

Each of our example programs that has a nontrivial command-line syntax provides a simple help facility for the user: if invoked with the `--help` option, the program displays a usage message that indicates the syntax for command-line options and arguments.

3.5.2 Common Functions and Header Files

Most of the example programs include a header file containing commonly required definitions, and they also use a set of common functions. We discuss the header file and functions in this section.

Common header file

Listing 3-1 is the header file used by nearly every program in this book. This header file includes various other header files used by many of the example programs, defines a *Boolean* data type, and defines macros for calculating the minimum and maximum of two numeric values. Using this header file allows us to make the example programs a bit shorter.

Listing 3-1: Header file used by most example programs

```
lib/tlpi_hdr.h

#ifndef TLPI_HDR_H
#define TLPI_HDR_H    /* Prevent accidental double inclusion */

#include <sys/types.h> /* Type definitions used by many programs */
#include <stdio.h>     /* Standard I/O functions */
#include <stdlib.h>    /* Prototypes of commonly used library functions,
                       plus EXIT_SUCCESS and EXIT_FAILURE constants */
#include <unistd.h>    /* Prototypes for many system calls */
#include <errno.h>     /* Declares errno and defines error constants */
#include <string.h>    /* Commonly used string-handling functions */

#include "get_num.h"   /* Declares our functions for handling numeric
                       arguments (getInt(), getLong()) */

#include "error_functions.h" /* Declares our error-handling functions */

typedef enum { FALSE, TRUE } Boolean;

#define min(m,n) ((m) < (n) ? (m) : (n))
#define max(m,n) ((m) > (n) ? (m) : (n))

#endif

lib/tlpi_hdr.h
```

Error-diagnostic functions

To simplify error handling in our example programs, we use the error-diagnostic functions whose declarations are shown in Listing 3-2.

Listing 3-2: Declarations for common error-handling functions

```
lib/error_functions.h
#ifndef ERROR_FUNCTIONS_H
#define ERROR_FUNCTIONS_H

void errMsg(const char *format, ...);

#ifdef __GNUC__

/* This macro stops 'gcc -Wall' complaining that "control reaches
   end of non-void function" if we use the following functions to
   terminate main() or some other non-void function. */

#define NORETURN __attribute__((__noreturn__))
#else
#define NORETURN
#endif

void errExit(const char *format, ...) NORETURN ;
void err_exit(const char *format, ...) NORETURN ;
void errExitEN(int errnum, const char *format, ...) NORETURN ;
void fatal(const char *format, ...) NORETURN ;
void usageErr(const char *format, ...) NORETURN ;
void cmdLineErr(const char *format, ...) NORETURN ;

#endif
lib/error_functions.h
```

To diagnose errors from system calls and library functions, we use *errMsg()*, *errExit()*, *err_exit()*, and *errExitEN()*.

```
#include "tspi_hdr.h"

void errMsg(const char *format, ...);
void errExit(const char *format, ...);
void err_exit(const char *format, ...);
void errExitEN(int errnum, const char *format, ...);
```

The *errMsg()* function prints a message on standard error. Its argument list is the same as for *printf()*, except that a terminating newline character is automatically appended to the output string. The *errMsg()* function prints the error text corresponding to the current value of *errno*—this consists of the error name, such as *EPERM*, plus the error description as returned by *strerror()*—followed by the formatted output specified in the argument list.

The *errExit()* function operates like *errMsg()*, but also terminates the program, either by calling *exit()* or, if the environment variable *EF_DUMPCORE* is defined with a

nonempty string value, by calling *abort()* to produce a core dump file for use with the debugger. (We explain core dump files in Section 22.1.)

The *err_exit()* function is similar to *errExit()*, but differs in two respects:

- It doesn't flush standard output before printing the error message.
- It terminates the process by calling *_exit()* instead of *exit()*. This causes the process to terminate without flushing *stdio* buffers or invoking exit handlers.

The details of these differences in the operation of *err_exit()* will become clearer in Chapter 25, where we describe the differences between *_exit()* and *exit()*, and consider the treatment of *stdio* buffers and exit handlers in a child created by *fork()*. For now, we simply note that *err_exit()* is especially useful if we write a library function that creates a child process that needs to terminate because of an error. This termination should occur without flushing the child's copy of the parent's (i.e., the calling process's) *stdio* buffers and without invoking exit handlers established by the parent.

The *errExitEN()* function is the same as *errExit()*, except that instead of printing the error text corresponding to the current value of *errno*, it prints the text corresponding to the error number (thus, the *EN* suffix) given in the argument *errnum*.

