第17章　与PostScript 打印机通信

17.1　引言

我们现在开发一个可以与 PostScript打印机通信的程序。 PostScript打印机目前使用很广，它一般通过RS-232端口与主机相连。这样就使得我们有可能使用第 11章中的终端I/O函数。同样，与PostScript打印机通信是全双工的，在发送数据给打印机时也要准备好从打印机读取状态消息。这样，又有可能使用 12.5节中的I/O多路转接函数：select 和poll。所开发的这个程序基于James Clark 所写的lprps程序。这个程序和其他一些程序共同组成 lprps软件包，可以在comp.sources.misc新闻组中找到（Volume 21，1991年7月)。

17.2　PostScript 通信机制

关于PostScript打印机所需要知道的第一件事就是我们并不是发送一个文件给打印机去打印 —— 而是发送一个 PostScript程序给打印机让它去执行。在 PostScript打印机中通常有一个PostScript解释器来执行这个程序，生成输出的页面。如果 PostScript程序有错误， PostScript打印机（实际上是PostScript解释器）返回一个错误消息，或许还会产生其他输出。

下面的PostScript程序在输出页面上生成一个熟悉的字符串“ hello, world ”(这里并不叙述PostScript编程，详细情况请参见Adobe Systems 〔1985和1986〕，而是着重在与PostScript打印机的通信上)。

```
%!
/Times-Roman findfont
15 scalefont            % point size of 15
setfont                 % establish current font
300 350 moveto          % x=300 y=350 (position on page)
(hello, world) show     % output the string to current page
showpage                % and output page to output device
```

如果将PostScript程序中的setfont改变为ssetfont，再把它发送到PostScript打印机，结果是什么也没有被打印。相反的，从打印机得到以下消息：

```
%% [ Error: undefined; OffendingCommand: ssetfont ]%%
%% [ Flushing: rest of job (to end-of-file) will be ignored ]%%
```

这些错误消息随时都可能产生，这也是处理 PostScript打印机复杂的地方。我们不能只是将整个PostScript程序发送给打印机后就不管了——还必须处理这些潜在的错误消息（本章所说的“ 打印机 ”，就是指PostScript解释器)。

PostScript打印机通常通过RS-232串口与主机相连。这就如同终端的连接一样，所以第 11章中的终端I/O函数在这里也适用（PostScript打印机也可以通过其他方式连接到主机上，例如逐渐流行的网络接口。但目前占主导地位的还是串口相连)。图17-1显示了典型的工作过程。一个PostScript程序可以产生两种形式的输出：通过 showpage操作输出到打印机页面上，或者通过print操作输出到它的标准输出（在这里是与主机的串口连接)。

PostScript解释器发送和接受的是7位ASCII字符。PostScript程序可包含所有可打印的ASCII

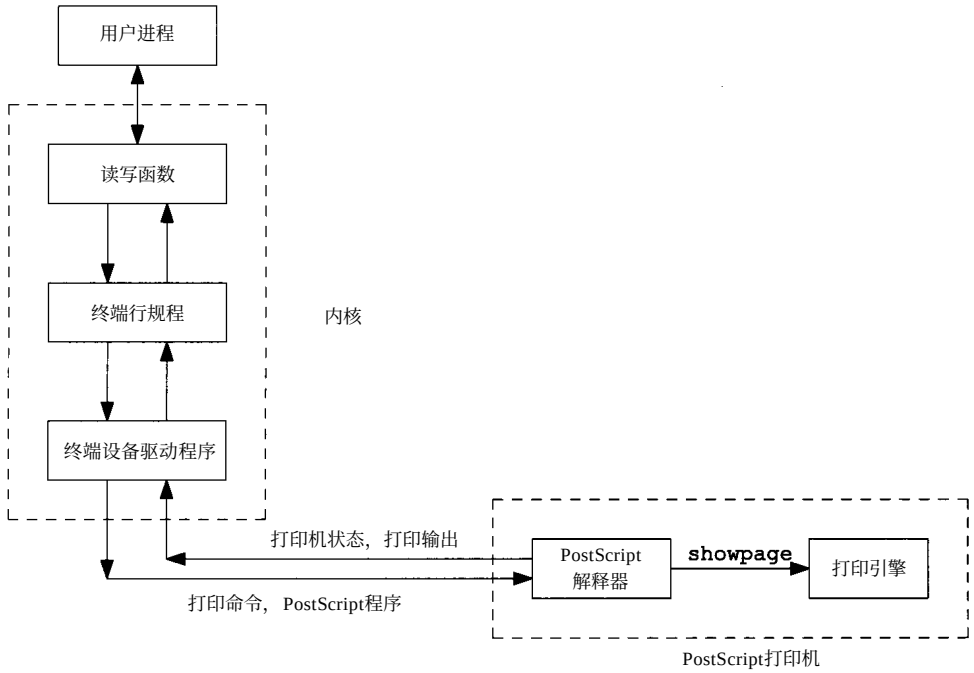

图17-1 用串口连接与PostScript打印机通信

字符。一些不可以打印的字符有着特殊的含义（见表17-1）。

表17-1 从主机发送到PostScript打印机的特殊字符

字 符	八进制值	说 明
Control-C	003	中断，导致PostScript解释器中断正在执行的操作。通常，这会终止正在解释的程序
Control-D	004	文件终止符
Line feed	012	行终止符，PostScript换行字符。如果顺序接收到回车和换行符，则只有换行符传给PostScript解释器
Rerurn	015	行终止符。它被解释为换行符
Control-Q	021	开始输出(XON流控制)
Control-S	023	结束输出(XOFF流控制)
Control-T	024	状态查询。PostScript解释器返回一个一行的状态消息

PostScript的文件终止符（Ctrl-D）用来同步打印机和主机。发送一个PostScript程序到打印机，然后再发送一个EOF到打印机。当打印机执行完PostScript程序后，它就发回一个EOF。

当PostScript解释器正在执行一个PostScript程序时，可以向它发送一个中断（Ctrl-C）。这通常使正在被打印机执行的程序终止。

状态查询消息（Ctrl-T）会使得打印机返回一个一行的状态消息。所有的打印机消息都是如下格式：

```
%% [ key : val ] %%
```

所有可能出现在状态消息中的*key: val*对，被分号分开。回忆前面例子的返回消息：

```
%% [ Error: undefined; OffendingCommand: ssetfont ]%%
%% [ Flushing: rest of job (to end-of-file) will be ignored ]%%
```

