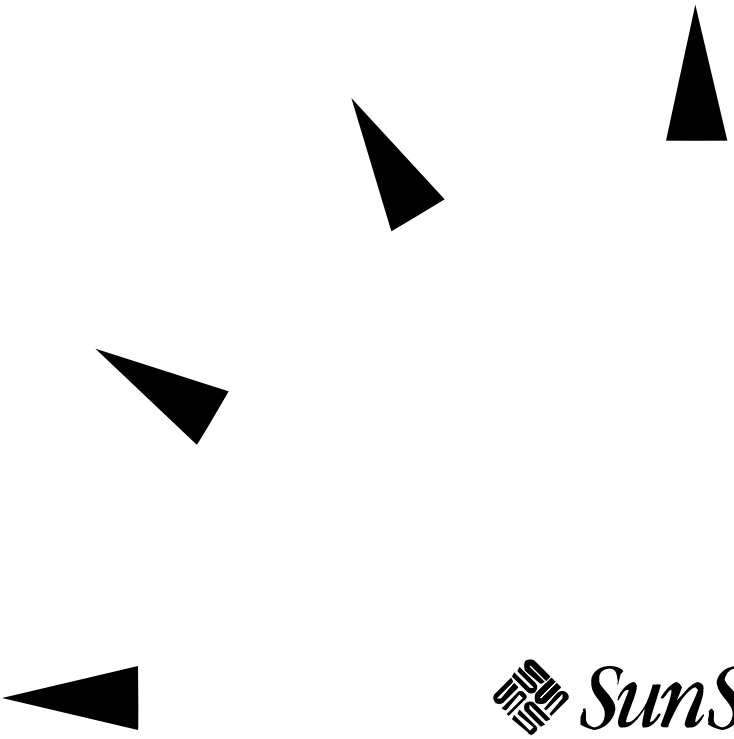
SunOS Reference Manual



Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

SunSoft
A Sun Microsystems, Inc. Business

Preface

OVERVIEW

A man page is provided for both the naive user, and sophisticated user who is familiar with the SunOS operating system and is in need of on-line information. A man page is intended to answer concisely the question “What does it do?” The man pages in general comprise a reference manual. They are not intended to be a tutorial.

The following contains a brief description of each section in the man pages and the information it references:

- Section 1 describes, in alphabetical order, commands available with the operating system.

- Section 1M describes, in alphabetical order, commands that are used chiefly for system maintenance and administration purposes.

- Section 2 describes all of the system calls. Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible returned value.

- Section 3 describes functions found in various libraries, other than those functions that directly invoke UNIX system primitives, which are described in Section 2 of this volume.

- Section 4 outlines the formats of various files. The C structure declarations for the file formats are given where applicable.

- Section 5 contains miscellaneous documentation such as character set tables, etc.

- Section 6 contains available games and demos.

- Section 7 describes various special files that refer to specific hardware peripherals, and device drivers. STREAMS software drivers, modules and the STREAMS-generic set of system calls are also described.

- Section 9 provides reference information needed to write device drivers in the kernel operating systems environment. It describes two device driver interface specifications: the Device Driver Interface (DDI) and the Driver–Kernel Interface (DKI).

- Section 9E describes the DDI/DKI, DDI-only, and DKI-only entry-point routines a developer may include in a device driver.

- Section 9F describes the kernel functions available for use by device drivers.

- Section 9S describes the data structures used by drivers to share information between the driver and the kernel.

Below is a generic format for man pages. The man pages of each manual section generally follow this order, but include only needed headings. For example, if there are no bugs to report, there is no BUGS section. See the intro pages for more information and detail about each section, and **man**(1) for more information about man pages in general.

NAME

This section gives the names of the commands or functions documented, followed by a brief description of what they do.

SYNOPSIS

This section shows the syntax of commands or functions. When a command or file does not exist in the standard path, its full pathname is shown. Literal characters (commands and options) are in **bold** font and variables (arguments, parameters and substitution characters) are in *italic* font. Options and

arguments are alphabetized, with single letter arguments first, and options with arguments next, unless a different argument order is required.

The following special characters are used in this section:

[ ] The option or argument enclosed in these brackets is optional. If the brackets are omitted, the argument *must* be specified.

... Ellipses. Several values may be provided for the previous argument, or the previous argument can be specified multiple times, for example, '*filename* ...'.

| Separator. Only one of the arguments separated by this character can be specified at time.

{ } Braces. The options and/or arguments enclosed within braces are interdependent, such that everything enclosed must be treated as a unit.

PROTOCOL

This section occurs only in subsection 3R to indicate the protocol description file. The protocol specification pathname is always listed in **bold** font.

AVAILABILITY

This section briefly states any limitations on the availabilty of the command. These limitations could be hardware or software specific.

A specification of a class of hardware platform, such as **x86** or **SPARC**, denotes that the command or interface is applicable for the hardware platform specified.

In Section 1 and Section 1M, **AVAILABILITY** indicates which package contains the command being described on the manual page. In order to use the command, the specified package must have been installed with the operating system. If the package was not installed, see **pkgadd**(1) for information on how to upgrade.

MT-LEVEL

This section lists the **MT-LEVEL** of the library functions described in the Section 3 manual pages. The **MT-LEVEL** defines the libraries' ability to support threads. See **Intro**(3) for more information.

DESCRIPTION

This section defines the functionality and behavior of the service. Thus it describes concisely what the command does. It does not discuss OPTIONS or cite EXAMPLES. Interactive commands, subcommands, requests, macros, functions and such, are described under USAGE.

IOCTL

This section appears on pages in Section 7 only. Only the device class which supplies appropriate parameters to the **ioctl**(2) system call is called **ioctl** and generates its own heading. **ioctl** calls for a specific device are listed alphabetically (on the man page for that specific device). **ioctl** calls are used for a particular class of devices all of which have an **io** ending, such as **mtio**(7).

OPTIONS

This lists the command options with a concise summary of what each option does. The options are listed literally and in the order they appear in the SYNOPSIS section. Possible arguments to options are discussed under the option, and where appropriate, default values are supplied.

OPERANDS

This section lists the command operands and describes how they affect the actions of the command.

OUTPUT

This section describes the output - standard output, standard error, or output files - generated by the command.

RETURN VALUES

If the man page documents functions that return values, this section lists these values and describes the conditions under which they are returned. If a function can return only constant values, such as 0 or –1, these values are listed in tagged paragraphs. Otherwise, a single paragraph describes the return values of each function. Functions declared as **void** do not return values, so they are not discussed in RETURN VALUES.

ERRORS

On failure, most functions place an error code in the global variable **errno** indicating why they failed. This section lists alphabetically all error codes a function can generate and describes the conditions that cause each error. When more than one condition can cause the same error, each condition is described in a separate paragraph under the error code.

USAGE

This section is provided as a *guidance* on use. This section lists special rules, features and commands that require in-depth explanations. The subsections listed below are used to explain built-in functionality:

Commands
Modifiers
Variables
Expressions
Input Grammar

EXAMPLES

This section provides examples of usage or of how to use a command or function. Wherever possible a complete example including command line entry and machine response is shown. Whenever an example is given, the prompt is shown as

example%

or if the user must be super-user,

example#

Examples are followed by explanations, variable substitution rules, or returned values. Most examples illustrate concepts from the SYNOPSIS, DESCRIPTION, OPTIONS and USAGE sections.

ENVIRONMENT

This section lists any environment variables that the command or function affects, followed by a brief description of the effect.

EXIT STATUS

This section lists the values the command returns to the calling program or shell and the conditions that cause these values to be returned. Usually, zero is returned for successful completion and values greater than zero for various error conditions.

FILES

This section lists all filenames referred to by the man page, files of interest, and files created or required by commands. Each is followed by a descriptive summary or explanation.

SEE ALSO

This section lists references to other man pages, in-house documentation and outside publications.

DIAGNOSTICS

This section lists diagnostic messages with a brief explanation of the condition causing the error. Messages appear in **bold** font with the exception of variables, which are in *italic* font.

WARNINGS

This section lists warnings about special conditions which could seriously affect your working conditions — this is not a list of diagnostics.

NOTES

This section lists additional information that does not belong anywhere else on the page. It takes the form of an *aside* to the user, covering points of special interest. Critical information is never covered here.

BUGS

This section describes known bugs and wherever possible suggests workarounds.

NAME

Intro, intro – introduction to miscellany

DESCRIPTION

This section contains miscellaneous documentation such as character set tables, etc.

Name	Description
advance(5)	See **regexp**(5)
ascii(5)	map of ASCII character set
charmap(5)	character set description file
compile(5)	See **regexp**(5)
environ(5)	user environment
eqnchar(5)	special character definitions for eqn
fcntl(5)	file control options
filesystem(5)	file system organization
floatingpoint(5)	IEEE floating point definitions
fnmatch(5)	file name pattern matching
fns(5)	overview of FNS
fns_initial_context(5)	overview of the FNS Initial Context
fns_policies(5)	Overview of the FNS Policies
fns_references(5)	Overview of FNS References
formats(5)	file format notation
iconv(5)	code set conversion tables
langinfo(5)	language information constants
locale(5)	subset of a user's environment that depends on language and cultural conventions
man(5)	macros to format Reference Manual pages
mansun(5)	macros to format Reference Manual pages
math(5)	math functions and constants
me(5)	macros for formatting papers
mm(5)	text formatting (memorandum) macros
ms(5)	text formatting macros
nl_types(5)	native language data types
prof(5)	profile within a function
regex(5)	internationalized basic and extended regular expression matching
regexp(5)	simple regular expression compile and match routines

siginfo(5)	signal generation information
signal(5)	base signals
standards(5)	standards and specifications supported by Solaris
stat(5)	data returned by stat system call
stdarg(5)	handle variable argument list
step(5)	See **regexp**(5)
sticky(5)	mark files for special treatment
term(5)	conventional names for terminals
types(5)	primitive system data types
ucontext(5)	user context
values(5)	machine-dependent values
varargs(5)	handle variable argument list
vgrindefs(5)	vgrind's language definition data base
wstat(5)	wait status
xpg4(5)	See **standards**(5)

NAME

ascii – map of ASCII character set

SYNOPSIS

cat /usr/pub/ascii

DESCRIPTION

/usr/pub/ascii is a map of the ASCII character set, to be printed as needed. It contains octal and hexadecimal values for each character. While not included in that file, a chart of decimal values is also shown here.

Octal — Character

000 NUL	001 SOH	002 STX	003 ETX	004 EOT	005 ENQ	006 ACK	007 BEL
010 BS	011 HT	012 NL	013 VT	014 NP	015 CR	016 SO	017 SI
020 DLE	021 DC1	022 DC2	023 DC3	024 DC4	025 NAK	026 SYN	027 ETB
030 CAN	031 EM	032 SUB	033 ESC	034 FS	035 GS	036 RS	037 US
040 SP	041 !	042 "	043 #	044 $	045 %	046 &	047 ´
050 (	051)	052 *	053 +	054 ,	055 –	056 .	057 /
060 0	061 1	062 2	063 3	064 4	065 5	066 6	067 7
070 8	071 9	072 :	073 ;	074 <	075 =	076 >	077 ?
100 @	101 A	102 B	103 C	104 D	105 E	106 F	107 G
110 H	111 I	112 J	113 K	114 L	115 M	116 N	117 O
120 P	121 Q	122 R	123 S	124 T	125 U	126 V	127 W
130 X	131 Y	132 Z	133 [	134 \	135]	136 ^	137 _
140 `	141 a	142 b	143 c	144 d	145 e	146 f	147 g
150 h	151 i	152 j	153 k	154 l	155 m	156 n	157 o
160 p	161 q	162 r	163 s	164 t	165 u	166 v	167 w
170 x	171 y	172 z	173 {	174 |	175 }	176 ~	177 DEL

Hexadecimal — Character

00 NUL	01 SOH	02 STX	03 ETX	04 EOT	05 ENQ	06 ACK	07 BEL
08 BS	09 HT	0A NL	0B VT	0C NP	0D CR	0E SO	0F SI
10 DLE	11 DC1	12 DC2	13 DC3	14 DC4	15 NAK	16 SYN	17 ETB
18 CAN	19 EM	1A SUB	1B ESC	1C FS	1D GS	1E RS	1F US
20 SP	21 !	22 "	23 #	24 $	25 %	26 &	27 ´
28 (	29)	2A *	2B +	2C ,	2D –	2E .	2F /
30 0	31 1	32 2	33 3	34 4	35 5	36 6	37 7
38 8	39 9	3A :	3B ;	3C <	3D =	3E >	3F ?
40 @	41 A	42 B	43 C	44 D	45 E	46 F	47 G
48 H	49 I	4A J	4B K	4C L	4D M	4E N	4F O
50 P	51 Q	52 R	53 S	54 T	55 U	56 V	57 W
58 X	59 Y	5A Z	5B [	5C \	5D]	5E ^	5F _
60 `	61 a	62 b	63 c	64 d	65 e	66 f	67 g
68 h	69 i	6A j	6B k	6C l	6D m	6E n	6F o
70 p	71 q	72 r	73 s	74 t	75 u	76 v	77 w
78 x	79 y	7A z	7B {	7C |	7D }	7E ~	7F DEL

Decimal — Character

0 NUL	1 SOH	2 STX	3 ETX	4 EOT	5 ENQ	6 ACK	7 BEL
8 BS	9 HT	10 NL	11 VT	12 NP	13 CR	14 SO	15 SI
16 DLE	17 DC1	18 DC2	19 DC3	20 DC4	21 NAK	22 SYN	23 ETB
24 CAN	25 EM	26 SUB	27 ESC	28 FS	29 GS	30 RS	31 US
32 SP	33 !	34 "	35 #	36 $	37 %	38 &	39 ´
40 (	41)	42 *	43 +	44 ,	45 –	46 .	47 /
48 0	49 1	50 2	51 3	52 4	53 5	54 6	55 7
56 8	57 9	58 :	59 ;	60 <	61 =	62 >	63 ?
64 @	65 A	66 B	67 C	68 D	69 E	70 F	71 G
72 H	73 I	74 J	75 K	76 L	77 M	78 N	79 O
80 P	81 Q	82 R	83 S	84 T	85 U	86 V	87 W
88 X	89 Y	90 Z	91 [	92 \	93]	94 ^	95 _
96 `	97 a	98 b	99 c	100 d	101 e	102 f	103 g
104 h	105 i	106 j	107 k	108 l	109 m	110 n	111 o
112 p	113 q	114 r	115 s	116 t	117 u	118 v	119 w
120 x	121 y	122 z	123 {	124		125 }	126 ~	127 DEL

FILES

/usr/pub/ascii On-line chart of octal and hexadecimal values for the ASCII character set.

NAME

charmap – character set description file

DESCRIPTION

A character set description file or *charmap* defines characteristics for a coded character set. Other information about the coded character set may also be in the file. Coded character set character values are defined using symbolic character names followed by character encoding values.

The character set description file provides:

- The capability to describe character set attributes (such as collation order or character classes) independent of character set encoding, and using only the characters in the portable character set. This makes it possible to create generic **localedef**(1) source files for all codesets that share the portable character set.
- Standardized symbolic names for all characters in the portable character set, making it possible to refer to any such character regardless of encoding.

Symbolic Names

Each symbolic name is included in the file and is mapped to a unique encoding value (except for those symbolic names that are shown with identical glyphs). If the control characters commonly associated with the symbolic names in the following table are supported by the implementation, the symbolic names and their corresponding encoding values are included in the file. Some of the encodings associated with the symbolic names in this table may be the same as characters in the portable character set table.

<ACK>	<DC2>	<ENQ>	<FS>	<IS4>	<SOH>
<BEL>	<DC3>	<EOT>	<GS>	<LF>	<STX>
<BS>	<DC4>	<ESC>	<HT>	<NAK>	<SUB>
<CAN>	<DEL>	<ETB>	<IS1>	<RS>	<SYN>
<CR>	<DLE>	<ETX>	<IS2>	<SI>	<US>
<DC1>	<EM>	<FF>	<IS3>	<SO>	<VT>

Declarations

The following declarations can precede the character definitions. Each must consist of the symbol shown in the following list, starting in column 1, including the surrounding brackets, followed by one or more blank characters, followed by the value to be assigned to the symbol.

<code_set_name> The name of the coded character set for which the character set description file is defined.

<mb_cur_max> The maximum number of bytes in a multi-byte character. This defaults to 1.

<mb_cur_min> An unsigned positive integer value that defines the minimum number of bytes in a character for the encoded character set.

<escape_char> The escape character used to indicate that the characters following will be interpreted in a special way, as defined later in this section. This defaults to backslash (\ ), which is the character glyph used

in all the following text and examples, unless otherwise noted.

<comment_char> The character that when placed in column 1 of a charmap line, is used to indicate that the line is to be ignored. The default character is the number sign (#).

Format

The character set mapping definitions will be all the lines immediately following an identifier line containing the string **CHARMAP** starting in column 1, and preceding a trailer line containing the string **END CHARMAP** starting in column 1. Empty lines and lines containing a *<comment_char>* in the first column will be ignored. Each non-comment line of the character set mapping definition (that is, between the **CHARMAP** and **END CHARMAP** lines of the file) must be in either of two forms:

"**%s %s %s\n",***<symbolic-name>*,*<encoding>*,*<comments>*

or

"**%s...%s %s %s\n",***<symbolic-name>*,*<symbolic-name>*, *<encoding>*,*<comments>*

In the first format, the line in the character set mapping definition defines a single symbolic name and a corresponding encoding. A character following an escape character is interpreted as itself; for example, the sequence <**\\\>**> represents the symbolic name \> enclosed between angle brackets.

In the second format, the line in the character set mapping definition defines a range of one or more symbolic names. In this form, the symbolic names must consist of zero or more non-numeric characters, followed by an integer formed by one or more decimal digits. The characters preceding the integer must be identical in the two symbolic names, and the integer formed by the digits in the second symbolic name must be equal to or greater than the integer formed by the digits in the first name. This is interpreted as a series of symbolic names formed from the common part and each of the integers between the first and the second integer, inclusive. As an example, **<j0101>...<j0104>** is interpreted as the symbolic names **<j0101>**, **<j0102>**, **<j0103>**, and **<j0104>**, in that order.

A character set mapping definition line must exist for all symbolic names and must define the coded character value that corresponds to the character glyph indicated in the table, or the coded character value that corresponds with the control character symbolic name. If the control characters commonly associated with the symbolic names are supported by the implementation, the symbolic name and the corresponding encoding value must be included in the file. Additional unique symbolic names may be included. A coded character value can be represented by more than one symbolic name.

The encoding part is expressed as one (for single-byte character values) or more concatenated decimal, octal or hexadecimal constants in the following formats:

"**%cd%d",***<escape_char>*,*<decimal byte value>*
"**%cx%x",***<escape_char>*,*<hexadecimal byte value>*
"**%c%o",***<escape_char>*,*<octal byte value>*

Decimal Constants

Decimal constants must be represented by two or three decimal digits, preceded by the escape character and the lower-case letter **d**; for example, **\d05**, **\d97**, or **\d143**. Hexadecimal constants must be represented by two hexadecimal digits, preceded by the escape

character and the lower-case letter **x**; for example, **\x05**, **\x61**, or **\x8f**. Octal constants must be represented by two or three octal digits, preceded by the escape character; for example, **\05**, **\141**, or **\217**. In a portable charmap file, each constant must represent an 8-bit byte. Implementations supporting other byte sizes may allow constants to represent values larger than those that can be represented in 8-bit bytes, and to allow additional digits in constants. When constants are concatenated for multi-byte character values, they must be of the same type, and interpreted in byte order from first to last with the least significant byte of the multi-byte character specified by the last constant.

Ranges of Symbolic Names

In lines defining ranges of symbolic names, the encoded value is the value for the first symbolic name in the range (the symbolic name preceding the ellipsis). Subsequent symbolic names defined by the range will have encoding values in increasing order. For example, the line

<j0101>...<j0104> **\d129\d254**

will be interpreted as:

<j0101> **\d129\d254**
<j0102> **\d129\d255**
<j0103> **\d130\d0**
<j0104> **\d130\d1**

Note that this line will be interpreted as the example even on systems with bytes larger than 8 bits. The comment is optional.

SEE ALSO

locale(1)**, localedef**(1)**, nl_langinfo**(3C)**, locale**(5)

NAME

environ – user environment

DESCRIPTION

When a process begins execution, **exec** routines make available an array of strings called the environment; see **exec**(2). By convention, these strings have the form *variable=value*, for example, **PATH=/sbin:/usr/sbin**. These environmental variables provide a way to make information about a program's environment available to programs.

A name may be placed in the environment by the **export** command and *name=value* arguments in **sh**(1), or by **exec**(2). It is unwise to conflict with certain shell variables that are frequently exported by **.profile** files: **MAIL**, **PS1**, **PS2**, **IFS**; see **profile**(4).

The following environmental variables can be used by applications and are expected to be set in the target run-time environment.

HOME The name of the user's login directory, set by **login**(1) from the password file; see **passwd**(4).

LANG The string used to specify internationalization information that allows users to work with different national conventions. The **setlocale**(3C) function checks the **LANG** environment variable when it is called with "" as the *locale* argument. **LANG** is used as the default locale if the corresponding environment variable for a particular category is unset or null. If, however, **LC_ALL** is set to a valid, non-empty value, its contents are used to override both the **LANG** and the other **LC_**∗ variables.

For example, when **setlocale**() is invoked as

setlocale(LC_CTYPE, ""),

setlocale() will query the **LC_CTYPE** environment variable first to see if it is set and non-null. If **LC_CTYPE** is not set or null, then **setlocale**() will check the **LANG** environment variable to see if it is set and non-null. If both **LANG** and **LC_CTYPE** are unset or **NULL**, the default "C" locale will be used to set the **LC_CTYPE** category.

Most commands will invoke

setlocale(LC_ALL, "")

prior to any other processing. This allows the command to be used with different national conventions by setting the appropriate environment variables.

The following environment variables correspond to each category of **setlocale**(3C):

LC_ALL If set to a valid, non-empty string value, override the values of **LANG** and all the other **LC_**∗ variables.

LC_COLLATE This category specifies the character collation sequence being used. The information corresponding to this category is stored in a database created by the **colltbl**(1M) command. This environment variable affects **strcoll**(3C) and **strxfrm**(3C).

LC_CTYPE This category specifies character classification, character conversion, and widths of multibyte characters. When **LC_CTYPE** is set to a valid value, the calling utility can display and handle text and file names containing valid characters for that locale; Extended Unix Code (EUC) characters where any individual character can be 1, 2, or 3 bytes wide; and EUC characters of 1, 2, or 3 column widths. The default "C" locale corresponds to the 7-bit ASCII character set; only characters from ISO 8859-1 are valid. The information corresponding to this category is stored in a database created by the **chrtbl**(1M) command. This environment variable is used by **ctype**(3C), **mbchar**(3C), and many commands, such as **cat**(1), **ed**(1), **ls**(1), and **vi**(1).

LC_MESSAGES This category specifies the language of the message database being used. For example, an application may have one message database with French messages, and another database with German messages. Message databases are created by the **mkmsgs**(1) command. This environment variable is used by **exstr**(1), **gettxt**(1), **srchtxt**(1), **gettxt**(3C), and **gettext**(3I).

LC_MONETARY This category specifies the monetary symbols and delimiters used for a particular locale. The information corresponding to this category is stored in a database created by the **montbl**(1M) command. This environment variable is used by **localeconv**(3C).

LC_NUMERIC This category specifies the decimal and thousands delimiters. The information corresponding to this category is stored in a database created by the **chrtbl**(1M) command. The default **C** locale corresponds to "**.**" as the decimal delimiter and no thousands delimiter. This environment variable is used by **localeconv**(3C), **printf**(3S), and **strtod**(3C).

LC_TIME This category specifies date and time formats. The information corresponding to this category is stored in a database specified in **strftime**(4). The default **C** locale corresponds to U.S. date and time formats. This environment variable is used by many commands and functions; for example: **at**(1), **calendar**(1), **date**(1), **strftime**(3C), and **getdate**(3C).

MSGVERB Controls which standard format message components **fmtmsg** selects when messages are displayed to **stderr**; see **fmtmsg**(1) and **fmtmsg**(3C).

NETPATH A colon-separated list of network identifiers. A network identifier is a character string used by the Network Selection component of the system to

provide application-specific default network search paths. A network identifier must consist of non-**NULL** characters and must have a length of at least 1. No maximum length is specified. Network identifiers are normally chosen by the system administrator. A network identifier is also the first field in any **/etc/netconfig** file entry. **NETPATH** thus provides a link into the **/etc/netconfig** file and the information about a network contained in that network's entry. **/etc/netconfig** is maintained by the system administrator. The library routines described in **getnetpath**(3N) access the **NETPATH** environment variable.

NLSPATH Contains a sequence of templates which **catopen**(3C) and **gettext**(3I) use when attempting to locate message catalogs. Each template consists of an optional prefix, one or more substitution fields, a filename and an optional suffix.

For example:

NLSPATH="/system/nlslib/%N.cat"

defines that **catopen**( ) should look for all message catalogs in the directory **/system/nlslib**, where the catalog name should be constructed from the *name* parameter passed to **catopen**( ), **%N**, with the suffix **.cat**.

Substitution fields consist of a **%** symbol, followed by a single-letter keyword. The following keywords are currently defined:

%N	The value of the *name* parameter passed to **catopen**( ).
%L	The value of **LANG** or **LC_MESSAGES**.
%l	The language element from **LANG** or **LC_MESSAGES**.
%t	The territory element from **LANG** or **LC_MESSAGES**.
%c	The codeset element from **LANG** or **LC_MESSAGES**.
%%	A single **%** character.

An empty string is substituted if the specified value is not currently defined. The separators ''**_**'' and ''**.**'' are not included in **%t** and **%c** substitutions.

Templates defined in **NLSPATH** are separated by colons (**:**). A leading colon or two adjacent colons (**::**) is equivalent to specifying **%N**.

For example:

NLSPATH=":%N.cat:/nlslib/%L/%N.cat"

indicates to **catopen**( ) that it should look for the requested message catalog in *name*, *name***.cat** and **/nlslib/$LANG/***name***.cat**. For **gettext( )**, **%N** automatically maps to "messages".

If **NLSPATH** is unset or **NULL**, **catopen( )** and **gettext( )** call **setlocale**(3C), which checks **LANG** and the **LC_***∗ variables to locate the message catalogs.

NLSPATH will normally be set up on a system wide basis (in **/etc/profile**) and thus makes the location and naming conventions associated with message catalogs transparent to both programs and users.

PATH The sequence of directory prefixes that **sh**(1), **time**(1), **nice**(1), **nohup**(1), and other utilities apply in searching for a file known by an incomplete path name. The prefixes are separated by colons (**:**). **login**(1) sets **PATH=/usr/bin**. For more detail, see **sh**(1).

SEV_LEVEL Define severity levels and associate and print strings with them in standard format error messages; see **addseverity**(3C), **fmtmsg**(1), and **fmtmsg**(3C).

TERM The kind of terminal for which output is to be prepared. This information is used by commands, such as **vi**(1), which may exploit special capabilities of that terminal.

TZ Timezone information. The contents of this environment variable are used by the functions **ctime**(3C), **localtime**(3C), **strftime**(3C), and **mktime**(3C) to override the default timezone. If **TZ** is not in the following form, it designates a path to a timezone database file relative to **/usr/share/lib/zoneinfo/**, ignoring the first character if it is a colon (**:**); otherwise, **TZ** has the form:

std offset **[** *dst* **[** *offset* **],[** *start* **[** */time* **],** *end* **[** */time* **]]]**

std and *dst* Three or more bytes that are the designation for the standard (*std*) and daylight savings time (*dst*) timezones. Only *std* is required. If *dst* is missing, then daylight savings time does not apply in this locale. Upper- and lower-case letters are allowed. Any characters except a leading colon (**:**), digits, a comma (**,**), a minus (–) or a plus (**+**) are allowed.

offset Indicates the value one must add to the local time to arrive at Coordinated Universal Time. The offset has the form:

hh **[:** *mm* **[:** *ss* **]]**

The minutes (*mm*) and seconds (*ss*) are optional. The hour (*hh*) is required and may be a single digit. The *offset* following *std* is required. If no *offset* follows *dst* , daylight savings time is assumed to be one hour ahead of standard time. One or more digits may be used; the value is always interpreted as a decimal number. The hour must be between 0 and 24, and the minutes (and seconds) if present between 0 and 59. Out of range values may cause unpredictable behavior. If preceded by a ‘‘–’’, the timezone is east of the Prime Meridian; otherwise it is west (which may be indicated by an optional preceding ‘‘+’’ sign).

start / *time*, *end* / *time*
Indicate when to change to and back from daylight savings time, where *start/time* describes when the change from standard time to daylight savings time occurs, and *end/time* describes when the change back happens. Each *time* field describes when, in current local time, the change is made.

The formats of *start* and *end* are one of the following:

J*n* The Julian day *n* ($1 \le n \le 365$). Leap days are not counted. That is, in all years, February 28 is day 59 and March 1 is day 60. It is impossible to refer to the occasional February 29.

n The zero-based Julian day ($0 \le n \le 365$). Leap days are counted, and it is possible to refer to February 29.

M*m.n.d* The d^{th} day, ($0 \le d \le 6$) of week *n* of month *m* of the year ($1 \le n \le 5$, $1 \le m \le 12$), where week 5 means ''the last *d*-day in month *m*'' which may occur in either the fourth or the fifth week). Week 1 is the first week in which the d^{th} day occurs. Day zero is Sunday.

Implementation specific defaults are used for *start* and *end* if these optional fields are not given.

The *time* has the same format as *offset* except that no leading sign (''–'' or ''+'') is allowed. The default, if *time* is not given is 02:00:00.

SEE ALSO

cat(1), **date**(1), **ed**(1), **fmtmsg**(1), **login**(1), **ls**(1), **mkmsgs**(1), **nice**(1), **nohup**(1), **sh**(1), **sort**(1), **time**(1), **vi**(1), **chrtbl**(1M), **colltbl**(1M), **montbl**(1M), **exec**(2), **addseverity**(3C), **catopen**(3C), **ctime**(3C), **ctype**(3C), **fmtmsg**(3C), **getdate**(3C), **getnetpath**(3N), **gettext**(3I), **gettxt**(3C), **localeconv**(3C), **mbchar**(3C), **mktime**(3C), **printf**(3S), **setlocale**(3C), **strcoll**(3C), **strftime**(3C), **strtod**(3C), **strxfrm**(3C), **netconfig**(4), **passwd**(4), **profile**(4), **strftime**(4), **TIMEZONE**(4)

NAME

eqnchar – special character definitions for eqn

SYNOPSIS

eqn /usr/share/lib/pub/eqnchar [*filename*] | **troff** [*options*]

neqn /usr/share/lib/pub/eqnchar [*filename*] | **nroff** [*options*]

DESCRIPTION

The **eqnchar** command contains **troff**(1) and **nroff**(1) character definitions for constructing characters that are not available on the Graphic Systems typesetter. These definitions are primarily intended for use with **eqn**(1) and **neqn**. It contains definitions for the following characters:

ciplus	⊕	*||*	‖	*square*	□
citimes	⊗	*langle*	⟨	*circle*	○
wig	~	*rangle*	⟩	*blot*	□
-wig	≈	*hbar*	ℏ	*bullet*	•
>wig	≳	*ppd*	⊥	*prop*	∝
<wig	≲	*<->*	↔	*empty*	∅
=wig	≅	*<=>*	⇔	*member*	∈
star	✻	*|<*	≮	*nomem*	∉
bigstar	✳	*|>*	≯	*cup*	∪
=dot	≐	*ang*	∠	*cap*	∩
orsign	∨	*rang*	⊦	*incl*	⊂⊃
andsign	∧	*3dot*	⋮	*subset*	⊂
=del	≜	*thf*	∴	*supset*	⊃
oppA	∀	*quarter*	¼	*!subset*	⊆
oppE	∃	*3quarter*	¾	*!supset*	⊇
angstrom	Å	*degree*	°		

FILES

/usr/share/lib/pub/eqnchar

SEE ALSO

eqn(1), **nroff**(1), **troff**(1)

NAME

fcntl – file control options

SYNOPSIS

#include <fcntl.h>

DESCRIPTION

The **<fcntl.h>** header defines the following requests and arguments for use by the functions **fcntl**(2) and **open**(2).

