Appendix B. Miscellaneous Source Code

B.1. Our Header File

Most programs in the text include the header `apue.h`, shown in Figure B.1. It defines constants (such as `MAXLINE`) and prototypes for our own functions.

Figure B.1. Our header: `apue.h`

```
/* Our own header, to be included before all standard system headers */

#ifndef _APUE_H
#define _APUE_H

#define _XOPEN_SOURCE   600  /* Single UNIX Specification, Version 3 */

#include <sys/types.h>       /* some systems still require this */
#include <sys/stat.h>
#include <sys/termios.h>     /* for winsize */
#ifndef TIOCGWINSZ
#include <sys/ioctl.h>
#endif
#include <stdio.h>     /* for convenience */
#include <stdlib.h>    /* for convenience */
#include <stddef.h>    /* for offsetof */
#include <string.h>    /* for convenience */
#include <unistd.h>    /* for convenience */
#include <signal.h>    /* for SIG_ERR */


#define MAXLINE 4096               /* max line length */

/*
 * Default file access permissions for new files.
 */
#define FILE_MODE   (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)

/*
 * Default permissions for new directories.
 */
#define DIR_MODE    (FILE_MODE | S_IXUSR | S_IXGRP | S_IXOTH)

typedef void   Sigfunc(int);   /* for signal handlers */

#if defined(SIG_IGN) && !defined(SIG_ERR)
#define SIG_ERR ((Sigfunc *)-1)
#endif

#define min(a,b)     ((a) < (b) ? (a) : (b))
#define max(a,b)     ((a) > (b) ? (a) : (b))

/*
 * Prototypes for our own functions.
 */
char    *path_alloc(int *);              /* Figure 2.15 */
```

```
long     open_max(void);                 /* Figure 2.16 */
void     clr_fl(int, int);               /* Figure 3.11 */
void     set_fl(int, int);               /* Figure 3.11 */
void     pr_exit(int);                   /* Figure 8.5 */
void     pr_mask(const char *);          /* Figure 10.14 */
Sigfunc *signal_intr(int, Sigfunc *);    /* Figure 10.19 */

int      tty_cbreak(int);                /* Figure 18.20 */
int      tty_raw(int);                   /* Figure 18.20 */
int      tty_reset(int);                 /* Figure 18.20 */
void     tty_atexit(void);               /* Figure 18.20 */
#ifdef  ECHO    /* only if <termios.h>  has been included */
struct termios  *tty_termios(void);      /* Figure 18.20 */
#endif

void     sleep_us(unsigned int);             /* Exercise 14.6 */
ssize_t  readn(int, void *, size_t);         /* Figure 14.29 */
ssize_t  writen(int, const void *, size_t);  /* Figure 14.29 */
void     daemonize(const char *);            /* Figure 13.1 */

int      s_pipe(int *);                 /* Figures 17.6 and 17.13 */
int      recv_fd(int, ssize_t (*func)(int,
                 const void *, size_t));/* Figures 17.21 and 17.23 */
int      send_fd(int, int);             /* Figures 17.20 and 17.22 */
int      send_err(int, int,
                  const char *);        /* Figure 17.19 */
int      serv_listen(const char *);     /* Figures 17.10 and 17.15 */
int      serv_accept(int, uid_t *);     /* Figures 17.11 and 17.16 */

int      cli_conn(const char *);        /* Figures 17.12 and 17.17 */
int      buf_args(char *, int (*func)(int,
                  char **));            /* Figure 17.32 */

int      ptym_open(char *, int);    /* Figures 19.8, 19.9, and 19.10 */
int      ptys_open(char *);         /* Figures 19.8, 19.9, and 19.10 */
#ifdef  TIOCGWINSZ
pid_t    pty_fork(int *, char *, int, const struct termios *,
                  const struct winsize *);      /* Figure 19.11 */
#endif

int     lock_reg(int, int, int, off_t, int, off_t); /* Figure 14.5 */
#define read_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_RDLCK, (offset), (whence), (len))
#define readw_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLKW, F_RDLCK, (offset), (whence), (len))
#define write_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_WRLCK, (offset), (whence), (len))
#define writew_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLKW, F_WRLCK, (offset), (whence), (len))
#define un_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_UNLCK, (offset), (whence), (len))

pid_t   lock_test(int, int, off_t, int, off_t);     /* Figure 14.6 */

#define is_read_lockable(fd, offset, whence, len) \
            (lock_test((fd), F_RDLCK, (offset), (whence), (len)) == 0)
#define is_write_lockable(fd, offset, whence, len) \
            (lock_test((fd), F_WRLCK, (offset), (whence), (len)) == 0)

void    err_dump(const char *, ...);        /* Appendix B */
void    err_msg(const char *, ...);
void    err_quit(const char *, ...);
void    err_exit(int, const char *, ...);
void    err_ret(const char *, ...);
void    err_sys(const char *, ...);

void    log_msg(const char *, ...);         /* Appendix B */
void    log_open(const char *, int, int);
void    log_quit(const char *, ...);
void    log_ret(const char *, ...);
void    log_sys(const char *, ...);

void    TELL_WAIT(void);        /* parent/child from Section 8.9 */
void    TELL_PARENT(pid_t);
void    TELL_CHILD(pid_t);
void    WAIT_PARENT(void);
void    WAIT_CHILD(void);

#endif  /* _APUE_H */
```

