Function libraries

In this chapter, we'll look at functions normally supplied in libraries with the system. As mentioned on page 151, if you have the sources, it is usually relatively trivial to port a single library function that doesn't require specific kernel functionality. There are hundreds of libraries, of course, and we can look only at those libraries that are available on a large number of systems. In the following sections, we'll look at:

- Functions that are found in the standard C library frequently enough that their absence can be a problem.
- Block memory functions, which modern compilers frequently treat as special cases.
- Regular expression libraries—five of them, all incompatible with each other, starting on page 300.
- *terminfo* and *termlib*, starting on page 307. We'll also briefly touch on *curses*.

Standard library functionality

The content of the standard C library *libc.a* reminds you of the joke "The nice thing about standards is that there are so many to choose from." Different systems have very different views on what should and should not be part of the standard library, and a number of systems don't supply the functions at all. In this section, we'll look at frequently used functions and what to do if your system doesn't have them.

alloca

`alloca` allocates memory in the stack, as opposed to `malloc`, which allocates memory in the heap:

```
void *alloca (size_t size);
```

This has a significant speed advantage over `malloc`: `malloc` needs to search and update a free space list. `alloca` typically just needs to change the value of a register, and is thus very fast. It is often included with the package you are compiling, but you may need to set a flag or modify the *Makefile* in order to include it. Versions for VAX, HP 300, i386 and Tahoe processors are located in the 4.4BSD Lite distribution as *lib/libc/<machine>/gen/alloca.s*. On the down side, it is a somewhat system-dependent function, and it's possible that it might break after a new kernel release. You can almost always replace `alloca` with `malloc`, though there may be a performance impact.

bcopy

`bcopy` is a BSD function that substantially corresponds to `memmove`:

```
#include <string.h>

void bcopy (const void *src, void *dst, size_t len);
```

Unlike `memcpy`, it is guaranteed to move correctly when the source and destination fields overlap. If your system has `memmove`, you can define it as:

```
#define bcopy(s, d, l) memmove (d, s, l)
```

The operands have a different sequence from those of `memmove`.

bzero

`bzero` is a BSD function to clear an area of memory to 0. It is a subset of the standard function `memset`, and you can define it as

```
#define bzero(d, l) memset (d, '\0', l)
```

fnmatch

`fnmatch` is a routine that matches patterns according to the shell file name rules:

```
#include <fnmatch.h>

int fnmatch (const char *pattern, const char *string, int flags);
```

`fnmatch` compares `string` against `pattern`. It returns 0 if `string` matches `pattern` and `FNM_NOMATCH` otherwise. The flags in Table 18-1 specify the kind of match to perform:

Table 18−1: `fnmatch` flags

Flag	Meaning
`FNM_NOESCAPE`	Interpret the backslash character (\) literally.
`FNM_PATHNAME`	Slash characters in string must be explicitly matched by slashes in `pattern`.
`FNM_PERIOD`	Leading periods in strings match periods in patterns. Not all versions of `fnmatch` implement this flag.

fnmatch is supplied with later BSD versions only. If you need it, it is in the 4.4BSD Lite distributio as *lib/libc/gen/fnmatch.c*.

getcwd and getwd

getcwd and getwd both return the name of the current working directory. This is the function behind the *pwd* command:

```
#include <stdio.h>

char *getcwd (char *buf, size_t size);
char *getwd (char *buf);
```

getwd has the great disadvantage that the function does not know the length of the pathname, and so it can write beyond the end of the buffer. As a result, it has been replaced by getcwd, which specifi es a maximum length for the returned string. You can defi ne getwd as:

```
#define getwd(d) getcwd (d, MAXPATHLEN)
```

MAXPATHLEN is a kernel constant defi ning the maximum path name length. It is normally defi ned in */usr/include/sys/param.h*.

gethostname and uname

There is no one function call which will return the name of the system on all UNIX platforms. On BSD systems, gethostname returns the current host name:

```
#include <unistd.h>
int gethostname (char *name, int namelen);
```

gethostname returns a null-terminated string if the space defi ned by namelen allows. This function is supported in System V.4, but not in standard versions of System V.3 and XENIX.

On System V systems, the system call uname returns a number of items of information about the system, including the name. It is also supported as a library function by most BSD systems.

```
#include <sys/utsname.h>
sys/utsname.h defines
struct utsname
  {
  char sysname [9];                    /* Internal system name */
  char nodename [9];                   /* External system name */
  char release [9];                    /* Operating system release */
  char version [9];                    /* Version of release */
  char machine [9];                    /* Processor architecture */
  };
int uname (struct utsname *name);
```

The systems that *do* support uname apply a different meaning to the fi eld sysname. For example, consider the output of the following program, which was compiled and run on Interactive UNIX/386 System V.3 Version 2.2 and BSD/386 version 1.1, both running on an Intel

486 platform:

```
#include <sys/utsname.h>
main ()
{
  struct utsname myname;
  uname (&myname);
  printf ("sysname %s nodename %s release %s version %s machine %s\n",
          myname.sysname,
          myname.nodename,
          myname.release,
          myname.version,
          myname.machine);
  }
$ uname            On the System V.3 machine:
sysname adagio nodename adagio release 3.2 version 2 machine i386
$ uname            On the BSD/386 machine:
sysname BSD/386 nodename allegro release 1.1 version 0 machine i386
```

System V puts the node name in `sysname`, whereas BSD uses it for the name of the operating system. This information is by no means complete: in particular, neither version tells you explicitly whether the system is running System V or BSD, and there is no indication of the vendor at all on the System V system.

index

`index` searches the string `s` forwards for the first occurrence of the *character* `c`. If it finds one, it returns a pointer to the character. Otherwise it returns `NULL`. It is essentially the same as the ANSI function `strchr`, and you can define it as:

```
#define index strchr
```

malloc

`malloc` has always been around, but the semantics have changed in the course of time. In the Seventh Edition and XENIX, a call to `malloc` with length 0 returned a valid pointer, whereas later versions return a `NULL` pointer, indicating an error. As a result, programs that ran on older versions might fail on more recent implementations.

memmove

`memmove` copies an area of memory:

```
#include <string.h>

void *memmove (void *dst, const void *src, size_t len);
```