Values for *cmd* used by **fcntl** (the following values are unique):

F_DUPFD	Duplicate file descriptor
F_GETFD	Get file descriptor flags
F_SETFD	Set file descriptor flags
F_GETFL	Get file status flags
F_SETFL	Set file status flags
F_GETLK	Get record locking information
F_SETLK	Set record locking information
F_SETLKW	Set record locking information; wait if blocked

File descriptor flags used for **fcntl**:

FD_CLOEXEC	Close the file descriptor upon execution of an exec function (see **exec**(2))

Values for **l_type** used for record locking with **fcntl** (the following values are unique):

F_RDLCK	Shared or read lock
F_UNLCK	Unlock
F_WRLCK	Exclusive or write lock

The following three sets of values are bitwise distinct: Values for **oflag** used by **open**:

O_CREAT	Create file if it does not exist
O_EXCL	Exclusive use flag
O_NOCTTY	Do not assign controlling tty
O_TRUNC	Truncate flag

File status flags used for **open** and **fcntl**:

O_APPEND	Set append mode
O_NDELAY	Non-blocking mode
O_NONBLOCK	Non-blocking mode (POSIX)
O_DSYNC	Write I/O operations on the file descriptor complete as defined by synchronized I/O data integrity completion
O_RSYNC	Read I/O operations on the file descriptor complete at the same level of integrity as specified by the the **O_DSYNC** and **O_SYNC** flags. If both **O_DSYNC** and **O_RSYNC** are set in *oflag*, all I/O operations on the file descriptor complete as defined by synchronized I/O data integrity completion. If both **O_SYNC** and **O_RSYNC** are set in *oflag*, all I/O operations on the file descriptor complete as defined by synchronized I/O file

integrity completion.

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. Write I/O operations on the file descriptor complete as defined by synchronized I/O file integrity completion.

Mask for use with file access modes:

O_ACCMODE Mask for file access modes

File access modes used for **open** and **fcntl**:

O_RDONLY Open for reading only
O_RDWR Open for reading and writing
O_WRONLY Open for writing only

The structure **flock** describes a file lock. It includes the following members:

```
short   l_type;     /* Type of lock */
short   l_whence;   /* Flag for starting offset */
off_t   l_start;    /* Relative offset in bytes */
off_t   l_len;      /* Size; if 0 then until EOF */
long    l_sysid;    /* Returned with F_GETLK */
pid_t   l_pid;      /* Returned with F_GETLK */
```

SEE ALSO

creat(2), **exec**(2), **fcntl**(2), **open**(2), **fsync**(3C), **fdatasync**(3R)

NOTES

Data is successfully transferred for a write operation to a regular file when the system ensures that all data written is readable on any subsequent open of the file (even one that follows a system or power failure) in the absense of a failure of the physical storage medium.

Data is successfully transferred for a read operation when an image of the data on the physical storage medium is available to the requesting process.

Synchronized I/O data integrity completion (see **fdatasync**(3R)):

For reads, the operation has been completed or diagnosed if unsuccessful. The read is complete only when an image of the data has been successfully transferred to the requesting process. If there were any pending write requests affecting the data to be read at the time that the synchronized read operation was requested, these write requests will be successfully transferred prior to reading the data.

For writes, the operation has been completed or diagnosed if unsuccessful. The write is complete only when the data specified in the write request is successfully transferred, and all file system information required to retrieve the data is successfully transferred.

File attributes that are not necessary for data retrieval (access time, modification time, status change time) need not be successfully transferred prior to returning to the calling process.

Synchronized I / O file integrity completion (see **fsync**(3C)):

Identical to a synchronized I / O data integrity completion with the addition that all file attributes relative tothe I / O operation (including access time, modification time, status change time) will be successfully transferred prior to returning to the calling process.

NAME

filesystem – file system organization

SYNOPSIS

/
/usr
/export

DESCRIPTION

The file system tree is organized for administrative convenience. Distinct areas within the file system tree are provided for files that are private to one machine, files that can be shared by multiple machines of a common architecture, files that can be shared by all machines, and home directories. This organization allows sharable files to be stored on one machine but accessed by many machines using a remote file access mechanism such as NFS. Grouping together similar files makes the file system tree easier to upgrade and manage.

The file system tree consists of a root file system and a collection of mountable file systems. The **mount**(2) program attaches mountable file systems to the file system tree at mount points (directory entries) in the root file system or other previously mounted file systems. Two file systems, **/** (the root) and **/usr**, must be mounted in order to have a completely functional system. The root file system is mounted automatically by the kernel at boot time; the **/usr** file system is mounted by the system start-up script, which is run as part of the booting process.

Root File System

The root file system contains files that are unique to each machine. It contains the following directories:

/dev	Primary location for special files. Typically, device files are built to match the kernel and hardware configuration of the machine.
/dev/dsk	Block disk devices.
/dev/pts	Pseudo-terminal devices.
/dev/rdsk	Raw disk devices.
/dev/rmt	Raw tape devices.
/dev/sad	Entry points for the STREAMS Administrative driver.
/dev/term	Terminal devices.
/etc	Host-specific administrative configuration files and databases. **/etc** may be viewed as the directory that defines the machine's identity.
/etc/acct	Accounting system configuration information.
/etc/cron.d	Configuration information for **cron**(1M).
/etc/default	Defaults information for various programs.
/etc/dfs	Configuration information for exported file systems.

/etc/fs	Binaries organized by file system types for operations required before **/usr** is mounted.
/etc/inet	Configuration files for Internet services.
/etc/init.d	Shell scripts for transitioning between run levels.
/etc/lib	Shared libraries needed during booting.
/etc/lp	Configuration information for the printer subsystem.
/etc/mail	Mail subsystem configuration.
/etc/net	Configuration information for transport independent network services.
/etc/opt	Configuration information for optional packages.
/etc/rc0.d	Scripts for entering or leaving run level 0. See **init**(1M).
/etc/rc1.d	Scripts for entering or leaving run level 1. See **init**(1M).
/etc/rc2.d	Scripts for entering or leaving run level 2. See **init**(1M).
/etc/rc3.d	Scripts for entering or leaving run level 3. See **init**(1M).
/etc/saf	Service Access Facility files.
/etc/skel	Default profile scripts for new user accounts. See **useradd**(1M).
/etc/sm	Status monitor information.
/etc/sm.bak	Backup status monitor information.
/etc/tm	Trademark files; contents displayed at boot time.
/etc/uucp	UUCP configuration information. See **uucp**(1C).
/export	Default root of the exported file system tree.
/home	Default root of a subtree for user directories.
/kernel	Subtree of Platform Independent loadable kernel modules required as part of the boot process. It includes the generic part of the core kernel that is platform-independent, **/kernel/genunix.** See **kernel**(1M).
/mnt	Default temporary mount point for file systems. This is an empty directory on which file systems may be temporarily mounted.
/opt	Root of a subtree for add-on application packages.
/platform	Subtree of Platform Specific objects which need to reside on the root filesystem. It contains a series of directories, one per supported platform. The semantics of the series of directories is equivalent to / (root).
/platform/*/kernel	Platform Dependent objects with semantics equivalent to /kernel. It includes the file **unix** ,the core kernel that is platform dependent. See **kernel**(1M).

/platform/*/lib	Platform Dependent objects with semantics equivalent to /lib.
/platform/*/sbin	Platform Dependent objects with semantics equivalent to /sbin.
/proc	Root of a subtree for the process file system.
/sbin	Essential executables used in the booting process and in manual system recovery. The full complement of utilities is available only after **/usr** is mounted.
/tmp	Temporary files; cleared during the boot operation.
/var	Root of a subtree for varying files. Varying files are files that are unique to a machine but that can grow to an arbitrary (that is, variable) size. An example is a log file.
/var/adm	System logging and accounting files.
/var/cron	Log files for **cron**(1M).
/var/mail	Directory where users' mail is kept.
/var/news	Community service messages. Note: this is not the same as USENET-style news.
/var/nis	NIS+ databases.
/var/opt	Root of a subtree for varying files associated with optional software packages.
/var/preserve	Backup files for **vi**(1) and **ex**(1).
/var/sadm	Databases maintained by the software package management utilities.
/var/saf	Service access facility logging and accounting files.
/var/spool	Root directory for files used in printer spooling, mail delivery, **cron**(1M), **at**(1), etc.
/var/spool/cron	**cron**(1M) and **at**(1) spooling files.
/var/spool/locks	Spooling lock files.
/var/spool/lp	Line printer spool files. See **lp**(1).
/var/spool/mqueue	Mail queued for delivery.
/var/spool/pkg	Spooled packages.
/var/spool/uucp	Queued **uucp**(1C) jobs.
/var/spool/uucppublic	Files deposited by **uucp**(1C).
/var/tmp	Transitory files; this directory is *not* cleared during the boot operation.
/var/uucp	**uucp**(1C) log and status files.
/var/yp	Databases needed for backwards compatibility with NIS and **ypbind**(1M); unnecessary after full transition to NIS+.

/usr File System

Because it is desirable to keep the root file system small and not volatile, on disk-based systems larger file systems are often mounted on **/home**, **/opt**, **/usr**, and **/var**.

The file system mounted on **/usr** contains architecture-dependent and architecture-independent sharable files. The subtree rooted at **/usr/share** contains architecture-independent sharable files; the rest of the **/usr** tree contains architecture-dependent files. By mounting a common remote file system, a group of machines with a common architecture may share a single **/usr** file system. A single **/usr/share** file system can be shared by machines of any architecture. A machine acting as a file server may export many different **/usr** file systems to support several different architectures and operating system releases. Clients usually mount **/usr** read-only so that they do not accidentally change any shared files.

The **/usr** file system contains the following subdirectories:

/usr/4lib	**a.out** libraries for the Binary Compatibility Package. See *Binary Compatibility Guide*.
/usr/bin	Primary location for standard system utilities.
/usr/bin/sunview1	SunView executables. This directory is only present when the Binary Compatibility Package is installed.
/usr/ccs	C compilation system.
/usr/ccs/bin	C compilation commands and system utilities.
/usr/ccs/lib	Libraries and auxiliary files.
/usr/demo	Demo programs and data.
/usr/dt	root of a subtree for CDE Motif.
/usr/dt/bin	Primary location for CDE Motif system utilities.
/usr/dt/include	Header files for CDE Motif.
/usr/dt/lib	Libraries for CDE Motif.
/usr/dt/man	On-line reference manual pages for CDE Motif.
/usr/games	Game binaries and data.
/usr/include	Include headers (for C programs, etc).
/usr/kernel	Subtree of Platform Independent loadable kernel modules, not needed in the root filesystem.
/usr/platform	Subtree of Platform Specific objects which does not need to reside on the root filesystem. It contains a series of directories, one per supported platform. The semantics of the series of directories is equivalent to /platform, except for subdirectories which don't provide utility under one or the other (for example: /platform/include isn't needed).
/platform/*/include	Platform Dependent headers with semantics equivalent to /usr/include.
/platform/*/kernel	Platform Dependent objects with semantics equivalent to

	/usr/kernel.
/platform/*/lib	Platform Dependent objects with semantics equivalent to /usr/lib.
/platform/*/sbin	Platform Dependent objects with semantics equivalent to /usr/sbin.
/usr/lib	Program libraries, various architecture-dependent databases, and executables not invoked directly by the user (system daemons, etc).
/usr/lib/acct	Accounting scripts and binaries. See **acct**(1M).
/usr/lib/dict	Database files for **spell**(1).
/usr/lib/class	Scheduling class-specific directories containing executables for **priocntl**(1) and **dispadmin**(1M).
/usr/lib/font	**troff**(1) font description files.
/usr/lib/fs	File system type dependent modules; generally not intended to be invoked directly by the user.
/usr/lib/iconv	Conversion tables for **iconv**(1).
/usr/lib/libp	Profiled libraries.
/usr/lib/locale	Localization databases.
/usr/lib/lp	Line printer subsystem databases and back-end executables.
/usr/lib/mail	Auxiliary programs for the **mail**(1) subsystem.
/usr/lib/netsvc	Internet network services.
/usr/lib/nfs	Auxiliary NFS-related programs and daemons.
/usr/lib/pics	Position Independent Code (PIC) archives needed to rebuild the run-time linker.
/usr/lib/refer	Auxiliary programs for **refer**(1).
/usr/lib/sa	Scripts and commands for the system activity report package. See **sar**(1).
/usr/lib/saf	Auxiliary programs and daemons related to the service access facility.
/usr/lib/spell	Auxiliary programs and databases for **spell**(1). This directory is only present when the Binary Compatibility Package is installed.
/usr/lib/uucp	Auxiliary programs and daemons for **uucp**(1C).
/usr/local	Commands local to a site.
/usr/net/servers	Entry points for foreign name service requests relayed using the network listener. See **listen**(1M).
/usr/oasys	Commands and files related to the optional Framed Access Command Environment (FACE) package. See **face**(1).

/usr/old	Programs that are being phased out.
/usr/openwin	Installation or mount point for the OpenWindows software.
/usr/sadm	System administration files and directories.
/usr/sadm/bin	Binaries for the Form and Menu Language Interpreter (FMLI) scripts. See **fmli**(1).
/usr/sadm/install	Executables and scripts for package management.
/usr/sbin	Executables for system administration.
/usr/sbin/static	Statically linked version of selected programs from **/usr/bin** and **/usr/sbin**. These are used to recover from broken dynamic linking and before all pieces necessary for dynamic linking are present.
/usr/share	Architecture-independent sharable files.
/usr/share/man	On-line reference manual pages (if present).
/usr/share/lib	Architecture-independent databases.
/usr/share/lib/keytables	Keyboard layout description tables.
/usr/share/lib/mailx	Help files for **mailx**(1).
/usr/share/lib/nterm	**nroff**(1) terminal tables.
/usr/share/lib/pub	Character set data files.
/usr/share/lib/spell	Auxiliary scripts and databases for **spell**(1).
/usr/share/lib/tabset	Tab setting escape sequences.
/usr/share/lib/terminfo	Terminal description files for **terminfo**(4).
/usr/share/lib/tmac	Macro packages and related files for text processing tools, for example, **nroff**(1) and **troff**(1).
/usr/share/lib/zoneinfo	Time zone information.
/usr/share/src	Source code for utilities and libraries.
/usr/snadm	SNAG files.
/usr/ucb	Berkeley compatibility package binaries. See *Source Compatibility Guide*.
/usr/ucbinclude	Berkeley compatibility package headers.
/usr/ucblib	Berkeley compatibility package libraries.
/usr/vmsys	Commands and files related to the optional FACE package. See **face**(1). Berkeley compatibility package libraries.

/export File System

A machine with disks may export root file systems, swap files, and **/usr** file systems to diskless or partially-disked machines that mount them into the standard file system hierarchy. The standard directory tree for sharing these file systems from a server is:

/export	The default root of the exported file system tree.

/export/exec/*architecture-name*

The exported **/usr** file system supporting *architecture-name* for the current release.

/export/exec/*architecture-name***.***release-name*
The exported **/usr** file system supporting *architecture-name* for *release-name*.

/export/exec/share The exported common **/usr/share** directory tree.

/export/exec/share.*release-name*
The exported common **/usr/share** directory tree for *release-name*.

/export/root/*hostname* The exported root file system for *hostname*.

/export/swap/*hostname* The exported swap file for *hostname*.

/export/var/*hostname* The exported **/var** directory tree for *hostname*.

SEE ALSO

at(1), **ex**(1), **face**(1), **fmli**(1), **iconv**(1), **lp**(1), **mail**(1), **mailx**(1), **nroff**(1), **priocntl**(1), **refer**(1), **sar**(1), **sh**(1), **spell**(1), **troff**(1), **uucp**(1C), **vi**(1), **acct**(1M), **cron**(1M), **dispadmin**(1M), **fsck**(1M), **init**(1M), **kernel**(1M), **mknod**(1M), **mount**(1M), **useradd**(1M), **ypbind**(1M), **mount**(2), **intro**(4), **terminfo**(4)

Binary Compatibility Guide
Source Compatibility Guide

NAME

floatingpoint – IEEE floating point definitions

SYNOPSIS

#include <floatingpoint.h>

DESCRIPTION

This file defines constants, types, and functions used to implement standard floating point according to ANSI/IEEE Std 754-1985. The functions are implemented in **libc**. The included header file **<sys/ieeefp.h>** defines certain types of interest to the kernel.

IEEE Rounding Modes:

fp_direction_type The type of the IEEE rounding direction mode. Note: the order of enumeration varies according to hardware.

fp_precision_type The type of the IEEE rounding precision mode, which only applies on systems that support extended precision such as machines based on the Intel 80387 FPU or the 80486.

SIGFPE handling:

sigfpe_code_type The type of a **SIGFPE** code.

sigfpe_handler_type The type of a user-definable **SIGFPE** exception handler called to handle a particular **SIGFPE** code.

SIGFPE_DEFAULT A macro indicating the default **SIGFPE** exception handling, namely to perform the exception handling specified by the user, if any, and otherwise to dump core using **abort**(3C).

SIGFPE_IGNORE A macro indicating an alternate **SIGFPE** exception handling, namely to ignore and continue execution.

SIGFPE_ABORT A macro indicating an alternate **SIGFPE** exception handling, namely to abort with a core dump.

IEEE Exception Handling:

N_IEEE_EXCEPTION The number of distinct IEEE floating-point exceptions.

fp_exception_type The type of the **N_IEEE_EXCEPTION** exceptions. Each exception is given a bit number.

fp_exception_field_type
The type intended to hold at least **N_IEEE_EXCEPTION** bits corresponding to the IEEE exceptions numbered by **fp_exception_type**. Thus **fp_inexact** corresponds to the least significant bit and **fp_invalid** to the fifth least significant bit. Note: some operations may set more than one exception.

IEEE Formats and Classification:

single; **extended**; **quadruple**
Definitions of IEEE formats.

fp_class_type An enumeration of the various classes of IEEE values and symbols.

IEEE Base Conversion:

The functions described under **floating_to_decimal**(3) and **decimal_to_floating**(3) satisfy not only the IEEE Standard, but also the stricter requirements of correct rounding for all arguments.

DECIMAL_STRING_LENGTH
: The length of a **decimal_string**.

decimal_string
: The digit buffer in a **decimal_record**.

decimal_record
: The canonical form for representing an unpacked decimal floating-point number.

decimal_form
: The type used to specify fixed or floating binary to decimal conversion.

decimal_mode
: A struct that contains specifications for conversion between binary and decimal.

decimal_string_form
: An enumeration of possible valid character strings representing floating-point numbers, infinities, or NaNs.

FILES

/usr/include/sys/ieeefp.h

SEE ALSO

abort(3C), **decimal_to_floating**(3), **econvert**(3), **floating_to_decimal**(3), **sigfpe**(3), **string_to_decimal**(3), **strtod**(3C)

NAME

fnmatch – file name pattern matching

DESCRIPTION

The pattern matching notation described below is used to specify patterns for matching strings in the shell. Historically, pattern matching notation is related to, but slightly different from, the regular expression notation. For this reason, the description of the rules for this pattern matching notation is based on the description of regular expression notation described on the **regex**(5) manual page.

Patterns Matching a Single Character

The following *patterns matching a single character* match a single character: *ordinary characters*, *special pattern characters* and *pattern bracket expressions*. The pattern bracket expression will also match a single collating element.

An ordinary character is a pattern that matches itself. It can be any character in the supported character set except for NUL, those special shell characters that require quoting, and the following three special pattern characters. Matching is based on the bit pattern used for encoding the character, not on the graphic representation of the character. If any character (ordinary, shell special, or pattern special) is quoted, that pattern will match the character itself. The shell special characters always require quoting.

When unquoted and outside a bracket expression, the following three characters will have special meaning in the specification of patterns:

? A question-mark is a pattern that will match any character.

* An asterisk is a pattern that will match multiple characters, as described in **Patterns Matching Multiple Characters**, below.

[The open bracket will introduce a pattern bracket expression.

The description of basic regular expression bracket expressions on the **regex**(5) manual page also applies to the pattern bracket expression, except that the exclamation-mark character **(!)** replaces the circumflex character (^) in its role in a *non-matching list* in the regular expression notation. A bracket expression starting with an unquoted circumflex character produces unspecified results.

The restriction on a circumflex in a bracket expression is to allow implementations that support pattern matching using the circumflex as the negation character in addition to the exclamation-mark. A portable application must use something like **[\^!]** to match either character.

When pattern matching is used where shell quote removal is not performed (such as in the argument to the **find** –**name** primary when **find** is being called using one of the **exec** functions, or in the *pattern* argument to the **fnmatch**(3C) function, special characters can be escaped to remove their special meaning by preceding them with a backslash character. This escaping backslash will be discarded. The sequence \ \ represents one literal backslash. All of the requirements and effects of quoting on ordinary, shell special and special pattern characters will apply to escaping in this context.

Both quoting and escaping are described here because pattern matching must work in three separate circumstances:

- Calling directly upon the shell, such as in pathname expansion or in a **case**

statement. All of the following will match the string or file **abc**:

abc "abc" a"b"c a\bc a[b]c
a["b"]c a[\b]c a["\b"]c a?c a*c

The following will not:

"a?c" a*c a\[b]c

- Calling a utility or function without going through a shell, as described for **find**(1) and the function **fnmatch**(3C)**.**
- Calling utilities such as **find**, **cpio**, **tar** or **pax** through the shell command line. In this case, shell quote removal is performed before the utility sees the argument. For example, in:

  find /bin -name e\c[\h]o -print

  after quote removal, the backslashes are presented to **find** and it treats them as escape characters. Both precede ordinary characters, so the **c** and **h** represent themselves and **echo** would be found on many historical systems (that have it in **/bin**). To find a file name that contained shell special characters or pattern characters, both quoting and escaping are required, such as:

  pax -r . . . "*a\ (\?"

  to extract a filename ending with **a(?**.

Conforming applications are required to quote or escape the shell special characters (sometimes called metacharacters). If used without this protection, syntax errors can result or implementation extensions can be triggered. For example, the KornShell supports a series of extensions based on parentheses in patterns; see **ksh**(1).

Patterns Matching Multiple Characters

The following rules are used to construct *patterns matching multiple characters* from *patterns matching a single character*:

- The asterisk (*) is a pattern that will match any string, including the null string.
- The concatenation of *patterns matching a single character* is a valid pattern that will match the concatenation of the single characters or collating elements matched by each of the concatenated patterns.
- The concatenation of one or more *patterns matching a single character* with one or more asterisks is a valid pattern. In such patterns, each asterisk will match a string of zero or more characters, matching the greatest possible number of characters that still allows the remainder of the pattern to match the string.

Since each asterisk matches zero or more occurrences, the patterns **a*b** and **a**b** have identical functionality.

Examples:

a[bc]	matches the strings **ab** and **ac**.
a*d	matches the strings **ad**, **abd** and **abcd**, but not the string **abc**.
a*d*	matches the strings **ad**, **abcd**, **abcdef**, **aaaad** and **adddd**.
***a*d**	matches the strings **ad**, **abcd**, **efabcd**, **aaaad** and **adddd**.

Patterns Used for Filename Expansion

The rules described so far in **Patterns Matching Multiple Characters** and **Patterns Matching a Single Character** are qualified by the following rules that apply when pattern matching notation is used for filename expansion.

1. The slash character in a pathname must be explicitly matched by using one or more slashes in the pattern; it cannot be matched by the asterisk or question-mark special characters or by a bracket expression. Slashes in the pattern are identified before bracket expressions; thus, a slash cannot be included in a pattern bracket expression used for filename expansion. For example, the pattern **a[b/c]d** will not match such pathnames as **abd** or **a/d**. It will only match a pathname of literally **a[b/c]d**.
2. If a filename begins with a period (.), the period must be explicitly matched by using a period as the first character of the pattern or immediately following a slash character. The leading period will not be matched by:
   - the asterisk or question-mark special characters
   - a bracket expression containing a non-matching list, such as :

     [!a]

     a range expression, such as:

     [%–0]

     or a character class expression, such as:

     [[:punct:]]

   It is unspecified whether an explicit period in a bracket expression matching list, such as:

   [.abc]

   can match a leading period in a filename.
3. Specified patterns are matched against existing filenames and pathnames, as appropriate. Each component that contains a pattern character requires read permission in the directory containing that component. Any component, except the last, that does not contain a pattern character requires search permission. For example, given the pattern:

   /foo/bar/x*/bam

   search permission is needed for directories **/** and **foo**, search and read permissions are needed for directory **bar**, and search permission is needed for each **x*** directory. If the pattern matches any existing filenames or pathnames, the pattern will be replaced with those filenames and pathnames, sorted according to the collating sequence in effect in the current locale. If the pattern contains an invalid bracket expression or does not match any existing filenames or pathnames, the pattern string is left unchanged.

SEE ALSO

find(1), **ksh**(1), **fnmatch**(3C), **regex**(5)

NAME

fns – overview of FNS

DESCRIPTION

Federated Naming Service (FNS) provides a method for federating multiple naming services under a single, simple interface for the basic naming operations. The service supports resolution of *composite* names — names that spans multiple naming systems — through the naming interface. In addition to the naming interface, FNS also specifies *policies* for composing names in the enterprise namespace. See **fns_policies**(5) and **fns_initial_context**(5).

Fundamental to the FNS model are the notions of composite names and *contexts.* A context provides operations for:

- associating (binding) names to objects
- resolving names to objects
- removing bindings, listing names, renaming and so on.

A context contains a set of names to reference bindings. A reference contains a list of communication end-points. Every naming operation in the FNS interface is performed on a context object.

The FNS is formed by contexts from one naming system being bound in the contexts of another naming system. Resolution of composite name proceeds from contexts within one naming system to those in the next until the name is resolved.

XFN

XFN is *X/Open Federated Naming* . The programming interface and policies that FNS supports are specified by XFN. See **xfn**(3N) and **fns_policies**(5).

Composite Names

A composite name is a name that spans multiple naming systems. It consists of an ordered list of components. Each component is a name from the namespace of a single naming system. FNS defines the syntax for constructing a composite name using names from component naming systems. Individual naming systems are responsible for the syntax of each component.

The syntax for composite names is that components are composed left to right using the slash character ('/') as the component separator. For example, the composite name **.../Wiz.Com/site/Oceanview.East** consists of four components: **...**, **Wiz.COM**, **site**, and **Oceanview.East**. See **fns_policies**(5) and **fns_initial_context**(5) for more examples of composite names.

Why FNS?

FNS is useful for the following reasons:

- A single uniform naming interface is provided to clients for accessing naming services. Consequently, the addition of new naming services does not require changes to applications or existing naming services. Furthermore, applications that use FNS will be portable across platforms because the interface exported by FNS is XFN, a public, open interface endorsed by other vendors and X/Open.
- Names can be composed in a uniform way (i.e. it supports a model in which composite names are constructed in a uniform syntactic way and can have any number of

components).

- Coherent naming is encouraged through the use of shared contexts and shared names.

FNS and NIS+

In Solaris, the FNS implementation consists of a set of enterprise-level naming services implemented on top of NIS+, the enterprise-wide information service in Solaris. FNS provides the FNS interface for performing naming and attribute operations on FNS enterprise objects (organization, site, user, host and service objects). FNS stores bindings for these objects in NIS+ and uses them in conjunction with existing NIS+ objects.

The command **fncreate** creates NIS+ tables and directories in the NIS+ hierarchy associated with the domain of the host it executes on. The invoker of **fncreate** and other FNS commands is expected to have the necessary NIS+ credentials. See **nis+**(1) and **nisdefaults**(1). The environment variable NIS_GROUP of the process specifies the group owner for the NIS+ objects thus created. In order to facilitate administration of the NIS+ objects, NIS_GROUP should be set to the name of the NIS+ administration group for the domain prior to executing **fncreate** and other FNS commands. Changes to NIS+-related properties, including default access control rights, could be effected using NIS+ administration tools and interfaces after the context has been created. The NIS+ object name that corresponds to an FNS composite name can be obtained using **fnlookup** and **fnlist.**

SEE ALSO

fncreate(1M), **fnlookup**(1), **fnlist**(1), **nis+**(1), **nisdefaults**(1), **nisls**(1), **nischgrp**(1), **nischmod**(1), **nischown**(1), **xfn**(3N), **fns_initial_context**(5), **fns_policies**(5), **fns_references**(5)

NAME

fns_initial_context – overview of the FNS Initial Context

DESCRIPTION

Every FNS name is interpreted relative to some context, and every FNS naming operation is performed on a context object. The FNS programming interface (XFN) provides a function that allows the client to obtain an *Initial Context* object. The Initial Context provides the initial pathway to other FNS contexts. FNS defines a set of bindings that the client can expect to find in this context,

FNS assumes that for every process:

1. There is a user associated with the process when **fn_ctx_handle_from_initial()** is invoked. This association is based on the effective uid of the process. In the following discussion this user is denoted by *U*. The association of user to process may change during the life of a process but does not affect the context handle originally returned by **fn_ctx_handle_from_initial()**.
2. The process is running on a host when **fn_ctx_handle_from_initial()** is invoked. In the following discussion this host is denoted by *H*.

The following atomic names can appear in the Initial Context: **...**, **thishost**, **thisorgunit**, **thisens**, **myself**, **myorgunit**, **myens**, **orgunit**, **site**, **user** and **host**. Except for **...**, these names with an added underscore ('_') prefix are also in the Initial Context and have the same binding as their counterpart (e.g. **thishost** and **_thishost** have the same binding). In addition, **org** has the same binding as **orgunit**, and **thisuser** has the same binding as **myself**. The bindings for these names are summarized in the following table.

Some of these names may not necessarily appear in all Initial Contexts. For example, a process owned by the super-user of a machine does not have any of the user-related bindings. Or, an installation that has not set up a site namespace will not have the site-related bindings.

...	global context for resolving DNS or X.500 names. Synonym: **/...**
thishost	*H*'s host context. Synonym: **_thishost**
thisens	the enterprise root of *H*. Synonym: **_thisens**
thisorgunit	*H*'s distinguished organizational unit context. In Solaris, this is *H*'s NIS+ home domain. Synonym: **_thisorgunit**
myself	*U*'s user context. Synonyms: **_myself**, **thisuser**
myens	the enterprise root of *U*. Synonym: **_myens**
myorgunit	*U*'s distinguished organizational unit context. In Solaris, this is *U*'s NIS+ home domain. Synonym: **_myorgunit**
user	the context in which users in the same organizational unit as *H* are named. Synonym: **_user**
host	the context in which hosts in the same organizational unit as *H* are named. Synonym: **_host**
org	the root context of the organizational unit namespace in *H*'s enterprise. In Solaris, this corresponds to the NIS+ root domain. Synonyms:

orgunit, **_orgunit**

site the root context of the site namespace in *H*'s enterprise, if the site namespace has been configured. Synonym: **_site**

EXAMPLES

The types of objects that may be named relative to the enterprise root are user, host, service, organizational unit, file and site. Here are some examples of names that begin with the enterprise root.

thisens/orgunit/multimedia.servers.engineering
names an organizational unit **multimedia.servers.engineering** in *H*'s enterprise.

thisens/site/northwing.floor3.admin
names the north wing site, on the third floor of the administrations building in *H*'s enterprise.

myens/user/hdiffie
names the user **hdiffie** in *U*'s enterprise.

myens/service/teletax
names the **teletax** service of *U*'s enterprise.