Mainly, we use *errExitEN()* in programs that employ the POSIX threads API. Unlike traditional UNIX system calls, which return *-1* on error, the POSIX threads functions diagnose an error by returning an error number (i.e., a positive number of the type normally placed in *errno*) as their function result. (The POSIX threads functions return *0* on success.)

We could diagnose errors from the POSIX threads functions using code such as the following:

```
errno = pthread_create(&thread, NULL, func, &arg);
if (errno != 0)
    errExit("pthread_create");
```

However, this approach is inefficient because *errno* is defined in threaded programs as a macro that expands into a function call that returns a modifiable lvalue. Thus, each use of *errno* results in a function call. The *errExitEN()* function allows us to write a more efficient equivalent of the above code:

```
int s;

s = pthread_create(&thread, NULL, func, &arg);
if (s != 0)
    errExitEN(s, "pthread_create");
```

In C terminology, an *lvalue* is an expression referring to a region of storage. The most common example of an lvalue is an identifier for a variable. Some operators also yield lvalues. For example, if *p* is a pointer to a storage area, then **p* is an lvalue. Under the POSIX threads API, *errno* is redefined to be a function that returns a pointer to a thread-specific storage area (see Section 31.3).

To diagnose other types of errors, we use *fatal()*, *usageErr()*, and *cmdLineErr()*.

```

#include "tspi_hdr.h"

void fatal(const char *format, ...);
void usageErr(const char *format, ...);
void cmdLineErr(const char *format, ...);

```

The *fatal()* function is used to diagnose general errors, including errors from library functions that don't set *errno*. Its argument list is the same as for *printf()*, except that a terminating newline character is automatically appended to the output string. It prints the formatted output on standard error and then terminates the program as with *errExit()*.

The *usageErr()* function is used to diagnose errors in command-line argument usage. It takes an argument list in the style of *printf()* and prints the string *Usage:* followed by the formatted output on standard error, and then terminates the program by calling *exit()*. (Some of the example programs in this book provide their own extended version of the *usageErr()* function, under the name *usageError()*.)

The *cmdLineErr()* function is similar to *usageErr()*, but is intended for diagnosing errors in the command-line arguments specified to a program.

The implementations of our error-diagnostic functions are shown in Listing 3-3.

Listing 3-3: Error-handling functions used by all programs

```

----- lib/error_functions.c

#include <stdarg.h>
#include "error_functions.h"
#include "tspi_hdr.h"
#include "ename.c.inc"          /* Defines ename and MAX_ENAME */

#ifdef __GNUC__
__attribute__((__noreturn__))
#endif
static void
terminate(Boolean useExit3)
{
    char *s;

    /* Dump core if EF_DUMP CORE environment variable is defined and
       is a nonempty string; otherwise call exit(3) or _exit(2),
       depending on the value of 'useExit3'. */

    s = getenv("EF_DUMP CORE");

    if (s != NULL && *s != '\0')
        abort();
    else if (useExit3)
        exit(EXIT_FAILURE);
    else
        _exit(EXIT_FAILURE);
}

```

```

static void
outputError(Boolean useErr, int err, Boolean flushStdout,
             const char *format, va_list ap)
{
#define BUF_SIZE 500
    char buf[BUF_SIZE], userMsg[BUF_SIZE], errText[BUF_SIZE];

    vsnprintf(userMsg, BUF_SIZE, format, ap);

    if (useErr)
        snprintf(errText, BUF_SIZE, " [%s %s]",
                 (err > 0 && err <= MAX_ENAME) ?
                 ename[err] : "?UNKNOWN?", strerror(err));
    else
        snprintf(errText, BUF_SIZE, ":");

    snprintf(buf, BUF_SIZE, "ERROR%s %s\n", errText, userMsg);

    if (flushStdout)
        fflush(stdout);          /* Flush any pending stdout */
    fputs(buf, stderr);
    fflush(stderr);             /* In case stderr is not line-buffered */
}

void
errMsg(const char *format, ...)
{
    va_list argList;
    int savedErrno;

    savedErrno = errno;         /* In case we change it here */

    va_start(argList, format);
    outputError(TRUE, errno, TRUE, format, argList);
    va_end(argList);

    errno = savedErrno;
}

void
errExit(const char *format, ...)
{
    va_list argList;

    va_start(argList, format);
    outputError(TRUE, errno, TRUE, format, argList);
    va_end(argList);

    terminate(TRUE);
}

