第11章 终 端 I/O

11.1 引言

在操作系统中，终端I/O处理是个非常繁琐的部分， UNIX也不例外。在很多版本的UNIX手册中，终端I/O的手册页常常是最长的部分之一。SVID中的termio手册页至少有16页。

70年代后期，UNIX终端I/O处理发展成两种不同的风格。一种是系统Ⅲ在V7的基础上进行了很多改变而形成的，这种风格由系统Ⅴ沿续下来；另一种则是 V7的风格，它正成为伯克利类系统的标准组成部分。如同信号一样， POSIX.1在这两种风格的基础上制定了终端I/O标准。本章将介绍POSIX.1的终端函数，以及SVR4和4.3+BSD的增加部分。

终端I/O的用途很广泛，包括：终端、计算机之间的直接连接、调制解调器、打印机等等，所以它就变得非常复杂。在后面的若干章中，开发了两个例示终端I/O的程序：一个与PostScript打印机进行通信（见第17章），另一个涉及调制解调器以及远程计算机登录（见第18章）。

11.2 综述

终端I/O有两种不同的工作方式：

(1) 规范方式输入处理。在这种方式中，终端输入以行为单位进行处理。对于每个读要求，终端驱动程序最多返回一行。

(2) 非规范方式输入处理。输入字符不以行为单位进行装配。

如果不作特殊处理，则默认方式是规范方式。例如：若shell的标准输入、输出是终端，在用read和write将标准输入复制到标准输出时，终端以规范方式进行工作，每次 read最多返回一行。处理整个屏幕的程序，例如vi编辑程序使用非规范方式，其原因是其命令是由不以新行符终止的一个或几个字符组成的。另外，该编辑程序使用了若干特殊字符作为编辑命令，所以它也不希望系统对特殊字符进行处理。例如， Ctrl-D字符通常是终端的文件结束符，但在vi中它是向下滚动半个屏幕的命令。

> V7和BSD类的终端驱动程序支持三种终端输入方式：(a) 精细加工方式（输入装配成行，并对特殊字符进行处理），(b) 原始方式（输入不装配成行，也不对特殊字符进行处理），(c) cbreak方式（输入不装配成行，但对某些特殊字符进行处理）。程序11-10显示了将终端设置为cbreak或原始方式的POSIX.1函数。

POSIX.1定义了11个特殊输入字符，其中9个可以改变。本章已经用到了其中几个，例如：文件结束符（通常是Ctrl-D），挂起字符（通常是Ctrl-Z）。11.3节对其中每个字符都进行了说明。

终端设备是由一般位于内核中的终端驱动程序所控制的。每个终端设备有一个输入队列，一个输出队列，见图11-1。

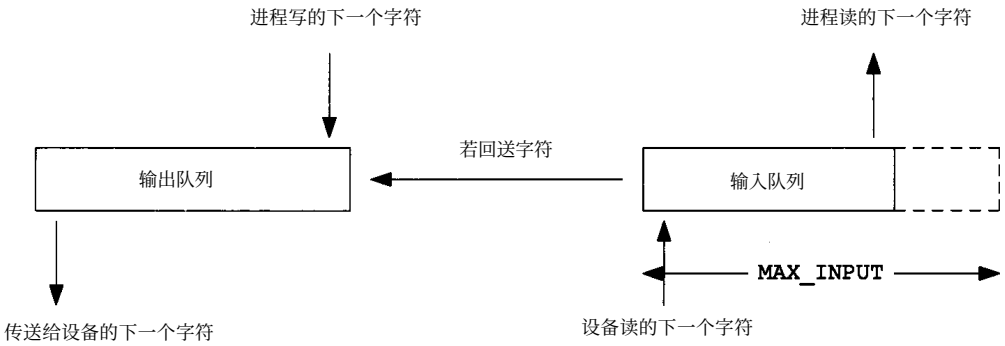

图11-1 终端设备的输入、输出队列的逻辑结构

对此图要说明下列几点：

• 如果需要回送，则在输入队列和输出队列之间有一个隐含的连接。

• 输入队列的长度MAX_INPUT（见表2-5）是有限值，当一个特定设备的输入队列已经填满时，系统作何种处理依赖于实现。当此发生时，大多数UNIX系统回送响铃字符。

• 图中没有显示另一个输入限制MAX_CANON，它是在一个规范输入行中的最大字节数。

• 虽然输出队列通常也是有限长度，但是程序不能存取定义其长度的常数。这是因为当输出队列要填满时，内核使写进程睡眠直至写队列中有可用的空间，所以程序无需关心该队列的长度。

• 我们将说明如何使用tcflush函数刷清输入或输出队列。与此类似，在说明tcsetattr函数时，将会了解到如何通知系统仅在输出队列空时改变一个终端的属性。（例如，正在改变输出属性时可能就要这样做。）我们也能通知系统，当它正在改变终端属性时，丢弃在输入队列中的任何东西。（如果正在改变输入属性，或者在规范和非规范方式之间进行转换，则可能希望这样做，以免以错误的方式对以前输入的字符进行解释。）