The types of objects that may be named relative to an organizational unit name are: user, host, service, file and site. Here are some examples of names that begin with organizational unit names (either explicitly via org, or implicitly via thisorgunit or myorgunit), and name objects relative to organizational unit names when resolved in the Initial Context.

org/accounts_payable.finance/site/videoconference.northwing
names a conference room **videoconference** located in the north wing of the site associated with the organizational unit **accounts_payable.finance.**

org/finance/user/mjones
names a user **mjones** in the organizational unit **finance**.

org/finance/host/inmail
names a machine **inmail** belonging to the organizational unit **finance**.

org/accounts_payable.finance/fs/pub/blue-and-whites/FY92-124
names a file **pub/blue-and-whites/FY92-124** belonging to the organizational unit **accounts_payable.finance**.

org/accounts_payable.finance/service/calendar
names the **calendar** service of the organizational unit **accounts_payable.finance**. This might manage the meeting schedules of the organizational unit.

thisorgunit/user/cmead
names the user **cmead** in *H*'s organizational unit.

myorgunit/fs/pub/project_plans/widget.ps
names the file **pub/project_plans/widget.ps** exported by *U*'s organizational unit's file system.

The types of objects that may be named relative to a site name are users, hosts, services and files. Here are some examples of names that begin with site names via site, and name objects relative to sites when resolved in the Initial Context.

site/b5.mtv/service/printer/speedy
names a printer **speedy** in the **b5.mtv** site.

site/admin/fs/usr/dist
names a file directory **usr/dist** available in the site **admin**.

The types of objects that may be named relative to a user name are services and files. Here are some examples of names that begin with user names (explicitly via user or implicitly via thisuser), and name objects relative to users when resolved in the Initial Context.

user/jsmith/service/calendar
names the **calendar** service of the user **jsmith**.

user/jsmith/fs/bin/games/riddles
names the file **bin/games/riddles** of the user **jsmith**.

thisuser/service/printer
names the **printer** service of *U*.

The types of objects that may be named relative to a host name are services and files. Here are some examples of names that begin with host names (explicitly via host or implicitly via thishost), and name objects relative to hosts when resolved in the Initial Context.

host/mailhop/service/mailbox
names the **mailbox** service associated with the machine **mailhop**.

host/mailhop/fs/pub/saf/archives.91
names the directory **pub/saf/archives.91** found under the root directory of the machine **mailhop**.

thishost/service/printer
names the **printer** service of *H*.

SEE ALSO **nis+**(1), **geteuid**(2), **fn_ctx_handle_from_initial**(3N), **xfn**(3N), **fns**(5), **fns_policies**(5)

NAME

fns_policies – Overview of the FNS Policies

DESCRIPTION

FNS defines policies for naming objects in the federated namespace. The goal of these policies is to allow easy and uniform composition of names. The policies use the basic rule that objects with narrower scopes are named relative to objects with wider scopes.

FNS policies are described in terms of the following three categories: global, enterprise and application.

Global naming service.

A global naming service is a naming service that has world-wide scope. Internet DNS and X.500 are examples of global naming services. The types of objects named at this global level are typically countries, states, provinces, cities, companies, universities, institutions, and government departments and ministries. These entities are refered to as *enterprises.*

Enterprise-level naming service.

Enterprise-level naming services are used to name objects within an enterprise. Within an enterprise, there are naming services that provide contexts for naming common entities such as organizational units, physical sites, human users and computers. Enterprise-level naming services are bound below the global naming services. Global naming services provide contexts in which the root contexts of enterprise-level naming services can be bound.

Application-level naming service.

Application-level naming services are incorporated in applications offering services such as file service, mail service, print service, and so on. Application-level naming services are bound below enterprise naming services. The enterprise-level naming services provide contexts in which contexts of application-level naming services can be bound.

FNS has policies for global and enterprise naming. Naming within applications is left to individual applications or groups of related applications and not specified by FNS.

FNS policy specifies that DNS and X.500 are global naming services that are used to name enterprises. The global namespace is named using the name "...". A DNS name or an X.500 name can appear after the "...". Support for federating global naming services is planned for a future release of FNS.

Within an enterprise, there are namespaces for organizational units, sites, hosts, users, files and services, referred to by the names **orgunit**, **site**, **host**, **user**, **fs**, and **service**. In addition, these namespaces can be named using these names with an added underscore ('_') prefix (e.g. **host** and **_host** have the same binding). The following table summarizes the FNS policies.

Context Type	Subordinate Context	Parent Context
org unit	site user host file system service	enterprise root
site	user host file system service	enterprise root org unit
user	service file system	enterprise root org unit
host	service file system	enterprise root org unit
service	not specified	enterprise root org unit site user host
file system	none	enterprise root org unit site user host

In Solaris, an organizational unit name corresponds to an NIS+ domain name and is identified using either the fully-qualified form of its NIS+ domain name, or its NIS+ domain name relative to the NIS+ root. Fully-qualified NIS+ domain names have a terminal dot ('.'). For example, assume that the NIS+ root domain is "Wiz.COM." and "sales" is a subdomain of that. Then, the names **org/sales.Wiz.COM.** and **org/sales** both refer to the organizational unit corresponding to the same NIS+ domain **sales.Wiz.COM.**.

User names correspond to names in the corresponding NIS+ *passwd.org_dir* table. The file system context associated with a user is obtained from his entry in the NIS+ *passwd.org_dir* table.

Host names correspond to names in the corresponding NIS+ *hosts.org_dir* table. The file system context associated with a host corresponds to the files systems exported by the host.

EXAMPLES

The types of objects that may be named relative to an organizational unit name are: user, host, service, file and site. Here are some examples of names name objects relative to

organizational unit names.

org/accounts_payable.finance/site/videoconference.northwing
names a conference room **videoconference** located in the north wing of the site associated with the organizational unit **accounts_payable.finance**.

org/finance/user/mjones
names a user **mjones** in the organizational unit **finance**.

org/finance/host/inmail
names a machine **inmail** belonging to the organizational unit **finance**.

org/accounts_payable.finance/fs/pub/blue-and-whites/FY92-124
names a file **pub/blue-and-whites/FY92-124** belonging to the organizational unit **accounts_payable.finance**.

org/accounts_payable.finance/service/calendar
names the **calendar** service of the organizational unit **accounts_payable.finance**. This might manage the meeting schedules of the organizational unit.

The types of objects that may be named relative to a site name are services and files. Here are some examples of names that name objects relative to sites.

site/b5.mtv/service/printer/speedy
names a printer **speedy** in the **b5.mtv** site.

site/admin/fs/usr/dist
names a file directory **usr/dist** available in the site **admin**.

The types of objects that may be named relative to a user name are services and files. Here are some examples of names that name objects relative to users.

user/jsmith/service/calendar
names the **calendar** service of the user **jsmith**.

user/jsmith/fs/bin/games/riddles
names the file **bin/games/riddles** of the user **jsmith**.

The types of objects that may be named relative to a host name are services and files. Here are some examples of names that name objects relative to hosts.

host/mailhop/service/mailbox
names the **mailbox** service associated with the machine **mailhop**.

host/mailhop/fs/pub/saf/archives.91
names the directory **pub/saf/archives.91** found under the root directory of the machine **mailhop**.

SEE ALSO **fncreate**(1M), **nis+**(1), **xfn**(3N), **fns**(5), **fns_initial_context**(5), **fns_references**(5)

NAME

fns_references – Overview of FNS References

DESCRIPTION

Every composite name in FNS is bound to a *reference*. A reference consists of a type and a list of addresses. The reference type is used to identify the type of object.

An address is something that can be used with some communication mechanism to invoke operations on an object or service. Multiple addresses are intended to identify multiple communication endpoints for a single conceptual object or service. Each address in a reference consists of an address type and an opaque buffer. The address type determines the format and interpretation of the address data. Together, the address's type and data specify how to reach the object. Many communication mechanisms are possible; FNS does not place any restrictions on them.

The following summarizes the reference and address types that are currently defined. New types should be registered with the Federated Naming Group at SunSoft.

Reference Types

All reference types use the **FN_ID_STRING** identifier format unless otherwise qualified.

onc_fn_enterprise
Enterprise root context.

onc_fn_organization
A context for naming objects related to an organizational unit.

onc_fn_hostname
A context for naming hosts.

onc_fn_username
A context for naming users.

onc_fn_user
A context for naming objects related to a user.

onc_fn_host
A context for naming objects related to a computer.

onc_fn_site
A context for naming sites.

onc_fn_service
A context for naming services.

onc_fn_nsid
A context for naming namespace identifiers.

onc_fn_generic
A context for naming application-specific objects.

onc_fn_fs
A context for naming files, directories, and file systems.

onc_fn_printername

A context for naming printers.

onc_printers
A printer object. When implemented on top of NIS+, this could also be a context for naming printers.

onc_fn_null
A null context.

fn_link_ref
An XFN link.

inet_domain
An Internet domain.

Address Types

All address types use the **FN_ID_STRING** identifier format unless otherwise qualfied. The format of address contents is determined by the corresponding address type.

onc_fn_nisplus
For an FNS enterprise-level object implemented on top of NIS+. The address contains the context type, context representation type (either normal or merged), version number of the reference, and the NIS+ name of the object. The only intended use of this reference is that it be passed to **fn_ctx_handle_from_ref( )**.

onc_fn_nis
For an FNS enterprise-level object implemented on top of NIS. The address contains the context type and version number of the reference, and the NIS name of the object. The only intended use of this reference is that it be passed to **fn_ctx_handle_from_ref( )**.

onc_fn_files
For an FNS enterprise-level object implemented on top of **/etc** files. The address contains the context type and version number of the reference, and the location of the object in the /etc file system. The only intended use of this reference is that it be passed to **fn_ctx_handle_from_ref()**.

onc_fn_fs_user_nisplus
For a user's home directory. The address contains the user's name and the name of the NIS+ passwd table where the user's home directory is stored.

onc_fn_fs_host
For all file systems exported by a host. The address contains the host's name.

onc_fn_fs_mount
For a single mount point. The address contains the mount options, the name of the servers and the exported path. See **mount**(1M).

fn_link_addr
For an XFN link address. The contents is the string form of the composite name.

inet_domain
For an Internet domain. The address contains the fully-qualified domain name (e.g. "Wiz.COM.")

inet_ipaddr_string

For an object with an Internet address. The address contains an internet IP address in dotted string form (e.g. "192.144.2.3").

x500 For an X.500 object. The address contains an X.500 Distinguished Name, in the syntax specified in the **X/Open DCE: Directory Services.**

osi_paddr
For an object with an OSI presentation address. The address contains the string encoding of an OSI Presentation Address as defined in **RFC 1278.**

onc_printers_bsaddr
For a printer that understands the BSD print protocol. The address contains the machine name and printer name used by the protocol.

onc_printers_use
For a printer alias. The address contains a printer name.

onc_printers_all
For a list of printers that are enumerated using the "all" option. The address contains a list of printer names.

onc_printers_location
For a printer's location. The address format is unspecified.

onc_printers_type
For a printer's type. The address format is unspecified.

onc_printers_speed
For a printer's speed. The address format is unspecified.

SEE ALSO **fn_ctx_handle_from_ref**(3N), **xfn**(3N), **fns**(5), **fns_policies**(5)

NAME

formats – file format notation

DESCRIPTION

Utility descriptions use a syntax to describe the data organization within files—stdin, stdout, stderr, input files, and output files—when that organization is not otherwise obvious. The syntax is similar to that used by the **printf**(3S) function. When used for stdin or input file descriptions, this syntax describes the format that could have been used to write the text to be read, not a format that could be used by the **scanf**(3S) function to read the input file.

Format

The description of an individual record is as follows:

"<*format*>", [<*arg1*>, <*arg2*>, . . ., <*argn*>]

The *format* is a character string that contains three types of objects defined below:

characters Characters that are not *escape sequences* or *conversion specifications*, as described below, are copied to the output.

escape sequences Represent non-graphic characters.

conversion specifications
Specifies the output format of each argument. (See below.)

The following characters have the following special meaning in the format string:

" " (An empty character position.) One or more blank characters

Δ Exactly one space character.

The notation for spaces allows some flexibility for application output. Note that an empty character position in *format* represents one or more blank characters on the output (not *white space*, which can include newline characters). Therefore, another utility that reads that output as its input must be prepared to parse the data using **scanf**(3S), **awk**(1), and so forth. The Δ character is used when exactly one space character is output.

Escape Sequences

The following table lists escape sequences and associated actions on display devices capable of the action.

Escape Sequence	Represents Character	Terminal Action
\\	backslash	None.
\a	alert	Attempts to alert the user through audible or visible notification.
\b	backspace	Moves the printing position to one column before the current position, unless the current position is the start of a line.
\f	form-feed	Moves the printing position to the initial printing position of the next logical page.
\n	newline	Moves the printing position to the start of the next line.
\r	carriage-return	Moves the printing position to the start of the current line.

\t	tab	Moves the printing position to the next tab position on the current line. If there are no more tab positions left on the line, the behaviour is undefined.
\v	vertical-tab	Moves the printing position to the start of the next vertical tab position. If there are no more vertical tab positions left on the page, the behaviour is undefined.

Conversion Specifications

Each conversion specification is introduced by the percent-sign character (%). After the character %, the following appear in sequence:

flags Zero or more *flags*, in any order, that modify the meaning of the conversion specification.

field width An optional string of decimal digits to specify a minimum *field width*. For an output field, if the converted value has fewer bytes than the field width, it is padded on the left (or right, if the left-adjustment flag (–), described below, has been given to the field width).

precision Gives the minimum number of digits to appear for the d, o, i, u, x or X conversions (the field is padded with leading zeros), the number of digits to appear after the radix character for the e and f conversions, the maximum number of significant digits for the g conversion; or the maximum number of bytes to be written from a string in s conversion. The precision takes the form of a period (.) followed by a decimal digit string; a null digit string is treated as zero.

conversion characters A conversion character (see below) that indicates the type of conversion to be applied.

flags

The *flags* and their meanings are:

– The result of the conversion is left-justified within the field.

+ The result of a signed conversion always begins with a sign (+ or –).

<*space*> If the first character of a signed conversion is not a sign, a space character is prefixed to the result. This means that if the space character and + flags both appear, the space character flag is ignored.

The value is to be converted to an alternative form. For c, d, i, u, and s conversions, the behaviour is undefined. For o conversion, it increases the precision to force the first digit of the result to be a zero. For x or X conversion, a non-zero result has 0x or 0X prefixed to it, respectively. For e, E, f, g, and G conversions, the result always contains a radix character, even if no digits follow the radix character. For g and G conversions, trailing zeros are not removed from the result as they usually are.

0 For d, i, o, u, x, X, e, E, f, g, and G conversions, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the 0 and – flags both appear, the 0 flag is ignored. For d, i, o, u, x and X conversions, if a precision is specified, the 0 flag is

ignored. For other conversions, the behaviour is undefined.

Conversion Characters

Each conversion character results in fetching zero or more arguments. The results are undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are ignored.

The *conversion characters* and their meanings are:

d,i,o,u,x,X The integer argument is written as signed decimal (d or i), unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal notation (x and X). The d and i specifiers convert to signed decimal in the style **[–]***dddd*. The x conversion uses the numbers and letters 0123456789abcdef and the X conversion uses the numbers and letters 0123456789ABCDEF. The *precision* component of the argument specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits than the specified minimum, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of 0 is no characters. If both the field width and precision are omitted, the implementation may precede, follow or precede and follow numeric arguments of types d, i and u with blank characters; arguments of type o (octal) may be preceded with leading zeros.

The treatment of integers and spaces is different from the **printf**(3S) function in that they can be surrounded with blank characters. This was done so that, given a format such as:

"%d \ n",<*foo*>

the implementation could use a **printf( )** call such as:

printf("%6d \ n", *foo*);

and still conform. This notation is thus somewhat like **scanf( )** in addition to **printf( ).**

f The floating point number argument is written in decimal notation in the style **[–]***ddd.ddd*, where the number of digits after the radix character (shown here as a decimal point) is equal to the *precision* specification. The **LC_NUMERIC** locale category determines the radix character to use in this format. If the *precision* is omitted from the argument, six digits are written after the radix character; if the *precision* is explicitly 0, no radix character appears.

e,E The floating point number argument is written in the style **[–]***d.ddde±dd* (the symbol ± indicates either a plus or minus sign), where there is one digit before the radix character (shown here as a decimal point) and the number of digits after it is equal to the precision. The **LC_NUMERIC** locale category determines the radix character to use in this format. When the precision is missing, six digits are written after the radix character; if the precision is 0, no radix character appears. The E conversion character produces a number with E instead of e introducing the exponent. The exponent always contains at least two digits. However, if the value to be written requires an

exponent greater than two digits, additional exponent digits are written as necessary.

g,G The floating point number argument is written in style f or e (or in style E in the case of a G conversion character), with the precision specifying the number of significant digits. The style used depends on the value converted: style g is used only if the exponent resulting from the conversion is less than –4 or greater than or equal to the precision. Trailing zeros are removed from the result. A radix character appears only if it is followed by a digit.

c The integer argument is converted to an **unsigned char** and the resulting byte is written.

s The argument is taken to be a string and bytes from the string are written until the end of the string or the number of bytes indicated by the *precision* specification of the argument is reached. If the precision is omitted from the argument, it is taken to be infinite, so all bytes up to the end of the string are written.

% Write a % character; no argument is converted.

In no case does a non-existent or insufficient *field width* cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. The term *field width* should not be confused with the term *precision* used in the description of %s.

One difference from the C function **printf()** is that the l and h conversion characters are not used. There is no differentiation between decimal values for type **int**, type **long**, or type **short**. The specifications %d or %i should be interpreted as an arbitrary length sequence of digits. Also, no distinction is made between single precision and double precision numbers (**float** or **double** in C). These are simply referred to as floating point numbers.

Many of the output descriptions use the term *line*, such as:

"%s", <*input line*>

Since the definition of *line* includes the trailing newline character already, there is no need to include a **\n** in the format; a double newline character would otherwise result.

EXAMPLES

To represent the output of a program that prints a date and time in the form Sunday, July 3, 10:02, where <*weekday*> and <*month*> are strings:

"%s,␣%s␣%d,␣%d:%.2d\n",<*weekday*>,<*month*>,<*day*>,<*hour*>,<*min*>

To show π written to 5 decimal places:

"pi␣=␣%.5f\n",<*value of* π>

To show an input file format consisting of five colon-separated fields:

"%s:%s:%s:%s:%s\n",<*arg1*>,<*arg2*>,<*arg3*>,<*arg4*>,<*arg5*>

SEE ALSO **awk**(1), **printf**(1), **printf**(3S), **scanf**(3S)

NAME

iconv – code set conversion tables

DESCRIPTION

The following code set conversions are supported:

Code Set Conversions Supported				
Code	Symbol	Target Code	Symbol	comment
ISO 646	646	ISO 8859-1	8859	US Ascii
ISO 646de	646de	ISO 8859-1	8859	German
ISO 646da	646da	ISO 8859-1	8859	Danish
ISO 646en	646en	ISO 8859-1	8859	English Ascii
ISO 646es	646es	ISO 8859-1	8859	Spanish
ISO 646fr	646fr	ISO 8859-1	8859	French
ISO 646it	646it	ISO 8859-1	8859	Italian
ISO 646sv	646sv	ISO 8859-1	8859	Swedish
ISO 8859-1	8859	ISO 646	646	7 bit Ascii
ISO 8859-1	8859	ISO 646de	646de	German
ISO 8859-1	8859	ISO 646da	646da	Danish
ISO 8859-1	8859	ISO 646en	646en	English Ascii
ISO 8859-1	8859	ISO 646es	646es	Spanish
ISO 8859-1	8859	ISO 646fr	646fr	French
ISO 8859-1	8859	ISO 646it	646it	Italian
ISO 8859-1	8859	ISO 646sv	646sv	Swedish

The conversions are performed according to the tables following. All values in the tables are given in octal.

ISO 646 (US ASCII) to ISO 8859-1

For the conversion of ISO 646 to ISO 8859-1 all characters in ISO 646 can be mapped unchanged to ISO 8859-1

ISO 646de (GERMAN) to ISO 8859-1

For the conversion of ISO 646de to ISO 8859-1 all characters not in the following table are mapped unchanged.

Conversions Performed	
ISO 646de	ISO 8859-1
100	247
133	304
134	326
135	334
173	344
174	366
175	374
176	337

ISO 646da (DANISH) to ISO 8859-1

For the conversion of ISO 646da to ISO 8859-1 all characters not in the following table are mapped unchanged.

Conversions Performed	
ISO 646da	ISO 8859-1
133	306
134	330
135	305
173	346
174	370
175	345

ISO 646en (ENGLISH ASCII) to ISO 8859-1

For the conversion of ISO 646en to ISO 8859-1 all characters not in the following table are mapped unchanged.

Conversions Performed	
ISO 646en	ISO 8859-1
043	243

ISO 646fr (FRENCH) to ISO 8859-1

For the conversion of ISO 646fr to ISO 8859-1 all characters not in the following table are mapped unchanged.

Conversions Performed	
ISO 646fr	ISO 8859-1
043	243
100	340
133	260
134	347
135	247
173	351
174	371
175	350
176	250

ISO 646it (ITALIAN) to ISO 8859-1

For the conversion of ISO 646it to ISO 8859-1 all characters not in the following table are mapped unchanged.

Conversions Performed	
ISO 646it	ISO 8859-1
043	243
100	247
133	260
134	347
135	351
140	371
173	340
174	362
175	350
176	354

ISO 646es (SPANISH) to ISO 8859-1

For the conversion of ISO 646es to ISO 8859-1 all characters not in the following table are mapped unchanged.

Conversions Performed	
ISO 646es	ISO 8859-1
100	247
133	241
134	321
135	277
173	260
174	361
175	347

ISO 646sv (SWEDISH) to ISO 8859-1

For the conversion of ISO 646sv to ISO 8859-1 all characters not in the following table are mapped unchanged.

Conversions Performed	
ISO 646sv	ISO 8859-1
100	311
133	304
134	326
135	305
136	334
140	351
173	344
174	366
175	345
176	374

ISO 8859-1 to ISO 646 (ASCII)

For the conversion of ISO 8859-1 to ISO 646 all characters not in the following table are mapped unchanged.

Converted to Underscore '_' (137)
200 201 202 203 204 205 206 207
210 211 212 213 214 215 216 217
220 221 222 223 224 225 226 227
230 231 232 233 234 235 236 237
240 241 242 243 244 245 246 247
250 251 252 253 254 255 256 257
260 261 262 263 264 265 266 267
270 271 272 273 274 275 276 277
300 301 302 303 304 305 306 307
310 311 312 313 314 315 316 317
320 321 322 323 324 325 326 327
330 331 332 333 334 335 336 337
340 341 342 343 344 345 346 347
350 351 352 353 354 355 356 357
360 361 362 363 364 365 366 367
370 371 372 373 374 375 376 377

ISO 8859-1 to ISO 646de (GERMAN)

For the conversion of ISO 8859-1 to ISO 646de all characters not in the following tables are mapped unchanged.

Conversions Performed	
ISO 8859-1	ISO 646de
247	100
304	133
326	134
334	135
337	176
344	173
366	174
374	175

Converted to Underscore '_' (137)
100 133 134 135 173 174 175 176
200 201 202 203 204 205 206 207
210 211 212 213 214 215 216 217
220 221 222 223 224 225 226 227
230 231 232 233 234 235 236 237
240 241 242 243 244 245 246
250 251 252 253 254 255 256 257
260 261 262 263 264 265 266 267
270 271 272 273 274 275 276 277
300 301 302 303 305 306 307
310 311 312 313 314 315 316 317
320 321 322 323 324 325 327
330 331 332 333 335 336 337
340 341 342 343 345 346 347
350 351 352 353 354 355 356 357
360 361 362 363 364 365 367
370 371 372 373 375 376 377

ISO 8859-1 to ISO 646da (DANISH)

For the conversion of ISO 8859-1 to ISO 646da all characters not in the following tables are mapped unchanged.

Conversions Performed	
ISO 8859-1	ISO 646da
305	135
306	133
330	134
345	175
346	173
370	174

Converted to Underscore '_' (137)
133 134 135 173 174 175
200 201 202 203 204 205 206 207
210 211 212 213 214 215 216 217
220 221 222 223 224 225 226 227
230 231 232 233 234 235 236 237
240 241 242 243 244 245 246 247
250 251 252 253 254 255 256 257
260 261 262 263 264 265 266 267
270 271 272 273 274 275 276 277
300 301 302 303 304 307
310 311 312 313 314 315 316 317
320 321 322 323 324 325 326 327
331 332 333 334 335 336 337
340 341 342 343 344 347
350 351 352 353 354 355 356 357
360 361 362 363 364 365 366 367
371 372 373 374 376 377

ISO 8859-1 to ISO 646en (ENGLISH ASCII)

For the conversion of ISO 8859-1 to ISO 646en all characters not in the following tables are mapped unchanged.

Conversions Performed	
ISO 8859-1	ISO 646en
243	043

Converted to Underscore '_' (137)
043
200 201 202 203 204 205 206 207
210 211 212 213 214 215 216 217
220 221 222 223 224 225 226 227
230 231 232 233 234 235 236 237
240 241 242 244 245 246 247
250 251 252 253 254 255 256 257
260 261 262 263 264 265 266 267
270 271 272 273 274 275 276 277
300 301 302 303 304 305 306 307
310 311 312 313 314 315 316 317
320 321 322 323 324 325 326 327
330 331 332 333 334 335 336 337
340 341 342 343 344 345 346 347
350 351 352 353 354 355 356 357
360 361 362 363 364 365 366 367
370 371 372 373 374 375 376 377

ISO 8859-1 to ISO 646fr (FRENCH)

For the conversion of ISO 8859-1 to ISO 646fr all characters not in the following tables are mapped unchanged.

Conversions Performed	
ISO 8859-1	ISO 646fr
243	043
247	135
250	176
260	133
340	100
347	134
350	175
351	173
371	174

Converted to Underscore '_' (137)
043
100 133 134 135 173 174 175 176
200 201 202 203 204 205 206 207
210 211 212 213 214 215 216 217
220 221 222 223 224 225 226 227
230 231 232 233 234 235 236 237
240 241 242 244 245 246
251 252 253 254 255 256 257
261 262 263 264 265 266 267
270 271 272 273 274 275 276 277
300 301 302 303 304 305 306 307
310 311 312 313 314 315 316 317
320 321 322 323 324 325 326 327
330 331 332 333 334 335 336 337
341 342 343 344 345 346
352 353 354 355 356 357
360 361 362 363 364 365 366 367
370 372 373 374 375 376 377

ISO 8859-1 to ISO 646it (ITALIAN)

For the conversion of ISO 8859-1 to ISO 646it all characters not in the following tables are mapped unchanged.

Conversions Performed	
ISO 8859-1	ISO 646it
243	043
247	100
260	133
340	173
347	134
350	175
351	135
354	176
362	174
371	140

Converted to Underscore '_' (137)
043
100 133 134 135 173 174 175 176
200 201 202 203 204 205 206 207
210 211 212 213 214 215 216 217
220 221 222 223 224 225 226 227
230 231 232 233 234 235 236 237
240 241 242 244 245 246
250 251 252 253 254 255 256 257
261 262 263 264 265 266 267
270 271 272 273 274 275 276 277
300 301 302 303 304 305 306 307
310 311 312 313 314 315 316 317
320 321 322 323 324 325 326 327
330 331 332 333 334 335 336 337
341 342 343 344 345 346
352 353 354 355 356 357
360 361 363 364 365 366 367
370 372 373 374 375 376 377

ISO 8859-1 to ISO 646es (SPANISH)

For the conversion of ISO 8859-1 to ISO 646es all characters not in the following tables are mapped unchanged.

Conversions Performed	
ISO 8859-1	ISO 646es
241	133
247	100
260	173
277	135
321	134
347	175
361	174

Converted to Underscore '_' (137)
100 133 134 135 173 174 175
200 201 202 203 204 205 206 207
210 211 212 213 214 215 216 217
220 221 222 223 224 225 226 227
230 231 232 233 234 235 236 237
240 242 243 244 245 246
250 251 252 253 254 255 256 257
261 262 263 264 265 266 267
270 271 272 273 274 275 276
300 301 302 303 304 305 306 307
310 311 312 313 314 315 316 317
320 322 323 324 325 326 327
330 331 332 333 334 335 336 337
340 341 342 343 344 345 346
350 351 352 353 354 355 356 357
360 362 363 364 365 366 367
370 371 372 373 374 375 376 377

ISO 8859-1 to ISO 646sv (SWEDISH)

For the conversion of ISO 8859-1 to ISO 646sv all characters not in the following tables are mapped unchanged.

Conversions Performed	
ISO 8859-1	ISO 646sv
304	133
305	135
311	100
326	134
334	136
344	173
345	175
351	140
366	174
374	176

Converted to Underscore '_' (137)
100 133 134 135 136 140
173 174 175 176
200 201 202 203 204 205 206 207
210 211 212 213 214 215 216 217
220 221 222 223 224 225 226 227
230 231 232 233 234 235 236 237
240 241 242 243 244 245 246 247
250 251 252 253 254 255 256 257
260 261 262 263 264 265 266 267
270 271 272 273 274 275 276 277
300 301 302 303 306 307
310 312 313 314 315 316 317
320 321 322 323 324 325 327
330 331 332 333 335 336 337
340 341 342 343 346 347
350 352 353 354 355 356 357
360 361 362 363 364 365 367
370 371 372 373 375 376 377

FILES

/usr/lib/iconv/*.so	conversion modules
/usr/lib/iconv/*.t	conversion tables
/usr/lib/iconv/iconv_data	list of conversions supported by conversion tables

SEE ALSO

iconv(1), **iconv**(3)

NAME

langinfo – language information constants

SYNOPSIS

#include <langinfo.h>

DESCRIPTION

This header contains the constants used to identify items of langinfo data. The mode of *items* is given in **nl_types**.