```

```

void
err_exit(const char *format, ...)
{
    va_list argList;

    va_start(argList, format);
    outputError(TRUE, errno, FALSE, format, argList);
    va_end(argList);

    terminate(FALSE);
}

void
errExitEN(int errnum, const char *format, ...)
{
    va_list argList;

    va_start(argList, format);
    outputError(TRUE, errnum, TRUE, format, argList);
    va_end(argList);

    terminate(TRUE);
}

void
fatal(const char *format, ...)
{
    va_list argList;

    va_start(argList, format);
    outputError(FALSE, 0, TRUE, format, argList);
    va_end(argList);

    terminate(TRUE);
}

void
usageErr(const char *format, ...)
{
    va_list argList;

    fflush(stdout);          /* Flush any pending stdout */

    fprintf(stderr, "Usage: ");
    va_start(argList, format);
    vfprintf(stderr, format, argList);
    va_end(argList);

    fflush(stderr);          /* In case stderr is not line-buffered */
    exit(EXIT_FAILURE);
}

```

```

void
cmdLineErr(const char *format, ...)
{
    va_list argList;

    fflush(stdout);          /* Flush any pending stdout */

    fprintf(stderr, "Command-line usage error: ");
    va_start(argList, format);
    vfprintf(stderr, format, argList);
    va_end(argList);

    fflush(stderr);          /* In case stderr is not line-buffered */
    exit(EXIT_FAILURE);
}

```

lib/error_functions.c

The file `ename.c.inc` included by Listing 3-3 is shown in Listing 3-4. This file defines an array of strings, `ename`, that are the symbolic names corresponding to each of the possible `errno` values. Our error-handling functions use this array to print out the symbolic name corresponding to a particular error number. This is a workaround to deal with the facts that, on the one hand, the string returned by `strerror()` doesn't identify the symbolic constant corresponding to its error message, while, on the other hand, the manual pages describe errors using their symbolic names. Printing out the symbolic name gives us an easy way of looking up the cause of an error in the manual pages.

The content of the `ename.c.inc` file is architecture-specific, because `errno` values vary somewhat from one Linux hardware architecture to another. The version shown in Listing 3-4 is for a Linux 2.6/x86-32 system. This file was built using a script (`lib/Build_ename.sh`) included in the source code distribution for this book. This script can be used to build a version of `ename.c.inc` that should be suitable for a specific hardware platform and kernel version.

Note that some of the strings in the `ename` array are empty. These correspond to unused error values. Furthermore, some of the strings in `ename` consist of two error names separate by a slash. These strings correspond to cases where two symbolic error names have the same numeric value.

From the `ename.c.inc` file, we can see that the `EAGAIN` and `EWOLDBLOCK` errors have the same value. (SUSv3 explicitly permits this, and the values of these constants are the same on most, but not all, other UNIX systems.) These errors are returned by a system call in circumstances in which it would normally block (i.e., be forced to wait before completing), but the caller requested that the system call return an error instead of blocking. `EAGAIN` originated on System V, and it was the error returned for system calls performing I/O, semaphore operations, message queue operations, and file locking (`fcntl()`). `EWOLDBLOCK` originated on BSD, and it was returned by file locking (`flock()`) and socket-related system calls.

Within SUSv3, `EWOLDBLOCK` is mentioned only in the specifications of various interfaces related to sockets. For these interfaces, SUSv3 permits either `EAGAIN` or `EWOLDBLOCK` to be returned by nonblocking calls. For all other nonblocking calls, only the error `EAGAIN` is specified in SUSv3.

