NAME

intro – introduction to miscellany

DESCRIPTION

This section describes miscellaneous facilities such as macro packages, character set tables, etc.

(5)

NAME

ascii – map of ASCII character set

DESCRIPTION

ascii is a map of the ASCII character set, giving both octal and hexadecimal equivalents of each character, to be printed as needed. It contains:

```
|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 |

| 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 |
```

(5)

NAME

environ – user environment

DESCRIPTION

An array of strings called the "environment" is made available by *exec*(2) when a process begins. By convention, these strings have the form "name=value". The following names are used by various commands:

PATH The sequence of directory prefixes that *sh*(1), *time*(1), *nice*(1), *nohup*(1), etc., apply in searching for a file known by an incomplete path name. The prefixes are separated by colons (:). *Login*(1) sets **PATH=:/bin:/usr/bin.**

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

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

TZ Time zone information. The format is **xxx***n***zzz** where **xxx** is standard local time zone abbreviation, *n* is the difference in hours from GMT, and **zzz** is the abbreviation for the daylight-saving local time zone, if any; for example, **EST5EDT.**

Further names 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 ALSO

exec(2).
env(1), login(1), sh(1), nice(1), nohup(1), time(1), tplot(1G) in the *User's Reference Manual.*

(5)

NAME

fcntl – file control options

SYNOPSIS

#include <fcntl.h>

DESCRIPTION

The *fcntl*(2) function provides for control over open files. This include file describes *requests* and *arguments* to *fcntl* and *open*(2).

```
/* Flag values accessible to open(2) and fcntl(2) */
/*  (The first three can only be set by open) */
#define O_RDONLY  0
#define O_WRONLY  1
#define O_RDWR    2
#define O_NDELAY  04       /* Non-blocking I/O */
#define O_APPEND  010      /* append (writes guaranteed at the end) */
#define O_SYNC    020      /* synchronous write option */

/* Flag values accessible only to open(2) */
#define O_CREAT   00400    /* open with file create (uses third open arg)*/
#define O_TRUNC   01000    /* open with truncation */
#define O_EXCL    02000    /* exclusive open */

/* fcntl(2) requests */
#define F_DUPFD   0        /* Duplicate fildes */
#define F_GETFD   1        /* Get fildes flags */
#define F_SETFD   2        /* Set fildes flags */
#define F_GETFL   3        /* Get file flags */
#define F_SETFL   4        /* Set file flags */
#define F_GETLK   5        /* Get file lock */
#define F_SETLK   6        /* Set file lock */
#define F_SETLKW  7        /* Set file lock and wait */
#define F_CHKFL   8        /* Check legality of file flag changes */
```

```
/* file segment locking control structure */
struct flock {
        short l_type;
        short l_whence;
        long  l_start;
        long  l_len;          /* if 0 then until EOF */
        short l_sysid;        /* returned with F_GETLK*/
        short l_pid;          /* returned with F_GETLK*/
}

/* file segment locking types */
#define F_RDLCK 01      /* Read lock */
#define F_WRLCK 02      /* Write lock */
#define F_UNLCK 03      /* Remove locks */
```

SEE ALSO

fcntl(2), open(2).

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 an 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.

For the definitions of various machine-dependent "constants," see the description of the *<values.h>* header file.

SEE ALSO

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

(5)

NAME

prof – profile within a function

SYNOPSIS

#define MARK
#include <prof.h>

void MARK (name)

DESCRIPTION

MARK will introduce a mark called *name* that will be treated the same as a function entry point. Execution of the mark will add to a counter for that mark, and program-counter time spent will be 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 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 file *<prof.h>* is included. This may be defined by a preprocessor directive as in the synopsis, or by a command line argument, i.e:

```
cc –p –DMARK foo.c
```

If MARK is not defined, the *MARK*(name) statements may be left in the source files containing them and will be ignored.

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>
foo( )
{
        int i, j;
        .
        .
        .
        MARK(loop1);
        for (i = 0; i < 2000; i++) {
                . . .
                }
                MARK(loop2);
                for (j = 0; j < 2000; j++) {
                        . . .
                }
        }
```

SEE ALSO

prof(1), profil(2), monitor(3C).

NAME

regexp – regular expression compile and match routines

SYNOPSIS

```
#define INIT <declarations>
#define GETC() <getc code>
#define PEEKC() <peekc code>
#define UNGETC(c) <ungetc code>
#define RETURN(pointer) <return code>
#define ERROR(val) <error code>