这个状态消息具有这个格式：

```
%% [status : idle ]%%
```

除了idle（没有作业）外，其他状态指示还有 busy（正在执行一个 PostScript程序）、waiting（正在等待执行PostScript程序）、printing（打印中）、initializing（初始化）和printing test page（正打印测试页）。

现在来考虑PostScript解释器自己产生的状态消息。可以看到以下的消息：

```
%% [ Error:error ; OffendingCommand operator ]%%
```

总共大约会发生 25种不同的 error。通常的 error有dictstackunderflow, invalidaccess, typecheck, 和undefined。这里的*operator*是产生错误的PostScript操作。

一个打印机的错误用以下形式来指示：

```
%% [ PrinterErrorreason ]%%
```

这儿的*reason*一般是Out Of Paper（缺纸）、Cover Open（盖打开）或Miscellaneous Error（其他错误）。

当错误发生时， PostScript解释器经常会发出另外一个消息：

```
%% [ Flushing : rest of job (to end-of-file) will be ignored ] %%
```

我们看一下在特殊字符序列%%[和]%%中的字符串，为了处理这个消息，必须分析该字符串。一个PostScript程序也可以通过PostScript的print操作来产生输出。这个输出应当传给发送程序给打印机的用户（虽然这并不是打印程序所期望解释的）。

表17-2列出了PostScript解释器传送给主机的特殊字符。

表17-2 PostScript解释器传送给主机的特殊字符

字 符	八进制值	说 明
Control-D	004	文件终止符
Line feed	012	换行符，如果在PostScript解释器的标准输出写入一个换行符，则它被解释为一个回车符加上一个换行符
Control-Q	021	开始输出(XON流控制)
Control-S	023	结束输出(XOFF流控制)

17.3 假脱机打印

本章所开发的程序通过两种方式发送 PostScript程序给PostScript打印机：单独的方式或者通过BSD行式打印机假脱机系统。通常使用假脱机系统，但提供一个独立的方式也很有用，如用于测试等。

UNIX SVR4同样提供了一个假脱机打印系统，在 AT&T手册〔1991〕第一部分以lp开头的手册页中可以找到假脱机系统的详细资料。 Stevens〔1990〕的第13章详细说明了BSD和pre-SVR4的假脱机系统。本章并不着重在假脱机系统上，而在于与 PostScript打印机的通信。

在BSD的假脱机系统中，以如下形式打印一个文件：

```
lpr -pps main.c
```

这个命令发送文件main.c到名为ps的打印机。如果没有指定-pps的选项，那么或者输出到PRINTER环境变量对应的打印机上，或者输出到缺省的lp打印机上。所用的打印机参数可以在/etc/printcap文件中查到。图17-2是对应一个PostScript打印机的一项。

```
lp|ps:\
    :br#19200:lp=/dev/ttyb:\
    :sf:sh:rw:\
    :fc#0000374:fs#0000003:xc#0:xs#0040040:\
    :af=/var/adm/psacct:lf=/var/adm/pslog:sd=/var/spool/pslpd:\
    :if=/usr/local/lib/psif:
```

图17-2 一个PostScript打印机对应的printcap项

第一行给出了该项的名称，ps或者lp。br的值指定了波特率是19 200。lp指定了该打印机的特殊设备文件路径名。sf是格式送纸，sh是指打印作业的开始加入一个打印页头，rw指定打印机以读写方式打开。如17.2节所述，这一项是PostScript打印机所必须的。

下面四个域指定了在旧版本BSD风格的stty结构中需要打开和关闭的位（这里对此进行叙述是因为大多数使用printcap文件的BSD系统都支持这种老式的设置终端方式的方法。在本章的源程序文件中，可以看到如何使用第11章所述的POSIX.1函数来设置所有的终端参数。）首先，fc掩码清除sg_flags元素中的下列值：EVENP和ODDP（关闭了奇偶校验）、RAW（关闭raw模式）、CRMOD（关闭了输入输出中的CR/LF映射）、ECHO（关闭回显）和LCASE（关闭输入输出中的大小写映射）。然后，fs掩码打开了以下位：CBREAK（一次输入一个字符）和TANDEM（主机产生Ctrl-S，Ctrl-Q流控制）。接着，xc掩码清除了本地模式字中各位。最后，xs掩码设置了本地模式字中的下列两位：LDECCTQ（Ctrl-Q重新开始输出，Ctrl-S则停止输出）和LLITOUT（压缩输出转换）。

af和lf字符串分别指定了记帐文件和日志文件。sd指定了假脱机的目录，而if指定了输入过滤程序。

输入过滤程序可被所有的打印文件所调用，格式如下：

filter -n *loginname* -h *hostname acctfile*

这里还有几个可选的参数（这些参数有可能被PostScript打印机所忽略）。要打印的文件在标准输入中，打印机（printcap文件中的lp项）设在标准输出。标准输入也可以是一个管道。

使用一个PostScript打印机，输入过滤程序首先查询输入文件的开始两个字符，确定这个文件是ASCII文本文件还是PostScript程序，通常的惯例是前两个字符为%!表示这是一个PostScript程序。如果这个文件是PostScript程序，lprps程序（下面将详细讨论）就把它发送到打印机。如果这个文件是文本文件，就使用其他程序将它转换成PostScript程序。

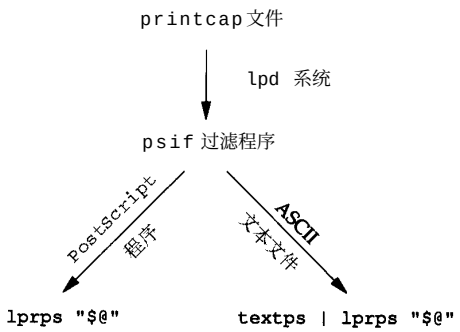

图17-3 lprps系统示意图

printcap文件中提到的psif过滤程序是lprps软件包提供的。这个包中的textps可以将文本文件转换成PostScript程序。图17-3概略表示了这些程序。