Listing 3-4: Linux error names (x86-32 version)

```

lib/ename.c.inc
static char *ename[] = {
    /* 0 */ "",
    /* 1 */ "EPERM", "ENOENT", "ESRCH", "EINTR", "EIO", "ENXIO", "E2BIG",
    /* 8 */ "ENOEXEC", "EBADF", "ECHILD", "EAGAIN/EWOULDBLOCK", "ENOMEM",
    /* 13 */ "EACCES", "EFAULT", "ENOTBLK", "EBUSY", "EEXIST", "EXDEV",
    /* 19 */ "ENODEV", "ENOTDIR", "EISDIR", "EINVAL", "ENFILE", "EMFILE",
    /* 25 */ "ENOTTY", "ETXTBSY", "EFBIG", "ENOSPC", "ESPIPE", "EROFS",
    /* 31 */ "EMLINK", "EPIPE", "EDOM", "ERANGE", "EDEADLK/DEADLOCK",
    /* 36 */ "ENAMETOOLONG", "ENOLCK", "ENOSYS", "ENOTEMPTY", "ELOOP", "",
    /* 42 */ "ENOMSG", "EIDRM", "ECHRNG", "EL2NSYNC", "EL3HLT", "EL3RST",
    /* 48 */ "ELNRNG", "EUNATCH", "ENOC SI", "EL2HLT", "EBAD", "EBADR",
    /* 54 */ "EXFULL", "ENOANO", "EBADRQC", "EBADSLT", "", "EBFONT", "ENOSTR",
    /* 61 */ "ENODATA", "ETIME", "ENOSR", "ENONET", "ENOPKG", "EREMOTE",
    /* 67 */ "ENOLINK", "EADV", "ESRMNT", "ECOMM", "EPROTO", "EMULTIHOP",
    /* 73 */ "EDOTDOT", "EBADMSG", "EOVERFLOW", "ENOTUNIQ", "EBADFD",
    /* 78 */ "EREMCHG", "ELIBACC", "ELIBBAD", "ELIBSCN", "ELIBMAX",
    /* 83 */ "ELIBEXEC", "EILSEQ", "ERESTART", "ESTRPIPE", "EUSERS",
    /* 88 */ "ENOTSOCK", "EDESTADDRREQ", "EMSGSIZE", "EPROTOTYPE",
    /* 92 */ "ENOPROTOOPT", "EPROTONOSUPPORT", "ESOCKTNOSUPPORT",
    /* 95 */ "EOPNOTSUPP/ENOTSUP", "EPFNOSUPPORT", "EAFNOSUPPORT",
    /* 98 */ "EADDRINUSE", "EADDRNOTAVAIL", "ENETDOWN", "ENETUNREACH",
    /* 102 */ "ENETRESET", "ECONNABORTED", "ECONNRESET", "ENOBUFS", "EISCONN",
    /* 107 */ "ENOTCONN", "ESHUTDOWN", "ETOOMANYREFS", "ETIMEDOUT",
    /* 111 */ "ECONNREFUSED", "EHOSTDOWN", "EHOSTUNREACH", "EALREADY",
    /* 115 */ "EINPROGRESS", "ESTALE", "EUCLEAN", "ENOTNAM", "ENAVAIL",
    /* 120 */ "EISNAM", "EREMOTEIO", "EDQUOT", "ENOMEDIUM", "EMEDIUMTYPE",
    /* 125 */ "ECANCELED", "ENOKEY", "EKEYEXPIRED", "EKEYREVOKED",
    /* 129 */ "EKEYREJECTED", "EOWNERDEAD", "ENOTRECOVERABLE", "ERFKILL"
};

#define MAX_ENAME 132

```

lib/ename.c.inc

Functions for parsing numeric command-line arguments

The header file in Listing 3-5 provides the declaration of two functions that we frequently use for parsing integer command-line arguments: *getInt()* and *getLong()*. The primary advantage of using these functions instead of *atoi()*, *atol()*, and *strtol()* is that they provide some basic validity checking of numeric arguments.

```

#include "tspi_hdr.h"

int getInt(const char *arg, int flags, const char *name);
long getLong(const char *arg, int flags, const char *name);

```

Both return *arg* converted to numeric form

The *getInt()* and *getLong()* functions convert the string pointed to by *arg* to an *int* or a *long*, respectively. If *arg* doesn't contain a valid integer string (i.e., only digits and the characters + and -), then these functions print an error message and terminate the program.

If the *name* argument is non-NULL, it should contain a string identifying the argument in *arg*. This string is included as part of any error message displayed by these functions.

The *flags* argument provides some control over the operation of the *getInt()* and *getLong()* functions. By default, these functions expect strings containing signed decimal integers. By ORing (|) one or more of the *GN_** constants defined in Listing 3-5 into *flags*, we can select alternative bases for conversion and restrict the range of the number to being nonnegative or greater than 0.

The implementations of the *getInt()* and *getLong()* functions are provided in Listing 3-6.

Although the *flags* argument allows us to enforce the range checks described in the main text, in some cases, we don't request such checks in our example programs, even though it might seem logical to do so. For example, in Listing 47-1, we don't check the *init-value* argument. This means that the user could specify a negative number as the initial value for a semaphore, which would result in an error (ERANGE) in the subsequent *semctl()* system call, because a semaphore can't have a negative value. Omitting range checks in such cases allows us to experiment not just with the correct use of system calls and library functions, but also to see what happens when invalid arguments are supplied. Real-world applications would usually impose stronger checks on their command-line arguments.