DAY_1	Locale's equivalent of 'sunday'
DAY_2	Locale's equivalent of 'monday'
DAY_3	Locale's equivalent of 'tuesday'
DAY_4	Locale's equivalent of 'wednesday'
DAY_5	Locale's equivalent of 'thursday'
DAY_6	Locale's equivalent of 'friday'
DAY_7	Locale's equivalent of 'saturday'
ABDAY_1	Locale's equivalent of 'sun'
ABDAY_2	Locale's equivalent of 'mon'
ABDAY_3	Locale's equivalent of 'tue'
ABDAY_4	Locale's equivalent of 'wed'
ABDAY_5	Locale's equivalent of 'thur'
ABDAY_6	Locale's equivalent of 'fri'
ABDAY_7	Locale's equivalent of 'sat'
MON_1	Locale's equivalent of 'january'
MON_2	Locale's equivalent of 'february'
MON_3	Locale's equivalent of 'march'
MON_4	Locale's equivalent of 'april'
MON_5	Locale's equivalent of 'may'
MON_6	Locale's equivalent of 'june'
MON_7	Locale's equivalent of 'july'
MON_8	Locale's equivalent of 'august'
MON_9	Locale's equivalent of 'september'
MON_10	Locale's equivalent of 'october'
MON_11	Locale's equivalent of 'november'
MON_12	Locale's equivalent of 'december'
ABMON_1	Locale's equivalent of 'jan'
ABMON_2	Locale's equivalent of 'feb'
ABMON_3	Locale's equivalent of 'mar'
ABMON_4	Locale's equivalent of 'apr'
ABMON_5	Locale's equivalent of 'may'
ABMON_6	Locale's equivalent of 'jun'
ABMON_7	Locale's equivalent of 'jul'
ABMON_8	Locale's equivalent of 'aug'
ABMON_9	Locale's equivalent of 'sep'
ABMON_10	Locale's equivalent of 'oct'
ABMON_11	Locale's equivalent of 'nov'
ABMON_12	Locale's equivalent of 'dec'

RADIXCHAR	Locale's equivalent of '.'
THOUSEP	Locale's equivalent of ','
YESSTR	Locale's equivalent of 'yes'
NOSTR	Locale's equivalent of 'no'
CRNCYSTR	Locale's currency symbol
D_T_FMT	Locale's default format for date and time
D_FMT	Locale's default format for the date
T_FMT	Locale's default format for the time
AM_STR	Locale's equivalent of 'AM'
PM_STR	Locale's equivalent of 'PM'

This information is retrieved by **nl_langinfo**.

The items **CRNCYSTR**, **RADIXCHAR** and **THOUSEP** are extracted from the fields **currency_symbol**, **decimal_point** and **thousands_sep** in the structure returned by **localeconv**.

The items **T_FMT**, **D_FMT**, **D_T_FMT**, **YESSTR** and **NOSTR** are retrieved from a special message catalog named **Xopen_info** which should be generated for each locale supported and installed in the appropriate directory [see **gettxt**(3C) and **mkmsgs**(1)]. This catalog should have the messages in the order **T_FMT**, **D_FMT**, **D_T_FMT**, **YESSTR** and **NOSTR**.

All other items are as returned by **strftime**.

SEE ALSO

mkmsgs(1), **chrtbl**(1M), **gettxt**(3C), **localeconv**(3C), **nl_langinfo**(3C), **strftime**(3C), **nl_types**(5)

NAME

locale – subset of a user's environment that depends on language and cultural conventions

DESCRIPTION

A *locale* is the definition of the subset of a user's environment that depends on language and cultural conventions. It is made up from one or more categories. Each category is identified by its name and controls specific aspects of the behavior of components of the system. Category names correspond to the following environment variable names:

LC_CTYPE	Character classification and case conversion.
LC_COLLATE	Collation order.
LC_TIME	Date and time formats.
LC_NUMERIC	Numeric formatting.
LC_MONETARY	Monetary formatting.
LC_MESSAGES	Formats of informative and diagnostic messages and interactive responses.

The standard utilities base their behavior on the current locale, as defined in the **ENVIRONMENT** section for each utility. The behavior of some of the C-language functions will also be modified based on the current locale, as defined by the last call to **setlocale**(3C).

Locales other than those supplied by the implementation can be created by the application via the **localedef**(1) utility. The value that is used to specify a locale when using environment variables will be the string specified as the *name* operand to **localedef** when the locale was created. The strings "C" and "POSIX" are reserved as identifiers for the POSIX locale.

Applications can select the desired locale by invoking the **setlocale( )** function with the appropriate value. If the function is invoked with an empty string, such as:

```
setlocale(LC_ALL, "");
```

the value of the corresponding environment variable is used. If the environment variable is unset or is set to the empty string, the **setlocale( )** function sets the appropriate environment.

Locale Definition

Locales can be described with the file format accepted by the **localedef** utility.

The locale definition file must contain one or more locale category source definitions, and must not contain more than one definition for the same locale category.

A category source definition consists of a category header, a category body and a category trailer. A category header consists of the character string naming of the category, beginning with the characters **LC_**. The category trailer consists of the string **END**, followed by one or more blank characters and the string used in the corresponding category header.

The category body consists of one or more lines of text. Each line contains an identifier, optionally followed by one or more operands. Identifiers are either keywords, identifying a particular locale element, or collating elements. Each keyword within a locale must

have a unique name (that is, two categories cannot have a commonly-named keyword); no keyword can start with the characters **LC_.** Identifiers must be separated from the operands by one or more blank characters.

Operands must be characters, collating elements or strings of characters. Strings must be enclosed in double-quotes. Literal double-quotes within strings must be preceded by the <*escape character*>, described below. When a keyword is followed by more than one operand, the operands must be separated by semicolons; blank characters are allowed both before and after a semicolon.

The first category header in the file can be preceded by a line modifying the comment character. It has the following format, starting in column 1:

"**comment_char %c\n",**<*comment character*>

The comment character defaults to the number sign (**#**). Blank lines and lines containing the <*comment character*> in the first position are ignored.

The first category header in the file can be preceded by a line modifying the escape character to be used in the file. It has the following format, starting in column 1:

"**escape_char %c\n",**<*escape character*>

The escape character defaults to backslash.

A line can be continued by placing an escape character as the last character on the line; this continuation character will be discarded from the input. Although the implementation need not accept any one portion of a continued line with a length exceeding **{LINE_MAX}** bytes, it places no limits on the accumulated length of the continued line. Comment lines cannot be continued on a subsequent line using an escaped newline character.

Individual characters, characters in strings, and collating elements must be represented using symbolic names, as defined below. In addition, characters can be represented using the characters themselves or as octal, hexadecimal or decimal constants. When non-symbolic notation is used, the resultant locale definitions will in many cases not be portable between systems. The left angle bracket (**<**) is a reserved symbol, denoting the start of a symbolic name; when used to represent itself it must be preceded by the escape character. The following rules apply to character representation:

1. A character can be represented via a symbolic name, enclosed within angle brackets **<** and **>**. The symbolic name, including the angle brackets, must exactly match a symbolic name defined in the charmap file specified via the **localedef** **–f** option, and will be replaced by a character value determined from the value associated with the symbolic name in the charmap file. The use of a symbolic name not found in the charmap file constitutes an error, unless the category is **LC_CTYPE** or **LC_COLLATE,** in which case it constitutes a warning condition (see **localedef**(1) for a description of action resulting from errors and warnings). The specification of a symbolic name in a **collating-element** or **collating-symbol** section that duplicates a symbolic name in the charmap file (if present) is an error. Use of the escape character or a right angle bracket within a symbolic name is invalid unless the character is preceded by the escape character.

Example:

<**c**>;<**c–cedilla**> "<**M**><**a**><**y**>"

2. A character can be represented by the character itself, in which case the value of the character is implementation-dependent. Within a string, the double-quote character, the escape character and the right angle bracket character must be escaped (preceded by the escape character) to be interpreted as the character itself. Outside strings, the characters

   , **;** **<** **>** *escape_char*

   must be escaped to be interpreted as the character itself.

   Example:

   c β "**May**"

3. A character can be represented as an octal constant. An octal constant is specified as the escape character followed by two or more octal digits. Each constant represents a byte value. Multi-byte values can be represented by concatenated constants specified in byte order with the last constant specifying the least significant byte of the character.

   Example:

   \143;\347;\143\150 "\115\141\171"

4. A character can be represented as a hexadecimal constant. A hexadecimal constant is specified as the escape character followed by an **x** followed by two or more hexadecimal digits. Each constant represents a byte value. Multi-byte values can be represented by concatenated constants specified in byte order with the last constant specifying the least significant byte of the character.

   Example:

   \x63;\xe7;\x63\x68 "\x4d\x61\x79"

5. A character can be represented as a decimal constant. A decimal constant is specified as the escape character followed by a **d** followed by two or more decimal digits. Each constant represents a byte value. Multi-byte values can be represented by concatenated constants specified in byte order with the last constant specifying the least significant byte of the character.

   Example:

   \d99;\d231;\d99\d104 "\d77\d97\d121"

Only characters existing in the character set for which the locale definition is created can be specified, whether using symbolic names, the characters themselves, or octal, decimal or hexadecimal constants. If a charmap file is present, only characters defined in the charmap can be specified using octal, decimal or hexadecimal constants. Symbolic names not present in the charmap file can be specified and will be ignored, as specified under item 1 above.

LC_CTYPE

The **LC_CTYPE** category defines character classification, case conversion and other character attributes. In addition, a series of characters can be represented by three adjacent periods representing an ellipsis symbol (. . .). The ellipsis specification is interpreted as meaning that all values between the values preceding and following it represent valid characters. The ellipsis specification is valid only within a single encoded character set; that is, within a group of characters of the same size. An ellipsis is interpreted as including in the list all characters with an encoded value higher than the encoded value of the character preceding the ellipsis and lower than the encoded value of the character following the ellipsis.

Example:

\x30;. . .;\x39;

includes in the character class all characters with encoded values between the endpoints.

The following keywords are recognized. In the descriptions, the term ''automatically included'' means that it is not an error either to include or omit any of the referenced characters.

The character classes **digit**, **xdigit**, **lower**, **upper**, and **space** have a set of automatically included characters. These only need to be specified if the character values (that is, encoding) differ from the implementation default values.

LC_CTYPE1
Begin definition of supplementary codeset 1.

LC_CTYPE2
Begin definition of supplementary codeset 2.

LC_CTYPE3
Begin definition of supplementary codeset 3.

cswidth Report byte count and screen width information.

upper Define characters to be classified as upper-case letters.

In the POSIX locale, the 26 upper-case letters are included:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

In a locale definition file, no character specified for the keywords **cntrl**, **digit**, **punct**, or **space** can be specified. The upper-case letters **A** to **Z** are automatically included in this class.

lower Define characters to be classified as lower-case letters.

In the POSIX locale, the 26 lower-case letters are included:

a b c d e f g h i j k l m n o p q r s t u v w x y z

In a locale definition file, no character specified for the keywords **cntrl**, **digit**, **punct**, or **space** can be specified. The lower-case letters **a** to **z** of the portable character set are automatically included in this class.

alpha Define characters to be classified as letters.

In the POSIX locale, all characters in the classes **upper** and **lower** are included. In a locale definition file, no character specified for the keywords **cntrl**, **digit**, **punct**, or **space** can be specified. Characters classified as either **upper** or **lower** are automatically included in this class.

digit Define the characters to be classified as numeric digits.

In the POSIX locale, only

0 1 2 3 4 5 6 7 8 9

are included.

In a locale definition file, only the digits **0**, **1**, **2**, **3**, **4**, **5**, **6**, **7**, **8**, and **9** can be specified, and in contiguous ascending sequence by numerical value. The digits **0** to **9** of the portable character set are automatically included in this class.

The definition of character class **digit** requires that only ten characters; the ones defining digits can be specified; alternative digits (for example, Hindi or Kanji) cannot be specified here.

space Define characters to be classified as white-space characters.

In the POSIX locale, at a minimum, the characters SPACE, FORMFEED, NEW-LINE, CARRIAGE RETURN, TAB, and VERTICAL TAB are included.

In a locale definition file, no character specified for the keywords **upper**, **lower**, **alpha**, **digit**, **graph**, or **xdigit** can be specified. The characters SPACE, FORMFEED, NEWLINE, CARRIAGE RETURN, TAB, and VERTICAL TAB of the portable character set, and any characters included in the class **blank** are automatically included in this class.

cntrl Define characters to be classified as control characters.

In the POSIX locale, no characters in classes **alpha** or **print** are included.

In a locale definition file, no character specified for the keywords **upper**, **lower**, **alpha**, **digit**, **punct**, **graph**, **print**, or **xdigit** can be specified.

punct Define characters to be classified as punctuation characters.

In the POSIX locale, neither the space character nor any characters in classes **alpha**, **digit**, or **cntrl** are included.

In a locale definition file, no character specified for the keywords **upper**, **lower**, **alpha**, **digit**, **cntrl**, **xdigit** or as the space character can be specified.

graph Define characters to be classified as printable characters, not including the space character.

In the POSIX locale, all characters in classes **alpha**, **digit**, and **punct** are included; no characters in class **cntrl** are included.

In a locale definition file, characters specified for the keywords **upper**, **lower**, **alpha**, **digit**, **xdigit**, and **punct** are automatically included in this class. No character specified for the keyword **cntrl** can be specified.

print Define characters to be classified as printable characters, including the space

character.

In the POSIX locale, all characters in class **graph** are included; no characters in class **cntrl** are included.

In a locale definition file, characters specified for the keywords **upper**, **lower**, **alpha**, **digit**, **xdigit**, **punct**, and the space character are automatically included in this class. No character specified for the keyword **cntrl** can be specified.

xdigit Define the characters to be classified as hexadecimal digits.

In the POSIX locale, only:

0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f

are included.

In a locale definition file, only the characters defined for the class **digit** can be specified, in contiguous ascending sequence by numerical value, followed by one or more sets of six characters representing the hexadecimal digits 10 to 15 inclusive, with each set in ascending order (for example **A**, **B**, **C**, **D**, **E**, **F**, **a**, **b**, **c**, **d**, **e**, **f**). The digits **0** to **9**, the upper-case letters **A** to **F** and the lower-case letters **a** to **f** of the portable character set are automatically included in this class.

The definition of character class **xdigit** requires that the characters included in character class **digit** be included here also.

blank Define characters to be classified as blank characters.
In the POSIX locale, only the space and tab characters are included.

In a locale definition file, the characters space and tab are automatically included in this class.

toupper Define the mapping of lower-case letters to upper-case letters.

In the POSIX locale, at a minimum, the 26 lower-case characters:

a b c d e f g h i j k l m n o p q r s t u v w x y z

are mapped to the corresponding 26 upper-case characters:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

In a locale definition file, the operand consists of character pairs, separated by semicolons. The characters in each character pair are separated by a comma and the pair enclosed by parentheses. The first character in each pair is the lower-case letter, the second the corresponding upper-case letter. Only characters specified for the keywords **lower** and **upper** can be specified. The lower-case letters **a** to **z**, and their corresponding upper-case letters **A** to **Z**, of the portable character set are automatically included in this mapping, but only when the **toupper** keyword is omitted from the locale definition.

tolower Define the mapping of upper-case letters to lower-case letters.
In the POSIX locale, at a minimum, the 26 upper-case characters

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

are mapped to the corresponding 26 lower-case characters:

a b c d e f g h i j k l m n o p q r s t u v w x y z

In a locale definition file, the operand consists of character pairs, separated by semicolons. The characters in each character pair are separated by a comma and the pair enclosed by parentheses. The first character in each pair is the upper-case letter, the second the corresponding lower-case letter. Only characters specified for the keywords **lower** and **upper** can be specified. If the **tolower** keyword is omitted from the locale definition, the mapping will be the reverse mapping of the one specified for **toupper**.

LC_COLLATE

The **LC_COLLATE** category provides a collation sequence definition for numerous utilities (such as **sort**(1), **uniq**(1), and so forth), regular expression matching (see **regex**(5)), and the **strcoll**(3C), **strxfrm**(3C), **wcscoll**(3I), and **wcsxfrm**(3I) functions.

A collation sequence definition defines the relative order between collating elements (characters and multi-character collating elements) in the locale. This order is expressed in terms of collation values; that is, by assigning each element one or more collation values (also known as collation weights). At least the following capabilities are provided:

1. **Multi-character collating elements**. Specification of multi-character collating elements (that is, sequences of two or more characters to be collated as an entity).
2. **User-defined ordering of collating elements**. Each collating element is assigned a collation value defining its order in the character (or basic) collation sequence. This ordering is used by regular expressions and pattern matching and, unless collation weights are explicity specified, also as the collation weight to be used in sorting.
3. **Multiple weights and equivalence classes**. Collating elements can be assigned one or more (up to the limit **{COLL_WEIGHTS_MAX})** collating weights for use in sorting. The first weight is hereafter referred to as the primary weight.
4. **One-to-Many mapping**. A single character is mapped into a string of collating elements.
5. **Equivalence class definition**. Two or more collating elements have the same collation value (primary weight).
6. **Ordering by weights**. When two strings are compared to determine their relative order, the two strings are first broken up into a series of collating elements; the elements in each successive pair of elements are then compared according to the relative primary weights for the elements. If equal, and more than one weight has been assigned, then the pairs of collating elements are recompared according to the relative subsequent weights, until either a pair of collating elements compare unequal or the weights are exhausted.

The following keywords are recognized in a collation sequence definition. They are described in detail in the following sections.

collating-element Define a collating-element symbol representing a multi-character

collating element. This keyword is optional.

collating-symbol Define a collating symbol for use in collation order statements. This keyword is optional.

order_start Define collation rules. This statement is followed by one or more collation order statements, assigning character collation values and collation weights to collating elements.

order_end Specify the end of the collation-order statements.

collating-element *keyword*

In addition to the collating elements in the character set, the **collating-element** keyword is used to define multi-character collating elements. The syntax is:

"**collating-element %s from \"%s\"\n",**<*collating-symbol*>,<*string*>

The <*collating-symbol*> operand is a symbolic name, enclosed between angle brackets (**<** and **>**), and must not duplicate any symbolic name in the current charmap file (if any), or any other symbolic name defined in this collation definition. The string operand is a string of two or more characters that collates as an entity. A <*collating-element*> defined via this keyword is only recognized with the **LC_COLLATE** category.

Example:

collating-element <**ch**> from "<**c**><**h**>"
collating-element <**e-acute**> from "<**acute**><**e**>"
collating-element <**ll**> from "**ll**"

collating-symbol *keyword*

This keyword will be used to define symbols for use in collation sequence statements; that is, between the **order_start** and the **order_end** keywords. The syntax is:

"**collating-symbol %s\n",**<*collating-symbol*>

The **<***collating-symbol***>** is a symbolic name, enclosed between angle brackets (**<** and **>**), and must not duplicate any symbolic name in the current charmap file (if any), or any other symbolic name defined in this collation definition.

A **collating-symbol** defined via this keyword is only recognized with the **LC_COLLATE** category.

Example:

collating-symbol <**UPPER_CASE**>
collating-symbol <**HIGH**>

The **collating-symbol** keyword defines a symbolic name that can be associated with a relative position in the character order sequence. While such a symbolic name does not represent any collating element, it can be used as a weight.

order_start *keyword*

The **order_start** keyword must precede collation order entries and also defines the number of weights for this collation sequence definition and other collation rules.

The syntax of the **order_start** keyword is:

"**order_start %s;%s;. . .;%s\n",***<sort-rules>*,*<sort-rules>*

The operands to the **order_start** keyword are optional. If present, the operands define rules to be applied when strings are compared. The number of operands define how many weights each element is assigned; if no operands are present, one **forward** operand is assumed. If present, the first operand defines rules to be applied when comparing strings using the first (primary) weight; the second when comparing strings using the second weight, and so on. Operands are separated by semicolons (**;**). Each operand consists of one or more collation directives, separated by commas (**,**). If the number of operands exceeds the **{COLL_WEIGHTS_MAX}** limit, the utility will issue a warning message. The following directives will be supported:

forward Specifies that comparison operations for the weight level proceed from start of string towards the end of string.

backward Specifies that comparison operations for the weight level proceed from end of string towards the beginning of string.

position Specifies that comparison operations for the weight level will consider the relative position of elements in the strings not subject to **IGNORE.** The string containing an element not subject to **IGNORE** after the fewest collating elements subject to **IGNORE** from the start of the compare will collate first. If both strings contain a character not subject to **IGNORE** in the same relative position, the collating values assigned to the elements will determine the ordering. In case of equality, subsequent characters not subject to **IGNORE** are considered in the same manner.

The directives **forward** and **backward** are mutually exclusive.

Example:

order_start forward;backward

If no operands are specified, a single **forward** operand is assumed.

The character (and collating element) order is defined by the order in which characters and elements are specified between the **order_start** and **order_end** keywords. This character order is used in range expressions in regular expressions (see **regex**(5)). Weights assigned to the characters and elements define the collation sequence; in the absence of weights, the character order is also the collation sequence.

The **position** keyword provides the capability to consider, in a compare, the relative position of characters not subject to **IGNORE.** As an example, consider the two strings ‘‘o-ring’’ and ‘‘or-ing’’. Assuming the hyphen is subject to **IGNORE** on the first pass, the two strings will compare equal, and the position of the hyphen is immaterial. On second pass, all characters except the hyphen are subject to **IGNORE,** and in the normal case the two strings would again compare equal. By taking position into account, the first collates before the second.

Collation Order

The **order_start** keyword is followed by collating identifier entries. The syntax for the collating element entries is

"%s %s;%s;...;%s\n"<*collating-identifier*>,<*weight*>,<*weight*>**,...**

Each *collating-identifier* consists of either a character described in **Locale Definition** above, a <*collating-element*>, a <*collating-symbol*>, an ellipsis, or the special symbol **UNDEFINED**. The order in which collating elements are specified determines the character order sequence, such that each collating element compares less than the elements following it. The NUL character compares lower than any other character.

A <*collating-element*> is used to specify multi-character collating elements, and indicates that the character sequence specified via the <*collating-element*> is to be collated as a unit and in the relative order specified by its place.

A <*collating-symbol*> is used to define a position in the relative order for use in weights. No weights are specified with a <*collating-symbol*>.

The ellipsis symbol specifies that a sequence of characters will collate according to their encoded character values. It is interpreted as indicating that all characters with a coded character set value higher than the value of the character in the preceding line, and lower than the coded character set value for the character in the following line, in the current coded character set, will be placed in the character collation order between the previous and the following character in ascending order according to their coded character set values. An initial ellipsis is interpreted as if the preceding line specified the NUL character, and a trailing ellipsis as if the following line specified the highest coded character set value in the current coded character set. An ellipsis is treated as invalid if the preceding or following lines do not specify characters in the current coded character set.

The symbol **UNDEFINED** is interpreted as including all coded character set values not specified explicitly or via the ellipsis symbol. Such characters are inserted in the character collation order at the point indicated by the symbol, and in ascending order according to their coded character set values. If no **UNDEFINED** symbol is specified, and the current coded character set contains characters not specified in this section, the utility will issue a warning message and place such characters at the end of the character collation order.

The optional operands for each collation-element are used to define the primary, secondary, or subsequent weights for the collating element. The first operand specifies the relative primary weight, the second the relative secondary weight, and so on. Two or more collation-elements can be assigned the same weight; they belong to the same *equivalence class* if they have the same primary weight. Collation behaves as if, for each weight level, elements subject to **IGNORE** are removed, unless the **position** collation directive is specified for the corresponding level with the **order_start** keyword. Then each successive pair of elements is compared according to the relative weights for the elements. If the two strings compare equal, the process is repeated for the next weight level, up to the limit {**COLL_WEIGHTS_MAX**}.

Weights are expressed as characters described in **Locale Definition** above, <*collating-symbol*>s, <*collating-element*>s, an ellipsis, or the special symbol **IGNORE.** A single character, a <*collating–symbol*> or a <*collating–element*> represent the relative position in the

character collating sequence of the character or symbol, rather than the character or characters themselves. Thus, rather than assigning absolute values to weights, a particular weight is expressed using the relative order value assigned to a collating element based on its order in the character collation sequence.

One-to-many mapping is indicated by specifying two or more concatenated characters or symbolic names. For example, if the character <**eszet**> is given the string "<**s**><**s**>" as a weight, comparisons are performed as if all occurrences of the character <**eszet**> are replaced by <**s**><**s**> (assuming that <**s**> has the collating weight <**s**>). If it is necessary to define <**eszet**> and <**s**><**s**> as an equivalence class, then a collating element must be defined for the string **ss**.

All characters specified via an ellipsis will by default be assigned unique weights, equal to the relative order of characters. Characters specified via an explicit or implicit **UNDEFINED** special symbol will by default be assigned the same primary weight (that is, belong to the same equivalence class). An ellipsis symbol as a weight is interpreted to mean that each character in the sequence has unique weights, equal to the relative order of their character in the character collation sequence. The use of the ellipsis as a weight is treated as an error if the collating element is neither an ellipsis nor the special symbol **UNDEFINED.**

The special keyword **IGNORE** as a weight indicates that when strings are compared using the weights at the level where **IGNORE** is specified, the collating element is ignored; that is, as if the string did not contain the collating element. In regular expressions and pattern matching, all characters that are subject to **IGNORE** in their primary weight form an equivalence class.

An empty operand is interpreted as the collating element itself.

For example, the order statement:

<**a**> <**a**>;<**a**>

is equal to:

<**a**>

An ellipsis can be used as an operand if the collating element was an ellipsis, and is interpreted as the value of each character defined by the ellipsis.

The collation order as defined in this section defines the interpretation of bracket expressions in regular expressions.

Example:

```
order_start     forward;backward
UNDEFINED       IGNORE;IGNORE
<LOW>
<space>         <LOW>;<space>
...             <LOW>;...
<a>             <a>;<a>
<a-acute>       <a>;<a-acute>
<a-grave>       <a>;<a-grave>
<A>             <a>;<A>
```

```
<A-acute>       <a>;<A-acute>
<A-grave>       <a>;<A-grave>
<ch>            <ch>;<ch>
<Ch>            <ch>;<Ch>
<s>             <s>;<s>
<eszet>         "<s><s>";"<eszet><eszet>"
order_end
```

This example is interpreted as follows:

1. The **UNDEFINED** means that all characters not specified in this definition (explicitly or via the ellipsis) are ignored for collation purposes; for regular expression purposes they are ordered first.
2. All characters between <**space**> and <**a**> have the same primary equivalence class and individual secondary weights based on their ordinal encoded values.
3. All characters based on the upper– or lower–case character **a** belong to the same primary equivalence class.
4. The multi-character collating element <**ch**> is represented by the collating symbol <**ch**> and belongs to the same primary equivalence class as the multi-character collating element <**Ch**>.

order_end *keyword*

The collating order entries must be terminated with an **order_end** keyword.

LC_MONETARY

The **LC_MONETARY** category defines the rules and symbols that are used to format monetary numeric information. This information is available through the **localeconv**(3C) function

The following items are defined in this category of the locale. The item names are the keywords recognized by the **localedef**(1) utility when defining a locale. They are also similar to the member names of the **lconv** structure defined in <**locale.h**>. The **localeconv** function returns **{CHAR_MAX}** for unspecified integer items and the empty string ("") for unspecified or size zero string items.

In a locale definition file the operands are strings. For some keywords, the strings can contain only integers. Keywords that are not provided, string values set to the empty string (""), or integer keywords set to **–1**, are used to indicate that the value is not available in the locale.

int_curr_symbol The international currency symbol. The operand is a four-character string, with the first three characters containing the alphabetic international currency symbol in accordance with those specified in the ISO 4217:1987 standard. The fourth character is the character used to separate the international currency symbol from the monetary quantity.

currency_symbol The string used as the local currency symbol.

mon_decimal_point The operand is a string containing the symbol that is used as the decimal delimiter (radix character) in monetary formatted

quantities. In contexts where standards (such as the ISO C standard) limit the **mon_decimal_point** to a single byte, the result of specifying a multi-byte operand is unspecified.

mon_thousands_sep The operand is a string containing the symbol that is used as a separator for groups of digits to the left of the decimal delimiter in formatted monetary quantities. In contexts where standards limit the **mon_thousands_sep** to a single byte, the result of specifying a multi-byte operand is unspecified.

mon_grouping Define the size of each group of digits in formatted monetary quantities. The operand is a sequence of integers separated by semicolons. Each integer specifies the number of digits in each group, with the initial integer defining the size of the group immediately preceding the decimal delimiter, and the following integers defining the preceding groups. If the last integer is not –1, then the size of the previous group (if any) will be repeatedly used for the remainder of the digits. If the last integer is –1, then no further grouping will be performed.

The following is an example of the interpretation of the **mon_grouping** keyword. Assuming that the value to be formatted is **123456789** and the **mon_thousands_sep** is **'**, then the following table shows the result. The third column shows the equivalent string in the ISO C standard that would be used by the **localeconv** function to accommodate this grouping.

mon_grouping	Formatted Value	ISO C String
3;–1	123456'789	"\3\177"
3	123'456'789	"\3"
3;2;–1	1234'56'789	"\3\2\177"
3;2	12'34'56'789	"\3\2"
–1	123456789	"\177"

In these examples, the octal value of **{CHAR_MAX}** is 177.

positive_sign A string used to indicate a non-negative-valued formatted monetary quantity.

negative_sign A string used to indicate a negative-valued formatted monetary quantity.

int_frac_digits An integer representing the number of fractional digits (those to the right of the decimal delimiter) to be written in a formatted monetary quantity using **int_curr_symbol**.

frac_digits An integer representing the number of fractional digits (those to the right of the decimal delimiter) to be written in a formatted monetary quantity using **currency_symbol**.

p_cs_precedes An integer set to 1 if the **currency_symbol** or **int_curr_symbol** precedes the value for a monetary quantity with a non-negative

value, and set to 0 if the symbol succeeds the value.

p_sep_by_space An integer set to 0 if no space separates the **currency_symbol** or **int_curr_symbol** from the value for a monetary quantity with a non-negative value, set to 1 if a space separates the symbol from the value, and set to 2 if a space separates the symbol and the sign string, if adjacent.

n_cs_precedes An integer set to 1 if the **currency_symbol** or **int_curr_symbol** precedes the value for a monetary quantity with a negative value, and set to 0 if the symbol succeeds the value.

n_sep_by_space An integer set to 0 if no space separates the **currency_symbol** or **int_curr_symbol** from the value for a monetary quantity with a negative value, set to 1 if a space separates the symbol from the value, and set to 2 if a space separates the symbol and the sign string, if adjacent.

p_sign_posn An integer set to a value indicating the positioning of the **positive_sign** for a monetary quantity with a non-negative value. The following integer values are recognized for both **p_sign_posn** and **n_sign_posn**:

- **0** Parentheses enclose the quantity and the **currency_symbol** or **int_curr_symbol**.
- **1** The sign string precedes the quantity and the **currency_symbol** or **int_curr_symbol**.
- **2** The sign string succeeds the quantity and the **currency_symbol** or **int_curr_symbol**.
- **3** The sign string precedes the **currency_symbol** or **int_curr_symbol**.
- **4** The sign string succeeds the **currency_symbol** or **int_curr_symbol**.

n_sign_posn An integer set to a value indicating the positioning of the **negative_sign** for a negative formatted monetary quantity.

The following table shows the result of various combinations:

		p_sep_by_space 2	1	0
p_cs_precedes = 1	**p_sign_posn** = 0	($1.25)	($ 1.25)	($1.25)
	p_sign_posn = 1	+ $1.25	+$ 1.25	+$1.25
	p_sign_posn = 2	$1.25 +	$ 1.25+	$1.25+
	p_sign_posn = 3	+ $1.25	+$ 1.25	+$1.25
	p_sign_posn = 4	$ +1.25	$+ 1.25	$+1.25
p_cs_precedes = 0	**p_sign_posn** = 0	(1.25 $)	(1.25 $)	(1.25$)
	p_sign_posn = 1	+1.25 $	+1.25 $	+1.25$
	p_sign_posn = 2	1.25$ +	1.25 $+	1.25$+
	p_sign_posn = 3	1.25+ $	1.25 +$	1.25+$
	p_sign_posn = 4	1.25$ +	1.25 $+	1.25$+

The monetary formatting definitions for the POSIX locale follow; the code listing depicting the **localedef**(1) input, the table representing the same information with the addition of **localeconv**(3C) and **nl_langinfo**(3C) formats. All values are unspecified in the POSIX locale.

```
LC_MONETARY
# This is the POSIX locale definition for
# the LC_MONETARY category.
#
int_curr_symbol         ""
currency_symbol         ""
mon_decimal_point       ""
mon_thousands_sep       ""
mon_grouping            -1
positive_sign           ""
negative_sign           ""
int_frac_digits         -1
p_cs_precedes           -1
p_sep_by_space          -1
n_cs_precedes           -1
n_sep_by_space          -1
p_sign_posn             -1
n_sign_posn             -1
#
END LC_MONETARY
```

The entry **n/a** indicates that the value is not available in the POSIX locale.