#include <regexp.h>

char *compile (instring, expbuf, endbuf, eof)
char *instring, *expbuf, *endbuf;
int eof;

int step (string, expbuf)
char *string, *expbuf;

extern char *loc1, *loc2, *locs;

extern int circf, sed, nbra;
```

DESCRIPTION

This page describes general-purpose regular expression matching routines in the form of *ed*(1), defined in <regexp.h> . Programs such as *ed*(1), *sed*(1), *grep*(1), *bs*(1), *expr*(1), etc., which perform regular expression matching use this source file. In this way, only this file need be changed to maintain regular expression compatibility.

The interface to this file is unpleasantly complex. Programs that include this file must have the following five macros declared before the "#include <regexp.h>" statement. These macros are used by the *compile* routine.

GETC() Return the value of the next character in the regular expression pattern. Successive calls to GETC() should return successive characters of the regular expression.

PEEKC() Return the next character in the regular expression. Successive calls to PEEKC() should return the same character [which should also be the next character returned by GETC()].

UNGETC(*c*) Cause the argument *c* to be returned by the next call to GETC( ) [and PEEKC( )]. No more that one character of pushback is ever needed and this character is guaranteed to be the last character read by GETC( ). The value of the macro UNGETC(*c*) is always ignored.

RETURN(*pointer*) This macro is used on normal exit of the *compile* routine. The value of the argument *pointer* 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 is the abnormal return from the *compile* routine. The argument *val* is an error number (see table below for meanings). This call should never return.

ERROR	MEANING
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.

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. For example, in *ed*(1), this character is usually a /.

Each program that includes this file must have a **#define** statement for INIT. This definition will be placed right after the declaration for the function *compile* and the opening curly brace ({). It is used for dependent declarations and initializations. Most often it is used to set a register variable to point 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 the example below of the declarations taken from *grep*(1).

There are other functions in this file which perform actual regular expression matching, one of which is the function *step*. The call to *step* is as follows:

```
step(string, expbuf)
```

The first parameter to *step* 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 of the function *compile*.

The function *step* returns non-zero if the given string matches the regular expression, and zero if the expressions do not match. If there is a match, two external character pointers are set as a side effect to the call to *step*. The variable set in *step* is *loc1*. This is a pointer to the first character that matched the regular expression. The variable *loc2*, which is set by the function *advance*, points to the character after the last character that matches the regular expression. Thus if the regular expression matches the entire line, *loc1* will point to the first character of *string* and *loc2* will point to the null at the end of *string*.

Step uses the external variable *circf* which is set by *compile* if the regular expression begins with ^. If this is set then *step* will try to match the regular expression to the beginning of the string only. If more than one regular expression is to be compiled before the first is executed the value of *circf* should be saved for each compiled expression and *circf* should be set to that saved value before each call to *step*.

The function *advance* is called from *step* with the same arguments as *step*. The purpose of *step* is to step through the *string* argument and call *advance* until *advance* returns non-zero indicating a match or until the end of *string* is reached. If one wants to constrain *string* to the beginning of the line in all cases, *step* need not be called; simply call *advance*.

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. This is used by *ed*(1) and *sed*(1) for substitutions done globally (not just the first occurrence, but the whole line) so, for example, expressions like **s/y*//g** do not loop forever.

The additional external variables *sed* and *nbra* are used for special purposes.

EXAMPLES

The following is an example of how the regular expression macros and calls look from *grep*(1):

```
#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();
```

SEE ALSO

ed(1), expr(1), grep(1), sed(1) in the *User's Reference Manual*.

NAME

stat – data returned by stat system call

SYNOPSIS

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

DESCRIPTION

The system calls *stat* and *fstat* return data whose structure is defined by this include file. The encoding of the field *st_mode* is defined in this file also.

Structure of the result of stat

```
struct   stat
{
         dev_t    st_dev;
         ushort   st_ino;
         ushort   st_mode;
         short    st_nlink;
         ushort   st_uid;
         ushort   st_gid;
         dev_t    st_rdev;
         off_t    st_size;
         time_t   st_atime;
         time_t   st_mtime;
         time_t   st_ctime;
};

#define S_IFMT    0170000 /* type of file */
#define S_IFDIR   0040000 /* directory */
#define S_IFCHR   0020000 /* character special */
#define S_IFBLK   0060000 /* block special */
#define S_IFREG   0100000 /* regular */
#define S_IFIFO   0010000 /* fifo */
#define S_ISUID   04000   /* set user id on execution */
#define S_ISGID   02000   /* set group id on execution */
#define S_ISVTX   01000   /* save swapped text even after use */
#define S_IREAD   00400   /* read permission, owner */
#define S_IWRITE  00200   /* write permission, owner */
#define S_IEXEC   00100   /* execute/search permission, owner */
#define S_ENFMT   S_ISGID /* record locking enforcement flag */
```