图中有一个程序psrev没有表示出来，该程序将PostScript程序生成的页面反转过来，当PostScript打印机在正面而不是在反面打印输出时，就可以使用此程序。

以下将介绍lprps程序的设计和编码。

17.4 源码

先看一下main调用的函数，以及它们是如何与打印机交互作用的。表 17-3详细表明了这种交互作用。表中第二列指定该函数是否可以通过接受 SIGINT信号而中断。第三列指定了各个函数的超时设置（以秒为单位）。注意，当发送用户的PostScript程序到打印机时，没有超时设置。这是因为一个 PostScript程序可能用任一长的时间来执行。函数 get_page行中的“ our PostScript program ”是指程序17-9，这个程序是用来记录当前页码的。

表17-3 main程序调用的函数

函 数	可中断?	超时?	发送给打印机	从打印机得到
get_status	否	5	Ctrl-T	%%[status: idle]%%
get_page	否	30	我们的PostScript程序 EOF EOF	%%[pagecount *n*]%%
send_file	是	无	用户的Postscript程序 EOF EOF	
get_page	否	30	我们的Postscript程序 EOF EOF	%%[pagecount *n*]%%

程序17-1列出了头文件lprps.h。该文件包含在所有的源文件中。该头文件包含了各个源程序所需的系统头文件，定义了全局变量和全局函数的原型。

程序17-1 lprps.h头文件

```
#include    <sys/types.h>
#include    <sys/time.h>
#include    <errno.h>
#include    <signal.h>
#include    <syslog.h>      /* since we're a daemon */
#include    "ourhdr.h"

#define EXIT_SUCCESS    0   /* defined by BSD spooling system */
#define EXIT_REPRINT    1
#define EXIT_THROW_AWAY 2

#define DEF_DEVICE  "/dev/ttyb" /* defaults for debug mode */
#define DEF_BAUD    B19200

                            /* modify following as appropriate */
#define MAILCMD     "mail -s \"printer job\" %s@%s < %s"

#define OBSIZE  1024    /* output buffer */
#define IBSIZE  1024    /* input buffer */
#define MBSIZE  1024    /* message buffer */

                /* declare global variables */
extern char *loginname;
extern char *hostname;
extern char *acct_file;
extern char  eofc;      /* PS end-of-file (004) */
extern int   debug;     /* true if interactive (not a daemon) */
extern int   in_job;    /* true if sending user's PS job to printer */
```

```
extern int    psfd;       /* file descriptor for PostScript printer */
extern int    start_page;/* starting page# */
extern int    end_page;   /* ending page# */
extern volatile sig_atomic_t    intr_flag; /* set if SIGINT is caught */
extern volatile sig_atomic_t    alrm_flag; /* set if SIGALRM goes off */

extern enum status {     /* printer status */
    INVALID, UNKNOWN, IDLE, BUSY, WAITING
} status;

                /* global function prototypes */
void    do_acct(void);              /* acct.c */

void    clear_alrm(void);           /* alarm.c */
void    handle_alrm(void);
void    set_alrm(unsigned int);

void    get_status(void);           /* getstatus.c */

void    init_input(int);            /* input.c */
void    proc_input_char(int);
void    proc_some_input(void);
void    proc_upto_eof(int);

void    clear_intr(void);           /* interrupt.c */
void    handle_intr(void);
void    set_intr(void);

void    close_mailfp(void);         /* mail.c */
void    mail_char(int);
void    mail_line(const char *, const char *);

void    msg_init(void);             /* message.c */
void    msg_char(int);
void    proc_msg(void);

void    out_char(int);              /* output.c */

void    get_page(int *);            /* pagecount.c */

void    send_file(void);            /* sendfile.c */

void    block_write(const char *, int); /* tty.c */
void    tty_flush(void);
void    set_block(void);
void    set_nonblock(void);
void    tty_open(void);
```

文件vars.c（见程序17-2）定义了全局变量。

程序17-3从执行main函数开始。因为此程序一般是作为精灵进程运行的，所以 main函数调用log_open函数（见附录B）。不能将错误消息写到标准错误上——而应使用 13.4.2节中描述的syslog设施。

程序17-2　声明全局变量

```
#include    "lprps.h"

char    *loginname;
char    *hostname;
char    *acct_file;
char    eofc = '\004';          /* Control-D = PostScript EOF */

int     psfd = STDOUT_FILENO;
int     start page = -1;
```

```
int     end_page = -1;
int     debug;
int     in_job;

volatile sig_atomic_t   intr_flag;
volatile sig_atomic_t   alrm_flag;

enum status     status = INVALID;
```

程序17-3 main函数

```
#include    "lprps.h"

static void usage(void);

int
main(int argc, char *argv[])
{
    int         c;

    log_open("lprps", LOG_PID, LOG_LPR);

    opterr = 0;     /* don't want getopt() writing to stderr */
    while ( (c = getopt(argc, argv, "cdh:i:l:n:x:y:w:")) != EOF) {
        switch (c) {
        case 'c':       /* control chars to be passed */
        case 'x':       /* horizontal page size */
        case 'y':       /* vertical page size */
        case 'w':       /* width */
        case 'l':       /* length */
        case 'i':       /* indent */
            break;  /* not interested in these */

        case 'd':       /* debug (interactive) */
            debug = 1;
            break;

        case 'n':       /* login name of user */
            loginname = optarg;
            break;

        case 'h':       /* host name of user */
            hostname = optarg;
            break;

    case '?':
        log_msg("unrecognized option: -%c", optopt);
        usage();
    }
}

if (hostname == NULL || loginname == NULL)
    usage();    /* require both hostname and loginname */

if (optind < argc)
    acct_file = argv[optind];   /* remaining arg = acct file */

if (debug)
    tty_open();

if (atexit(close_mailfp) < 0)   /* register func for exit() */
    log_sys("main: atexit error");

get_status();

get_page(&start_page);

send_file();            /* copies stdin to printer */
```

```
        get_page(&end_page);

        do_acct();

        exit(EXIT_SUCCESS);
    }

    static void
    usage(void)
    {
        log_msg("lprps: invalid arguments");
        exit(EXIT_THROW_AWAY);
    }
```