LC_NUMERIC

The **LC_NUMERIC** category defines the rules and symbols that will be used to format non-monetary numeric information. This information is available through the **localeconv**(3C) function.

The following items are defined in this category of the locale. The item names are the keywords recognized by the **localedef** utility when defining a locale. They are also similar to the member names of the *lconv* structure defined in <**locale.h**>. The **localeconv( )** function returns **{CHAR_MAX}** for unspecified integer items and the empty string ("") for unspecified or size zero string items.

In a locale definition file the operands are strings. For some keywords, the strings only can contain integers. Keywords that are not provided, string values set to the empty string (""), or integer keywords set to –1, will be used to indicate that the value is not available in the locale. The following keywords are recognized:

decimal_point The operand is a string containing the symbol that is used as the decimal delimiter (radix character) in numeric, non-monetary formatted quantities. This keyword cannot be omitted and cannot be set to the empty string. In contexts where standards limit the **decimal_point** to a single byte, the result of specifying a multi-byte operand is unspecified.

thousands_sep The operand is a string containing the symbol that is used as a separator for groups of digits to the left of the decimal delimiter in numeric, non-monetary formatted monetary quantities. In contexts where standards limit the **thousands_sep** to a single byte, the result of specifying a multi-byte operand is unspecified.

grouping Define the size of each group of digits in formatted non-monetary quantities. The operand is a sequence of integers separated by semicolons. Each integer specifies the number of digits in each group, with the initial integer defining the size of the group immediately preceding the decimal delimiter, and the following integers defining the preceding groups. If the last integer is not –1, then the size of the previous group (if any) will be repeatedly used for the remainder of the digits. If the last integer is –1, then no further grouping will be performed.

The non-monetary numeric formatting definitions for the POSIX locale follow; the code listing depicting the **localedef** input, the table representing the same information with the addition of **localeconv** values and **nl_langinfo** constants.

```
LC_NUMERIC
# This is the POSIX locale definition for
# the LC_NUMERIC category.
#
decimal_point       "<period>"
thousands_sep       ""
grouping            -1
#
END LC_NUMERIC
```

Item	POSIX locale Value	langinfo Constant	localeconv() Value	localedef Value
decimal_point	**"."**	**RADIXCHAR**	**"."**	**.**
thousands_sep	**n/a**	**THOUSEP**	**""**	**""**
grouping	**n/a**	**-**	**""**	**–1**

The entry **n/a** indicates that the value is not available in the POSIX locale.

LC_TIME

The **LC_TIME** category defines the interpretation of the field descriptors supported by **date**(1) and affects the behavior of the **strftime**(3C), **wcsftime**(3I), **strptime**(3C), and **nl_langinfo**(3C) functions. Because the interfaces for C-language access and locale definition differ significantly, they are described separately.

For locale definition, the following mandatory keywords are recognized:

abday Define the abbreviated weekday names, corresponding to the **%a** field descriptor (conversion specification in the **strftime( )**, **wcsftime( )**, and **strptime( )** functions). The operand consists of seven semicolon-separated strings, each surrounded by double-quotes. The first string is the abbreviated name of the day corresponding to Sunday, the second the abbreviated name of the day corresponding to Monday, and so on.

day Define the full weekday names, corresponding to the **%A** field descriptor. The operand consists of seven semicolon-separated strings, each surrounded by double-quotes. The first string is the full name of the day corresponding to Sunday, the second the full name of the day corresponding to Monday, and so on.

abmon Define the abbreviated month names, corresponding to the **%b** field descriptor. The operand consists of twelve semicolon-separated strings, each surrounded by double-quotes. The first string is the abbreviated name of the first month of the year (January), the second the abbreviated name of the second month, and so on.

mon Define the full month names, corresponding to the **%B** field descriptor. The operand consists of twelve semicolon-separated strings, each surrounded by double-quotes. The first string is the full name of the first month of the year (January), the second the full name of the second month, and so on.

d_t_fmt Define the appropriate date and time representation, corresponding to the **%c** field descriptor. The operand consists of a string, and can contain any combination of characters and field descriptors. In addition, the string can contain the escape sequences ****, **\a**, **\b**, **\f**, **\n**, **\r**, **\t**, **\v**.

date_fmt Define the appropriate date and time representation, corresponding to the **%C** field descriptor. The operand consists of a string, and can contain any combination of characters and field descriptors. In addition, the string can contain the escape sequences ****, **\a**, **\b**, **\f**, **\n**, **\r**, **\t**, **\v**.

d_fmt Define the appropriate date representation, corresponding to the **%x** field descriptor. The operand consists of a string, and can contain any combination of characters and field descriptors. In addition, the string can contain the escape sequences ****, **\a**, **\b**, **\f**, **\n**, **\r**, **\t**, **\v**.

t_fmt Define the appropriate time representation, corresponding to the **%X** field descriptor. The operand consists of a string, and can contain any combination of characters and field descriptors. In addition, the string can contain the escape sequences ****, **\a**, **\b**, **\f**, **\n**, **\r**, **\t**, **\v**.

am_pm Define the appropriate representation of the *ante meridiem* and *post meridiem* strings, corresponding to the **%p** field descriptor. The operand consists of two strings, separated by a semicolon, each surrounded by double-quotes. The first string represents the *ante meridiem* designation, the last string the *post meridiem* designation.

t_fmt_ampm Define the appropriate time representation in the 12-hour clock format with **am_pm**, corresponding to the **%r** field descriptor. The operand consists of a string and can contain any combination of characters and field descriptors. If the string is empty, the 12-hour format is not supported in the locale.

era Define how years are counted and displayed for each era in a locale. The operand consists of semicolon-separated strings. Each string is an era description segment with the format:

direction:*offset*:*start_date*:*end_date*:*era_name*:*era_format*

according to the definitions below. There can be as many era description segments as are necessary to describe the different eras.

The start of an era might not be the earliest point For example, the Christian era B.C. starts on the day before January 1, A.D. 1, and increases with earlier time.

direction Either a + or a – character. The + character indicates that years closer to the *start_date* have lower numbers than those closer to the *end_date*. The – character indicates that years closer to the *start_date* have higher numbers than those closer to the *end_date*.

offset The number of the year closest to the *start_date* in the era, corresponding to the **%Ey** field descriptor.

start_date A date in the form *yyyy* / *mm* / *dd*, where *yyyy*, *mm*, and *dd* are the year, month and day numbers respectively of the start of the era. Years prior to A.D. 1 are represented as negative numbers.

end_date The ending date of the era, in the same format as the *start_date*, or one of the two special values –∗ or +∗. The value –∗ indicates that the ending date is the beginning of time. The value +∗ indicates that the ending date is the end

of time.

era_name A string representing the name of the era, corresponding to the **%EC** field descriptor.

era_format A string for formatting the year in the era, corresponding to the **%EY** field descriptor.

era_d_fmt Define the format of the date in alternative era notation, corresponding to the **%Ex** field descriptor.

era_t_fmt Define the locale's appropriate alternative time format, corresponding to the **%EX** field descriptor.

era_d_t_fmt Define the locale's appropriate alternative date and time format, corresponding to the **%Ec** field descriptor.

alt_digits Define alternative symbols for digits, corresponding to the **%O** field descriptor modifier. The operand consists of semicolon-separated strings, each surrounded by double-quotes. The first string is the alternative symbol corresponding with zero, the second string the symbol corresponding with one, and so on. Up to 100 alternative symbol strings can be specified. The **%O** modifier indicates that the string corresponding to the value specified via the field descriptor will be used instead of the value.

LC_TIME *C-language Access*

The following information can be accessed. These correspond to constants defined in <**langinfo.h**> and used as arguments to the **nl_langinfo**(3C) function.

ABDAY_*x* The abbreviated weekday names (for example Sun), where *x* is a number from 1 to 7.

DAY_*x* The full weekday names (for example Sunday), where *x* is a number from 1 to 7.

ABMON_*x* The abbreviated month names (for example Jan), where *x* is a number from 1 to 12.

MON_*x* The full month names (for example January), where *x* is a number from 1 to 12.

D_T_FMT The appropriate date and time representation.

D_FMT The appropriate date representation.

T_FMT The appropriate time representation.

AM_STR The appropriate ante-meridiem affix.

PM_STR The appropriate post-meridiem affix.

T_FMT_AMPM The appropriate time representation in the 12-hour clock format with **AM_STR** and **PM_STR.**

ERA The era description segments, which describe how years are counted and displayed for each era in a locale. Each era description segment has the format:

direction:*offset*:*start_date*:*end_date*:*era_name*:*era_format*

according to the definitions below. There will be as many era description segments as are necessary to describe the different eras. Era description segments are separated by semicolons.

The start of an era might not be the earliest point For example, the Christian era B.C. starts on the day before January 1, A.D. 1, and increases with earlier time.

direction Either a + or a – character. The + character indicates that years closer to the start_date have lower numbers than those closer to the *end_date*. The – character indicates that years closer to the start_date have higher numbers than those closer to the *end_date*.

offset The number of the year closest to the start_date in the era.

start_date A date in the form *yyyy* / *mm* / *dd*, where *yyyy*, *mm*, and *dd* are the year, month and day numbers respectively of the start of the era. Years prior to AD 1 are represented as negative numbers.

end_date The ending date of the era, in the same format as the *start_date*, or one of the two special values –∗ or +∗. The value –∗ indicates that the ending date is the beginning of time. The value +∗ indicates that the ending date is the end of time.

era_name The era, corresponding to the **%EC** conversion specification.

era_format The format of the year in the era, corresponding to the **%EY** conversion specification.

ERA_D_FMT The era date format.

ERA_T_FMT The locale's appropriate alternative time format, corresponding to the **%EX** field descriptor.

ERA_D_T_FMT The locale's appropriate alternative date and time format, corresponding to the **%Ec** field descriptor.

ALT_DIGITS The alternative symbols for digits, corresponding to the **%O** conversion specification modifier. The value consists of semicolon-separated symbols. The first is the alternative symbol corresponding to zero, the second is the symbol corresponding to one, and so on. Up to 100 alternative symbols may be specified.

The following table displays the correspondence between the items described above and the conversion specifiers used by **date**(1) and the **strftime**(3C), **wcsftime**(3I), and **strptime**(3C) functions.

localedef Keyword	langinfo Constant	Conversion Specifier
abday	**ABDAY_***x*	**%a**
day	**DAY_***x*	**%A**
abmon	**ABMON_***x*	**%b**
mon	**MON**	**%B**
d_t_fmt	**D_T_FMT**	**%c**
date_fmt	**DATE_FMT**	**%C**
d_fmt	**D_FMT**	**%x**
t_fmt	**T_FMT**	**%X**
am_pm	**AM_STR**	**%p**
am_pm	**PM_STR**	**%p**
t_fmt_ampm	**T_FMT_AMPM**	**%r**
era	**ERA**	**%EC, %Ey, %EY**
era_d_fmt	**ERA_D_FMT**	**%Ex**
era_t_fmt	**ERA_T_FMT**	**%EX**
era_d_t_fmt	**ERA_D_T_FMT**	**%Ec**
alt_digits	**ALT_DIGITS**	**%O**

LC_TIME *General Information*

Although certain of the field descriptors in the POSIX locale (such as the name of the month) are shown with initial capital letters, this need not be the case in other locales. Programs using these fields may need to adjust the capitalization if the output is going to be used at the beginning of a sentence.

The **LC_TIME** descriptions of **abday**, **day**, **mon**, and **abmon** imply a Gregorian style calendar (7-day weeks, 12-month years, leap years, and so forth). Formatting time strings for other types of calendars is outside the scope of this document set.

As specified under **date** in **Locale Definition** and **strftime**(3C), the field descriptors corresponding to the optional keywords consist of a modifier followed by a traditional field descriptor (for instance **%Ex**). If the optional keywords are not supported by the implementation or are unspecified for the current locale, these field descriptors are treated as the traditional field descriptor. For instance, assume the following keywords:

```
alt_digits      "0th";"1st";"2nd";"3rd";"4th";"5th"; \
                "6th";"7th";"8th";"9th";"10th"
d_fmt           "The %Od day of %B in %Y"
```

On 7/4/1776, the **%x** field descriptor would result in “The 4th day of July in 1776” while 7/14/1789 would come out as “The 14 day of July in 1789” It can be noted that the above example is for illustrative purposes only; the **%O** modifier is primarily intended to provide for Kanji or Hindi digits in **date** formats.

LC_MESSAGES

The **LC_MESSAGES** category defines the format and values for affirmative and negative responses.

The following keywords are recognized as part of the locale definition file. The **nl_langinfo**(3C) function accepts upper-case versions of the first four keywords.

yesexpr The operand consists of an extended regular expression (see **regex**(5)) that describes the acceptable affirmative response to a question expecting an affirmative or negative response.

noexpr The operand consists of an extended regular expression that describes the acceptable negative response to a question expecting an affirmative or negative response.

yesstr The operand consists of a fixed string (not a regular expression) that can be used by an application for composition of a message that lists an acceptable affirmative response, such as in a prompt.

nostr The operand consists of a fixed string that can be used by an application for composition of a message that lists an acceptable negative response.

The format and values for affirmative and negative responses of the POSIX locale follow; the code listing depicting the **localedef** input, the table representing the same information with the addition of **nl_langinfo( )** constants.

```
LC_MESSAGES
# This is the POSIX locale definition for
# the LC_MESSAGES category.
#
yesexpr "<circumflex><left-square-bracket><y><Y><right-square-bracket>"
#
noexpr  "<circumflex><left-square-bracket><n><N><right-square-bracket>"
#
yesstr          "yes"
nostr           "no"
END LC_MESSAGES
```

localedef Keyword	langinfo Constant	POSIX Locale Value
yesexpr	YESEXPR	"^[yY]"
noexpr	NOEXPR	"^[nN]"
yesstr	YESSTR	"yes"
nostr	NOSTR	"no"

SEE ALSO **date**(1), **locale**(1), **localedef**(1), **sort**(1), **uniq**(1), **localeconv**(3C), **nl_langinfo**(3C), **setlocale**(3C), **strcoll**(3C), **strftime**(3C), **strptime**(3C), **strxfrm**(3C), **wcscoll**(3I), **wcsftime**(3I), **wcsxfrm**(3I), **charmap**(5), **regex**(5)

NAME

man – macros to format Reference Manual pages

SYNOPSIS

nroff –**man** *filename*...

troff –**man** *filename*...

DESCRIPTION

These macros are used to lay out the reference pages in this manual. Note: if *filename* contains format input for a preprocessor, the commands shown above must be piped through the appropriate preprocessor. This is handled automatically by the **man**(1) command. See the ''Conventions'' section.

Any text argument *t* may be zero to six words. Quotes may be used to include SPACE characters in a "word". If *text* is empty, the special treatment is applied to the next input line with text to be printed. In this way **.I** may be used to italicize a whole line, or **.SB** may be used to make small bold letters.

A prevailing indent distance is remembered between successive indented paragraphs, and is reset to default value upon reaching a non-indented paragraph. Default units for indents *i* are ens.

Type font and size are reset to default values before each paragraph, and after processing font and size setting macros.

These strings are predefined by –**man**:

*R	'®', '(Reg)' in **nroff**.
*S	Change to default type size.

Requests

* n.t.l. = next text line; p.i. = prevailing indent

Request	*Cause Break*	*If no Argument*	*Explanation*
.B *t*	no	*t*=n.t.l.*	Text is in bold font.
.BI *t*	no	*t*=n.t.l.	Join words, alternating bold and italic.
.BR *t*	no	*t*=n.t.l.	Join words, alternating bold and roman.
.DT	no	.5i 1i...	Restore default tabs.
.HP *i*	yes	*i*=p.i.*	Begin paragraph with hanging indent. Set prevailing indent to *i*.
.I *t*	no	*t*=n.t.l.	Text is italic.
.IB *t*	no	*t*=n.t.l.	Join words, alternating italic and bold.
.IP *x i*	yes	*x*=""	Same as **.TP** with tag *x*.
.IR *t*	no	*t*=n.t.l.	Join words, alternating italic and roman.
.IX *t*	no	-	Index macro, for SunSoft internal use.
.LP	yes	-	Begin left-aligned paragraph. Set prevailing indent to .5i.
.P	yes	-	Same as **.LP**.
.PD *d*	no	*d*=.4v	Set vertical distance between paragraphs.
.PP	yes	-	Same as **.LP**.
.RE	yes	-	End of relative indent. Restores prevailing indent.
.RB *t*	no	*t*=n.t.l.	Join words, alternating roman and bold.

.RI *t*	no	*t*=n.t.l.	Join words, alternating roman and italic.
.RS *i*	yes	*i*=p.i.	Start relative indent, increase indent by *i*. Sets prevailing indent to .5i for nested indents.
.SB *t*	no	-	Reduce size of text by 1 point, make text bold.
.SH *t*	yes	-	Section Heading.
.SM *t*	no	*t*=n.t.l.	Reduce size of text by 1 point.
.SS *t*	yes	*t*=n.t.l.	Section Subheading.
.TH *n s d f m*	yes	-	Begin reference page *n*, of of section *s*; *d* is the date of the most recent change. If present, *f* is the left page footer; *m* is the main page (center) header. Sets prevailing indent and tabs to .5i.
.TP *i*	yes	*i*=p.i.	Begin indented paragraph, with the tag given on the next text line. Set prevailing indent to *i*.
.TX *t p*	no	-	Resolve the title abbreviation *t*; join to punctuation mark (or text) *p*.

Conventions

When formatting a manual page, **man** examines the first line to determine whether it requires special processing. For example a first line consisting of:

'\" t

indicates that the manual page must be run through the **tbl**(1) preprocessor.

A typical manual page for a command or function is laid out as follows:

.TH *title* [1-9]

The name of the command or function, which serves as the title of the manual page. This is followed by the number of the section in which it appears.

.SH NAME

The name, or list of names, by which the command is called, followed by a dash and then a one-line summary of the action performed. All in roman font, this section contains no **troff**(1) commands or escapes, and no macro requests. It is used to generate the **windex** database, which is used by the **whatis**(1) command.

.SH SYNOPSIS

Commands:

The syntax of the command and its arguments, as typed on the command line. When in boldface, a word must be typed exactly as printed. When in italics, a word can be replaced with an argument that you supply. References to bold or italicized items are not capitalized in other sections, even when they begin a sentence.

Syntactic symbols appear in roman face:

[ ] An argument, when surrounded by brackets is optional.

| Arguments separated by a vertical bar are exclusive. You can supply only one item from such a list.

... Arguments followed by an ellipsis can be repeated. When an ellipsis follows a bracketed set, the expression within the brackets can be repeated.

Functions:

If required, the data declaration, or **#include** directive, is shown first, followed by the function declaration. Otherwise, the function declaration is shown.

.SH DESCRIPTION

A narrative overview of the command or function's external behavior. This includes how it interacts with files or data, and how it handles the standard input, standard output and standard error. Internals and implementation details are normally omitted. This section attempts to provide a succinct overview in answer to the question, "what does it do?"

Literal text from the synopsis appears in constant width, as do literal filenames and references to items that appear elsewhere in the reference manuals. Arguments are italicized.

If a command interprets either subcommands or an input grammar, its command interface or input grammar is normally described in a USAGE section, which follows the OPTIONS section. The DESCRIPTION section only describes the behavior of the command itself, not that of subcommands.

.SH OPTIONS

The list of options along with a description of how each affects the command's operation.

.SH RETURN VALUES

A list of the values the library routine will return to the calling program and the conditions that cause these values to be returned.

.SH EXIT STATUS

A list of the values the utility will return to the calling program or shell, and the conditions that cause these values to be returned.

.SH FILES

A list of files associated with the command or function.

.SH SEE ALSO

A comma-separated list of related manual pages, followed by references to other published materials.

.SH DIAGNOSTICS

A list of diagnostic messages and an explanation of each.

.SH BUGS
A description of limitations, known defects, and possible problems associated with the command or function.

FILES

/usr/share/lib/tmac/an
/usr/share/man/windex

SEE ALSO

man(1), **nroff**(1), **troff**(1), **whatis**(1)

Dale Dougherty and Tim O'Reilly, *Unix Text Processing*

NAME

mansun – macros to format Reference Manual pages

SYNOPSIS

nroff –**mansun** *filename*...

troff –**mansun** *filename*...

DESCRIPTION

These macros are used to lay out the reference pages in this manual. Note: if *filename* contains format input for a preprocessor, the commands shown above must be piped through the appropriate preprocessor. This is handled automatically by **man**(1). See the ''Conventions'' section.

Any text argument *t* may be zero to six words. Quotes may be used to include SPACE characters in a "word". If *text* is empty, the special treatment is applied to the next input line with text to be printed. In this way **.I** may be used to italicize a whole line, or **.SB** may be used to make small bold letters.

A prevailing indent distance is remembered between successive indented paragraphs, and is reset to default value upon reaching a non-indented paragraph. Default units for indents *i* are ens.

Type font and size are reset to default values before each paragraph, and after processing font and size setting macros.

These strings are predefined by **–mansun**:

***R** '®', '(Reg)' in **nroff**.
***S** Change to default type size.

Requests

* n.t.l. = next text line; p.i. = prevailing indent

Request	*Cause Break*	*If no Argument*	*Explanation*
.B *t*	no	*t*=n.t.l.*	Text is in bold font.
.BI *t*	no	*t*=n.t.l.	Join words, alternating bold and italic.
.BR *t*	no	*t*=n.t.l.	Join words, alternating bold and Roman.
.DT	no	.5i 1i...	Restore default tabs.
.HP *i*	yes	*i*=p.i.*	Begin paragraph with hanging indent. Set prevailing indent to *i*.
.I *t*	no	*t*=n.t.l.	Text is italic.
.IB *t*	no	*t*=n.t.l.	Join words, alternating italic and bold.
.IP *x i*	yes	*x*=""	Same as **.TP** with tag *x*.
.IR *t*	no	*t*=n.t.l.	Join words, alternating italic and Roman.
.IX *t*	no	-	Index macro, for SunSoft internal use.
.LP	yes	-	Begin left-aligned paragraph. Set prevailing indent to .5i.
.P	yes	-	Same as **.LP**.
.PD *d*	no	*d*=.4v	Set vertical distance between paragraphs.
.PP	yes	-	Same as **.LP**.
.RE	yes	-	End of relative indent. Restores prevailing indent.
.RB *t*	no	*t*=n.t.l.	Join words, alternating Roman and bold.

.RI *t*	no	*t*=n.t.l.	Join words, alternating Roman and italic.
.RS *i*	yes	*i*=p.i.	Start relative indent, increase indent by *i*. Sets prevailing indent to .5i for nested indents.
.SB *t*	no	-	Reduce size of text by 1 point, make text bold.
.SH *t*	yes	-	Section Heading.
.SM *t*	no	*t*=n.t.l.	Reduce size of text by 1 point.
.SS *t*	yes	*t*=n.t.l.	Section Subheading.
.TH *n s d f m*	yes	-	Begin reference page *n*, of of section *s*; *d* is the date of the most recent change. If present, *f* is the left page footer; *m* is the main page (center) header. Sets prevailing indent and tabs to .5i.
.TP *i*	yes	*i*=p.i.	Begin indented paragraph, with the tag given on the next text line. Set prevailing indent to *i*.
.TX *t p*	no	-	Resolve the title abbreviation *t*; join to punctuation mark (or text) *p*.

Conventions

When formatting a manual page, **mansun** examines the first line to determine whether it requires special processing. For example a first line consisting of:

'\" t

indicates that the manual page must be run through the **tbl**(1) preprocessor.

A typical manual page for a command or function is laid out as follows:

.TH *title* [1-8]

The name of the command or function, which serves as the title of the manual page. This is followed by the number of the section in which it appears.

.SH NAME

The name, or list of names, by which the command is called, followed by a dash and then a one-line summary of the action performed. All in Roman font, this section contains no **troff**(1) commands or escapes, and no macro requests. It is used to generate the **windex** database, which is used by the **whatis**(1) command.

.SH SYNOPSIS

Commands:

The syntax of the command and its arguments, as typed on the command line. When in boldface, a word must be typed exactly as printed. When in italics, a word can be replaced with an argument that you supply. References to bold or italicized items are not capitalized in other sections, even when they begin a sentence.

Syntactic symbols appear in Roman face:

[ ] An argument, when surrounded by brackets is optional.

| Arguments separated by a vertical bar are exclusive. You can supply only one item from such a list.

... Arguments followed by an ellipsis can be repeated. When an ellipsis follows a bracketed set, the expression within the brackets can be repeated.

Functions:

If required, the data declaration, or **#include** directive, is shown first, followed by the function declaration. Otherwise, the function declaration is shown.

.SH DESCRIPTION

A narrative overview of the command or function's external behavior. This includes how it interacts with files or data, and how it handles the standard input, standard output and standard error. Internals and implementation details are normally omitted. This section attempts to provide a succinct overview in answer to the question, "what does it do?"

Literal text from the synopsis appears in constant width, as do literal filenames and references to items that appear elsewhere in the reference manuals. Arguments are italicized.

If a command interprets either subcommands or an input grammar, its command interface or input grammar is normally described in a USAGE section, which follows the OPTIONS section. The DESCRIPTION section only describes the behavior of the command itself, not that of subcommands.

.SH OPTIONS

The list of options along with a description of how each affects the command's operation.

.SH FILES

A list of files associated with the command or function.

.SH SEE ALSO

A comma-separated list of related manual pages, followed by references to other published materials.

.SH DIAGNOSTICS

A list of diagnostic messages and an explanation of each.

.SH BUGS

A description of limitations, known defects, and possible problems associated with the command or function.

FILES

/usr/share/lib/tmac/ansun
/usr/share/man/windex

SEE ALSO

man(1), **nroff**(1), **troff**(1), **whatis**(1)

Dale Dougherty and Tim O'Reilly, *Unix Text Processing*

NAME

math – math functions and constants

SYNOPSIS

#include <math.h>

DESCRIPTION

This file contains declarations of all the functions in the Math Library (described in Section 3M), as well as various functions in the C Library (Section 3C) that return floating-point values.

It defines the structure and constants used by the **matherr**(3M) error-handling mechanisms, including the following constant used as a error-return value:

HUGE The maximum value of a single-precision floating-point number.

The following mathematical constants are defined for user convenience:

M_E The base of natural logarithms (*e*).

M_LOG2E The base-2 logarithm of *e*.

M_LOG10E The base-10 logarithm of *e*.

M_LN2 The natural logarithm of 2.

M_LN10 The natural logarithm of 10.

M_PI π, the ratio of the circumference of a circle to its diameter.

M_PI_2 $\pi / 2$.

M_PI_4 $\pi / 4$.

M_1_PI $1 / \pi$.

M_2_PI $2 / \pi$.

M_2_SQRTPI $2 / \sqrt{\pi}$.

M_SQRT2 The positive square root of 2.

M_SQRT1_2 The positive square root of 1/2.

The following mathematical constants are also defined in this header file:

MAXFLOAT The maximum value of a non-infinite single-precision floating point number.

HUGE_VAL positive infinity.

For the definitions of various machine-dependent constants see **values**(5).

SEE ALSO

intro(3), **matherr**(3M), **values**(5)

NAME

me – macros for formatting papers

SYNOPSIS

nroff –me [*options*] *filename* . . .
troff –me [*options*] *filename* . . .

DESCRIPTION

This package of **nroff** and **troff** macro definitions provides a canned formatting facility for technical papers in various formats. When producing 2-column output on a terminal, filter the output through **col**(1).

The macro requests are defined below. Many **nroff** and **troff** requests are unsafe in conjunction with this package, however, these requests may be used with impunity after the first .pp:

.bp begin new page
.br break output line here
.sp *n* insert n spacing lines
.ls *n* (line spacing) n=1 single, n=2 double space
.na no alignment of right margin
.ce *n* center next n lines
.ul *n* underline next n lines
.sz +*n* add n to point size

Output of the **eqn**(1), **neqn**(1), **refer**(1), and **tbl**(1) preprocessors for equations and tables is acceptable as input.

REQUESTS

In the following list, "initialization" refers to the first **.pp, .lp**, **.ip**, **.np**, **.sh**, or **.uh** macro. This list is incomplete.

Request	Initial Value	Cause Break	Explanation
.(c	-	yes	Begin centered block.
.(d	-	no	Begin delayed text.
.(f	-	no	Begin footnote.
.(l	-	yes	Begin list.
.(q	-	yes	Begin major quote.
.(x*x*	-	no	Begin indexed item in index *x*.
.(z	-	no	Begin floating keep.
.)c	-	yes	End centered block.
.)d	-	yes	End delayed text.
.)f	-	yes	End footnote.
.)l	-	yes	End list.
.)q	-	yes	End major quote.
.)x	-	yes	End index item.
.)z	-	yes	End floating keep.
.++ *m H*	-	no	Define paper section. *m* defines the part of the paper, and can be **C** (chapter), **A** (appendix), **P** (preliminary, for instance,

			abstract, table of contents, etc.), **B** (bibliography), **RC** (chapters renumbered from page one each chapter), or **RA** (appendix renumbered from page one).
.+c *T*	-	yes	Begin chapter (or appendix, etc., as set by **.++**). *T* is the chapter title.
.1c	1	yes	One column format on a new page.
.2c	1	yes	Two column format.
.EN	-	yes	Space after equation produced by **eqn** or **neqn**.
.EQ *x y*	-	yes	Precede equation; break out and add space. Equation number is *y*. The optional argument *x* may be *I* to indent equation (default), *L* to left-adjust the equation, or *C* to center the equation.
.GE	-	yes	End *gremlin* picture.
.GS	-	yes	Begin *gremlin* picture.
.PE	-	yes	End **pic** picture.
.PS	-	yes	Begin **pic** picture.
.TE	-	yes	End table.
.TH	-	yes	End heading section of table.
.TS *x*	-	yes	Begin table; if *x* is *H* table has repeated heading.
.ac *A N*	-	no	Set up for ACM style output. *A* is the Author's name(s), *N* is the total number of pages. Must be given before the first initialization.
.b *x*	no	no	Print *x* in boldface; if no argument switch to boldface.
.ba *+n*	0	yes	Augments the base indent by *n*. This indent is used to set the indent on regular text (like paragraphs).
.bc	no	yes	Begin new column.
.bi *x*	no	no	Print *x* in bold italics (nofill only).
.bu	-	yes	Begin bulleted paragraph.
.bx *x*	no	no	Print *x* in a box (nofill only).
.ef *´x´y´z*	´´´´´	no	Set even footer to *x y z*.
.eh *´x´y´z*	´´´´´	no	Set even header to *x y z*.
.fo *´x´y´z*	´´´´´	no	Set footer to *x y z*.
.hx	-	no	Suppress headers and footers on next page.
.he *´x´y´z*	´´´´´	no	Set header to *x y z*.