```
#define S_IRWXU 00700    /* read,write, execute: owner */
#define S_IRUSR 00400    /* read permission: owner */
#define S_IWUSR 00200    /* write permission: owner */
#define S_IXUSR 00100    /* execute permission: owner */
#define S_IRWXG 00070    /* read, write, execute: group */
#define S_IRGRP 00040    /* read permission: group */
#define S_IWGRP 00020    /* write permission: group */
#define S_IXGRP 00010    /* execute permission: group */
#define S_IRWXO 00007    /* read, write, execute: other */
#define S_IROTH 00004    /* read permission: other */
#define S_IWOTH 00002    /* write permission: other */
#define S_IXOTH 00001    /* execute permission: other */
```

SEE ALSO

stat(2), types(5).

NAME

term – conventional names for terminals

DESCRIPTION

These names are used by certain commands (e.g., *man*(1), *tabs*(1), *tput*(1), *vi*(1) and *curses*(3X)) and are maintained as part of the shell environment in the environment variable **TERM** (see *sh*(1), *profile*(4), and *environ*(5)).

Entries in *terminfo*(4) source files consist of a number of comma-separated fields. (To obtain the source description for a terminal, use the **–I** option of *infocmp*(1M).) White space after each comma is ignored. The first line of each terminal description in the *terminfo*(4) database gives the names by which *terminfo*(4) 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 VT100 terminal, **vt100**. 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 [a–z0–9], 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, a VT100 terminal in 132 column mode would be **vt100–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 (e.g., –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.

Some of the known terminal names may include the following (for a complete list, type: **ls -C /usr/lib/terminfo/?**):

155	Motorola EXORterm 155
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
450	DASI 450 (same as Diablo 1620)
450–12	DASI 450 in 12-pitch mode
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
sync	generic name for synchronous Teletype Model 4540-compatible terminals
vt100	DEC VT100

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/lib/terminfo/?/* compiled terminal description database

SEE ALSO

curses(3X), profile(4), terminfo(4), environ(5).
man(1), sh(1), stty(1), tabs(1), tput(1), tplot(1G), vi(1) in the *User's Reference Manual.*
infocmp(1M) in the *System Administrator's Reference Manual*.
Chapter 10 of the *Programmer's Guide.*

NOTES

Not all programs follow the above naming conventions.

NAME

types – primitive system data types

SYNOPSIS

#include <sys/types.h>

DESCRIPTION

The data types defined in the include file are used in the operating system code; some data of these types are accessible to user code:

```
typedef  struct { int r[1]; } *physadr;
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  ushort            ino_t;
typedef  short             cnt_t;
typedef  long              time_t;
typedef  int               label_t[10];
typedef  short             dev_t;
typedef  long              off_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;
```

The form *daddr_t* is used for disk addresses except in an i-node on disk, see *fs*(4). Times are encoded in seconds since 00:00:00 GMT, 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.

SEE ALSO

fs(4).

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 (e.g., int).

HIBITS The value of a short integer with only the high-order bit set (in most implementations, 0x8000).

HIBITL The value of a long integer with only the high-order bit set (in most implementations, 0x80000000).

HIBITI The value of a regular integer with only the high-order bit set (usually the same as HIBITS or HIBITL).

MAXSHORT The maximum value of a signed short integer (in most implementations, 0x7FFF ≡ 32767).

MAXLONG The maximum value of a signed long integer (in most implementations, 0x7FFFFFFF ≡ 2147483647).

MAXINT The maximum value of a signed regular integer (usually the same as MAXSHORT or MAXLONG).

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

void va_start(pvar)
va_list pvar;

type **va_arg(pvar,** *type***)**
va_list pvar;

void va_end(pvar)
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 nonportable, 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* ... *va_end,* are possible.

EXAMPLE

This example is a possible implementation of *execl*(2).

```
#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];
        int argno = 0;

        va_start(ap);
        file = va_arg(ap, char *);
        while ((args[argno++] = va_arg(ap, char *)) != (char *)0)
                ;
        va_end(ap);
        return execv(file, args);
}
```

SEE ALSO

exec(2), printf(3S), vprintf(3S).

NOTES

It is up to the calling routine to specify how many arguments there are, since it is not always possible to determine this 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.