然后处理命令行参数，很多参数会被PostScript打印机忽略。-d标志指示这个程序是交互运行，而不是作为精灵进程。如果设置了这个标志，则需要初始化终端模式（tty_open）。将函数close_mailfp指定为退出处理程序。

然后可以调用在表17-3中提到的函数：取得打印机状态保证它是就绪的(get_status)，得到打印机的起始页码(get_page)，发送文件（PostScript程序）到打印机(send_file)，得到打印机的结束页码(get_page)，写记帐记录(do_acct)，然后终止。

文件acct.c定义了函数do_acct (见程序17-4)。它在main函数的结尾处被调用，用来写下记帐记录。记帐文件的路径和名字从 printcap文件中的相应记录项（见图 17-2）获得，并作为命令行的最后一个参数。

程序17-4　do_acct函数

```
#include    "lprps.h"

/* Write the number of pages, hostname, and loginname to the
 * accounting file.  This function is called by main() at the end
 * if all was OK, by printer_flushing(), and by handle_intr() if
 * an interrupt is received. */

void
do_acct(void)
{
    FILE    *fp;

    if (end_page > start_page &&
        acct_file != NULL &&
        (fp = fopen(acct_file, "a")) != NULL) {
            fprintf(fp, "%7.2f %s:%s\n",
                        (double)(end_page - start_page),
                        hostname, loginname);
            if (fclose(fp) == EOF)
                log_sys("do_acct: fclose error");
    }
}
```

从历史上看，所有的BSD打印过滤程序都使用%7.2f printf格式，把输出的页数写到记帐文件中。这样就允许光栅设备不使用页数，而以英尺为单位报告输出长度。

下面一个文件是tty.c（见程序17-5），它包含了所有的终端I/O函数。这些函数调用第3章中提到的函数（fcntl、write和open）和第11章中的POSIX.1终端函数（tcflush、tcgetattr、tcsetattr、cfsetispeed 和cfsetospeed）。如果允许发生写阻塞，那么要调用block_write函数。如果不希望发生阻塞，则调用set_nonblock函数，然后再调用read或者write函数。因为PostScript是一个全双工的设备，打印机有可能发送数据回来（例如出错消息等），所以不希望阻塞一个方向的写操作。

如果打印机发送错误消息时，我们正因向其发送数据而处于阻塞状态，则会出现死锁。

一般是内核为终端输入和输出进行缓存，所以如果发生错误，则可以调用 tty_flush来刷清输入和输出队列。

如果以交互方式运行该程序，那么 main函数将调用函数tty_open。需要把终端设为非规范模式，设置波特率和其他一些终端标志。注意各种 PostScript打印机的这些设置并不都一样。检查你的打印机手册确定它的设置。（数据的位数可能是7位或8位，起始位、停止位的数目以及奇偶校验等都可能因打印机而异。）

程序17-5 终端函数

```
#include    "lprps.h"
#include    <fcntl.h>
#include    <termios.h>

static int      block_flag = 1;     /* default is blocking I/O */

void
set_block(void)     /* turn off nonblocking flag */
{                   /* called only by block_write() below */
    int     val;

    if (block_flag == 0) {
        if ( (val = fcntl(psfd, F_GETFL, 0)) < 0)
            log_sys("set_block: fcntl F_GETFL error");
        val &= ~O_NONBLOCK;
        if (fcntl(psfd, F_SETFL, val) < 0)
            log_sys("set_block: fcntl F_SETFL error");

        block_flag = 1;
    }
}

void
set_nonblock(void)  /* set descriptor nonblocking */
{
    int     val;

    if (block_flag) {
        if ( (val = fcntl(psfd, F_GETFL, 0)) < 0)
            log_sys("set_nonblock: fcntl F_GETFL error");
        val |= O_NONBLOCK;
        if (fcntl(psfd, F_SETFL, val) < 0)
            log_sys("set_nonblock: fcntl F_SETFL error");

        block_flag = 0;
    }
}

void
block_write(const char *buf, int n)
{
    set_block();
    if (write(psfd, buf, n) != n)
        log_sys("block_write: write error");
}

void
tty_flush(void)     /* flush (empty) tty input and output queues */
{
    if (tcflush(psfd, TCIOFLUSH) < 0)
        log_sys("tty_flush: tcflush error");
}
```

```
void
tty_open(void)
{
    struct termios  term;

    if ( (psfd = open(DEF_DEVICE, O_RDWR)) < 0)
        log_sys("tty_open: open error");

    if (tcgetattr(psfd, &term) < 0)     /* fetch attributes */
        log_sys("tty_open: tcgetattr error");
    term.c_cflag  = CS8 |           /* 8-bit data */
                    CREAD |         /* enable receiver */
                    CLOCAL;         /* ignore modem status lines */
                                    /* no parity, 1 stop bit */
    term.c_oflag &= ~OPOST;         /* turn off post processing */
    term.c_iflag  = IXON | IXOFF |  /* Xon/Xoff flow control */
                    IGNBRK |        /* ignore breaks */
                    ISTRIP |        /* strip input to 7 bits */
                    IGNCR;          /* ignore received CR */
    term.c_lflag  = 0;      /* everything off in local flag:
                               disables canonical mode, disables
                               signal generation, disables echo */
    term.c_cc[VMIN]  = 1;   /* 1 byte at a time, no timer */
    term.c_cc[VTIME] = 0;
    cfsetispeed(&term, DEF_BAUD);
    cfsetospeed(&term, DEF_BAUD);
    if (tcsetattr(psfd, TCSANOW, &term) < 0)    /* set attributes */
        log_sys("tty_open: tcsetattr error");
}
```

该程序处理两个信号：SIGINT和SIGALRM。处理SIGINT是对BSD假脱机系统调用的任何一种过滤程序的要求。如果打印机作业被lprm(1)命令删除，那么这个信号被发送给过滤程序。使用SIGALRM来设置超时，对这两个信号用类似的方式处理：提供了set_XXX函数来建立信号处理器，clear_XXX函数来取消这个信号处理器。如果有信号传送给这个进程，信号处理器在设置一个全局的标记intr_flag和alrm_flag后返回。程序的其他部分可在适当的时间来检测这些标记，有一个明显的时间是在I/O函数返回错误EINTR时，该程序然后调用handle_intr或者handle_alrm来处理这种情况。调用signal_intr函数（见程序10-13）来中断一个慢速的系统调用。程序17-6是处理SIGINT信号的interrupt.c文件。