.hl	-	yes	Draw a horizontal line.
.i *x*	no	no	Italicize *x*; if *x* missing, italic text follows.
.ip *x y*	no	yes	Start indented paragraph, with hanging tag *x*. Indentation is *y* ens (default 5).
.lp	yes	yes	Start left-blocked paragraph.
.lo	-	no	Read in a file of local macros of the form **.***x*. Must be given before initialization.
.np	1	yes	Start numbered paragraph.
.of ´*x*´*y*´*z*	´´´´´	no	Set odd footer to x y z.
.oh ´*x*´*y*´*z*	´´´´´	no	Set odd header to x y z.
.pd	-	yes	Print delayed text.
.pp	no	yes	Begin paragraph. First line indented.
.r	yes	no	Roman text follows.
.re	-	no	Reset tabs to default values.
.sc	no	no	Read in a file of special characters and diacritical marks. Must be given before initialization.
.sh *n x*	-	yes	Section head follows, font automatically bold. *n* is level of section, *x* is title of section.
.sk	no	no	Leave the next page blank. Only one page is remembered ahead.
.sm *x*	-	*no*	*Set x in a smaller pointsize.*
.sz +*n*	10p	no	Augment the point size by *n* points.
.th	no	no	Produce the paper in thesis format. Must be given before initialization.
.tp	no	yes	Begin title page.
.u *x*	-	no	Underline argument (even in **troff**). (Nofill only).
.uh	-	yes	Like **.sh** but unnumbered.
.xp *x*	-	no	Print index *x*.

FILES

/usr/share/lib/tmac/e
/usr/share/lib/tmac/∗.me

SEE ALSO

eqn(1), **nroff**(1), **refer**(1), **tbl**(1), **troff**(1)

NAME

mm – text formatting (memorandum) macros

SYNOPSIS

nroff –**mm** [*options*] *filename* . . .

troff –**mm** [*options*] *filename* . . .

AVAILABILITY

SUNWdoc

DESCRIPTION

This package of **nroff**(1) and **troff**(1) macro definitions provides a formatting facility for various styles of articles, theses, and books. When producing 2-column output on a terminal or lineprinter, or when reverse line motions are needed, filter the output through **col**(1). All external –**mm** macros are defined below.

Note: this –**mm** macro package is an extended version written at Berkeley and is a superset of the standard –**mm** macro packages as supplied by Bell Labs. Some of the Bell Labs macros have been removed; for instance, it is assumed that the user has little interest in producing headers stating that the memo was generated at Whippany Labs.

Many **nroff** and **troff** requests are unsafe in conjunction with this package. However, the first four requests below may be used with impunity after initialization, and the last two may be used even before initialization:

.bp begin new page
.br break output line
.sp *n* insert n spacing lines
.ce *n* center next n lines
.ls *n* line spacing: *n*=**1** single, *n*=**2** double space
.na no alignment of right margin

Font and point size changes with **\f** and **\s** are also allowed; for example, **\fIword\fR** will italicize *word*. Output of the **tbl**(1), **eqn**(1) and **refer**(1) preprocessors for equations, tables, and references is acceptable as input.

REQUESTS

Macro Name	Initial Value	Break? Reset?	Explanation
.1C	on	y,y	one column format on a new page
.2C [*l*]	–	y,y	two column format *l*=line length
.AE	–	y	end abstract
.AL [*t*] [*i*] [*s*]	*t*=**1**;*i*=**.Li**;*s*=**0**	y	Start automatic list type *t*=[**1**,**A**,**a**,**I**,**i**] **1**=arabic numbers; **A**=uppercase lett **a**=lowercase letters; **I**=uppercase Ro numerals; **i**=lowercase Roman nume indentation *i*; separation *s*
.AS *m* [*n*]	*n*=**0**	y	begin abstract
.AU	–	y	author's name
.AV *x*	–	y	signature and date line of verifier *x*
.B *x*	–	n	embolden *x*; if no *x*, switch to boldfa
.BE	–	y	end block text

.BI *x y*	–	n	embolden *x* and underline *y*
.BL	–	y	bullet list
.BR *x y*	–	n	embolden *x* and use Roman font for
.BS	–	n	start block text
.CN	–	y	same as **.DE** (**nroff**)
.CS	–	y	cover sheet
.CW	–	n	same as **.DS I** (**nroff**)
.DE	–	y	end display
.DF [*p*] [*f*] [*rp*]	*p*=**L**;*f*=**N**	y	start floating display; position *p*=[**L**,**C** **L**=left; **I**=indent; **C**=center; **CB**=cente fill *f*=[**N**,**Y**]; right position *rp* (fill only
.DL [*i*] [*s*]	–	y	start dash list
.DS [*p*] [*f*] [*rp*]	*p*=**L**;*f*=**N**	y	begin static display (see **.DF** for argument descriptions)
.EC *x* [*n*]	*n*=**1**	y	equation title; equation *x*; number *n*
.EF *x*	–	n	even footer appears at the bottom of even-numbered pages; *x*="*l*'*c*'*r*" *l*=left; *c*=center; *r*=right
.EH *x*	–	n	even header appears at the top of even-numbered pages; *x*="*l*'*c*'*r*" *l*=left; *c*=center; *r*=right
.EN	–	y	end displayed equation produced by
.EQ	–	y	break out equation produced by **eqn**
.EX *x* [*n*]	*n*=**1**	y	exhibit title; exhibit *x* number *n*
.FD [*f*] [*r*]	*f*=**10**;*r*=**1**	n	set footnote style format *f*=[**0-11**]; renumber *r*=[**0**,**1**]
.FE	–	y	end footnote
.FG *x* [*n*]	*n*=**1**	y	figure title; figure *x*; number *n*
.FS	–	n	start footnote
.H *l* [*t*]	–	y	produce numbered heading level *l*=[**1-7**]; title *t*
.HU *t*	–	y	produce unnumbered heading; title *t*
.I *x*	–	n	underline *x*
.IB *x y*	–	n	underline *x* and embolden *y*
.IR *x y*	–	n	underline *x* and use Roman font on *y*
.LE [*s*]	*s*=**0**	y	end list; separation *s*
.LI [*m*] [*p*]	–	y	start new list item; mark *m* prefix *p* (mark only)
.ML *m* [*i*] [*s*]	*s*=**0**	y	start marked list; mark *m* indentation *i*; separation *s*=[**0**,**1**]
.MT *x*		y	memo title; title *x*
.ND *x*		n	no date in page footer; *x* is date on co
.NE	–	y	end block text
.NS	–	y	start block text

.OF *x*	–	n	odd footer appears at the bottom of odd-numbered pages; *x*="*l*'*c*'*r*" *l*=left; *c*=center; *r*=right
.OF *x*	–	n	odd header appears at the top of odd-numbered pages; *x*="*l*'*c*'*r*" *l*=left; *c*=center; *r*=right
.OP	–	y	skip to the top of an odd-number pag
.P [*t*]	*t*=**0**	y,y	begin paragraph; *t*=[**0**,**1**] **0**=justified; **1**=indented
.PF *x*	–	n	page footer appears at the bottom of every page; *x*="*l*'*c*'*r*" *l*=left; *c*=center; *r*=right
.PH *x*	–	n	page header appears at the top of every page; *x*="*l*'*c*'*r*" *l*=left; *c*=center; *r*=right
.R	on	n	return to Roman font
.RB *x y*	–	n	use Roman on *x* and embolden *y*
.RI *x y*	–	n	use Roman on *x* and underline *y*
.RP *x*	-	y,y	released paper format ? *x*=no stops title on first
.RS	5n	y,y	right shift: start level of relative inde
.S *m n*	–	n	set character point size & vertical spa character point size *m*; vertical space
.SA *x*	*x*=**1**	n	justification; *x*=[**0**,**1**]
.SK *x*	–	y	skip *x* pages
.SM	–	n	smaller; decrease point size by 2
.SP [*x*]	–	y	leave *x* blank lines
.TB *x* [*n*]	*n*=**1**	y	table title; table *x*; number *n*
.TC	–	y	print table of contents (put at end of input file)
.TE	–	y	end of table processed by tbl
.TH	–	y	end multi-page header of table
.TL	–	n	title in boldface and two points large
.TM	–	n	UC Berkeley thesis mode
.TP *i*	y	y	*i*=p.i. Begin indented paragraph, with the tag given on the next text lin Set prevailing indent to *i*.
.TS *x*	–	y,y	begin table; if *x*=**H** table has multi-page header
x P 0 (view:<–y>Contents) link-dest			
.TY	–	y	display centered title **CONTENTS**
.VL *i* [*m*] [*s*]	*m*=**0**;*s*=**0**	y	start variable-item list; indentation *i* mark-indentation *m*; separation *s*

REGISTERS

Formatting distances can be controlled in **–mm** by means of built-in number registers. For example, this sets the line length to 6.5 inches:

.nr LL 6.5i

Here is a table of number registers and their default values:

Name	Register Controls	Takes Effect	Default
Cl	contents level	table of contents	2
De	display eject	display	0
Df	display floating	display	5
Ds	display spacing	display	1v
Hb	heading break	heading	2
Hc	heading centering	heading	0
Hi	heading indent	heading	1
Hi	heading spacing	heading	1
Hu	heading unnumbered	heading	2
Li	list indentation	list	6 (**nroff**) 5 (**troff**)
Ls	list spacing	list	6
Pi	paragraph indent	paragraph	5
Pt	paragraph type	paragraph	1
Si	static indent	display	5 (**nroff**) 3 (**troff**)

When resetting these values, make sure to specify the appropriate units. Setting the line length to 7, for example, will result in output with one character per line. Setting **Pi** to 0 suppresses paragraph indentation

Here is a list of string registers available in **–mm**; they may be used anywhere in the text:

Name	String's Function
***Q**	quote (" in **nroff,** **“** in **troff**)
***U**	unquote (" in **nroff,** **”** in **troff**)
***–**	dash (**--** in **nroff,** **—** in **troff**)
***(MO**	month (month of the year)
***(DY**	day (current date)
******	automatically numbered footnote
***´**	acute accent (before letter)
***`**	grave accent (before letter)
***^**	circumflex (before letter)
***,**	cedilla (before letter)
***:**	umlaut (before letter)
***~**	tilde (before letter)
\(BU	bullet item
\(DT	date (*month day*, *yr*)
\(EM	em dash
\(Lf	**LIST OF FIGURES** title

\(Lt **LIST OF TABLES** title
\(Lx **LIST OF EXHIBITS** title
\(Le **LIST OF EQUATIONS** title
\(Rp **REFERENCES** title
\(Tm trademark character (TM)

When using the extended accent mark definitions available with **.AM**, these strings should come after, rather than before, the letter to be accented.

FILES

/usr/share/lib/tmac/m
/usr/share/lib/tmac/mm.[nt]
nroff and **troff** definitions of **mm**.

SEE ALSO

col(1), **eqn**(1), **nroff**(1), **refer**(1), **tbl**(1), **troff**(1)

BUGS

Floating keeps and regular keeps are diverted to the same space, so they cannot be mixed together with predictable results.

NAME

ms – text formatting macros

SYNOPSIS

nroff –ms [*options*] *filename* . . .

troff –ms [*options*] *filename* . . .

DESCRIPTION

This package of **nroff**(1) and **troff**(1) macro definitions provides a formatting facility for various styles of articles, theses, and books. When producing 2-column output on a terminal or lineprinter, or when reverse line motions are needed, filter the output through **col**(1). All external **–ms** macros are defined below.

Note: this **–ms** macro package is an extended version written at Berkeley and is a superset of the standard **–ms** macro packages as supplied by Bell Labs. Some of the Bell Labs macros have been removed; for instance, it is assumed that the user has little interest in producing headers stating that the memo was generated at Whippany Labs.

Many **nroff** and **troff** requests are unsafe in conjunction with this package. However, the first four requests below may be used with impunity after initialization, and the last two may be used even before initialization:

.bp	begin new page
.br	break output line
.sp *n*	insert n spacing lines
.ce *n*	center next n lines
.ls *n*	line spacing: *n*=**1** single, *n*=**2** double space
.na	no alignment of right margin

Font and point size changes with **\f** and **\s** are also allowed; for example, **\fIword\fR** will italicize *word*. Output of the **tbl**(1), **eqn**(1) and **refer**(1) preprocessors for equations, tables, and references is acceptable as input.

REQUESTS

Macro Name	Initial Value	Break? Reset?	Explanation
.AB *x*	–	y	begin abstract; if *x*=no do not label abstract
.AE	–	y	end abstract
.AI	–	y	author's institution
.AM	–	n	better accent mark definitions
.AU	–	y	author's name
.B *x*	–	n	embolden *x*; if no *x*, switch to boldface
.B1	–	y	begin text to be enclosed in a box
.B2	–	y	end boxed text and print it
.BT	date	n	bottom title, printed at foot of page
.BX *x*	–	n	print word *x* in a box
.CM	if t	n	cut mark between pages
.CT	–	y,y	chapter title: page number moved to CF (TM only)
.DA *x*	if n	n	force date *x* at bottom of page; today if no *x*
.DE	–	y	end display (unfilled text) of any kind
.DS *x y*	I	y	begin display with keep; *x*=I, L, C, B; *y*=indent

.ID *y*	8n,.5i	y	indented display with no keep; *y*=indent
.LD	–	y	left display with no keep
.CD	–	y	centered display with no keep
.BD	–	y	block display; center entire block
.EF *x*	–	n	even page footer *x* (3 part as for **.tl**)
.EH *x*	–	n	even page header *x* (3 part as for **.tl**)
.EN	–	y	end displayed equation produced by **eqn**
.EQ *x y*	–	y	break out equation; *x*=L,I,C; *y*=equation number
.FE	–	n	end footnote to be placed at bottom of page
.FP	–	n	numbered footnote paragraph; may be redefined
.FS *x*	–	n	start footnote; *x* is optional footnote label
.HD	undef	n	optional page header below header margin
.I *x*	–	n	italicize *x*; if no *x*, switch to italics
.IP *x y*	–	y,y	indented paragraph, with hanging tag *x*; *y*=indent
.IX *x y*	–	y	index words *x y* and so on (up to 5 levels)
.KE	–	n	end keep of any kind
.KF	–	n	begin floating keep; text fills remainder of page
.KS	–	y	begin keep; unit kept together on a single page
.LG	–	n	larger; increase point size by 2
.LP	–	y,y	left (block) paragraph.
.MC *x*	–	y,y	multiple columns; *x*=column width
.ND *x*	if t	n	no date in page footer; *x* is date on cover
.NH *x y*	–	y,y	numbered header; *x*=level, *x*=0 resets, *x*=S sets to *y*
.NL	10p	n	set point size back to normal
.OF *x*	–	n	odd page footer *x* (3 part as for **.tl**)
.OH *x*	–	n	odd page header *x* (3 part as for **.tl**)
.P1	if TM	n	print header on first page
.PP	–	y,y	paragraph with first line indented
.PT	- % -	n	page title, printed at head of page
.PX *x*	–	y	print index (table of contents); *x*=no suppresses title
.QP	–	y,y	quote paragraph (indented and shorter)
.R	on	n	return to Roman font
.RE	5n	y,y	retreat: end level of relative indentation
.RP *x*	–	n	released paper format; *x*=no stops title on first page
.RS	5n	y,y	right shift: start level of relative indentation
.SH	–	y,y	section header, in boldface
.SM	–	n	smaller; decrease point size by 2
.TA	8n,5n	n	set TAB characters to 8n 16n ... (**nroff**) 5n 10n ... (**troff**)
.TC *x*	–	y	print table of contents at end; *x*=no suppresses title
.TE	–	y	end of table processed by **tbl**

.TH	–	y	end multi-page header of table
.TL	–	y	title in boldface and two points larger
.TM	off	n	UC Berkeley thesis mode
.TS *x*	–	y,y	begin table; if *x*=H table has multi-page header
.UL *x*	–	n	underline *x*, even in **troff**
.UX *x*	–	n	UNIX; trademark message first time; *x* appended
.XA *x y*	–	y	another index entry; *x*=page or no for none; y=indent
.XE	–	y	end index entry (or series of **.IX** entries)
.XP	–	y,y	paragraph with first line indented, others indented
.XS *x y*	–	y	begin index entry; *x*=page or no for none; *y*=indent
.1C	on	y,y	one column format, on a new page
.2C	–	y,y	begin two column format
.]–	–	n	beginning of **refer** reference
.[0	–	n	end of unclassifiable type of reference
.[N	–	n	N= 1:journal-article, 2:book, 3:book-article, 4:report

REGISTERS

Formatting distances can be controlled in **–ms** by means of built-in number registers. For example, this sets the line length to 6.5 inches:

```
.nr LL 6.5i
```

Here is a table of number registers and their default values:

Name	Register Controls	Takes Effect	Default
PS	point size	paragraph	10
VS	vertical spacing	paragraph	12
LL	line length	paragraph	6i
LT	title length	next page	same as **LL**
FL	footnote length	next **.FS**	5.5i
PD	paragraph distance	paragraph	1v (if n), .3v (if t)
DD	display distance	displays	1v (if n), .5v (if t)
PI	paragraph indent	paragraph	5n
QI	quote indent	next **.QP**	5n
FI	footnote indent	next **.FS**	2n
PO	page offset	next page	0 (if n), ~1i (if t)
HM	header margin	next page	1i
FM	footer margin	next page	1i
FF	footnote format	next **.FS**	0 (1, 2, 3 available)

When resetting these values, make sure to specify the appropriate units. Setting the line length to 7, for example, will result in output with one character per line. Setting **FF** to 1 suppresses footnote superscripting; setting it to 2 also suppresses indentation of the first line; and setting it to 3 produces an **.IP**-like footnote paragraph.

Here is a list of string registers available in **–ms**; they may be used anywhere in the text:

Name	String's Function
***Q**	quote (" in **nroff,** `` in **troff**)
***U**	unquote (" in **nroff,** '' in **troff**)
***–**	dash (**--** in **nroff,** **—** in **troff**)
***(MO**	month (month of the year)
***(DY**	day (current date)
******	automatically numbered footnote
***´**	acute accent (before letter)
***`**	grave accent (before letter)
***^**	circumflex (before letter)
***,**	cedilla (before letter)
***:**	umlaut (before letter)
***~**	tilde (before letter)

When using the extended accent mark definitions available with **.AM**, these strings should come after, rather than before, the letter to be accented.

FILES

/usr/share/lib/tmac/s
/usr/share/lib/tmac/ms.???

SEE ALSO

col(1), **eqn**(1), **nroff**(1), **refer**(1), **tbl**(1), **troff**(1)

BUGS

Floating keeps and regular keeps are diverted to the same space, so they cannot be mixed together with predictable results.

NAME

nl_types – native language data types

SYNOPSIS

#include <nl_types.h>

DESCRIPTION

This header contains the following definitions:

nl_catd	Used by the message catalog functions **catopen**, **catgets** and **catclose** to identify a catalogue.
nl_item	Used by **nl_langinfo** to identify items of langinfo data. Values for objects of type **nl_item** are defined in **<langinfo.h>**.
NL_SETD	Used by **gencat** when no **$set** directive is specified in a message text source file. This constant can be used in subsequent calls to **catgets** as the value of the set identifier parameter.
NL_MGSMAX	Maximum number of messages per set.
NL_SETMAX	Maximum number of sets per catalogue.
NL_TEXTMAX	Maximum size of a message.

SEE ALSO

gencat(1), **catgets**(3C), **catopen**(3C), **nl_langinfo**(3C), **langinfo**(5)

NAME

prof – profile within a function

SYNOPSIS

```
#define MARK
#include <prof.h>
void MARK( name);
```

DESCRIPTION

MARK introduces a mark called *name* that is treated the same as a function entry point. Execution of the mark adds to a counter for that mark, and program-counter time spent is accounted to the immediately preceding mark or to the function if there are no preceding marks within the active function.

name may be any combination of letters, numbers, or underscores. Each *name* in a single compilation must be unique, but may be the same as any ordinary program symbol.

For marks to be effective, the symbol **MARK** must be defined before the header **prof.h** is included, either by a preprocessor directive as in the synopsis, or by a command line argument:

```
cc –p –DMARK work.c
```

If **MARK** is not defined, the **MARK(***name***)** statements may be left in the source files containing them and are ignored. **prof –g** must be used to get information on all labels.

EXAMPLE

In this example, marks can be used to determine how much time is spent in each loop. Unless this example is compiled with **MARK** defined on the command line, the marks are ignored.

```
#include <prof.h>
work( )
{
        int i, j;
        . . .
        MARK(loop1);
        for (i = 0; i < 2000; i++) {
                . . .
        }
        MARK(loop2);
        for (j = 0; j < 2000; j++) {
                . . .
        }
}
```

SEE ALSO

profil(2), **monitor**(3C)

NAME

regex – internationalized basic and extended regular expression matching

DESCRIPTION

Regular Expressions (REs) provide a mechanism to select specific strings from a set of character strings. The Internationalized Regular Expressions described below differ from the Simple Regular Expressions described on the **regexp**(5) manual page in the following ways:

- both Basic and Extended Regular Expressions are supported
- the Internationalization features—character class, equivalence class, and multi-character collation—are supported.

The Basic Regular Expression (BRE) notation and construction rules described in the **BASIC REGULAR EXPRESSIONS** section apply to most utilities supporting regular expressions. Some utilities, instead, support the Extended Regular Expressions (ERE) described in the **EXTENDED REGULAR EXPRESSIONS** section; any exceptions for both cases are noted in the descriptions of the specific utilities using regular expressions. Both BREs and EREs are supported by the Regular Expression Matching interfaces **regcomp**(3C) and **regexec**(3C).

BASIC REGULAR EXPRESSIONS

BREs Matching a Single Character

A BRE ordinary character, a special character preceded by a backslash, or a period matches a single character. A bracket expression matches a single character or a single collating element. See **RE Bracket Expression**, below.

BRE Ordinary Characters

An ordinary character is a BRE that matches itself: any character in the supported character set, except for the BRE special characters listed in **BRE Special Characters**, below.

The interpretation of an ordinary character preceded by a backslash (\) is undefined, except for:

1. the characters), (, {, and }
2. the digits 1 to 9 inclusive (see **BREs Matching Multiple Characters**, below)
3. a character inside a bracket expression.

BRE Special Characters

A *BRE special character* has special properties in certain contexts. Outside those contexts, or when preceded by a backslash, such a character will be a BRE that matches the special character itself. The BRE special characters and the contexts in which they have their special meaning are:

. [\ The combination of period, left-bracket and backslash is special except when used in a bracket expression (see **RE Bracket Expression**, below). An expression containing a [that is not preceded by a backslash and is not part of a bracket expression produces undefined results.

`*` The asterisk is special except when used:

- in a bracket expression
- as the first character of an entire BRE (after an initial ^, if any)
- as the first character of a subexpression (after an initial ^, if any); see **BREs Matching Multiple Characters**, below.

`^` The circumflex is special when used:

- as an anchor (see **BRE Expression Anchoring**, below).
- as the first character of a bracket expression (see **RE Bracket Expression**, below).

$ The dollar sign is special when used as an anchor.

Periods in BREs

A period (**.**), when used outside a bracket expression, is a BRE that matches any character in the supported character set except NUL.

RE Bracket Expression

A bracket expression (an expression enclosed in square brackets, [ ]) is an RE that matches a single collating element contained in the non-empty set of collating elements represented by the bracket expression.

The following rules and definitions apply to bracket expressions:

1. A *bracket expression* is either a matching list expression or a non-matching list expression. It consists of one or more expressions: collating elements, collating symbols, equivalence classes, character classes, or range expressions (see rule 7 below). Portable applications must not use range expressions, even though all implementations support them. The right-bracket (]) loses its special meaning and represents itself in a bracket expression if it occurs first in the list (after an initial circumflex (^), if any). Otherwise, it terminates the bracket expression, unless it appears in a collating symbol (such as [.].]) or is the ending right-bracket for a collating symbol, equivalence class, or character class. The special characters:

   `. * [ \`

   (period, asterisk, left-bracket and backslash, respectively) lose their special meaning within a bracket expression.

   The character sequences:

   `[. [= [:`

   (left-bracket followed by a period, equals-sign, or colon) are special inside a bracket expression and are used to delimit collating symbols, equivalence class expressions, and character class expressions. These symbols must be followed by a valid expression and the matching terminating sequence .], =] or :], as described in the following items.

2. A *matching list* expression specifies a list that matches any one of the expressions represented in the list. The first character in the list must not be the circumflex. For example, **[abc]** is an RE that matches any of the characters **a**, **b** or **c**.

3. A *non-matching list* expression begins with a circumflex (^), and specifies a list that matches any character or collating element except for the expressions represented in the list after the leading circumflex. For example, [^abc] is an RE that matches any character or collating element except the characters **a, b** or **c**. The circumflex will have this special meaning only when it occurs first in the list, immediately following the left-bracket.

4. A *collating symbol* is a collating element enclosed within bracket-period ([. .]) delimiters. Multi-character collating elements must be represented as collating symbols when it is necessary to distinguish them from a list of the individual characters that make up the multi-character collating element. For example, if the string **ch** is a collating element in the current collation sequence with the associated collating symbol <ch>, the expression [[.ch.]] will be treated as an RE matching the character sequence **ch,** while [ch] will be treated as an RE matching **c** or **h.** Collating symbols will be recognized only inside bracket expressions. This implies that the RE **[[.ch.]]***__c__ matches the first to fifth character in the string chchch. If the string is not a collating element in the current collating sequence definition, or if the collating element has no characters associated with it, the symbol will be treated as an invalid expression.

5. An *equivalence class expression* represents the set of collating elements belonging to an equivalence class. Only primary equivalence classes will be recognised. The class is expressed by enclosing any one of the collating elements in the equivalence class within bracket-equal ([= =]) delimiters. For example, if a, à and â belong to the same equivalence class, then [[=a=]b], [[=à=]b] and [[=â=]b] will each be equivalent to [aàâb]. If the collating element does not belong to an equivalence class, the equivalence class expression will be treated as a *collating symbol* .

6. A *character class expression* represents the set of characters belonging to a character class, as defined in the **LC_CTYPE** category in the current locale. All character classes specified in the current locale will be recognized. A character class expression is expressed as a character class name enclosed within bracket-colon ([: :]) delimiters.

   The following character class expressions are supported in all locales:

[:alnum:]	**[:cntrl:]**	**[:lower:]**	**[:space:]**
[:alpha:]	**[:digit:]**	**[:print:]**	**[:upper:]**
[:blank:]	**[:graph:]**	**[:punct:]**	**[:xdigit:]**

   In addition, character class expressions of the form:

   [:name:]

   are recognized in those locales where the *name* keyword has been given a **charclass** definition in the **LC_CTYPE** category.

7. A *range expression* represents the set of collating elements that fall between two elements in the current collation sequence, inclusively. It is expressed as the starting point and the ending point separated by a hyphen (–).

Range expressions must not be used in portable applications because their behavior is dependent on the collating sequence. Ranges will be treated according to the current collating sequence, and include such characters that fall within the range based on that collating sequence, regardless of character values. This, however, means that the interpretation will differ depending on collating sequence. If, for instance, one collating sequence defines ä as a variant of **a,** while another defines it as a letter following **z,** then the expression [ä–z] is valid in the first language and invalid in the second.

In the following, all examples assume the collation sequence specified for the POSIX locale, unless another collation sequence is specifically defined.

The starting range point and the ending range point must be a collating element or collating symbol. An equivalence class expression used as a starting or ending point of a range expression produces unspecified results. An equivalence class can be used portably within a bracket expression, but only outside the range. For example, the unspecified expression [[=e=]–f] should be given as [[=e=]e–f]. The ending range point must collate equal to or higher than the starting range point; otherwise, the expression will be treated as invalid. The order used is the order in which the collating elements are specified in the current collation definition. One-to-many mappings (see **locale**(5)) will not be performed. For example, assuming that the character eszet (β) is placed in the collation sequence after **r** and **s,** but before **t,** and that it maps to the sequence **ss** for collation purposes, then the expression [r–s] matches only **r** and **s,** but the expression [s–t] matches **s,** β or **t.**

The interpretation of range expressions where the ending range point is also the starting range point of a subsequent range expression (for instance [a–m–o]) is undefined.

The hyphen character will be treated as itself if it occurs first (after an initial ^, if any) or last in the list, or as an ending range point in a range expression. As examples, the expressions [–ac] and [ac–] are equivalent and match any of the characters **a, c,** or **–;** [^–ac] and [^ac–] are equivalent and match any characters except **a, c,** or –; the expression [%––] matches any of the characters between % and – inclusive; the expression [––@] matches any of the characters between – and @ inclusive; and the expression [a––@] is invalid, because the letter **a** follows the symbol – in the POSIX locale. To use a hyphen as the starting range point, it must either come first in the bracket expression or be specified as a collating symbol, for example: [][.–.]–0], which matches either a right bracket or any character or collating element that collates between hyphen and 0, inclusive.

If a bracket expression must specify both – and], the] must be placed first (after the ^, if any) and the – last within the bracket expression.

BREs Matching Multiple Characters

The following rules can be used to construct BREs matching multiple characters from BREs matching a single character:

1. The concatenation of BREs matches the concatenation of the strings matched by

each component of the BRE.

2. A *subexpression* can be defined within a BRE by enclosing it between the character pairs \(and \). Such a subexpression matches whatever it would have matched without the \(and \), except that anchoring within subexpressions is optional behavior; see **BRE Expression Anchoring**, below. Subexpressions can be arbitrarily nested.

3. The *back-reference* expression *n* matches the same (possibly empty) string of characters as was matched by a subexpression enclosed between \(and \) preceding the *n*. The character *n* must be a digit from 1 to 9 inclusive, *n*th subexpression (the one that begins with the *n*th \(and ends with the corresponding paired \)). The expression is invalid if less than *n* subexpressions precede the *n*. For example, the expression ^\(.*\)\1$ matches a line consisting of two adjacent appearances of the same string, and the expression \(a\)*\1 fails to match **a.** The limit of nine back-references to subexpressions in the RE is based on the use of a single digit identifier. This does not imply that only nine subexpressions are allowed in REs. The following is a valid BRE with ten subexpressions:

   \(\(\(ab\)*c\)*d\)\(ef\)*\(gh\)\{2\}\(ij\)*\(kl\)*\(mn\)*\(op\)*\(qr\)*

4. When a BRE matching a single character, a subexpression or a back-reference is followed by the special character asterisk (*), together with that asterisk it matches what zero or more consecutive occurrences of the BRE would match. For example, **[ab]* and [ab][ab]** are equivalent when matching the string **ab.**

5. When a BRE matching a single character, a subexpression, or a back-reference is followed by an *interval expression* of the format \{*m*\}, \{*m*,\} or \{*m*,*n*\}, together with that interval expression it matches what repeated consecutive occurrences of the BRE would match. The values of *m* and *n* will be decimal integers in the range $0 \le m \le n \le \{$**RE_DUP_MAX**$\}$, where *m* specifies the exact or minimum number of occurrences and *n* specifies the maximum number of occurrences. The expression \{*m*\} matches exactly *m* occurrences of the preceding BRE, \{*m*,\} matches at least *m* occurrences and \{*m*,*n*\} matches any number of occurrences between *m* and *n*, inclusive.

   For example, in the string **abababccccccd,** the BRE c\{3\} is matched by characters seven to nine, the BRE \(ab\)\{4,\} is not matched at all and the BRE c\{1,3\}d is matched by characters ten to thirteen.

The behavior of multiple adjacent duplication symbols **(** * and intervals) produces undefined results.