Listing 3-5: Header file for *get_num.c*

```

----- lib/get_num.h
#ifndef GET_NUM_H
#define GET_NUM_H

#define GN_NONNEG      01      /* Value must be >= 0 */
#define GN_GT_0        02      /* Value must be > 0 */

                                /* By default, integers are decimal */
#define GN_ANY_BASE    0100    /* Can use any base - like strtol(3) */
#define GN_BASE_8      0200    /* Value is expressed in octal */
#define GN_BASE_16     0400    /* Value is expressed in hexadecimal */

long getLong(const char *arg, int flags, const char *name);

int getInt(const char *arg, int flags, const char *name);

#endif
----- lib/get_num.h

```

Listing 3-6: Functions for parsing numeric command-line arguments

```

----- lib/get_num.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include <errno.h>
#include "get_num.h"

```

```

static void
gnFail(const char *fname, const char *msg, const char *arg, const char *name)
{
    fprintf(stderr, "%s error", fname);
    if (name != NULL)
        fprintf(stderr, " (in %s)", name);
    fprintf(stderr, ": %s\n", msg);
    if (arg != NULL && *arg != '\0')
        fprintf(stderr, "    offending text: %s\n", arg);

    exit(EXIT_FAILURE);
}

static long
getNum(const char *fname, const char *arg, int flags, const char *name)
{
    long res;
    char *endptr;
    int base;

    if (arg == NULL || *arg == '\0')
        gnFail(fname, "null or empty string", arg, name);

    base = (flags & GN_ANY_BASE) ? 0 : (flags & GN_BASE_8) ? 8 :
        (flags & GN_BASE_16) ? 16 : 10;

    errno = 0;
    res = strtol(arg, &endptr, base);
    if (errno != 0)
        gnFail(fname, "strtol() failed", arg, name);

    if (*endptr != '\0')
        gnFail(fname, "nonnumeric characters", arg, name);

    if ((flags & GN_NONNEG) && res < 0)
        gnFail(fname, "negative value not allowed", arg, name);

    if ((flags & GN_GT_0) && res <= 0)
        gnFail(fname, "value must be > 0", arg, name);

    return res;
}

long
getLong(const char *arg, int flags, const char *name)
{
    return getNum("getLong", arg, flags, name);
}

int
getInt(const char *arg, int flags, const char *name)
{
    long res;

    res = getNum("getInt", arg, flags, name);
}

```

```

    if (res > INT_MAX || res < INT_MIN)
        gnFail("getInt", "integer out of range", arg, name);

    return (int) res;
}

```

lib/get_num.c

3.6 Portability Issues

In this section, we consider the topic of writing portable system programs. We introduce feature test macros and the standard system data types defined by SUSv3, and then look at some other portability issues.

3.6.1 Feature Test Macros

Various standards govern the behavior of the system call and library function APIs (see Section 1.3). Some of these standards are defined by standards bodies such as The Open Group (Single UNIX Specification), while others are defined by the two historically important UNIX implementations: BSD and System V Release 4 (and the associated System V Interface Definition).

Sometimes, when writing a portable application, we may want the various header files to expose only the definitions (constants, function prototypes, and so on) that follow a particular standard. To do this, we define one or more of the *feature test macros* listed below when compiling a program. One way that we can do this is by defining the macro in the program source code before including any header files:

```
#define _BSD_SOURCE 1
```

Alternatively, we can use the `-D` option to the C compiler:

```
$ cc -D_BSD_SOURCE prog.c
```

The term *feature test macro* may seem confusing, but it makes sense if we look at things from the perspective of the implementation. The implementation decides which of the *features* available in each header it should make visible, by *testing* (with `#if`) which values the application has defined for these *macros*.

The following feature test macros are specified by the relevant standards, and consequently their usage is portable to all systems that support these standards:

`_POSIX_SOURCE`

If defined (with any value), expose definitions conforming to POSIX.1-1990 and ISO C (1990). This macro is superseded by `_POSIX_C_SOURCE`.

`_POSIX_C_SOURCE`

If defined with the value 1, this has the same effect as `_POSIX_SOURCE`. If defined with a value greater than or equal to 199309, also expose definitions for POSIX.1b (realtime). If defined with a value greater than or equal to 199506, also expose definitions for POSIX.1c (threads). If defined with the value 200112, also expose definitions for the POSIX.1-2001 base specification (i.e., the XSI extension is excluded). (Prior to version 2.3.3, the