当一个中断发生时，必须发送PostScript的中断字符（Ctrl-C）给打印机，接着发送一个文件终止符(EOF)。这通常引起PostScript解释器终止它正在解释的程序，然后等待从打印机返回的EOF（稍后将描述proc_upto_eof函数）。此时读取最后的页码，写下记帐记录，然后就可以终止了。

程序17-6　处理中断信号的interrupt.c文件

```
#include    "lprps.h"

static void
sig_int(int signo)      /* SIGINT handler */
{
    intr_flag = 1;
    return;
}

/* This function is called after SIGINT has been delivered,
 * and the main loop has recognized it.  (It not called as
 * a signal handler, set_intr() above is the handler.) */
```

```
void
handle_intr(void)
{
    char    c;

    intr_flag = 0;
    clear_intr();           /* turn signal off */

    set_alrm(30);       /* 30 second timeout to interrupt printer */

    tty_flush();            /* discard any queued output */
    c = '\003';
    block_write(&c, 1);     /* Control-C interrupts the PS job */
    block_write(&eofc, 1);  /* followed by EOF */
    proc_upto_eof(1);       /* read & ignore up through EOF */

    clear_alrm();

    get_page(&end_page);
    do_acct();
    exit(EXIT_SUCCESS);     /* success since user lprm'ed the job */
}

void
set_intr(void)      /* enable signal handler */
{
    if (signal_intr(SIGINT, sig_int) == SIG_ERR)
        log_sys("set_intr: signal_intr error");
}

void
clear_intr(void)    /* ignore signal */
{
    if (signal(SIGINT, SIG_IGN) == SIG_ERR)
        log_sys("clear_intr: signal error");
}
```

表17-3写明了哪些函数设置超时。仅在以下情况下设置超时：查询打印机状态 (get_status)、读取打印机的页码(get_page)或者正中断打印机时（handle_intr）。如果发生了超时，只需要记录下错误，过一段时间后终止。程序17-7是alarm.c文件。

程序17-7 处理超时的alarm.c文件

```
#include    "lprps.h"

static void
sig_alrm(int signo)         /* SIGALRM handler */
{
    alrm_flag = 1;
    return;
}

void
handle_alrm(void)
{
    log_ret("printer not responding");
    sleep(60);      /* it will take at least this long to warm up */

    exit(EXIT_REPRINT);
}

void        /* Establish the signal handler and set the alarm. */
set_alrm(unsigned int nsec)
```

```
{
    alrm_flag = 0;
    if (signal_intr(SIGALRM, sig_alrm) == SIG_ERR)
        log_sys("set_alrm: signal_intr error");
    alarm(nsec);
}

void
clear_alrm(void)
{
    alarm(0);
    if (signal(SIGALRM, SIG_IGN) == SIG_ERR)
        log_sys("clear_alrm: signal error");
    alrm_flag = 0;
}
```

程序17-8是函数get_status，这个函数由main函数调用。它发送一个Ctrl-T到打印机以获取打印机的状态消息。打印机回送一行消息。如果接收到的消息是：

```
%%[ status : idle ]%%
```

则意味着打印机准备好执行一个新的作业。这个消息由函数 proc_some_input读取和处理（下面将讨论这个函数）。

程序17-8　get_status函数

```
#include    "lprps.h"

/* Called by main() before printing job.
 * We send a Control-T to the printer to fetch its status.
 * If we timeout before reading the printer's status, something
 * is wrong. */

void
get_status(void)
{
    char    c;

    set_alrm(5);            /* 5 second timeout to fetch status */

    tty_flush();
    c = '\024';
    block_write(&c, 1);     /* send Control-T to printer */

    init_input(0);
    while (status == INVALID)
        proc_some_input();  /* wait for something back */

    switch (status) {
    case IDLE:      /* this is what we're looking for ... */
        clear_alrm();
        return;

    case WAITING:   /* printer thinks it's in the middle of a job */
        block_write(&eofc, 1);  /* send EOF to printer */
        sleep(5);
        exit(EXIT_REPRINT);

    case BUSY:
    case UNKNOWN:
        sleep(15);
        exit(EXIT_REPRINT);
    }
}
```

如果接收到下列消息：

```
%%[ status: waiting ]%%
```

则说明打印机正等待我们发送更多的数据以用于当前正打印的作业，这很可能是前一打印作业出了些问题。为了清除这个状态，发送给打印机一个EOF终止符。

PostScript打印机维护着一个页码计数器。这个计数器每打印一页就会增加，它即使在关闭电源时也会保存。为了读此计数器，需要发送给打印机一个PostScript程序。文件pagecount.c（见程序17-9）包含了这个小PostScript程序（含有大约10个PostScript操作符）和函数get_page。

程序17-9 pagecount.c文件 —— 得到打印机的计数器值

```
#include    "lprps.h"

/* PostScript program to fetch the printer's pagecount.
 * Notice that the string returned by the printer:
 *      %%[ pagecount: N ]%%
 * will be parsed by proc_msg(). */

static char pagecount_string[] =
    "(%%[ pagecount: ) print "  /* print writes to current output file */
    "statusdict begin pagecount end "   /* push pagecount onto stack */
    "20 string "        /* creates a string of length 20 */
    "cvs "              /* convert to string */
    "print "            /* write to current output file */
    "( ]%%) print "
    "flush\n";          /* flush current output file */

/* Read the starting or ending pagecount from the printer.
 * The argument is either &start_page or &end_page. */

void
get_page(int *ptrcount)
{
    set_alrm(30);           /* 30 second timeout to read pagecount */

    tty_flush();
    block_write(pagecount_string, sizeof(pagecount_string) - 1);
                            /* send query to printer */
    init_input(0);
    *ptrcount = -1;
    while (*ptrcount < 0)
        proc_some_input();  /* read results from printer */

    block_write(&eofc, 1);  /* send EOF to printer */
    proc_upto_eof(0);       /* wait for EOF from printer */

    clear_alrm();
}
```

pagecount_string数组包含了这个小PostScript程序。虽然我们可以用如下方法得到并打印页码：

```
statusdict begin pagecount end = flush
```