BRE Precedence

The order of precedence is as shown in the following table:

BRE Precedence (from high to low)	
collation-related bracket symbols	[= =] [: :] [. .]
escaped characters	\<*special character*>
bracket expression	[ ]
subexpressions/back-references	\(\) *n*
single-character-BRE duplication	* \{*m*,*n*\}
concatenation	
anchoring	^ $

BRE Expression Anchoring

A BRE can be limited to matching strings that begin or end a line; this is called *anchoring*. The circumflex and dollar sign special characters will be considered BRE anchors in the following contexts:

1. A circumflex **(^)** is an anchor when used as the first character of an entire BRE. The implementation may treat circumflex as an anchor when used as the first character of a subexpression. The circumflex will anchor the expression to the beginning of a string; only sequences starting at the first character of a string will be matched by the BRE. For example, the BRE ^ab matches **ab** in the string **abcdef,** but fails to match in the string **cdefab.** A portable BRE must escape a leading circumflex in a subexpression to match a literal circumflex.
2. A dollar sign **($)** is an anchor when used as the last character of an entire BRE. The implementation may treat a dollar sign as an anchor when used as the last character of a subexpression. The dollar sign will anchor the expression to the end of the string being matched; the dollar sign can be said to match the end-of-string following the last character.
3. A BRE anchored by both **^** and **$** matches only an entire string. For example, the BRE **^abcdef$** matches strings consisting only of **abcdef.**
4. **^** and **$** are not special in subexpressions.

EXTENDED REGULAR EXPRESSIONS

The rules specififed for BREs apply to Extended Regular Expressions (EREs) with the following exceptions:

- The characters **|**, **+**, and **?** have special meaning, as defined below
- The subexpression **( )** and duplication **{}** operators need not be preceded by a backslash (\)
- The back reference operator is not supported.
- Anchoring (**^$**) is supported in subexpressions.

EREs Matching a Single Character

An ERE ordinary character, a special character preceded by a backslash, or a period matches a single character. A bracket expression matches a single character or a single collating element. An *ERE matching a single character* enclosed in parentheses matches the same as the ERE without parentheses would have matched.

ERE Ordinary Characters

An *ordinary character* is an ERE that matches itself. An ordinary character is any character in the supported character set, except for the ERE special characters listed in **ERE Special Characters** below. The interpretation of an ordinary character preceded by a backslash (\) is undefined.

ERE Special Characters

An *ERE special character* has special properties in certain contexts. Outside those contexts, or when preceded by a backslash, such a character is an ERE that matches the special character itself. The extended regular expression special characters and the contexts in which they have their special meaning are:

`. [ \ (` The period, left-bracket, backslash and left-parenthesis are special except when used in a bracket expression (see **RE Bracket Expression**, above). Outside a bracket expression, a left-parenthesis immediately followed by a right-parenthesis produces undefined results.

`)` The right-parenthesis is special when matched with a preceding left-parenthesis, both outside a bracket expression.

`* + ? {` The asterisk, plus-sign, question-mark and left-brace are special except when used in a bracket expression (see **RE Bracket Expression**, above). Any of the following uses produce undefined results:

- if these characters appear first in an ERE, or immediately following a vertical-line, circumflex or left-parenthesis
- if a left-brace is not part of a valid interval expression.

`|` The vertical-line is special except when used in a bracket expression (see **RE Bracket Expression**, above). A vertical-line appearing first or last in an ERE, or immediately following a vertical-line or a left-parenthesis, or immediately preceding a right-parenthesis, produces undefined results.

`^` The circumflex is special when used:

- as an anchor (see **ERE Expression Anchoring**, below).
- as the first character of a bracket expression (see **RE Bracket Expression**, above).

`$` The dollar sign is special when used as an anchor.

Periods in EREs

A period (**.**), when used outside a bracket expression, is an ERE that matches any character in the supported character set except NUL.

ERE Bracket Expression

The rules for ERE Bracket Expressions are the same as for Basic Regular Expressions; see **RE Bracket Expression**, above).

EREs Matching Multiple Characters

The following rules will be used to construct EREs matching multiple characters from EREs matching a single character:

1. A *concatenation of EREs* matches the concatenation of the character sequences matched by each component of the ERE. A concatenation of EREs enclosed in parentheses matches whatever the concatenation without the parentheses matches. For example, both the ERE **cd** and the ERE **(cd)** are matched by the third and

fourth character of the string **abcdefabcdef.**

2. When an ERE matching a single character or an ERE enclosed in parentheses is followed by the special character plus-sign (+), together with that plus-sign it matches what one or more consecutive occurrences of the ERE would match. For example, the ERE **b+(bc)** matches the fourth to seventh characters in the string **acabbbcde**; **[ab]** + and **[ab][ab]*** are equivalent.
3. When an ERE matching a single character or an ERE enclosed in parentheses is followed by the special character asterisk (*), together with that asterisk it matches what zero or more consecutive occurrences of the ERE would match. For example, the ERE **b*c** matches the first character in the string **cabbbcde,** and the ERE **b*cd** matches the third to seventh characters in the string **cabbbcdebbbbbbcdbc.** And, **[ab]*** and **[ab][ab]** are equivalent when matching the string **ab.**
4. When an ERE matching a single character or an ERE enclosed in parentheses is followed by the special character question-mark (**?**), together with that question-mark it matches what zero or one consecutive occurrences of the ERE would match. For example, the ERE **b?c** matches the second character in the string **acabbbcde.**
5. When an ERE matching a single character or an ERE enclosed in parentheses is followed by an *interval expression* of the format {*m*}, {*m*,} or {*m*,*n*}, together with that interval expression it matches what repeated consecutive occurrences of the ERE would match. The values of *m* and *n* will be decimal integers in the range $0 \le m \le n \le$ **{RE_DUP_MAX}**, where *m* specifies the exact or minimum number of occurrences and *n* specifies the maximum number of occurrences. The expression {*m*} matches exactly *m* occurrences of the preceding ERE, {*m*,} matches at least *m* occurrences and {*m*,*n*} matches any number of occurrences between *m* and *n*, inclusive.

For example, in the string **abababccccccd** the ERE c{3} is matched by characters seven to nine and the ERE (ab){2,} is matched by characters one to six.

The behavior of multiple adjacent duplication symbols **(**+**,** *, **?** and intervals) produces undefined results.

ERE Alternation

Two EREs separated by the special character vertical-line (|) match a string that is matched by either. For example, the ERE a((bc) | d) matches the string abc and the string ad. Single characters, or expressions matching single characters, separated by the vertical bar and enclosed in parentheses, will be treated as an ERE matching a single character.

ERE Precedence

The order of precedence will be as shown in the following table:

ERE Precedence (from high to low)	
collation-related bracket symbols	[= =] [: :] [. .]
escaped characters	\<*special character*>
bracket expression	[ ]
grouping	( )
single-character-ERE duplication	* + ? {*m*,*n*}
concatenation	
anchoring	^ $
alternation	|

For example, the ERE **abba | cde** matches either the string **abba** or the string **cde** (rather than the string **abbade** or **abbcde**, because concatenation has a higher order of precedence than alternation).

ERE Expression Anchoring

An ERE can be limited to matching strings that begin or end a line; this is called *anchoring*. The circumflex and dollar sign special characters are considered ERE anchors when used anywhere outside a bracket expression. This has the following effects:

1. A circumflex (^) outside a bracket expression anchors the expression or subexpression it begins to the beginning of a string; such an expression or subexpression can match only a sequence starting at the first character of a string. For example, the EREs ^ab and (^ab) match ab in the string abcdef, but fail to match in the string cdefab, and the ERE a^b is valid, but can never match because the **a** prevents the expression ^b from matching starting at the first character.
2. A dollar sign **($)** outside a bracket expression anchors the expression or subexpression it ends to the end of a string; such an expression or subexpression can match only a sequence ending at the last character of a string. For example, the EREs ef$ and (ef$) match ef in the string abcdef, but fail to match in the string cdefab, and the ERE e$f is valid, but can never match because the **f** prevents the expression e$ from matching ending at the last character.

SEE ALSO

localedef(1), **regcomp**(3C), **environ**(5), **locale**(5), **regexp**(5)

NAME

regexp, compile, step, advance – simple regular expression compile and match routines

SYNOPSIS

```
#define INIT declarations
#define GETC(void) getc code
#define PEEKC(void) peekc code
#define UNGETC(void) ungetc code
#define RETURN(ptr) return code
#define ERROR(val) error code

#include <regexp.h>

char *compile(char *instring, char *expbuf, char *endbuf, int eof);

int step(char *string, char *expbuf);

int advance(char *string, char *expbuf);

extern char *loc1, *loc2, *locs;
```

DESCRIPTION

Regular Expressions (REs) provide a mechanism to select specific strings from a set of character strings. The Simple Regular Expressions described below differ from the Internationalized Regular Expressions described on the **regex**(5) manual page in the following ways:

- only Basic Regular Expressions are supported
- the Internationalization features—character class, equivalence class, and multi-character collation—are not supported.

The functions **step( )**, **advance( )**, and **compile( )** are general purpose regular expression matching routines to be used in programs that perform regular expression matching. These functions are defined by the **<regexp.h>** header.

The functions **step( )** and **advance( )** do pattern matching given a character string and a compiled regular expression as input.

The function **compile( )** takes as input a regular expression as defined below and produces a compiled expression that can be used with **step( )** or **advance( )**.

Basic Regular Expressions

A regular expression specifies a set of character strings. A member of this set of strings is said to be matched by the regular expression. Some characters have special meaning when used in a regular expression; other characters stand for themselves.

The following *one-character RE*s match a *single* character:

1.1 An ordinary character (*not* one of those discussed in 1.2 below) is a one-character RE that matches itself.

1.2 A backslash (\ ) followed by any special character is a one-character RE that matches the special character itself. The special characters are:

a. **.**, *, **[**, and \ (period, asterisk, left square bracket, and backslash, respectively), which are always special, *except* when they appear within square brackets (**[ ]**; see 1.4 below).

b. ^ (caret or circumflex), which is special at the *beginning* of an *entire* RE (see 4.1 and 4.3 below), or when it immediately follows the left of a pair of square brackets (**[ ]**) (see 1.4 below).

c. **$** (dollar sign), which is special at the **end** of an *entire* RE (see 4.2 below).

d. The character used to bound (that is, delimit) an entire RE, which is special for that RE (for example, see how slash (/) is used in the **g** command, below.)

1.3 A period (**.**) is a one-character RE that matches any character except new-line.

1.4 A non-empty string of characters enclosed in square brackets (**[ ]**) is a one-character RE that matches *any one* character in that string. If, however, the first character of the string is a circumflex (^), the one-character RE matches any character *except* new-line and the remaining characters in the string. The ^ has this special meaning *only* if it occurs first in the string. The minus (–) may be used to indicate a range of consecutive characters; for example, **[0–9]** is equivalent to **[0123456789]**. The – loses this special meaning if it occurs first (after an initial ^, if any) or last in the string. The right square bracket (**]**) does not terminate such a string when it is the first character within it (after an initial ^, if any); for example, **[ ]a–f]** matches either a right square bracket (**]**) or one of the ASCII letters **a** through **f** inclusive. The four characters listed in 1.2.a above stand for themselves within such a string of characters.

The following rules may be used to construct REs from one-character REs:

2.1 A one-character RE is a RE that matches whatever the one-character RE matches.

2.2 A one-character RE followed by an asterisk (*) is a RE that matches **0** or more occurrences of the one-character RE. If there is any choice, the longest leftmost string that permits a match is chosen.

2.3 A one-character RE followed by **\{***m***\}**, **\{***m***,\}**, or **\{***m***,***n***\}** is a RE that matches a *range* of occurrences of the one-character RE. The values of *m* and *n* must be non-negative integers less than 256; **\{***m***\}** matches *exactly m* occurrences; **\{***m***,\}** matches *at least m* occurrences; **\{***m***,***n***\}** matches *any number* of occurrences *between m* and *n* inclusive. Whenever a choice exists, the RE matches as many occurrences as possible.

2.4 The concatenation of REs is a RE that matches the concatenation of the strings matched by each component of the RE.

2.5 A RE enclosed between the character sequences **\(** and **\)** is a RE that matches whatever the unadorned RE matches.

2.6 The expression **** *n* matches the same string of characters as was matched by an expression enclosed between **\(** and **\)** *earlier* in the same RE. Here *n* is a digit; the sub-expression specified is that beginning with the *n*-th occurrence of **\(** counting from the left. For example, the expression **^\(.*\)\1$** matches a line consisting of two repeated appearances of the same string.

A RE may be constrained to match words.

3.1 **\ <** constrains a RE to match the beginning of a string or to follow a character that is not a digit, underscore, or letter. The first character matching the RE must be a digit, underscore, or letter.

3.2 **\ >** constrains a RE to match the end of a string or to precede a character that is not a digit, underscore, or letter.

An *entire RE* may be constrained to match only an initial segment or final segment of a line (or both).

4.1 A circumflex (^) at the beginning of an entire RE constrains that RE to match an *initial* segment of a line.

4.2 A dollar sign (**$**) at the end of an entire RE constrains that RE to match a *final* segment of a line.

4.3 The construction *^entire RE* **$** constrains the entire RE to match the entire line.

The null RE (for example, **//**) is equivalent to the last RE encountered.

Addressing with REs

Addresses are constructed as follows:

1. The character "**.**" addresses the current line.
2. The character "**$**" addresses the last line of the buffer.
3. A decimal number *n* addresses the *n*-th line of the buffer.
4. *'x* addresses the line marked with the mark name character *x*, which must be an ASCII lower-case letter (**a**–**z**). Lines are marked with the **k** command described below.
5. A RE enclosed by slashes (**/**) addresses the first line found by searching *forward* from the line *following* the current line toward the end of the buffer and stopping at the first line containing a string matching the RE. If necessary, the search wraps around to the beginning of the buffer and continues up to and including the current line, so that the entire buffer is searched.
6. A RE enclosed in question marks (**?**) addresses the first line found by searching *backward* from the line *preceding* the current line toward the beginning of the buffer and stopping at the first line containing a string matching the RE. If necessary, the search wraps around to the end of the buffer and continues up to and including the current line.
7. An address followed by a plus sign (**+**) or a minus sign (–) followed by a decimal number specifies that address plus (respectively minus) the indicated number of lines. A shorthand for .+5 is .5.
8. If an address begins with **+** or –, the addition or subtraction is taken with respect to the current line; for example, **–5** is understood to mean **.–5**.
9. If an address ends with **+** or –, then 1 is added to or subtracted from the address, respectively. As a consequence of this rule and of Rule 8, immediately above, the address – refers to the line preceding the current line. (To maintain compatibility with earlier versions of the editor, the character ^ in addresses is entirely

equivalent to –.) Moreover, trailing **+** and – characters have a cumulative effect, so —– refers to the current line less 2.

10. For convenience, a comma (**,**) stands for the address pair **1,$**, while a semicolon (**;**) stands for the pair **.,$**.

Characters With Special Meaning

Characters that have special meaning except when they appear within square brackets (**[ ]**) or are preceded by \ are: **.**, ∗, **[**, \ . Other special characters, such as **$** have special meaning in more restricted contexts.

The character ^ at the beginning of an expression permits a successful match only immediately after a newline, and the character **$** at the end of an expression requires a trailing newline.

Two characters have special meaning only when used within square brackets. The character – denotes a range, **[** *c*–*c***]**, unless it is just after the open bracket or before the closing bracket, **[** –*c***]** or **[** *c*–**]** in which case it has no special meaning. When used within brackets, the character ^ has the meaning *complement of* if it immediately follows the open bracket (example: **[**^*c***]**); elsewhere between brackets (example: **[***c*^**]**) it stands for the ordinary character ^.

The special meaning of the \ operator can be escaped only by preceding it with another \ , for example \ \ .

Macros

Programs must have the following five macros declared before the **#include <regexp.h>** statement. These macros are used by the **compile( )** routine. The macros **GETC**, **PEEKC**, and **UNGETC** operate on the regular expression given as input to **compile( )**.

GETC This macro returns the value of the next character (byte) in the regular expression pattern. Successive calls to **GETC** should return successive characters of the regular expression.

PEEKC This macro returns the next character (byte) in the regular expression. Immediately successive calls to **PEEKC** should return the same character, which should also be the next character returned by **GETC**.

UNGETC This macro causes the argument **c** to be returned by the next call to **GETC** and **PEEKC**. No more than one character of pushback is ever needed and this character is guaranteed to be the last character read by **GETC**. The return value of the macro **UNGETC(c)** is always ignored.

RETURN(*ptr***)** This macro is used on normal exit of the **compile( )** routine. The value of the argument *ptr* is a pointer to the character after the last character of the compiled regular expression. This is useful to programs which have memory allocation to manage.

ERROR(*val***)** This macro is the abnormal return from the **compile( )** routine. The argument *val* is an error number (see ERRORS below for meanings). This call should never return.

compile()

The syntax of the **compile()** routine is as follows:

compile(*instring***,** *expbuf***,** *endbuf***,** *eof***)**

The first parameter, *instring*, is never used explicitly by the **compile()** routine but is useful for programs that pass down different pointers to input characters. It is sometimes used in the **INIT** declaration (see below). Programs which call functions to input characters or have characters in an external array can pass down a value of **(char *)0** for this parameter.

The next parameter, *expbuf*, is a character pointer. It points to the place where the compiled regular expression will be placed.

The parameter *endbuf* is one more than the highest address where the compiled regular expression may be placed. If the compiled expression cannot fit in **(endbuf–expbuf)** bytes, a call to **ERROR(50)** is made.

The parameter *eof* is the character which marks the end of the regular expression. This character is usually a **/**.

Each program that includes the **<regexp.h>** header file must have a **#define** statement for **INIT**. It is used for dependent declarations and initializations. Most often it is used to set a register variable to point to the beginning of the regular expression so that this register variable can be used in the declarations for **GETC**, **PEEKC**, and **UNGETC**. Otherwise it can be used to declare external variables that might be used by **GETC**, **PEEKC** and **UNGETC**. (See **EXAMPLES** below.)

step(), advance()

The first parameter to the **step()** and **advance()** functions is a pointer to a string of characters to be checked for a match. This string should be null terminated.

The second parameter, *expbuf*, is the compiled regular expression which was obtained by a call to the function **compile()**.

The function **step()** returns non-zero if some substring of *string* matches the regular expression in *expbuf* and **0** if there is no match. If there is a match, two external character pointers are set as a side effect to the call to **step()**. The variable **loc1** points to the first character that matched the regular expression; the variable **loc2** points to the character after the last character that matches the regular expression. Thus if the regular expression matches the entire input string, **loc1** will point to the first character of *string* and **loc2** will point to the null at the end of *string*.

The function **advance()** returns non-zero if the initial substring of *string* matches the regular expression in *expbuf*. If there is a match, an external character pointer, **loc2**, is set as a side effect. The variable **loc2** points to the next character in *string* after the last character that matched.

When **advance()** encounters a * or **\{ \}** sequence in the regular expression, it will advance its pointer to the string to be matched as far as possible and will recursively call itself trying to match the rest of the string to the rest of the regular expression. As long as there is no match, **advance()** will back up along the string until it finds a match or reaches the point in the string that initially matched the * or **\{ \}**. It is sometimes desirable to stop this backing up before the initial point in the string is reached. If the external

character pointer **locs** is equal to the point in the string at sometime during the backing up process, **advance( )** will break out of the loop that backs up and will return zero.

The external variables **circf**, **sed**, and **nbra** are reserved.

EXAMPLES

The following is an example of how the regular expression macros and calls might be defined by an application program:

```
#define INIT       register char *sp = instring;
#define GETC       (*sp++)
#define PEEKC      (*sp)
#define UNGETC(c)  (--sp)
#define RETURN(*c)  return;
#define ERROR(c)   regerr

#include <regexp.h>
...
    (void) compile(*argv, expbuf, &expbuf[ESIZE],'\0');
...
    if (step(linebuf, expbuf))
            succeed;
```

DIAGNOSTICS

The function **compile( )** uses the macro **RETURN** on success and the macro **ERROR** on failure (see above). The functions **step( )** and **advance( )** return non-zero on a successful match and zero if there is no match. Errors are:

11 range endpoint too large.

16 bad number.

25 \ *digit* out of range.

36 illegal or missing delimiter.

41 no remembered search string.

42 **\(\)** imbalance.

43 too many **\(**.

44 more than 2 numbers given in **\{ \}**.

45 **}** expected after \.

46 first number exceeds second in **\{ \}**.

49 **[ ]** imbalance.

50 regular expression overflow.

SEE ALSO

regex(5)

NAME

siginfo – signal generation information

SYNOPSIS

#include <siginfo.h>

DESCRIPTION

If a process is catching a signal, it may request information that tells why the system generated that signal (see **sigaction**(2)). If a process is monitoring its children, it may receive information that tells why a child changed state (see **waitid**(2)). In either case, the system returns the information in a structure of type **siginfo_t**, which includes the following information:

```
int             si_signo    /* signal number */
int             si_errno    /* error number */
int             si_code     /* signal code */
union sigval    si_value    /* signal value */
```

si_signo contains the system-generated signal number. For the **waitid**(2) function, **si_signo** is always **SIGCHLD**.

If **si_errno** is non-zero, it contains an error number associated with this signal, as defined in **<errno.h>**.

si_code contains a code identifying the cause of the signal.

If the value of the **si_code** member is **SI_NOINFO**, only the **si_signo** member of **siginfo_t** is meaningful, and the value of all other members is unspecified.

User Signals

If the value of **si_code** is less than or equal to 0, then the signal was generated by a user process (see **kill**(2), **_lwp_kill**(2), **sigqueue**(3R), **sigsend**(2), **abort**(3C), and **raise**(3C)) and the **siginfo** structure contains the following additional information:

```
typedef long pid_t    si_pid    /* sending process ID */
typedef long uid_t    si_uid    /* sending user ID */
```

If the signal was generated by a user process, the following values are defined for **si_code**:

SI_USER the implementation sets **si_code** to **SI_USER** if the signal was sent by **kill**(2), **sigsend**(2), **raise**(3C) or **abort**(3C).

SI_LWP the signal was sent by **_lwp_kill**(2).

SI_QUEUE the signal was sent by **sigqueue**(3R).

SI_TIMER the signal was generated by the expiration of a timer created by **timer_settime**(3R).

SI_ASYNCIO the signal was generated by the completion of an asynchronous I/O request.

SI_MESGQ the signal was generated by the arrival of a message on an empty message queue. (see **mq_notify**(3R)).

si_value contains the application specified value, which is passed to the application's signal-catching function at the time of the signal delivery, if **si_code** is any of **SI_QUEUE**, **SI_TIMER**, **SI_ASYNCHIO**, or **SI_MESGQ**.

System Signals

Otherwise, **si_code** contains a positive value reflecting the reason why the system generated the signal:

Signal	Code	Reason
SIGILL	**ILL_ILLOPC**	illegal opcode
	ILL_ILLOPN	illegal operand
	ILL_ILLADR	illegal addressing mode
	ILL_ILLTRP	illegal trap
	ILL_PRVOPC	privileged opcode
	ILL_PRVREG	privileged register
	ILL_COPROC	co-processor error
	ILL_BADSTK	internal stack error
SIGFPE	**FPE_INTDIV**	integer divide by zero
	FPE_INTOVF	integer overflow
	FPE_FLTDIV	floating point divide by zero
	FPE_FLTOVF	floating point overflow
	FPE_FLTUND	floating point underflow
	FPE_FLTRES	floating point inexact result
	FPE_FLTINV	invalid floating point operation
	FPE_FLTSUB	subscript out of range
SIGSEGV	**SEGV_MAPERR**	address not mapped to object
	SEGV_ACCERR	invalid permissions for mapped object
SIGBUS	**BUS_ADRALN**	invalid address alignment
	BUS_ADRERR	non-existent physical address
	BUS_OBJERR	object specific hardware error
SIGTRAP	**TRAP_BRKPT**	process breakpoint
	TRAP_TRACE	process trace trap
SIGCHLD	**CLD_EXITED**	child has exited
	CLD_KILLED	child was killed
	CLD_DUMPED	child terminated abnormally
	CLD_TRAPPED	traced child has trapped
	CLD_STOPPED	child has stopped
	CLD_CONTINUED	stopped child had continued
SIGPOLL	**POLL_IN**	data input available
	POLL_OUT	output buffers available
	POLL_MSG	input message available
	POLL_ERR	I/O error
	POLL_PRI	high priority input available
	POLL_HUP	device disconnected

In addition, the following signal-dependent information is available for kernel-generated signals:

Signal	Field	Value
SIGILL **SIGFPE**	**caddr_t si_addr**	address of faulting instruction
SIGSEGV **SIGBUS**	**caddr_t si_addr**	address of faulting memory reference
SIGCHLD	**pid_t si_pid** **int si_status**	child process ID exit value or signal
SIGPOLL	**long si_band**	band event for **POLL_IN**, **POLL_OUT**, or **POLL_MSG**

SEE ALSO

_lwp_kill(2), **kill**(2), **sigaction**(2), **sigsend**(2), **waitid**(2), **abort**(3C), **raise**(3C), **aio_read**(3R), **mq_notify**(3R), **sigqueue**(3R), **timer_create**(3R), **timer_settime**(3R), **signal**(5)

NOTES

For **SIGCHLD** signals, if **si_code** is equal to **CLD_EXITED**, then **si_status** is equal to the exit value of the process; otherwise, it is equal to the signal that caused the process to change state. For some implementations, the exact value of **si_addr** may not be available; in that case, **si_addr** is guaranteed to be on the same page as the faulting instruction or memory reference.

NAME

signal – base signals

SYNOPSIS

#include <signal.h>

DESCRIPTION

A signal is an asynchronous notification of an event. A signal is said to be generated for (or sent to) a process when the event associated with that signal first occurs. Examples of such events include hardware faults, timer expiration and terminal activity, as well as the invocation of the **kill**(2) or **sigsend**(2) system calls. In some circumstances, the same event generates signals for multiple processes. A process may request a detailed notification of the source of the signal and the reason why it was generated (see **siginfo**(5)).

A process responds to signals in similar ways whether it is using threads (see **thr_create**(3T)) or it is using lightweight processes (LWPs). Each process may specify a system action to be taken in response to each signal sent to it, called the signal's disposition. All threads or LWPs in the process share the disposition. The set of system signal actions for a process is initialized from that of its parent. Once an action is installed for a specific signal, it usually remains installed until another disposition is explicitly requested by a call to either **sigaction, signal** or **sigset,** or until the process **execs** (see **sigaction**(2) and **signal**(3C)). When a process execs, all signals whose disposition has been set to catch the signal will be set to **SIG_DFL.** Alternatively, a process may request that the system automatically reset the disposition of a signal to **SIG_DFL** after it has been caught (see **sigaction**(2) and **signal**(3C)).

A signal is said to be delivered to a process when a thread or LWP within the process takes the appropriate action for the disposition and signal. Delivery of a signal can be blocked. Each thread or LWP has a signal mask (see **thr_sigsetmask**(3T) or **sigprocmask**(2)) that defines the set of signals currently blocked from delivery to it. The signal mask of the main thread or LWP is inherited from the signal mask of the thread or LWP that created it in the parent process. The selection of the thread or LWP within the process that is to take the appropriate action for the signal is based on the method of signal generation and the signal masks of the threads or LWPs in the receiving process. Signals that are a generated by action of a particular thread or LWP such as hardware faults are delivered to the thread or LWP that caused the signal. Also, see **alarm**(2) for current semantics of delivery of **SIGALRM.** Signals that are directed to a particular thread or LWP (see **thr_kill**(3T) or **_lwp_kill**(2)) are delivered to the targeted thread or LWP. If the selected thread or LWP has blocked the signal, it remains pending on the thread or LWP until it is unblocked. For all other types of signal generation (e.g. **kill**(2), **sigsend**(2), terminal activity, and other external events not ascribable to a particular thread or LWP) one of the threads or LWPs that does not have the signal blocked is selected to process the signal. If all the threads or LWPs within the process block the signal, it remains pending on the process until a thread or LWP in the process unblocks it. If the action associated with a signal is set to ignore the signal then both currently pending and subsequently generated signals of this type are discarded immediately for this process.

The determination of which action is taken in response to a signal is made at the time the signal is delivered to a thread or LWP within the process, allowing for any changes since the time of generation. This determination is independent of the means by which the signal was originally generated.

The signals currently defined by **<signal.h>** are as follows:

Name	Value	Default	Event
SIGHUP	1	Exit	Hangup (see **termio**(7I))
SIGINT	2	Exit	Interrupt (see **termio**(7I))
SIGQUIT	3	Core	Quit (see **termio**(7I))
SIGILL	4	Core	Illegal Instruction
SIGTRAP	5	Core	Trace/Breakpoint Trap
SIGABRT	6	Core	Abort
SIGEMT	7	Core	Emulation Trap
SIGFPE	8	Core	Arithmetic Exception
SIGKILL	9	Exit	Killed
SIGBUS	10	Core	Bus Error
SIGSEGV	11	Core	Segmentation Fault
SIGSYS	12	Core	Bad System Call
SIGPIPE	13	Exit	Broken Pipe
SIGALRM	14	Exit	Alarm Clock
SIGTERM	15	Exit	Terminated
SIGUSR1	16	Exit	User Signal 1
SIGUSR2	17	Exit	User Signal 2
SIGCHLD	18	Ignore	Child Status Changed
SIGPWR	19	Ignore	Power Fail/Restart
SIGWINCH	20	Ignore	Window Size Change
SIGURG	21	Ignore	Urgent Socket Condition
SIGPOLL	22	Exit	Pollable Event (see **streamio**(7I))
SIGSTOP	23	Stop	Stopped (signal)
SIGTSTP	24	Stop	Stopped (user) (see **termio**(7I))
SIGCONT	25	Ignore	Continued
SIGTTIN	26	Stop	Stopped (tty input) (see **termio**(7I))
SIGTTOU	27	Stop	Stopped (tty output) (see **termio**(7I))
SIGVTALRM	28	Exit	Virtual Timer Expired
SIGPROF	29	Exit	Profiling Timer Expired
SIGXCPU	30	Core	CPU time limit exceeded (see **getrlimit**(2))
SIGXFSZ	31	Core	File size limit exceeded (see **getrlimit**(2))
SIGWAITING	32	Ignore	Concurrency signal reserved by threads library
SIGLWP	33	Ignore	Inter-LWP signal reserved by threads library
SIGFREEZE	34	Ignore	Check point Freeze
SIGTHAW	35	Ignore	Check point Thaw
SIGCANCEL	36	Ignore	Cancellation signal reserved by threads library
SIGRTMIN	*	Exit	First real time signal
(SIGRTMIN + 1)	*	Exit	Second real time signal
...			
(SIGRTMAX - 1)	*	Exit	Second-to-last real time signal
SIGRTMAX	*	Exit	Last real time signal

(The symbols **SIGRTMIN** through **SIGRTMAX** are evaluated dynamically in order to permit future configurability)

A process, using a **signal**(3C), **sigset**(3C) or **sigaction**(2) system call, may specify one of three dispositions for a signal: take the default action for the signal, ignore the signal, or catch the signal.

Default Action: SIG_DFL

A disposition of **SIG_DFL** specifies the default action. The default action for each signal is listed in the table above and is selected from the following:

Exit When it gets the signal, the receiving process is to be terminated with all the consequences outlined in **exit**(2).

Core When it gets the signal, the receiving process is to be terminated with all the consequences outlined in **exit**(2). In addition, a ''core image'' of the process is constructed in the current working directory.

Stop When it gets the signal, the receiving process is to stop. When a process is stopped, all the threads and LWPs within the process also stop executing.