This is the same function as `memcpy`, except that `memmove` is guaranteed to move overlapping data correctly. Except for the parameter sequence, it is the same as `bcopy` (see page 294). If you don't have either function, you can find `bcopy` in the 4.4BSD library source (*lib/libc/string/bcopy.c*), as well as versions in assembler:

lib/libc/vax/string/memmove.s
lib/libc/hp300/string/bcopy.s
lib/libc/tahoe/string/bcopy.s

A generic version of `memmove` in C is in the GNU C library in *sysdeps/generic/memmove.c*. See Appendix E, *Where to get sources* to locate all these sources. Note also the comments about memory move functions on page 299.

remove

```
#include <stdio.h>

int remove (const char *path);
```

On BSD systems, `remove` is a synonym for the system call `unlink`. This means that it makes sense to use it only for files. On System V.4 systems, it is slightly more complicated: if called for a file, it does the same thing as `unlink`, for directories it does the same thing as `rmdir`.

rindex

`rindex` (reverse *index*) searches the string `s` for the last occurrence of *character* `c` and returns a pointer to the character if it is found, and `NULL` otherwise. It is essentially the same function as `strrchr`, and you can define it as:

```
#define rindex strrchr
```

snprintf and vsnprintf

`snprintf` and `vsnprintf` are versions of `sprintf` and `vsprintf` that limit the length of the output string:

```
int sprintf (char *str, const char *format, ...);
int snprintf (char *str, size_t size, const char *format, ...);
int vsprintf (char *str, char *format, va_list ap);
int vsnprintf (char *str, size_t size, const char *format, va_list ap);
```

The argument `size` specifies the maximum length of the output string, including the trailing `'\0'`. These functions are supplied in 4.4BSD Lite as *usr/src/lib/libc/stdio/snprintf.c* and *usr/src/lib/libc/stdio/vsnprintf.c*. Alternatively, you can remove the second parameter and use `sprintf` or `vsprintf` instead.

strcasecmp and strncasecmp

`strcasecmp` and `strncasecmp` perform an unsigned comparison of two ASCII strings ignoring case (they consider `a` to be the same thing as `A`):

```
#include <string.h>

int strcasecmp (const char *s1, const char *s2);
int strncasecmp (const char *s1, const char *s2, size_t len);
```

`strncasecmp` differs from `strcasecmp` by comparing at most `len` characters. Both functions stop comparing when a `NUL` character is detected in one of the strings. You can find both functions in the 4.4BSD Lite distribution (*lib/libc/string/strcasecmp.c*).

strdup

`strdup` allocates memory with `malloc` and copies a string to it:

```
#include <string.h>

char *strdup (const char *str);
```

It is included in the 4.4BSD Lite distribution (*lib/libc/string/strdup.c*).

strerror and sys_errlist

`strerror` returns an error message string for a specific error:

```
#include <string.h>

extern char *sys_errlist [];
extern int sys_nerr;
char *strerror (int errnum);
```

`errnum` is the number of the error; `strerror` returns a pointer to a text for the error, or NULL if none is found.

Most library implementations also define `sys_errlist`, an array of description strings for errors, and `sys_nerr`, the total number of error messages, in other words, the number of messages in `sys_errlist`. If you don't find this function anywhere in your man pages, don't give up: it's frequently hidden in an unexpected library. For example, NonStop UX version B22 doesn't define or document `sys_errlist` anywhere, but it is in *libc.a* all the same.

The implementation of `strerror` is trivial:

```
char *strerror (int errnum)
  {
  if (errnum < sys_nerr)
    return sys_errlist [errnum];
  else
    {
    static char bogus [80];
    sprintf (bogus, "Unknown error: %d", errnum);
    return bogus;
    }
  }
```

Don't assume that your system doesn't have `sys_errlist` just because you can't find a defi nition in the header fi les. Many systems install it via the back door because packages such as X11 use them. The safest way to fi nd out is to search the system libraries. The shell script *findf*, described in Chapter 21, *Object files and friends*, page 374, will do this for you.

stricmp and strnicmp

These are somewhat uncommon alternatives to `strcasecmp` and `strncasecmp` which are supplied on some systems (see page 297).

Block memory access functions

Many programs spend the bulk of their execution time moving areas of memory about or comparing them. The C language supplies a rich set of functions, such as `memcpy`, `memmove`, `strcpy`, `strchr`, `strlen`, and friends. Unfortunately, their performance frequently leaves something to be desired. Many C libraries still write the move in C. You can write `memcpy` as:

```
char *memcpy (char *d, char *s, int len)
  {
  char *dest = d;
  while (len--)
    *d++ = *s++;
  return dest;
  }
```

On an Intel 386 architecture, *gcc* compiles quite a tight little loop with only 7 instructions,* but it also requires another 15 instructions to set up the function environment and remove it again. In addition, the calling sequence `memcpy (bar, foo, 10)` might compile in 5 instructions. Many machines supply special instructions for block memory operations, but even those that don't can do it faster without a function call. The block memory functions are thus ideal candidates for inline functions that the compiler can optimize. Many compilers now do so, including *gcc* and the System V.4 CCS compilers. In this situation, the compiler can recognize that there are only a few bytes to be moved, and that they are word-aligned, so it can use native load and store operations. When you enable optimization, *gcc* can compiles the `memcpy (bar, foo, 10)` into only 6 simple instructions: the loop has disappeared completely, and we just have 3 load and 3 store instructions. This approach isn't appropriate for moves of 1000 bytes, of course. Here, the compiler uses 4 instructions to set up a block move instruction, for a total of 5 instructions.

These examples are typical of what a smart compiler can do if it has inside information about what the function does. Normally this information is compiled into the compiler, and it doesn't need a header fi le to know about the function. This can have a number of consequences for you:

- The compiler "knows" the parameter types for the function. If you defi ne the function differently, you get a possibly confusing error message:

* See Chapter 21, *Object files and friends*, page 377 for a discussion of parameter passing.

```
    memcpy.c:3: warning: conflicting types for built-in function ‘memcpy’
```

If you get this message, you can either ignore it or, better, remove the definition. The compiler knows anyway.

- When debugging, you can't just put a breakpoint on `memcpy`. There is no such function, or if it has been included to satisfy references from modules compiled by other compilers, it may not be called when you expect it to.
- If you have a program written for a compiler that knows about the block memory functions, you may need to add definitions if your compiler doesn't support them.