但我们希望得到类似于打印机返回的状态消息的输出格式：

```
%% [ pagecount N ]%%
```

然后，proc_some_input函数处理这个消息，其方式与处理打印机的状态消息相类似。

程序17-10中的函数send_file由main函数调用，它将用户的PostScript程序发送到打印机上。

程序17-10 send_file函数

```
#include    "lprps.h"

void
send_file(void)     /* called by main() to copy stdin to printer */
{
    int     c;

    init_input(1);
    set_intr();             /* we catch SIGINT */

    while ( (c = getchar()) != EOF)     /* main loop of program */
        out_char(c);        /* output each character */
    out_char(EOF);          /* output final buffer */

    block_write(&eofc, 1);  /* send EOF to printer */
    proc_upto_eof(0);       /* wait for printer to send EOF back */
}
```

这个函数主要是一个while循环，首先读取标准输入（getchar），然后调用函数out_char把字符发送给打印机。当在标准输入上遇到EOF时，就发送一个EOF给打印机（指示作业完成），然后等待从打印机返回一个EOF(proc_upto_eof)。

回忆图17-1，连接在串口的PostScript解释器的输出可能是打印机状态消息或者是PostScript的print操作符的输出。所以，我们所认为的“文件被打印”甚至可能一页都不输出。这个PostScript程序文件执行后，把它的结果送回主机。PostScript不是一种编程语言，但是，有时确实需要发送一个PostScript程序到打印机并将结果返回主机，而不是打印在纸上。一个例子是取页码数的PostScript程序，用其可以了解打印机的使用情况。

```
%!
statusdict begin pagecount end =
```

如果从PostScript解释器返回的不是状态消息，就以电子邮件形式发送给用户。程序17-11的mail.c完成这一功能。

程序17-11 mail.c文件

```
#include    "lprps.h"

static FILE *mailfp;
static char temp_file[L_tmpnam];
static void open_mailfp(void);

/* Called by proc_input_char() when it encounters characters
 * that are not message characters.  We have to send these
 * characters back to the user. */

void
mail_char(int c)
{
    static int  done_intro = 0;

    if (in_job && (done_intro || c != '\n')) {
        open_mailfp();
        if (done_intro == 0) {
            fputs("Your PostScript printer job "
                  "produced the following output:\n", mailfp);
            done_intro = 1;
        }
        putc(c, mailfp);
    }
}
```

```
/* Called by proc_msg() when an "Error" or "OffendingCommand" key
 * is returned by the PostScript interpreter.  Send the key and
 * val to the user. */
void
mail_line(const char *msg, const char *val)
{
    if (in_job) {
        open_mailfp();
        fprintf(mailfp, msg, val);
    }
}
/* Create and open a temporary mail file, if not already open.
 * Called by mail_char() and mail_line() above. */
static void
open_mailfp(void)
{
    if (mailfp == NULL) {
        if ( (mailfp = fopen(tmpnam(temp_file), "w")) == NULL)
            log_sys("open_mailfp: fopen error");
    }
}
/* Close the temporary mail file and send it to the user.
 * Registered to be called on exit() by atexit() in main(). */
void
close_mailfp(void)
{
    char    command[1024];
    if (mailfp != NULL) {
        if (fclose(mailfp) == EOF)
            log_sys("close_mailfp: fclose error");
        sprintf(command, MAILCMD, loginname, hostname, temp_file);
        system(command);
        unlink(temp_file);
    }
}
```

每次在打印机返回一个字符，而且这个字符不是状态消息的一部分时，那么就调用函数mail_char（下面将讨论函数proc_input_char，它调用mail_char）。只有当函数send_file正发送一个文件给打印机时，变量in_job才被设置。在其他时候，例如正在读取打印机的状态消息或者打印机的页码计数器值时，它都不会被设置。然后调用函数mail_line，它将一行写入邮件文件中。

当第一次调用函数open_mailfp时，它生成一个临时文件并把它打开。函数 close_mailfp由main函数设置为终止处理程序，当调用 exit时就会调用该函数。如果此时临时邮件文件已经产生，那么关闭这个文件，邮件传给用户。

如果发送一行的PostScript程序：

```
%!
statusdict begin pagecount end =
```

来获得打印机的页码计数，那么返回的邮件消息是：

```
Your PostScript printer job produced the following output:
11185
```

output.c（见程序17-12）包含了函数out_char，send_file调用此函数以便将字符输出到打印机。

程序17-12 output.c 文件

```
#include    "lprps.h"

static char outbuf[OBSIZE];
static int  outcnt = OBSIZE;    /* #bytes remaining */
static char *outptr = outbuf;

static void out_buf(void);

/* Output a single character.
 * Called by main loop in send_file(). */

void
out_char(int c)
{
    if (c == EOF) {
        out_buf();      /* flag that we're all done */
        return;
    }

    if (outcnt <= 0)
        out_buf();      /* buffer is full, write it first */

    *outptr++ = c;      /* just store in buffer */
    outcnt--;
}

/* Output the buffer that out_char() has been storing into.
 * We have our own output function, so that we never block on a write
 * to the printer.  Each time we output our buffer to the printer,
 * we also see if the printer has something to send us.  If so,
 * we call proc_input_char() to process each character. */

static void
out_buf(void)
{
    char    *wptr, *rptr, ibuf[IBSIZE];
    int     wcnt, nread, nwritten;
    fd_set  rfds, wfds;

    FD_ZERO(&wfds);
    FD_ZERO(&rfds);
    set_nonblock();         /* don't want the write() to block */
    wptr = outbuf;          /* ptr to first char to output */
    wcnt = outptr - wptr;   /* #bytes to output */
    while (wcnt > 0) {
        FD_SET(psfd, &wfds);
        FD_SET(psfd, &rfds);
        if (intr_flag)
            handle_intr();
        while (select(psfd + 1, &rfds, &wfds, NULL, NULL) < 0) {
            if (errno == EINTR) {
                if (intr_flag)
                    handle_intr();      /* no return */
            } else
                log_sys("out_buf: select error");
        }
        if (FD_ISSET(psfd, &rfds)) {        /* printer is readable */
            if ( (nread = read(psfd, ibuf, IBSIZE)) < 0)
                log_sys("out_buf: read error");
            rptr = ibuf;
            while (--nread >= 0)
                proc_input_char(*rptr++);
        }
        if (FD_ISSET(psfd, &wfds)) {        /* printer is writeable */
```

```
            if ( (nwritten = write(psfd, wptr, wcnt)) < 0)
                log_sys("out_buf: write error");
            wcnt -= nwritten;
            wptr += nwritten;
        }
    }
    outptr = outbuf;    /* reset buffer pointer and count */
    outcnt = OBSIZE;
}
```