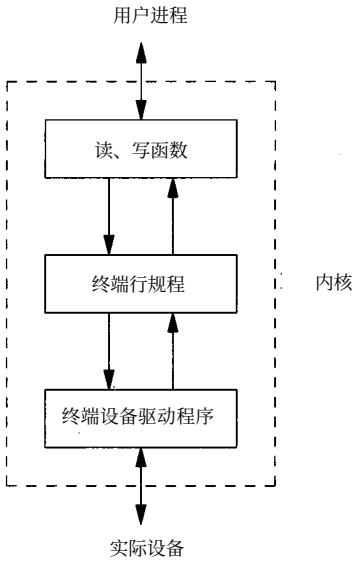

图11-2 终端行规程

大多数UNIX系统在一个称为终端行规程（*terminal line discipline*）的模块中进行规范处理。它是位于内核类属读、写函数和实际设备驱动程序之间的模块，见图11-2。

12.4节讨论流I/O系统以及第19章讨论伪终端时还将使用此图。

所有我们可以检测和更改的终端设备特性都包含在 termios结构中。该结构在头文件<termios.h>中定义，本章经常使用这一头文件。

```
struct termios {
  tcflag_t  c_iflag;    /* input flags */
  tcflag_t  c_oflag;    /* output flags */
  tcflag_t  c_cflag;    /* control flags */
  tcflag_t  c_lflag;    /* local flags */
  cc_t      c_cc[NCCS]; /* control characters */
};
```

粗略而言，输入标志由终端设备驱动程序用来控制输入特性（剥除输入字节的第 8位，允许输入奇偶校验等等），输出标志则控制输出特性（执行输出处理，将新行映照为 CR/LF等），控制标志影响到RS-232串行线（忽略调制解调器的状态线，每个字符的一个或两个停止位等等），本地标志影响驱动程序和用户之间的界面（回送的开或关，可视的擦除符，允许终端产生的信号，对后台作业输出的控制停止信号等）。

类型tcflag_t的长度是以保持每个标志值。它经常被定义为 unsigned long。c_cc数组包含了所有可以更改的特殊字符。 NCCS是该数组的长度，其典型值在 11~18之间（大多数UNIX实现支持的特殊字符较POSIX.1所定义的11个更多）。cc_t类型的长度足以保持每个专用字符，典型的是unsigned char。

> 系统V的早期版本有一个名为 <termio.h>的头文件，一个名为 termio的数据结构，为了区别于这些名字，POSIX.1在这些名字后加了一个s。

表11-1列出了所有可以更改以影响终端设备特性的终端标志。注意， POSIX.1定义的标志是SVR4和4.3+BSD都支持的，但是它们还各有自己的扩充部分。这些扩充部分与这两个系统的各自历史发展过程有关。 11.5节将详细讨论这些标志值。

给出了表11-1中的所有选择项后，如何才能检测和更改终端设备的这些特性呢？表11-2列出了POSIX.1所定义的对终端设备进行操作的各个函数。(9.7节已说明了tcgetpgrp和tcsetpgrp函数。)

表11-1 终端标志

字 段	标 志	说 明	POSIX.1	SVR4 扩展	4.3+BSD 扩展
c_iflag	BRKINT	接到BREAK时产生SIGINT	•		
	ICRNL	将输入的CR转换为NL	•		
	IGNBRK	忽略BREAK条件	•		
	IGNCR	忽略CR	•		
	IGNPAR	忽略奇偶错字符	•		
	IMAXBEL	在输入队列空时振铃		•	•
	INLCR	将输入的NL转换为CR	•		
	INPCK	打开输入奇偶校验	•		
	ISTRIP	剥除输入字符的第8位	•		
	IUCLC	将输入的大写字符转换成小写字符		•	
	IXANY	使任一字符都重新起动输出		•	•
	IXOFF	使起动/停止输入控制流起作用	•		
	IXON	使起动/停止输出控制流起作用	•		
	PARMRK	标记奇偶错	•		
c_oflag	BSDLY	退格延迟屏蔽		•	
	CRDLY	CR延迟屏蔽		•	
	FFDLY	换页延迟屏蔽		•	
	NLDLY	NL延迟屏蔽		•	
	OCRNL	将输出的CR转换为NL		•	
	OFDEL	填充符为DEL，否则为NUL		•	

(续)