Most programs need to include the following headers: `<stdio.h>`, `<stdlib.h>` (for the `exit` function prototype), and `<unistd.h>` (for all the standard UNIX function prototypes). So our header automatically includes these system headers, along with `<string.h>`. This also reduces the size of all the program listings in the text.

The reasons we include our header before all the normal system headers are to allow us to define anything that might be required by headers before they are included, control the order in which header files are included, and allow us to redefine anything that needs to be fixed up to hide the differences between systems.

B.2 Standard Error Routines

Two sets of error functions are used in most of the examples throughout the text to handle error conditions. One set begins with `err_` and outputs an error message to standard error. The other set begins with `log_` and is for daemon processes (Chapter 13) that probably have no controlling terminal.

The reason for our own error functions is to let us write our error handling with a single line of C code, as in

```
    if (error condition)
              err_dump(printf format with any number of arguments);
```

instead of

```
    if (error condition){
              char buf[200];
              sprintf(buf, printf format with any number of arguments);
              perror(buf);
              abort();
    }
```

Our error functions use the variable-length argument list facility from ISO C. See Section 7.3 of Kernighan and Ritchie [1988] for additional details. Be aware that this ISO C facility differs from the `varargs` facility provided by earlier systems (such as SVR3 and 4.3BSD). The names of the macros are the same, but the arguments to some of the macros have changed.

Figure B.2 summarizes the differences between the various error functions.

Figure B.2. Our standard error functions

Function	Adds string from `strerror` ?	Parameter to `strerror`	Terminate ?
`err_dump`	yes	`errno`	`abort();`
`err_exit`	yes	explicit parameter	`exit(1);`
`err_msg`	no		`return;`
`err_quit`	no		`exit(1);`
`err_ret`	yes	`errno`	`return;`
`err_sys`	yes	`errno`	`exit(1);`
`log_msg`	no		`return;`
`log_quit`	no		`exit(2);`
`log_ret`	yes	`errno`	`return;`
`log_sys`	yes	`errno`	`exit(2);`

Figure B.3 shows the error functions that output to standard error.