当传送给out_char的参数是EOF时，表明输入已经结束，最后的输出缓存内容应当发送到打印机。

函数out_char把每个字符放到输出缓存中，当缓存满了时调用 out_buf函数。编写out_buf函数必须小心：我们发送数据到打印机，打印机也可能同时送回数据。为了避免写操作的阻塞，必须把描述符设置为非阻塞的。（回忆程序12-1）。使用select 函数来多路转接双向的I/O：输入和输出。在读取和写入时都设置同一个描述符。还有一种可能是 select函数可能会被一个信号（SIGINT）所中断，所以必须在任何错误返回时对此进行检查。

如果从打印机接收到异步输入，则调用 proc_input_char来处理每一个字符。这个输入可能是打印机状态消息或者发送给用户的邮件。

当向打印机write时，必须处理write返回的计数比期望的数量少的情况。同样，回忆程序12-1的例子，其中在每次write时终端可以接收任一数量的数据。

文件input.c（见程序17-13），定义了处理所有从打印机输入的函数。这种输入可以是打印机状态消息或者给用户的输出。

程序17-13 input.c文件 —— 读取和处理打印机的输入

```
#include    "lprps.h"

static int  eof_count;
static int  ignore_input;
static enum parse_state {   /* state of parsing input from printer */
    NORMAL,
    HAD_ONE_PERCENT,
    HAD_TWO_PERCENT,
    IN_MESSAGE,
    HAD_RIGHT_BRACKET,
    HAD_RIGHT_BRACKET_AND_PERCENT
} parse_state;

/* Initialize our input machine. */

void
init_input(int job)
{
    in_job = job;       /* only true when send_file() calls us */
    parse_state = NORMAL;
    ignore_input = 0;
}

/* Read from the printer until we encounter an EOF.
 * Whether or not the input is processed depends on "ignore". */

void
proc_upto_eof(int ignore)
{
    int     ec;

    ignore_input = ignore;
```

```
    ec = eof_count;     /* proc_input_char() increments eof_count */
    while (ec == eof_count)
        proc_some_input();
}

/* Wait for some data then read it.
 * Call proc_input_char() for every character read. */

void
proc_some_input(void)
{
    char    ibuf[IBSIZE];
    char    *ptr;
    int     nread;
    fd_set  rfds;

    FD_ZERO(&rfds);
    FD_SET(psfd, &rfds);
    set_nonblock();
    if (intr_flag)
        handle_intr();
    if (alrm_flag)
        handle_alrm();
    while (select(psfd + 1, &rfds, NULL, NULL, NULL) < 0) {
        if (errno == EINTR) {
            if (alrm_flag)
                handle_alrm();      /* doesn't return */
            else if (intr_flag)
                handle_intr();      /* doesn't return */
        } else
            log_sys("proc_some_input: select error");
    }
    if ( (nread = read(psfd, ibuf, IBSIZE)) < 0)
        log_sys("proc_some_input: read error");
    else if (nread == 0)
        log_sys("proc_some_input: read returned 0");

    ptr = ibuf;
    while (--nread >= 0)
        proc_input_char(*ptr++);    /* process each character */
}

/* Called by proc_some_input() above after some input has been read.
 * Also called by out_buf() whenever asynchronous input appears. */

void
proc_input_char(int c)
{
    if (c == '\004') {
        eof_count++;    /* just count the EOFs */
        return;
    } else if (ignore_input)
        return;         /* ignore everything except EOFs */

    switch (parse_state) {      /* parse the input */
    case NORMAL:
        if (c == '%')
            parse_state = HAD_ONE_PERCENT;
        else
            mail_char(c);
        break;

case HAD_ONE_PERCENT:
    if (c == '%')
        parse_state = HAD_TWO_PERCENT;
    else {
        mail_char('%'); mail_char(c);
```

```
                parse_state = NORMAL;
            }
            break;
        case HAD_TWO_PERCENT:
            if (c == '[') {
                msg_init();     /* message starting; init buffer */
                parse_state = IN_MESSAGE;
            } else {
                mail_char('%'); mail_char('%'); mail_char(c);
                parse_state = NORMAL;
            }
            break;
        case IN_MESSAGE:
            if (c == ']')
                parse_state = HAD_RIGHT_BRACKET;
            else
                msg_char(c);
            break;
        case HAD_RIGHT_BRACKET:
            if (c == '%')
                parse_state = HAD_RIGHT_BRACKET_AND_PERCENT;
            else {
                msg_char(']'); msg_char(c);
                parse_state = IN_MESSAGE;
            }
            break;
        case HAD_RIGHT_BRACKET_AND_PERCENT:
            if (c == '%') {
                parse_state = NORMAL;
                proc_msg();     /* we have a message; process it */
            } else {
                msg_char(']'); msg_char('%'); msg_char(c);
                parse_state = IN_MESSAGE;
            }
            break;
        default:
            abort();
        }
    }
```

每当等待从打印机返回EOF时就会调用函数proc_upto_eof。

函数proc_some_input从串口读取。注意我们调用 select 函数来确定什么时候该描述符是可以读取的，这是因为 select 函数通常被一个捕捉到的信号所中断 —— 它并不自动地重起动。因为select 函数能被SIGALRM或SIGINT所中断，故我们并不希望它重起动。回忆一下 12.5节中关于select函数被正常中断的讨论，同样回忆 10.5节中设置SA_RESTART来说明当一个特定信号出现时，应当自动重起动的 I/O函数。但是因为并不总是有一个附加的标志，使得我们可以说明I/O 函数不应当重起动。如果不设置 SA_RESTART，我们只能依赖系统的缺省值，而这可能是自动重新起动被中断的 I/O函数。当从打印机返回输入时，我们以非阻塞模式读取，得到打印机准备就绪的数据，然后调用函数proc_input_char来处理各个字符。