Regular expression routines

Regular expressions are coded descriptions of text strings. They are used by editors and utilities such as `grep` for searching and matching actual text strings. There's nothing very special about routines that process regular expressions, but there is no agreement about standards and there is no agreement about how to write a regular expression. The only regular thing about them is the regularity with which programs fail to link due to missing functions. There are at least five different packages, and the names are similar enough to cause significant confusion.

In all cases, the intention of the routines is the same:

- A *compilation* function converts a string representation of a regular expression into an internal representation that is faster to interpret.
- A *search* function performs the search.

In addition, the Eighth Edition *regex* package has support for a replacement function based on the results of a previous search.

Regular expression syntax also comes in two flavours:

- The documentation of the older syntax usually states that it is the same syntax that *ed* uses. *ed* is an editor that is now almost completely obsolete,* so it's good to know that the stream editor *sed*, which is still in current use, uses the same syntax.
- The newer syntax is the same that *egrep* uses. It is similar to that of *ed*, but includes a number of more advanced expressions.

If you get software that expects one package, but you have to substitute another, you should expect changes in behaviour. Regular expressions that worked with the old package may not work with the new one, or they may match differently. A particularly obvious example is the use of parentheses to group expressions. All forms of regular expressions perform this grouping, but the old syntax requires the parentheses to be escaped: `\(expr\)`, whereas the new syntax does not: `(expr)`.

* *ed* does have its uses, though. If you have serious system problems (like */usr* crashed), it's nice to have a copy of *ed* on the root file system. It's also useful when your only connection to the machine is via a slow modem line: over a 2400 bps line, redrawing a 24x80 screen with *vi* or *emacs* takes 8 seconds, and things are a lot faster with *ed*.

Apart from the intention of the functions, they also perform their task in *very* different ways. They might store compiled program in an area that the caller supplies, they might *malloc* it, or they might hide it where the user can't find it. In some cases, the compiled expression is stored in a *struct* along with other information intended for use by the calling functions. In others this information is not returned at all, and in others again it is returned in global arrays, and in others it is returned via optional arguments. Translating from one flavour to another takes a lot of time. Three packages are generally available: Henry Spencer's Eighth Edition package, the 4.4BSD POSIX.2 version, and the GNU POSIX.2 version. See Appendix E, *Where to get sources* for sources of these packages. If you *do* have to port to a different regular expression package, choose a POSIX.2 implementation. Although it is by far the most complicated to understand, it is probably the only one that will be around in a few years.

Regular expressions are used for two purposes: searching and replacing. When replacing one regular expression with another, it's nice to be able to refer to parts of the expression. By convention, you define these subexpressions with parentheses: the expression `foo\(.*\)bar\(.*\)baz` defines two such subexpressions. The regular expression will match all strings containing the texts `foo`, `bar`, and `baz` with anything in between them. The first marked subexpression is the text between `foo` and `bar`, and the second one is the text between `bar` and `baz`.

regexpr

The *regexpr* routines have traditionally been supplied with System V and XENIX systems. They were originally part of the *ed* editor and thus implement the *ed* style of regular expressions. Despite the name, there is no function called `regexpr`.

The routines themselves are normally in a library *libgen.a*. In addition, for some reason many versions of System V and XENIX include the complete *source* to the functions in the header file *regexp.h*, whereas the header file *regexpr.h* contains only the declarations. There are three routines:

```
#include <regexpr.h>

extern char *loc1, *loc2, *locs;
extern int nbra, regerrno, reglength;
extern char *braslist [], *braelist [];

char *compile (const char *instring,
               char *expbuf,
               char *endbuf);
int step (const char *string, char *expbuf);
int advance (const char *string, char *expbuf);
```

- `compile` compiles the regular expression `instring`. The exact behaviour depends on the value of `expbuf`. If `expbuf` is `NULL`, `compile` *malloc*s space for the compiled version of the expression and returns a pointer to it. If `expbuf` is non-`NULL`, `compile` places the compiled form there if it fits between `expbuf` and `endbuf`, and it returns a pointer to the first *free* byte. If the regular expression does not fit in this space, `compile` aborts. If the compilation succeeds, the length of the compiled expression is stored in the global variable `reglength`.

If the compilation fails, `compile` returns `NULL` and sets the variable `regerrno` to one of the values in Table 18-2:

Table 18−2: `regcomp` error codes

Error code	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 overfbw.

- `step` compares the string `string` with the compiled expression at `expbuf`. It returns non-zero if it fi nds a match, and 0 if it does not. If there is a match, the pointer `loc1` is set to point to the fi rst matching character, and `loc2` is set to point past the last character of the match.

  If the regular expression contains *subexpressions*, expressions bracketed by the character sequences `\(` and `\)`, `step` stores the locations of the start and end of each matching string in the global arrays `braslist` (start) and `braelist` (end). It stores the total number of such subexpressions in `nbra`.

- `advance` has the same function as `step`, but it restricts its matching to the beginning of the string. In other words, a match always causes `loc1` to point to the beginning of the string.

regcmp

regcmp is another regular expression processor used in System V and XENIX. Like *regexpr*, they implement the *ed* style of regular expressions with some extensions. They are also normally part of the library *libgen.a*.

```
#include <libgen.h>

char *regcmp (const char *string1,
              /* char *string2 */ ...
              (char *) 0);
```

```
char *regex (const char *re,
             const char *subject,
             /* char *ret0, *ret1, ... *retn */ ... );
extern char *__loc1;
```

- `regcmp` can take multiple input arguments, which it concatenates before compilation. This can be useful when the expression is supplied on a number of input lines, for example. It always *malloc*s space for its compiled expression, and returns a pointer to it.
- `regex` searches for the string `subject` in the compiled regular expression `re`. On success, it returns a pointer to the next unmatched character and sets the global pointer `__loc1` to the address of the first character of the match. Optionally, it returns up to ten strings at `ret0` and the parameters that follow. You specify them with the `$n` regular expression element discussed below.