Ignore When it gets the signal, the receiving process is to ignore it. This is identical to setting the disposition to **SIG_IGN.**

Ignore Signal: SIG_IGN

A disposition of **SIG_IGN** specifies that the signal is to be ignored. Setting a signal action to **SIG_IGN** for a signal that is pending causes the pending signal to be discarded, whether or not it is blocked. Any queued values pending are also discarded, and the resources used to queue them are released and made available to queue other signals.

Catch Signal: *function address*

A disposition that is a function address specifies that, when it gets the signal, the thread or LWP within the process that is selected to process the signal will execute the signal handler at the specified address. Normally, the signal handler is passed the signal number as its only argument; if the disposition was set with the **sigaction** function however, additional arguments may be requested (see **sigaction**(2)). When the signal handler returns, the receiving process resumes execution at the point it was interrupted, unless the signal handler makes other arrangements. If an invalid function address is specified, results are undefined.

If the disposition has been set with the **sigset** or **sigaction** function, the signal is automatically blocked in the thread or LWP while it is executing the signal catcher. If a **longjmp** (see **setjmp**(3C)) is used to leave the signal catcher, then the signal must be explicitly unblocked by the user (see **signal**(3C) and **sigprocmask**(2)).

If execution of the signal handler interrupts a blocked system call, the handler is executed and the interrupted system call returns a –1 to the calling process with **errno** set to **EINTR.** However, if the **SA_RESTART** flag is set the system call will be transparently restarted.

Some signal–generating functions, such as high resolution timer expiration, asynchronous I/O completion, inter-process message arrival, and the **sigqueue**(3R) function, support the specification of an application defined value, either explicitly as a parameter to the function, or in a **sigevent** structure parameter. The **sigevent** structure is defined by **<signal.h>** and contains at least the following members:

Member Type	Member Name	Description
int	**sigev_notify**	Notification type
int	**sigev_signo**	Signal number
union sigval	**sigev_value**	Signal value

The **sigval** union is defined by **<signal.h>** and contains at least the following members:

Member Type	Member Name	Description
int	**sival_int**	Integer signal value
void *	**sival_ptr**	Pointer signal value

sigev_notify specifies the notification mechanism to use when an asynchronous event occurs. **sigev_notify** may be defined with the following values:

SIGEV_NONE No asynchronous notification is delivered when the event of interest occurs.

SIGEV_SIGNAL A queued signal, with its value application-defined, is generated when the event of interest occurs.

Your implementation may define additional notification mechanisms.

sigev_signo specifies the signal to be generated.

sigev_value references the application defined value to be passed to the signal-catching function at the time of the signal delivery as the **si_value** member of the **siginfo_t** structure.

The **sival_int** member will be used when the application defined value is of type **int**; and the **sival_ptr** member will be used when the application defined value is a pointer.

When a signal is generated by **sigqueue**(3R) or any signal–generating function which supports the specification of an application defined value, the signal is marked pending and, if the **SA_SIGINFO** flag is set for that signal, the signal is queued to the process along with the application specified signal value. Multiple occurrences of signals so generated are queued in FIFO order. If the **SA_SIGINFO** flag is not set for that signal, later occurrences of that signal's generation, when a signal is already queued, are silently discarded.

SEE ALSO

_lwp_sigredirect(2), **alarm**(2), **intro**(2), **exit**(2), **getrlimit**(2), **kill**(2), **pause**(2), **sigaction**(2), **sigaltstack**(2), **sigprocmask**(2), **sigsend**(2), **sigsuspend**(2), **wait**(2), **signal**(3C), **sigsetops**(3C), **sigqueue**(3R), **siginfo**(5), **ucontext**(5)

NOTES

The dispositions of the **SIGKILL** and **SIGSTOP** signals cannot be altered from their default values. The system generates an error if this is attempted.

The **SIGKILL** and **SIGSTOP** signals cannot be blocked. The system silently enforces this restriction.

Whenever a process receives a **SIGSTOP, SIGTSTP, SIGTTIN,** or **SIGTTOU** signal, regardless of its disposition, any pending **SIGCONT** signal are discarded.

Whenever a process receives a **SIGCONT** signal, regardless of its disposition, any pending **SIGSTOP, SIGTSTP, SIGTTIN,** and **SIGTTOU** signals is discarded. In addition, if the process was stopped, it is continued.

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.**

If the disposition of the **SIGCHLD** signal has been set with **signal** or **sigset**, or with **sigaction** and the **SA_NOCLDSTOP** flag has been specified, it will only be sent to the calling process when its children exit; otherwise, it will also be sent when the calling process's children are stopped or continued due to job control.

The name **SIGCLD** is also defined in this header and identifies the same signal as **SIGCHLD. SIGCLD** is provided for backward compatibility, new applications should use **SIGCHLD.**

The disposition of signals that are inherited as **SIG_IGN** should not be changed.

A signal directed via **kill**(2), **sigqueue**(2), **sigsend**(2), terminal activity, and other external events not ascribable to a particular thread or LWP, such as the **SIGXFSZ** or **SIGPIPE** signal, to a multi-threaded process, i.e. a process linked with **-lthread** or **-lpthread**, is routed to this process through a special, designated LWP within this process, called the **aslwp** (short for "Asynchronous Signal LWP").

The **aslwp** within the multi-threaded process receives notification of any signal directed to this process. Upon receiving this notification, the **aslwp** forwards it to a thread within the process that has the signal unmasked. Actual signal delivery to the thread occurs only when the thread is running on an LWP. If no threads exist having that signal number unblocked, the signal remains pending. The **aslwp** is usually blocked in a call to **_signotifywait**(2), waiting for such notifications. The eventual target thread receives the signal via a call to **_lwp_sigredirect**(2), made either by the **aslwp** or the thread itself, redirecting the signal to the LWP that the target thread is running on.

NAME

standards, xpg4 – standards and specifications supported by Solaris

DESCRIPTION

Solaris 2.5 supports the following standards and specifications:

XPG4 The X/Open Portability Guide, Issue 4 (XPG4) specification contains internationalized utilities, headers, and interfaces.

If the behavior required by XPG4 utilities conflicts with existing Solaris behavior, the original Solaris version is not changed. Rather, a new version that is XPG4-compliant can be found in **/usr/xpg4/bin**. Therefore, when setting the **PATH** (**sh** or **ksh**) or **path** (**csh**) environment variables, **/usr/xpg4/bin** should precede any other directories in which XPG4 utilities are found, such as **/usr/bin**, **/bin**, **/usr/ucb** and **/usr/ccs/bin** for applications that want to take advantage of XPG4 features. Note that XPG4 includes all of the utilities in IEEE Std 1003.2-1992 and IEEE Std 1003.2a-1992 (together, commonly called POSIX.2).

An XPG4-compliant implementation must include an ANSI X3.159-1989 (ANSI C Language) standard conforming compilation system and the **cc** and **c89** utilities. Solaris 2.5 was tested with the **cc** and **c89** utilities and the compilation system in the SPROcc package provided by **SPARCompiler™ C 4.0** in the SPARC environment and **ProCompiler™ 3.0.1** (it is expected that a patch will be needed for the 2.5 FCS release) in the x86 environment. As specified by XPG4, applications must define the **_XOPEN_VERSION** macro to have value **4** before including any headers specified by XPG4 to get the features specified by XPG4. When **cc** is used to link applications, **/usr/ccs/lib/values-xpg4.o** must be specified on any link/load command line.

SEE ALSO

Standards Conformance Guide

NAME

stat – data returned by stat system call

SYNOPSIS

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

DESCRIPTION

The system calls **stat**, **lstat** and **fstat** return data in a **stat** structure, which is defined in **stat.h**.

The constants used in the **st_mode** field are also defined in this file:

```
#define   S_IFMT    /* type of file */
#define   S_IAMB    /* access mode bits */
#define   S_IFIFO   /* fifo */
#define   S_IFCHR   /* character special */
#define   S_IFDIR   /* directory */
#define   S_IFNAM   /* XENIX special named file */
#define   S_INSEM   /* XENIX semaphore subtype of IFNAM */
#define   S_INSHD   /* XENIX shared data subtype of IFNAM */
#define   S_IFBLK   /* block special */
#define   S_IFREG   /* regular */
#define   S_IFLNK   /* symbolic link */
#define   S_ISUID   /* set user id on execution */
#define   S_ISGID   /* set group id on execution */
#define   S_ISVTX   /* save swapped text even after use */
#define   S_IREAD   /* read permission, owner */
#define   S_IWRITE  /* write permission, owner */
#define   S_IEXEC   /* execute/search permission, owner */
#define   S_ENFMT   /* record locking enforcement flag */
#define   S_IRWXU   /* read, write, execute: owner */
#define   S_IRUSR   /* read permission: owner */
#define   S_IWUSR   /* write permission: owner */
#define   S_IXUSR   /* execute permission: owner */
#define   S_IRWXG   /* read, write, execute: group */
#define   S_IRGRP   /* read permission: group */
#define   S_IWGRP   /* write permission: group */
#define   S_IXGRP   /* execute permission: group */
#define   S_IRWXO   /* read, write, execute: other */
#define   S_IROTH   /* read permission: other */
#define   S_IWOTH   /* write permission: other */
#define   S_IXOTH   /* execute permission: other */
```

The following macros are for POSIX conformance:

```
#define  S_ISBLK(mode)    block special file
#define  S_ISCHR(mode)    character special file
#define  S_ISDIR(mode)    directory file
#define  S_ISFIFO(mode)   pipe or fifo file
#define  S_ISREG(mode)    regular file
```

SEE ALSO **stat**(2), **types**(5)

NAME

stdarg – handle variable argument list

SYNOPSIS

```
#include <stdarg.h>

va_list pvar;

void va_start(va_list pvar, parmN);

type va_arg(va_list pvar, type);

void va_end(va_list pvar);
```

DESCRIPTION

This set of macros allows portable procedures that accept variable numbers of arguments of variable types to be written. Routines that have variable argument lists (such as **printf**) but do not use *stdarg* are inherently non-portable, as different machines use different argument-passing conventions.

va_list is a type defined for the variable used to traverse the list.

The **va_start()** macro is invoked before any access to the unnamed arguments and initializes **pvar** for subsequent use by **va_arg()** and **va_end()**. The parameter *parmN* is the identifier of the rightmost parameter in the variable parameter list in the function definition (the one just before the **, ...**). If this parameter is declared with the **register** storage class or with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions, the behavior is undefined.

The parameter *parmN* is required under strict ANSI C compilation. In other compilation modes, *parmN* need not be supplied and the second parameter to the **va_start()** macro can be left empty (for example, **va_start(pvar,);**). This allows for routines that contain no parameters before the **...** in the variable parameter list.

The **va_arg()** macro expands to an expression that has the type and value of the next argument in the call. The parameter **pvar** should have been previously initialized by **va_start()**. Each invocation of **va_arg()** modifies **pvar** so that the values of successive arguments are returned in turn. The parameter *type* is the type name of the next argument to be returned. The type name must be specified in such a way so that the type of a pointer to an object that has the specified type can be obtained simply by postfixing a * to *type*. If there is no actual next argument, or if *type* is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), the behavior is undefined.

The **va_end()** macro is used to clean up.

Multiple traversals, each bracketed by **va_start** and **va_end**, are possible.

EXAMPLE

This example gathers into an array a list of arguments that are pointers to strings (but not more than **MAXARGS** arguments) with function **f1**, then passes the array as a single argument to function **f2**. The number of pointers is specified by the first argument to **f1**.

```
#include <stdarg.h>
#define MAXARGS      31

void f1(int n_ptrs, ...)
{
        va_list ap;
        char *array[MAXARGS];
        int ptr_no = 0;

        if (n_ptrs > MAXARGS)
                n_ptrs = MAXARGS;
        va_start(ap, n_ptrs);
        while (ptr_no < n_ptrs)
                array[ptr_no++] = va_arg(ap, char*);
        va_end(ap);
        f2(n_ptrs, array);
}
```

Each call to **f1** shall have visible the definition of the function or a declaration such as

```
void f1(int, ...)
```

SEE ALSO

vprintf(3S)

NOTES

It is up to the calling routine to specify in some manner how many arguments there are, since it is not always possible to determine the number of arguments from the stack frame. For example, **execl** is passed a zero pointer to signal the end of the list. **printf** can tell how many arguments there are by the format. It is non-portable to specify a second argument of **char**, **short**, or **float** to **va_arg**, because arguments seen by the called function are not **char**, **short**, or **float**. C converts **char** and **short** arguments to **int** and converts **float** arguments to **double** before passing them to a function.

NAME

sticky – mark files for special treatment

DESCRIPTION

The *sticky bit* (file mode bit **01000**, see **chmod**(2)) is used to indicate special treatment of certain files and directories. A directory for which the sticky bit is set restricts deletion of files it contains. A file in a sticky directory may only be removed or renamed by a user who has write permission on the directory, and either owns the file, owns the directory, or is the super-user. This is useful for directories such as **/tmp**, which must be publicly writable, but should deny users permission to arbitrarily delete or rename the files of others.

If the sticky bit is set on a regular file and no execute bits are set, the system's page cache will not be used to hold the file's data. This bit is normally set on swap files of diskless clients so that accesses to these files do not flush more valuable data from the system's cache. Moreover, by default such files are treated as swap files, whose inode modification times may not necessarily be correctly recorded on permanent storage.

Any user may create a sticky directory. See **chmod** for details about modifying file modes.

FILES

/tmp

SEE ALSO

chmod(1), **chmod**(2), **chown**(2), **mkdir**(2)

BUGS

mkdir(2) will not create a directory with the sticky bit set.

NAME

term – conventional names for terminals

DESCRIPTION

Terminal names are maintained as part of the shell environment in the environment variable **TERM** (see **sh**(1), **profile**(4), and **environ**(5)). These names are used by certain commands (for example, **tabs**, **tput**, and **vi**) and certain functions (for example, see **curses**(3X)).

Files under **/usr/share/lib/terminfo** are used to name terminals and describe their capabilities. These files are in the format described in **terminfo**(4). Entries in **terminfo** source files consist of a number of comma-separated fields. To print a description of a terminal *term*, use the command **infocmp –I** *term* (see **infocmp**(1M)). White space after each comma is ignored. The first line of each terminal description in the **terminfo** database gives the names by which **terminfo** knows the terminal, separated by bar (|) characters. The first name given is the most common abbreviation for the terminal (this is the one to use to set the environment variable **TERMINFO** in **$HOME/.profile**; see **profile**(4)), the last name given should be a long name fully identifying the terminal, and all others are understood as synonyms for the terminal name. All names but the last should contain no blanks and must be unique in the first 14 characters; the last name may contain blanks for readability.

Terminal names (except for the last, verbose entry) should be chosen using the following conventions. The particular piece of hardware making up the terminal should have a root name chosen, for example, for the AT&T 4425 terminal, **att4425**. This name should not contain hyphens, except that synonyms may be chosen that do not conflict with other names. Up to 8 characters, chosen from the set **a** through **z** and **0** through **9**, make up a basic terminal name. Names should generally be based on original vendors rather than local distributors. A terminal acquired from one vendor should not have more than one distinct basic name. Terminal sub-models, operational modes that the hardware can be in, or user preferences should be indicated by appending a hyphen and an indicator of the mode. Thus, an AT&T 4425 terminal in 132 column mode is **att4425–w**. The following suffixes should be used where possible:

Suffix	Meaning	Example
–w	Wide mode (more than 80 columns)	**att4425–w**
–am	With auto. margins (usually default)	**vt100–am**
–nam	Without automatic margins	**vt100–nam**
–n	Number of lines on the screen	**aaa–60**
–na	No arrow keys (leave them in local)	**c100–na**
–np	Number of pages of memory	**c100–4p**
–rv	Reverse video	**att4415–rv**

To avoid conflicts with the naming conventions used in describing the different modes of a terminal (for example, **–w**), it is recommended that a terminal's root name not contain hyphens. Further, it is good practice to make all terminal names used in the **terminfo**(4) database unique. Terminal entries that are present only for inclusion in other entries via the **use=** facilities should have a '**+**' in their name, as in **4415+nl**.

Here are some of the known terminal names: (For a complete list, enter the command **ls -C /usr/share/lib/terminfo/?**).

2621,hp2621	Hewlett-Packard 2621 series
2631	Hewlett-Packard 2631 line printer
2631–c	Hewlett-Packard 2631 line printer, compressed mode
2631–e	Hewlett-Packard 2631 line printer, expanded mode
2640,hp2640	Hewlett-Packard 2640 series
2645,hp2645	Hewlett-Packard 2645 series
3270	IBM Model 3270
33,tty33	AT&T Teletype Model 33 KSR
35,tty35	AT&T Teletype Model 35 KSR
37,tty37	AT&T Teletype Model 37 KSR
4000a	Trendata 4000a
4014,tek4014	TEKTRONIX 4014
40,tty40	AT&T Teletype Dataspeed 40/2
43,tty43	AT&T Teletype Model 43 KSR
4410,5410	AT&T 4410/5410 in 80-column mode, version 2
4410–nfk,5410–nfk	AT&T 4410/5410 without function keys, version 1
4410–nsl,5410–nsl	AT&T 4410/5410 without pln defined
4410–w,5410–w	AT&T 4410/5410 in 132-column mode
4410v1,5410v1	AT&T 4410/5410 in 80-column mode, version 1
4410v1–w,5410v1–w	AT&T 4410/5410 in 132-column mode, version 1
4415,5420	AT&T 4415/5420 in 80-column mode
4415–nl,5420–nl	AT&T 4415/5420 without changing labels
4415–rv,5420–rv	AT&T 4415/5420 80 columns in reverse video
4415–rv–nl,5420–rv–nl	AT&T 4415/5420 reverse video without changing labels
4415–w,5420–w	AT&T 4415/5420 in 132-column mode
4415–w–nl,5420–w–nl	AT&T 4415/5420 in 132-column mode without changing labels
4415–w–rv,5420–w–rv	AT&T 4415/5420 132 columns in reverse video
4418,5418	AT&T 5418 in 80-column mode
4418–w,5418–w	AT&T 5418 in 132-column mode
4420	AT&T Teletype Model 4420
4424	AT&T Teletype Model 4424
4424-2	AT&T Teletype Model 4424 in display function group ii
4425,5425	AT&T 4425/5425
4425–fk,5425–fk	AT&T 4425/5425 without function keys

4425–nl,5425–nl	AT&T 4425/5425 without changing labels in 80-column mode
4425–w,5425–w	AT&T 4425/5425 in 132-column mode
4425–w–fk,5425–w–fk	AT&T 4425/5425 without function keys in 132-column mode
4425–nl–w,5425–nl–w	AT&T 4425/5425 without changing labels in 132-column mode
4426	AT&T Teletype Model 4426S
450	DASI 450 (same as Diablo 1620)
450–12	DASI 450 in 12-pitch mode
500,att500	AT&T-IS 500 terminal
510,510a	AT&T 510/510a in 80-column mode
513bct,att513	AT&T 513 bct terminal
5320	AT&T 5320 hardcopy terminal
5420_2	AT&T 5420 model 2 in 80-column mode
5420_2–w	AT&T 5420 model 2 in 132-column mode
5620,dmd	AT&T 5620 terminal 88 columns
5620–24,dmd–24	AT&T Teletype Model DMD 5620 in a 24x80 layer
5620–34,dmd–34	AT&T Teletype Model DMD 5620 in a 34x80 layer
610,610bct	AT&T 610 bct terminal in 80-column mode
610–w,610bct–w	AT&T 610 bct terminal in 132-column mode
630,630MTG	AT&T 630 Multi-Tasking Graphics terminal
7300,pc7300,unix_pc	AT&T UNIX PC Model 7300
735,ti	Texas Instruments TI735 and TI725
745	Texas Instruments TI745
dumb	generic name for terminals that lack reverse line-feed and other special escape sequences
hp	Hewlett-Packard (same as 2645)
lp	generic name for a line printer
pt505	AT&T Personal Terminal 505 (22 lines)
pt505–24	AT&T Personal Terminal 505 (24-line mode)
sync	generic name for synchronous Teletype Model 4540-compatible terminals

Commands whose behavior depends on the type of terminal should accept arguments of the form **–T***term* where *term* is one of the names given above; if no such argument is present, such commands should obtain the terminal type from the environment variable **TERM**, which, in turn, should contain *term*.

FILES

/usr/share/lib/terminfo/?/*
compiled terminal description database

SEE ALSO

sh(1), **stty**(1), **tabs**(1), **tput**(1), **vi**(1), **infocmp**(1M), **curses**(3X), **profile**(4), **terminfo**(4), **environ**(5)

NAME

types – primitive system data types

SYNOPSIS

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

DESCRIPTION

The data types defined in **types.h** are used in UNIX System code. Some data of these types are accessible to user code:

```
typedef  struct { int r[1]; } *physadr;
typedef  long               clock_t;
typedef  long               daddr_t;
typedef  char *             caddr_t;
typedef  unsigned char      unchar;
typedef  unsigned short     ushort;
typedef  unsigned int       uint;
typedef  unsigned long      ulong;
typedef  unsigned long      ino_t;
typedef  long               uid_t;
typedef  long               gid_t;
typedef  ulong              nlink_t;
typedef  ulong              mode_t;
typedef  short              cnt_t;
typedef  long               time_t;
typedef  int                label_t[10];
typedef  ulong              dev_t;
typedef  long               off_t;
typedef  long               pid_t;
typedef  long               paddr_t;
typedef  int                key_t;
typedef  unsigned char      use_t;
typedef  short              sysid_t;
typedef  short              index_t;
typedef  short              lock_t;
typedef  unsigned int       size_t;
typedef  long               clock_t;
typedef  long               pid_t;
```

The form **daddr_t** is used for disk addresses except in an inode on disk. Times are encoded in seconds since 00:00:00 UTC, January 1, 1970. The major and minor parts of a device code specify kind and unit number of a device and are installation-dependent. Offsets are measured in bytes from the beginning of a file. The **label_t** variables are used to save the processor state while another process is running.

NAME

ucontext – user context

SYNOPSIS

```
#include <ucontext.h>
```

DESCRIPTION

The **ucontext** structure defines the context of a thread of control within an executing process.

This structure includes at least the following members:

```
ucontext_t  uc_link
sigset_t    uc_sigmask
stack_t     uc_stack
mcontext_t  uc_mcontext
```

uc_link is a pointer to the context that to be resumed when this context returns. If **uc_link** is equal to 0, then this context is the main context, and the process exits when this context returns.

uc_sigmask defines the set of signals that are blocked when this context is active [see **sigprocmask**(2)].

uc_stack defines the stack used by this context [see **sigaltstack**(2)].

uc_mcontext contains the saved set of machine registers and any implementation specific context data. Portable applications should not modify or access **uc_mcontext**.

SEE ALSO

getcontext(2), **sigaction**(2), **sigaltstack**(2), **sigprocmask**(2), **makecontext**(3C)

NAME

values – machine-dependent values

SYNOPSIS

#include <values.h>

DESCRIPTION

This file contains a set of manifest constants, conditionally defined for particular processor architectures.

The model assumed for integers is binary representation (one's or two's complement), where the sign is represented by the value of the high-order bit.

BITS(*type***)** The number of bits in a specified type (for example, **int**).

HIBITS The value of a short integer with only the high-order bit set.

HIBITL The value of a long integer with only the high-order bit set.

HIBITI The value of a regular integer with only the high-order bit set.

MAXSHORT The maximum value of a signed short integer.

MAXLONG The maximum value of a signed long integer.

MAXINT The maximum value of a signed regular integer.

MAXFLOAT, **LN_MAXFLOAT**
The maximum value of a single-precision floating-point number, and its natural logarithm.

MAXDOUBLE, **LN_MAXDOUBLE**
The maximum value of a double-precision floating-point number, and its natural logarithm.

MINFLOAT, **LN_MINFLOAT**
The minimum positive value of a single-precision floating-point number, and its natural logarithm.

MINDOUBLE, **LN_MINDOUBLE**
The minimum positive value of a double-precision floating-point number, and its natural logarithm.

FSIGNIF The number of significant bits in the mantissa of a single-precision floating-point number.

DSIGNIF The number of significant bits in the mantissa of a double-precision floating-point number.

SEE ALSO

intro(3)**, math**(5)

NAME

varargs – handle variable argument list

SYNOPSIS

```
#include <varargs.h>
va_alist
va_dcl
va_list pvar;
void va_start(va_list pvar);
type va_arg(va_list pvar, type);
void va_end(va_list pvar);
```

DESCRIPTION

This set of macros allows portable procedures that accept variable argument lists to be written. Routines that have variable argument lists (such as **printf**(3S)) but do not use **varargs** are inherently non-portable, as different machines use different argument-passing conventions.

va_alist is used as the parameter list in a function header.

va_dcl is a declaration for **va_alist**. No semicolon should follow **va_dcl**.

va_list is a type defined for the variable used to traverse the list.

va_start is called to initialize **pvar** to the beginning of the list.

va_arg will return the next argument in the list pointed to by **pvar**. *type* is the type the argument is expected to be. Different types can be mixed, but it is up to the routine to know what type of argument is expected, as it cannot be determined at runtime.

va_end is used to clean up.

Multiple traversals, each bracketed by **va_start** and **va_end**, are possible.

EXAMPLE

This example is a possible implementation of **execl** (see **exec**(2)).

```
#include <unistd.h>
#include <varargs.h>
#define MAXARGS      100

/*      execl is called by
                execl(file, arg1, arg2, ..., (char *)0);
*/
execl(va_alist)
va_dcl
{
        va_list ap;
        char *file;
        char *args[MAXARGS];                    /* assumed big enough*/
        int argno = 0;

        va_start(ap);
```

```
        file = va_arg(ap, char *);
        while ((args[argno++] = va_arg(ap, char *)) != 0)
                ;
        va_end(ap);
        return execv(file, args);
}
```

SEE ALSO

exec(2), **printf**(3S), **vprintf**(3S), **stdarg**(5)

NOTES

It is up to the calling routine to specify in some manner how many arguments there are, since it is not always possible to determine the number of arguments from the stack frame. For example, **execl** is passed a zero pointer to signal the end of the list. **printf** can tell how many arguments are there by the format.

It is non-portable to specify a second argument of **char**, **short**, or **float** to **va_arg**, since arguments seen by the called function are not **char**, **short**, or **float**. C converts **char** and **short** arguments to **int** and converts **float** arguments to **double** before passing them to a function.

stdarg is the preferred interface.

NAME

vgrindefs – vgrind's language definition data base

SYNOPSIS

/usr/lib/vgrindefs

DESCRIPTION

vgrindefs contains all language definitions for **vgrind**(1). Capabilities in **vgrindefs** are of two types: Boolean capabilities which indicate that the language has some particular feature and string capabilities which give a regular expression or keyword list. Entries may continue onto multiple lines by giving a \ as the last character of a line. Lines starting with # are comments.

Capabilities

The following table names and describes each capability.

Name	Type	Description
ab	str	Regular expression for the start of an alternate form comment
ae	str	Regular expression for the end of an alternate form comment
bb	str	Regular expression for the start of a block
be	str	Regular expression for the end of a lexical block
cb	str	Regular expression for the start of a comment
ce	str	Regular expression for the end of a comment
id	str	String giving characters other than letters and digits that may legally occur in identifiers (default '_')
kw	str	A list of keywords separated by spaces
lb	str	Regular expression for the start of a character constant
le	str	Regular expression for the end of a character constant
oc	bool	Present means upper and lower case are equivalent
pb	str	Regular expression for start of a procedure
pl	bool	Procedure definitions are constrained to the lexical level matched by the 'px' capability
px	str	A match for this regular expression indicates that procedure definitions may occur at the next lexical level. Useful for lisp-like languages in which procedure definitions occur as subexpressions of defuns.
sb	str	Regular expression for the start of a string
se	str	Regular expression for the end of a string
tc	str	Use the named entry as a continuation of this one
tl	bool	Present means procedures are only defined at the top lexical level

Regular Expressions

vgrindefs uses regular expressions similar to those of **ex**(1) and **lex**(1). The characters '^', '$', ':', and '\' are reserved characters and must be 'quoted' with a preceding \ if they are to be included as normal characters. The metasymbols and their meanings are:

$	The end of a line
^	The beginning of a line
\d	A delimiter (space, tab, newline, start of line)
\a	Matches any string of symbols (like '.*' in lex)
\p	Matches any identifier. In a procedure definition (the 'pb' capability) the string

that matches this symbol is used as the procedure name.

() Grouping

| Alternation

? Last item is optional

\e Preceding any string means that the string will not match an input string if the input string is preceded by an escape character (\). This is typically used for languages (like C) that can include the string delimiter in a string by escaping it.

Unlike other regular expressions in the system, these match words and not characters. Hence something like '(tramp | steamer)flies?' would match 'tramp', 'steamer', 'trampflies', or 'steamerflies'. Contrary to some forms of regular expressions, **vgrindef** alternation binds very tightly. Grouping parentheses are likely to be necessary in expressions involving alternation.

Keyword List

The keyword list is just a list of keywords in the language separated by spaces. If the 'oc' boolean is specified, indicating that upper and lower case are equivalent, then all the keywords should be specified in lower case.

EXAMPLE

The following entry, which describes the C language, is typical of a language entry.

```
C | c | the C programming language:\
   :pb=^\d?*?\d?\p\d?(\a?\)(\d|{):bb={:be=}:cb=/*:ce=*/:sb=":se=\e":\
   :le=\e':tl:\
   :kw=asm auto break case char continue default do double else enum\
   extern float for fortran goto if int long register return short\
   sizeof static struct switch typedef union unsigned void while #define\
   #else #endif #if #ifdef #ifndef #include #undef # define endif\
   ifdef ifndef include undef defined:
```

Note that the first field is just the language name (and any variants of it). Thus the C language could be specified to **vgrind**(1) as 'c' or 'C'.

FILES

/usr/lib/vgrindefs file containing vgrind descriptions

SEE ALSO

ex(1), **lex**(1), **troff**(1), **vgrind**(1)

NAME

wstat – wait status

SYNOPSIS

#include <sys/wait.h>

DESCRIPTION

When a process waits for status from its children via either the **wait** or **waitpid** function, the status returned may be evaluated with the following macros, defined in **<sys/wait.h>**. These macros evaluate to integral expressions. The *stat* argument to these macros is the integer value returned from **wait** or **waitpid**.

WIFEXITED(*stat***)** Evaluates to a non-zero value if status was returned for a child process that terminated normally.

WEXITSTATUS(*stat***)** If the value of **WIFEXITED(***stat***)** is non-zero, this macro evaluates to the exit code that the child process passed to **_exit( )** (see **exit**(2)) or **exit**(3C), or the value that the child process returned from **main**.

WIFSIGNALED(*stat***)** Evaluates to a non-zero value if status was returned for a child process that terminated due to the receipt of a signal.

WTERMSIG(*stat***)** If the value of **WIFSIGNALED(***stat***)** is non-zero, this macro evaluates to the number of the signal that caused the termination of the child process.

WIFSTOPPED(*stat***)** Evaluates to a non-zero value if status was returned for a child process that is currently stopped.

WSTOPSIG(*stat***)** If the value of **WIFSTOPPED(***stat***)** is non-zero, this macro evaluates to the number of the signal that caused the child process to stop.

WIFCONTINUED(*stat*) Evaluates to a non-zero value if status was returned for a child process that has continued.

WCOREDUMP(*stat*) If the value of **WIFSIGNALED** (*stat*) is non-zero, this macro evaluates to a non-zero value if a core image of the terminated child was created.

SEE ALSO

exit(2), **wait**(2), **waitpid**(2), **exit**(3C)

Index

I

L

M

N

P

R