处理打印机发送给我们的消息是由 proc_input_char完成的。必须检查每一个字符并记住状态。变量parse_state跟踪记录当前状态。调用 msg_char函数把序列%%[以后所有的字符串储存在消息缓存中。当遇到结束序列]%%时，调用 proc_msg来处理消息。除了开始 %%[和最后]%%序列以及二者之间的状态消息其他字符串，都被认为是用户的输出，被邮递给用户（调用 mail_char）。

现在查看那些处理由输入函数积累消息的函数。程序17-14是文件message.c。

当检测到%%[后，调用函数msg_init，它只是初始化缓存计数器。然后对于消息中的每一个字符都调用msg_char函数。

函数proc_msg将消息分解为*key:val*对，并检查*key*。调用ANSI C的strtok函数将消息分解为记号，每个*key*: *val*对用分号分隔。

一个以下形式的消息：

```
%%[ Flushing : rest of job (to end-of-file) will be ignored ]%%
```

引起函数printer_flushing 被调用。它清理终端的缓存，发送一个EOF给打印机，然后等待打印机返回一个EOF。

如果接收到一个以下形式的消息：

```
%%[ PrinterError: reason]%%
```

则调用log_msg函数来记录这个错误。 *key*为*Error*的出错消息邮递传送回用户。这些一般是PostScript程序的错误。

如果返回一个*key*为status的状态消息，它很可能是由于函数get_status发送给打印机一个状态请求（Ctrl-T）而引起的。我们查看*val*，并按照它来设置变量status。

程序17-14　message.c文件，处理从打印机返回的消息

```
#include    "lprps.h"
#include    <ctype.h>

static char msgbuf[MBSIZE];
static int  msgcnt;
static void printer_flushing(void);

/* Called by proc_input_char() after it's seen the "%%[" that
 * starts a message. */
void
msg_init(void)
{
    msgcnt = 0;     /* count of chars in message buffer */
}

/* All characters received from the printer between the starting
 * %%[ and the terminating ]%% are placed into the message buffer
 * by proc_some_input().  This message will be examined by
 * proc_msg() below. */
void
msg_char(int c)
{
    if (c != '\0' && msgcnt < MBSIZE - 1)
        msgbuf[msgcnt++] = c;
}

/* This function is called by proc_input_char() only after the final
 * percent in a "%%[ <message> ]%%" has been seen.  It parses the
 * <message>, which consists of one or more "key: val" pairs.
 * If there are multiple pairs, "val" can end in a semicolon. */
void
proc_msg(void)
{
    char    *ptr, *key, *val;
    int     n;

    msgbuf[msgcnt] = 0;     /* null terminate message */
    for (ptr = strtok(msgbuf, ";"); ptr != NULL;
```

```
                                        ptr = strtok(NULL, ";")) {
        while (isspace(*ptr))
            ptr++;          /* skip leading spaces in key */
        key = ptr;

        if ( (ptr = strchr(ptr, ':')) == NULL)
            continue;   /* missing colon, something wrong, ignore */
        *ptr++ = '\0';  /* null terminate key (overwrite colon) */

        while (isspace(*ptr))
            ptr++;      /* skip leading spaces in val */
        val = ptr;
                        /* remove trailing spaces in val */
        ptr = strchr(val, '\0');
        while (ptr > val && isspace(ptr[-1]))
            --ptr;
        *ptr = '\0';

        if (strcmp(key, "Flushing") == 0) {
            printer_flushing();     /* never returns */

        } else if (strcmp(key, "PrinterError") == 0) {
            log_msg("proc_msg: printer error: %s", val);

        } else if (strcmp(key, "Error") == 0) {
            mail_line("Your PostScript printer job "
                      "produced the error `%s'.\n", val);

        } else if (strcmp(key, "status") == 0) {
            if (strcmp(val, "idle") == 0)
                status = IDLE;
            else if (strcmp(val, "busy") == 0)
                status = BUSY;
            else if (strcmp(val, "waiting") == 0)
                status = WAITING;
            else
                status = UNKNOWN;   /* "printing", "PrinterError",
                       "initializing", or "printing test page". */

        } else if (strcmp(key, "OffendingCommand") == 0) {
            mail_line("The offending command was `%s'.\n", val);

        } else if (strcmp(key, "pagecount") == 0) {
            if (sscanf(val, "%d", &n) == 1 && n >= 0) {
                if (start_page < 0)
                    start_page = n;
                else
                    end_page = n;
            }
        }
    }
}

/* Called only by proc_msg() when the "Flushing" message
 * is received from the printer.  We exit. */

static void
printer_flushing(void)
{
    clear_intr();           /* don't catch SIGINT */

    tty_flush();            /* empty tty input and output queues */
    block_write(&eofc, 1);  /* send an EOF to the printer */

    proc_upto_eof(1);       /* this call won't be recursive,
                               since we specify to ignore input */

    get_page(&end_page);
    do_acct();
    exit(EXIT_SUCCESS);
}
```

key为OffendingCommand一般总是与其他*key: val*对一起出现，如

```
%% [ Error : stackunderflow ; OffendingCommand : pop ]%%
```

则在送回给用户的邮件中就要添加一行。

最后，函数get_page（见程序17-9）中的PostScript程序产生一个pagecount的*key*。我们调用sscanf把*val*转换为二进制，设置起始或结束页面值变量。函数 get_page中的while循环在等待这个变量变成非负值。

17.5 小结

本章实现了一个完整的程序 —— 它发送一个 PostScript程序给连接在 RS-232端口的PostScript打印机。这给我们一个实践机会，把前些章所介绍的很多函数用到一个实用的程序中：I/O多路转接、非阻塞I/O、终端I/O和信号等。

习题

17.1 我们需要使用lprps来打印标准输入的文件，它也可能是一个管道。因为程序 psif一定要查看输入文件的前两个字节，那么应当如何开发 psif程序（见图17-3）来处理这种情况呢？

17.2 实现psif过滤程序，处理前一个习题中的实例。

17.3 参考Adobe Systems〔1998〕12.5节中关于在PostScript程序中字体请求的处理，修改本章中的lprps程序以处理字体请求。