The regular expression syntax is slightly different from that of *ed* and *sed*:

- The character `$` represents the end of the string, not the end of the line. Use `\n` to specify the end of the line.
- You can use the syntax `[a-f]` to represent `[abcdef]`.
- You can use the syntax `x+` to represent one or more occurrences of `x`.
- You can use the syntax `{m}`, where `m` is an integer, to represent that the previous subexpression should be applied `m` times.
- You can use the syntax `{m,}`, where `m` is an integer, to represent that the previous subexpression should be applied at least `m` times.
- You can use the syntax `{m,u}`, where `m` and `u` are integers, to represent that the previous subexpression should be applied at least `m` and at most `u` times.
- The syntax `(exp)` groups the characters `exp` so that operators such as `*` and `+` work on the whole expression and not just the preceding character. For example, `abcabcabcabc` matches the regular expression `(abc)+`, and `abcccccc` matches `abc+`.
- The syntax `(exp)$n`, where `n` is an integer, matches the expression `exp`, and returns the address of the matched string to the call parameter `ret`*n* of the call to `regex`. It will even try to return it if you didn't supply parameter `ret`*n*, so it's good practice to supply all the parameters unless you are in control of the regular expressions.

regex: re_comp and re_exec

regex is the somewhat confusing name given to the functions `re_comp` and `re_exec`, which were introduced in 4.0BSD. Note particularly that there is *no* function called `regex`, and that the name is spelt without a final *p*. *regex* also implements *ed*-style regular expressions. There are two functions:

```
char *re_comp (char *sp);
int re_exec (char *p1);
```

- re_comp compiles the regular expression sp and stores the compiled form internally. On successful compilation, it returns a NULL pointer, and on error a pointer to an error message.
- re_exec searches the string p1 against the internally stored regular expression. It returns 1 if the string p1 matches the last compiled regular expression, 0 if the string p1 fails to match the last compiled regular expression, and -1 if the compiled regular expression is invalid.

No public-domain versions of regex are available, but it's relatively simple to defi ne them in terms of POSIX.2 regex.

Eighth edition regexp

The *Eighth edition regexp* package has gained wider popularity due to a widely available implementation by Henry Spencer of the University of Toronto. It consists of the functions regcomp, regexec, regsub and regerror:

```
#include <regexp.h>

regexp * regcomp (const char *exp);
int regexec (const regexp *prog, const char *string);
void regsub (const regexp *prog, const char *source, char *dest);
void regerror (const char *msg);
```

In contrast to earlier packages, Eighth edition regexp implements *egrep*-style regular expressions. Also in contrast to other packages, the compiled form of the regular expression includes two arrays which regexec sets for the convenience of the programmer: char *startp [] is an array of start addresses of up to nine subexpressions (expressions enclosed in parentheses), and char *endp [] is an array of the corresponding end addresses. The subexpressions are indexed 1 to 9; startp [0] refers to the complete expression.

regcomp compiles the regular expression exp and stores the compiled version in an area that it *malloc*s. It returns a pointer to the compiled version on success or NULL on failure.

regexec matches the string string against the compiled regular expression prog. It returns 1 if a match was found and 0 otherwise. In addition, it stores the start and end addresses of the fi rst ten parenthesized subexpressions in prog->startp and prog->endp.

regsub performs a regular expression substitution, a function not offered by the other packages. You use it after regcomp fi nds a match and stores subexpression start and end information in startp and endp. It copies the input string source to dest, replacing expressions of the type &n, where n is a single digit, by the substring defi ned by startp [n] and endp [n].

regerror determines the action to be taken if an error is detected in *regcomp*, *regexec* or *regsub*. The default regerror prints the message and terminates the program. If you want, you can replace it with your own routine.

POSIX.2 regex

As if there weren't enough regular expression libraries already, POSIX.2 also has a version of *regex*. It is intended to put an end to the myriad other flavours of regular expressions, and thus supplies all the functionality of the other packages. Unfortunately, it re-uses the function names of Eighth Edition *regexp*. This is the only similarity: the POSIX.2 functions take completely different parameters. The header file of the 4.4BSD package starts with

```
#ifndef _REGEX_H_
#define _REGEX_H_                    /* never again */

#ifdef _REGEXP_H_
BAD NEWS -- POSIX regex.h and V8 regexp.h are incompatible
#endif
```

The Eighth Edition *regexp.h* contains similar code, so if you accidentally try to use both, you get an error message when you try to compile it.

The POSIX.2 *regex* package offers a seemingly infinite variety of flags and options. It consists of the functions `regcomp`, `regexec`, `regerror` and `regfree`. They match regular expressions according to the POSIX.2 regular expression format, either *ed* format (so-called *basic* regular expressions, the default) or *egrep* format (*extended* regular expressions). The 4.4BSD implementations refer to them as *obsolete regular expressions* and *modern regular expressions* respectively. You choose the kind of expression via flag bits passed to the compilation function `regcomp`. Alternatively, you can specify that the string is to be matched exactly—no characters have any special significance any more.

Here are the functions:

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

int regcomp (regex_t *preg, const char *pattern, int cflags);
int regexec (const regex_t *preg,
             const char *string,
             size_t nmatch,
             regmatch_t pmatch [],
             int eflags);
size_t regerror (int errcode,
                 const regex_t *preg,
                 char *errbuf,
                 size_t errbuf_size);
void regfree (regex_t *preg);
```

- `regcomp` compiles the regular expression `pattern` and stores the result at `preg`. It returns 0 on success and an error code on failure. `cflags` modifies the way in which the compilation is performed. There are a number of flags, listed in Table 18-3:

Table 18−3: `cflags` bits for `regcomp`

Flag bit	Function
`REG_BASIC`	Compile basic ("obsolete") REs. This is the default.
`REG_EXTENDED`	Compile extended ("modern") REs.
`REG_NOSPEC`	Compile a literal expression (no special characters). This is not specified by POSIX.2. You may not combine `REG_EXTENDED` and `REG_NOSPEC`.
`REG_ICASE`	Compile an expression that ignores case.
`REG_NOSUB`	Compile to report only whether the text matched, don't return subexpression information. This makes the search faster.
`REG_NEWLINE`	Compile for newline-sensitive matching. By default, a newline character does not have any special meaning. With this flag, `^` and `$` match beginnings and ends of lines, and expressions bracketed with `[]` do not match new lines.
`REG_PEND`	Specify that the expression ends at `re_endp`, thus allowing `NUL` characters in expressions. This is not defined in POSIX.2

- `regexec` matches the string `string` against the compiled regular expression `preg`. If `nmatch` is non-zero and `pmatch` is non-NULL, start and end information for up to `nmatch` subexpressions is returned to the array `pmatch`. `regexec` also supports a number of flags in `eflags`. They are described in Table 18-4:

Table 18−4: `eflags` bits for `regexec`

Flag bit	Function
`REG_NOTBOL`	Do not match the beginning of the expression with `^`.
`REG_NOTEOL`	Do not match the end of the expression wtih `$`.
`REG_STARTEND`	Specify that the string starts at `string + pmatch[0].rm_so` and ends at `string + pmatch[0].rm_eo`. This can be used with `cflags` value `REG_PEND` to match expressions containing `NUL` characters.

- `regerror` is analogous to the C library function `perror`: it converts the error code `errcode` for regular expression `preg` into a human-readable error message at `errbuf`, up to a maximum of `errbuf_size` bytes.

As in Eighth Edition *regexp*, `regcomp` returns additional information about the expression:

- If you compile the expression with the `REG_PEND` bit set in `cflags`, you can set `re_endp` to point to the real end of the regular expression string supplied to `regcomp`, thus allowing `NUL` characters in the regular expression.
- After compilation, `regcomp` sets `re_nsub` to the number of subexpressions it found. For each subexpression, `regexec` can return start and end address information if you call it with appropriate parameters.

In addition, `regexec` stores information about matched subexpressions in a structure of type `regmatch_t`, unless you specify the flag `REG_NOSUB`. This contains at least the fields `rm_so` and `rm_eo`, which are *offsets* from the start of the string of the first and last character of the match. They are of type `regmatch_t`.

No less than three versions of POSIX.2 *regex* are generally available: Henry Spencer's *regex* is included in the 4.4BSD distribution, and the GNU project has the older *regex* and newer *rx*. See Appendix E, *Where to get sources*.

termcap and terminfo

When full-screen editors started to appear in the late 70s, Bill Joy at UCB wrote *ex*, out of which grew *vi*. One of the problems he had was that just about every terminal has a different command set. Even commands as simple as positioning the cursor were different from one terminal to the next. In order to get the editor to work over a range of terminals, he devised *termcap*, the terminal capabilities database, which, with a few access routines, allowed a program to determine the correct control sequences for most common functions.

A few years later, while writing the game *rogue*, Ken Arnold extended the routines and created a new package, *curses*, which offered higher-level functions.

Finally, USG improved on curses and introduced *terminfo*, a new implementation of the same kind of functionality as termcap. It was designed to address some of the weaknesses of termcap, and as a result is not compatible with it. More programs use termcap than terminfo: terminfo is usually restricted to System V systems, and termcap has been around for longer, and is available on most systems. Some programs use both, but there aren't very many of them.

There are a number of gotchas waiting for you with termcap, termlib and curses:

- Termcap isn't perfect, to put it mildly: it relies on a marginally human-readable definition file format, which means that the access routines are slow. When AT&T incorporated *termcap* into System V, they addressed this problem and created *terminfo*, with a so-called *compiled* representation of the terminal data.
- Both termcap and terminfo are passive services: your program has to explicitly ask for them in order for them to work. Many programs don't use either, and many use only one or the other.
- Though the BSD *curses* package offered a bewildering number of functions, users asked for more. When AT&T incorporated curses into System V, they added significant enhancements. Curses goes a level of complexity beyond termcap and terminfo, and supplies a huge number of functions to perform all kinds of functions on a terminal. It's

like a can of worms: avoid opening it if you can, otherwise refer to *UNIX Curses Explained* by Berny Goodheart. If you have a BSD system and need System V curses, you can try the *ncurses* package (see Appendix E, *Where to get sources*).

- BSD versions of UNIX have still not incorporated *terminfo* or the additional curses routines, although they have been available for some time. In this section, we'll look at the differences in the implementations and what can be done about them. For more information, read *Programming with curses*, by John Strang, for a description of BSD curses, *Termcap and Terminfo*, by John Strang, Tim O'Reilly and Linda Mui for a description of termcap and terminfo, and *UNIX Curses Explained*, by Berny Goodheart, for a description of both versions of curses.

termcap

The heart of termcap is the terminal description. This may be specified with an environment variable, or it may be stored in a file containing definitions for a number of terminals. There is no complete agreement on the location of this file:

- If the termcap routines find the environment variable `TERMCAP`, and it doesn't start with a slash (`/`), they try to interpret it as a termcap entry.
- If the termcap routines find the environment variable `TERMCAP`, and it *does* start with a slash, they try to interpret it as the name of the termcap file.
- If `TERMCAP` isn't specified, the location depends on a constant which is compiled into the termcap library. Typical directories are */etc*, */usr/lib*, and */usr/share*. Don't rely on finding only one of these files: it's not uncommon to find multiple copies on a machine, only one of which is of any use to you. If the system documentation forgets to tell you where the termcap file is located, you can usually find out with the aid of *strings* and *grep*. For example, BSD/OS gives

  ```
  $ strings libtermcap.a |grep /termcap
  .termcap /usr/share/misc/termcap
  ```

  and SCO UNIX gives

  ```
  $ strings libtermcap.a |grep /termcap
  /etc/termcap
  /etc/termcap
  ```

The file *termcap* contains entries for all terminals known to the system. On some systems it can be up to 200 kB in length. This may not seem very long, but every program that uses termcap must read it until it finds the terminal definition: if the terminal isn't defined, it reads the full length of the file.

Here's a typical entry:

```
vs|xterm|vs100|xterm terminal emulator (X Window System):\
        :AL=\E[%dL:DC=\E[%dP:DL=\E[%dM:DO=\E[%dB:IC=\E[%d@:UP=\E[%dA:\
        :al=\E[L:am:\
        :bs:cd=\E[J:ce=\E[K:cl=\E[H\E[2J:cm=\E[%i%d;%dH:co#80:\
```