Figure B.3. Error functions that output to standard error

```
#include "apue.h"
#include <errno.h>      /* for definition of errno */
#include <stdarg.h>     /* ISO C variable aruments */

static void err_doit(int, int, const char *, va_list);

/*
 * Nonfatal error related to a system call.
 * Print a message and return.
 */
void
err_ret(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(1, errno, fmt, ap);
    va_end(ap);
}

/*
 * Fatal error related to a system call.
 * Print a message and terminate.
 */
void
err_sys(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(1, errno, fmt, ap);
    va_end(ap);
    exit(1);
}

/*
 * Fatal error unrelated to a system call.
 * Error code passed as explict parameter.
 * Print a message and terminate.
 */
void
err_exit(int error, const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(1, error, fmt, ap);
    va_end(ap);
    exit(1);
}

/*
 * Fatal error related to a system call.
 * Print a message, dump core, and terminate.
 */
void
err_dump(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(1, errno, fmt, ap);
    va_end(ap);
    abort();        /* dump core and terminate */
    exit(1);        /* shouldn't get here */
}

/*
 * Nonfatal error unrelated to a system call.
 * Print a message and return.
 */
void
err_msg(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(0, 0, fmt, ap);
    va_end(ap);
}

/*
 * Fatal error unrelated to a system call.
 * Print a message and terminate.
 */
void
err_quit(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(0, 0, fmt, ap);
    va_end(ap);
    exit(1);
}

/*
 * Print a message and return to caller.
 * Caller specifies "errnoflag".
 */
static void
err_doit(int errnoflag, int error, const char *fmt, va_list ap)
{
    char    buf[MAXLINE];
   vsnprintf(buf, MAXLINE, fmt, ap);
   if (errnoflag)
       snprintf(buf+strlen(buf), MAXLINE-strlen(buf), ": %s",
         strerror(error));
   strcat(buf, "\n");
   fflush(stdout);     /* in case stdout and stderr are the same */
   fputs(buf, stderr);
   fflush(NULL);       /* flushes all stdio output streams */
}
```

Figure B.4 shows the `log_xxx` error functions. These require the caller to define the variable `log_to_stderr` and set it nonzero if the process is not running as a daemon. In this case, the error messages are sent to standard error. If the `log_to_stderr` flag is 0, the `syslog` facility (Section 13.4) is used.

Figure B.4. Error functions for daemons

```c
/*
 * Error routines for programs that can run as a daemon.
 */

#include "apue.h"
#include <errno.h>      /* for definition of errno */
#include <stdarg.h>     /* ISO C variable arguments */
#include <syslog.h>

static void log_doit(int, int, const char *, va_list ap);

/*
 * Caller must define and set this: nonzero if
 * interactive, zero if daemon
 */
extern int log_to_stderr;

/*
 * Initialize syslog(), if running as daemon.
 */
void
log_open(const char *ident, int option, int facility)
{
    if (log_to_stderr == 0)
        openlog(ident, option, facility);
}

/*
 * Nonfatal error related to a system call.
 * Print a message with the system's errno value and return.
 */
void
log_ret(const char *fmt, ...)
{
    va_list     ap;
    va_start(ap, fmt);
    log_doit(1, LOG_ERR, fmt, ap);
    va_end(ap);
}

/*
 * Fatal error related to a system call.
 * Print a message and terminate.
 */
void
log_sys(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    log_doit(1, LOG_ERR, fmt, ap);
    va_end(ap);
    exit(2);
}

/*
 * Nonfatal error unrelated to a system call.
 * Print a message and return.
 */
void
log_msg(const char *fmt, ...)
{
    va_list ap;

    va_start(ap, fmt);
    log_doit(0, LOG_ERR, fmt, ap);
    va_end(ap);
}

/*
 * Fatal error unrelated to a system call.
 * Print a message and terminate.
 */
void
log_quit(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    log_doit(0, LOG_ERR, fmt, ap);
    va_end(ap);
    exit(2);
}

/*
 * Print a message and return to caller.
 * Caller specifies "errnoflag" and "priority".
 */
static void
log_doit(int errnoflag, int priority, const char *fmt, va_list ap)
{
    int     errno_save;
    char    buf[MAXLINE];

    errno_save = errno;     /* value caller might want printed */
    vsnprintf(buf, MAXLINE, fmt, ap);
    if (errnoflag)
        snprintf(buf+strlen(buf), MAXLINE-strlen(buf), ": %s",
          strerror(errno_save));
    strcat(buf, "\n");
    if (log_to_stderr) {
        fflush(stdout);
        fputs(buf, stderr);
        fflush(stderr);
    } else {
        syslog(priority, buf);
    }
}
```