```
        :cs=\E[%i%d;%dr:ct=\E[3k:\
        :dc=\E[P:dl=\E[M:\
        :im=\E[4h:ei=\E[4l:mi:\
        :ho=\E[H:\
        :is=\E[r\E[m\E[2J\E[H\E[?7h\E[?1;3;4;6l\E[4l:\
        :rs=\E[r\E[m\E[2J\E[H\E[?7h\E[?1;3;4;6l\E[4l\E<:\
        :k1=\EOP:k2=\EOQ:k3=\EOR:k4=\EOS:kb=^H:kd=\EOB:ke=\E[?1l\E>:\
        :kl=\EOD:km:kn#4:kr=\EOC:ks=\E[?1h\E=:ku=\EOA:\
        :li#65:md=\E[1m:me=\E[m:mr=\E[7m:ms:nd=\E[C:pt:\
        :sc=\E7:rc=\E8:sf=\n:so=\E[7m:se=\E[m:sr=\EM:\
        :te=\E[2J\E[?47l\E8:ti=\E7\E[?47h:\
        :up=\E[A:us=\E[4m:ue=\E[m:xn:
v2|xterms|vs100s|xterm terminal emulator, small window (X Window System):\
        :co#80:li#24:tc=xterm:
vb|xterm-bold|xterm with bold instead of underline:\
        :us=\E[1m:tc=xterm:
#
# vi may work better with this termcap, because vi
# doesn't use insert mode much
vi|xterm-ic|xterm-vi|xterm with insert character instead of insert mode:\
        :im=:ei=:mi@:ic=\E[@:tc=xterm:
```

The lines starting with the hash mark (#) are comments. The other lines are terminal capability defi nitions: each entry is logically on a single line, and the lines are continued with the standard convention of terminating them with a backslash (\). As in many other old UNIX fi les, fi elds are separated by colons (`:`).

The fi rst fi eld of each description is the *label*, the name of the terminal to which the defi nition applies. The terminal may have multiple names, separated by vertical bars (`|`). In our example, the fi rst entry has the names `vs`, `xterm` and `vs100`. The last part of the name fi eld is a description of the kind of terminal. In the second entry, the names are `vi`, `xterm-ic` and `xterm-vi`.

Both 4.4BSD termcap and System V terminfo recommend the following conventions for naming terminals:

- Start with the name of the physical hardware, for example, *hp2621*.
- Avoid hyphens in the name.
- Describe operational modes or confi guration preferences for the terminal with an indicator, separated by a hyphen. Use the following suffi xes where possible:

Suffi x	Meaning	Example
`-w`	Wide mode (more than 80 columns)	`vt100-w`
`-am`	With automatic margins (usually default)	`vt100-am`
`-nam`	Without automatic margins	`vt100-nam`
`-n`	Number of lines on screen	`aaa-60`
`-na`	No arrow keys (leave them in local)	`concept100-na`
`-np`	Number of pages of memory	`concept100-4p`
`-rv`	Reverse video	`concept100-rv`

The following fields describe individual capabilities in the form *capability=definition*. The capabilities are abbreviated to two characters, and case is significant. See *Programming with curses*, by John Strang, for a list of the currently defined abbreviations and their meaning. Depending on the capability, the definition may be a truth value (true or false), a number, or a string. For example,

- The first entry for `vs` (`AL=\E[%dL`) states that the capability `AL` (insert *n* new blank lines) can be invoked with the string `\E[%dL`. `\E` represents an `ESC` character. The characters `[` and `L` are used literally. The program uses `sprintf` to replace the `%d` with the number of lines to insert.
- The next entry, `am`, has no parameter. This is a boolean or truth value, in this case meaning that the terminal supports automatic margins. The presence of the capability means that it is true, the absence means that it is false.
- The entry `co#80` specifies a numeric parameter and states that the capability `co` (number of columns) is 80.

There is almost nothing in the syntax of the definition file that requires that a particular capability have a particular type, or even which capabilities exist: this is simply a matter of agreement between the program and the capabilities database: if your program wants the capability `co`, and wants it to be numeric, it calls `tgetnum`. For example, the following code checks first the information supplied by `ioctl TIOCGWINSZ` (see Chapter 15, *Terminal drivers*, page 259), then the termcap entry, and if both of them are not defined, it defaults to a configurable constant:

```
if (! (maxcols = winsize.ws_col)
    && (! (maxcols = tgetnum ("co"))) )
     maxcols = MAXCOLS;
```

The only exception to this rule is the capability `tc`, which refers to another capability. In the example above, the entry for `vi` and friends consists of only 5 entries, but the last one is a `tc` entry that refers to the `vs` entry above.

This lack of hard and fast rules means that termcap is extensible: if you have a terminal that can change the number of colours which can be displayed at one time, and for some reason you want to use this feature, you might define a termcap variable `XC` specifying the string to output to the terminal to perform this function. The danger here is, of course, that somebody else might write a program that uses the variable `XC` for some other purpose.

termcap functions: termlib

Along with termcap come library functions, normally in a library called *libtermcap.a*, though a number of systems include them in *libc.a*. They help you query the database and perform user-visible functions such as writing to the terminal. There aren't many functions, but they are nonetheless confusing enough:

```
char PC;                                /* padding character */
char *BC;                               /* backspace string */
char *UP;                               /* Cursor up string */
```

```
short ospeed;                          /* terminal output speed */

tgetent (char *bp, char *name);
tgetnum (char *id);
tgetflag (char *id);
char *tgetstr (char *id, char **sbp);
char *tgoto (char *cm, int destcol, int destline);
tputs (register char *cp, int affcnt, int (*outc) ());
```

Before you start, you need two areas of memory: a 1 kB temporary buffer for the termcap entry, which we call `buf`, and a buffer for the string capabilities that you need, which we call `sbuf`. The initialization function `tgetent` fills `sbuf` with the termcap entry, and the function `tgetstr` transfers strings from `buf` to `sbuf`. After initialization is complete, `buf` can be deallocated.

In addition, you need a `char` pointer `sbp` which must be initialized to point to `sbuf`. `tgetstr` uses it to note the next free location in `sbuf`.

If you don't specify a specific termcap string as the value of the `TERMCAP` environment variable, you need to know the name of your terminal in order to access it in the *termcap* file. By convention, this is the value of the `TERM` environment variable, and you can retrieve it with the library function `getenv`.

- `tgetent` searches the *termcap* file for a definition for the terminal called `name` and places the entry into the buffer, which must be 1024 bytes long, at `buf`. All subsequent calls use this buffer. The confusing thing is that they do not explicitly reference the buffer: `tgetent` saves its address in a static pointer internal to the termcap routines. `tgetent` returns 1 on success, 0 if the terminal name was not found, or -1 if the termcap database could not be found.
- `tgetnum` looks for a numeric capability `id` and returns its value if found. If the value is not found, it returns -1.
- `tgetflag` looks for a boolean capability `id` and returns 1 if it is present and 0 otherwise.
- `tgetstr` looks for a string capability `id`. If it is found, it copies it into the buffer pointed to by the pointer at `*sbp`. It then updates `sbp` to point past the string. It returns the address of the string in `sbuf` if found, or `NULL` if not. This method ensures that the strings are null-terminated (which is not the case in `buf`), and that they are stored efficiently (though a 1 kB buffer is no longer the overhead it used to be).
- `tgoto` generates a positioning string to move the cursor to column `destcol` and line `destline` using the `cm` (cursor position) capability. This capability contains format specifications that `tgoto` replaces with the representation of the actual row and column numbers. It returns a pointer to the positioning string, which again is stored in a static buffer in the package. It also attempts to avoid the use of the characters `\n`, `CTRL-D` or `CTRL-@`. The resulting string may contain binary information that corresponds to tab characters. Depending on how it is set up, the terminal driver may replace these tab characters with blanks, which is obviously not a good idea. To ensure that this does not happen, turn off `TAB3` on a *termio* or *termios* system (see Chapter 15, *Terminal drivers*, page 243) or reset `XTABS` in `sgttyp.sg_flags` with the old terminal driver (see page 240).

This is all that tgoto does. It does not actually output anything to the terminal.

- tputs writes the string at cp to the screen. This seems unnecessary, since write and fwrite already do the same thing. The problem is that the output string cp may contain padding information for serial terminals, and only tputs interprets this information correctly. affcnt is the number of lines on the screen affected by this output, and outc is the address of a function that outputs the characters correctly. Often enough, this is something like putchar.

Problems with termcap

Sooner or later you'll run into problems with termcap. Here are some of the favourite ones:

Missing description

It could still happen that there isn't a description for your terminal in the termcap data base. This isn't the problem it used to be, since most modern terminals come close to the ANSI standard. In case of doubt, try ansi or vt100. This is, of course, not a good substitute for complete documentation for your terminal.

Incomplete description

It's much more likely that you *will* find a terminal description for your terminal, but it's incomplete. This happens surprisingly often. For example, the xterm definition supplied in X11R5 has 56 capabilities, and the definition supplied with X11R6 has 85. xterm hasn't changed significantly between X11R5 and X11R6: the capabilities were just missing from the entry in X11R5. Frequently you'll find that a feature you're looking for, in particular the code generated by a particular function key, is missing from your termcap entry. If nothing else helps, you can find out what codes the key generates with od:

```
$ od -c                          display stdin in character format
^[[11~^[[12~^[[13~^[[14~ RETURN
0000000  033   [   1   1   ~ 033   [   1   2   ~ 033   [   1   3   ~ 0000020
033   [   1   4   ~  \n
0000025
```

In this example, I pressed the keys *F1*, *F2*, *F3* and *F4* on an *xterm*: this is what echos on the first line. *od* doesn't display anything until its read completes, so I pressed RETURN to show the text. It shows that the sequences generated are:

- 033 (ESC, which is represented as \E in termcap entries).
- [1 and the number of the function key and a tilde (~).

These sequences can then be translated into the termcap capabilities:

```
k1=\E[11~:k2=\E[12~:k3=\E[13~:k4=\E[14~:
```

Incorrect description

If we look at the previous example more carefully, we'll notice something strange: these capabilities aren't the same as the ones in the example for *xterm* on page 308. What's wrong with this picture? A good question. Both under X11R5 and X11R6, *xterm* on an Intel architecture gives you the codes shown above. The codes for F5 to F10 are as shown in the termcap entry, but the entries for F1 to F4 are just plain wrong. I don't know of any way to generate them with *xterm*. This is typical of termcap entries: if you run into trouble, first make sure that your descriptions are correct.

Obsolete information

Another interesting thing about the *xterm* example is that it tells you the size of the terminal: `co#80` says that this terminal has 80 columns, and `li#65` says that it has 65 lines. This information can be an approximation at best, since X11 allows you to resize the window. Most modern systems supply the `SIGWINCH` signal, which is delivered to the controlling process when a window is resized (see Chapter 15, *Terminal drivers*, page 259). This information is just plain misleading, but there's a lot of it in just about any *termcap* file. The 4.4BSD man page flags a number of capabilities that are considered obsolete, including things as the character used to backspace or the number of function keys.

terminfo

terminfo is the System V replacement for *termcap*. At first sight it looks very similar, but there are significant differences:

- Instead of reading the *termcap* file, the *terminfo* routines require a "compiled" version.
- Instead of storing all entries in a single file, *terminfo* stores each entry in a different file, and puts them in a directory whose name is the initial letter of the terminal name. For example, the *terminfo* description for *xterm* might be stored in */usr/lib/terminfo/x/xterm*.
- The substitution syntax has been significantly expanded: in termcap, only `tgoto` could handle parameter substitution (see page 311); in terminfo, the substitution syntax is more general and offers many more features.
- The program *tic* (terminfo compiler) compiles terminfo descriptions.
- The programs *infocmp*, which is supplied with System V, and *untic*, which is part of *ncurses*, dump compiled terminfo entries in source form.

As an example of a terminfo definition, let's look at the definition for an xterm. This should contain the same information as the termcap entry on page 308:

```
xterm|xterm-24|xterms|vs100|xterm terminal emulator (X Window System),
        is2=\E7\E[r\E[m\E[?7h\E[?1;3;4;6l\E[4l\E8\E>,
        rs2=\E7\E[r\E[m\E[?7h\E[?1;3;4;6l\E[4l\E8\E>,
        am, bel=^G,
        cols#80, lines#24,
        clear=\E[H\E[2J, cup=\E[%i%p1%d;%p2%dH,
        csr=\E[%i%p1%d;%p2%dr,
        cud=\E[%p1%dB, cud1=\n, cuu=\E[%p1%dA, cuu1=\E[A,
        cub=\E[%p1%dD, cub1=\b,       cuf=\E[%p1%dC, cuf1=\E[C,
        el=\E[K, ed=\E[J,
        home=\E[H, ht=^I, ind=^J, cr=^M,
        km,
        smir=\E[4h, rmir=\E[4l, mir,
        smso=\E[7m, rmso=\E[m, smul=\E[4m, rmul=\E[m,
        bold=\E[1m, rev=\E[7m, blink@, sgr0=\E[m, msgr,
        enacs=\E)0, smacs=^N, rmacs=^O,
        smkx=\E[?1h\E=, rmkx=\E[?1l\E>,
        kf1=\EOP, kf2=\EOQ, kf3=\EOR, kf4=\EOS,
        kf5=\E[15~, kf6=\E[17~, kf7=\E[18~, kf8=\E[19~, kf9=\E[20~,
        kf10=\E[21~,
        kf11=\E[23~, kf12=\E[24~, kf13=\E[25~, kf14=\E[26~, kf15=\E[28~,
        kf16=\E[29~, kf17=\E[31~, kf18=\E[32~, kf19=\E[33~, kf20=\E[34~,
        kfnd=\E[1~, kich1=\E[2~, kdch1=\E[3~,
        kslt=\E[4~, kpp=\E[5~, knp=\E[6~,
        kbs=\b,        kcuu1=\EOA, kcud1=\EOB, kcuf1=\EOC, kcub1=\EOD,
        meml=\El, memu=\Em,
        smcup=\E7\E[?47h, rmcup=\E[2J\E[?47l\E8,
        sc=\E7, rc=\E8,
        il=\E[%p1%dL, dl=\E[%p1%dM, il1=\E[L, dl1=\E[M,
        ri=\EM,
        dch=\E[%p1%dP, dch1=\E[P,
        tbc=\E[3g,
        xenl,
xterm-65|xterm with tall window 65x80 (X Window System),
        lines#65,
        use=xterm,
xterm-bold|xterm with bold instead of underline (X Window System),
        smul=\E[1m, use=xterm,
#
# vi may work better with this entry, because vi
# doesn't use insert mode much
xterm-ic|xterm-vi|xterm with insert character instead of insert mode,
        smir@, rmir@, mir@, ich1=\E[@, ich=\E[%p1%d@, use=xterm,
```

The entries look very similar, but there are a few minor differences:

- The names for the capabilities may be up to 5 characters long. As a result, the names are different from the *termcap* names.
- Capabilities are separated by commas instead of colons.
- Definitions may be spread over several lines: there is no need to terminate each line of a definition with a `\`.
- The last character in each entry must be a comma (`,`). If you remove it, you will thoroughly confuse *tic*.

terminfo functions

terminfo has a number of functions that correspond closely to *termlib*. They are also the low-level routines for curses:

```
#include <curses.h>
#include <term.h>
TERMINAL *cur_term;

int setupterm (char *term, int fd, int *error);
int setterm (char *term);
int set_curterm (TERMINAL *nterm);
int del_curterm (TERMINAL *oterm);
int restartterm (char *term, int fildes, int *errret);
char *tparm (char *str, long int p1 ... long int p9);
int tputs (char *str, int affcnt, int  (*putc) (char));
int putp (char *str);
int vidputs (chtype attrs, int  (*putc) (char));
int vidattr (chtype attrs);
int mvcur (int oldrow, int oldcol, int newrow, int newcol);
int tigetflag (char *capname);
int tigetnum (char *capname);
int tigetstr (char *capname);
```

Terminfo can use an environment variable `TERMINFO`, which has a similar function to `TERMCAP`: it contains the name of a *directory* to search for terminfo entries. Since terminfo is compiled, there is no provision for stating the capabilities directly in `TERMINFO`.

- `setupterm` corresponds to the termcap function `tgetent`: it reads in the terminfo data and sets up all necessary capabilities. The parameter `term` is the name of the terminal, but may be `NULL`, in which case `setupterm` calls `getenv` to get the name of the terminal from the environment variable `TERM`. `fd` is the file descriptor of the output terminal (normally 1 for `stdout`), and `error` is an error status return address.
- `setterm` is a simplified version of `setupterm`: it is equivalent to `setupterm (term, 1, NULL)`.
- `setupterm` allocates space for the terminal information and stores the address in the global pointer `cur_term`. You can use `set_curterm` to set it to point to a different terminal entry in order to handle multiple terminals.
- `del_curterm` deallocates the space allocated by `setupterm` for the terminal `oterm`.
- `restartterm` performs a subset of `setupterm`: it assumes that the `cur_term` is valid, but that terminal type and transmission speed may change.
- `tparm` substitutes the real values of up to 9 parameters (`p1` to `p9`) into the string `str`. This can be used, for example, to create a cursor positioning string like `tgoto`, but it is much more flexible.
- `tputs` is effectively the same function as termcap `puts` described on page 312.

- `putp` is effectively `tputs (str, stdout, putchar)`.
- `vidputs` sets the terminal attributes for a video terminal.
- `vidattr` is effectively `vidputs (attr, putchar)`.
- `mvcur` provides optimized cursor motion depending on the terminal capabilities and the relationship between the current cursor position *(oldrow, oldcol)* and the new position *(newrow, newcol)*.
- `tigetflag`, `tigetnum` and `tigetstr` correspond to the termcap functions `tgetnum`, `tgetflag` and `tgetstr` described on page 311.

printcap

Termcap was developed in the days where “terminal” did not always mean a display terminal, and the capabilities include a number of functions relating to hardcopy terminals. Nevertheless, they are not sufficient to describe the properties of modern printers, and BSD systems have developed a parallel system called *printcap*, which is a kind of termcap for printers. It is used primarily by the spooling system.

Printcap differs from termcap in a number of ways:

- The printcap capabilities are stored in a file called */etc/printcap*, even in systems where *termcap* is stored in some other directory.
- The syntax of the *printcap* file is identical to the syntax of *termcap*, but the capabilities differ in name and meaning from termcap capabilities.
- There is no environment variable corresponding to `TERMCAP`.
- There are no user-accessible routines. The functions used by the spool system actually have the same names as the corresponding termcap functions, but they are private to the spooling system. In a BSD source tree, they are located in the file *usr.sbin/lpr/common_source/printcap.c*.

A better way to look at printcap is to consider it part of the spooling system, but occasionally you’ll need to modify */etc/printcap* for a new printer.