字 段	标 志	说 明	POSIX.1	SVR4 扩展	4.3+BSD 扩展
	OFILL	对于延迟使用填充符		•	
	OLCUC	将输出的小写字符转换为大写字符		•	
	ONLCR	将NL转换为CR-NL		•	•
	ONLRET	NL执行CR功能		•	
	ONOCR	在0列不输出CR		•	
	ONOEOT	在输出中删除EOT字符			•
	OPOST	执行输出处理	•		
	OXTABS	将制表符扩充为空格			•
	TABDLY	水平制表符延迟屏蔽		•	
	VTDLY	垂直制表符延迟屏蔽		•	
c_cflag	CCTS_OFLOW	输出的CTS流控制			•
	CIGNORE	忽略控制标志			•
	CLOCAL	忽略解制-解调器状态行	•		
	CREAD	启用接收装置	•		
	CRTS_IFLOW	输入的RTS流控制			•
	CSIZE	字符大小屏蔽	•		
	CSTOPB	送两个停止位，否则为1位	•		
	HUPCL	最后关闭时断开	•		
	MDMBUF	经载波的流控输出			•
	PARENB	进行奇偶校	•		
	PARODD	奇校，否则为偶校	•		
c_lflag	ALTWERASE	使用替换WERASE算法			•
	ECHO	进行回送	•		
	ECHOCTL	回送控制字符为^(char)		•	•
	ECHOE	可见擦除符	•		
	ECHOK	回送kill符	•		
	ECHOKE	kill的可见擦除		•	•
	ECHONL	回送NL	•		
	ECHOPRT	硬拷贝的可见擦除方式		•	•
	FLUSHO	刷清输出		•	•
	ICANON	规范输入	•		
	IEXTEN	使扩充的输入字符处理起作用	•		
	ISIG	使终端产生的信号起作用	•		
	NOFLSH	在中断或退出键后不刷清	•		
	NOKERNINFO	STATUS不使内核输出			•
	PENDIN	重新打印		•	•
	TOSTOP	对于后台输出发送SIGTTOU	•		
	XCASE	规范大/小写表示		•	

注意，对终端设备， POSIX.1没有使用ioctl，而使用了表11-2中列出的12个函数。这样做的理由是：对于终端设备的ioctl函数，其最后一个参数的数据类型随执行动作的不同而不同。

虽然只有12个函数对终端设备进行操作，但是应当理解的是，表 11-2中头两个函数tcgetattr和tcsetattr处理大约50种不同的标志（见表11-1）。对于终端设备有大量选择项可供使用，对于一个特定设备（终端、调制解调器、激光打印机等等）又要决定所需的选择项，这些都使对终端设备的处理变得复杂起来。

表11-2 POSIX.1终端I/O函数

函 数	说 明
tcgetattr	取属性(termios结构)
tcsetattr	设置属性(termios结构)
cfgetispeed	得到输入速度
cfgetospeed	得到输出速度
cfsetispeed	设置输入速度
cfsetospeed	设置输出速度
tcdrain	等待所有输出都被传输
tcflow	挂起传输或接收
tcflush	刷清未决输入和/或输出
tcsendbreak	送BREAK字符
tcgetpgrp	得到前台进程组ID
tcsetpgrp	设置前台进程组ID

表11-2中列出的12个函数之间的关系示于图11-3中。

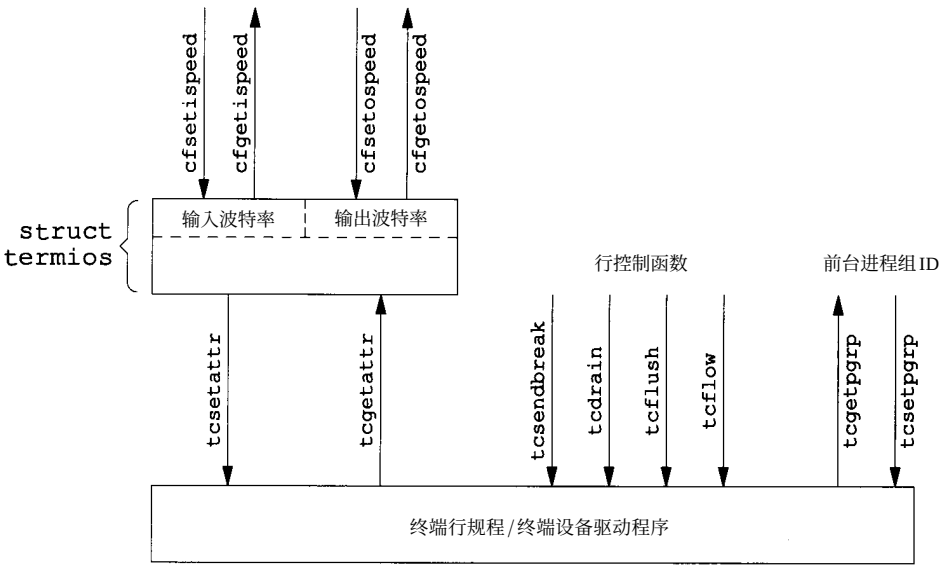

图11-3 与终端有关的函数之间的关系

POSIX.1没有规定在termios结构中何处存放波特率信息，这是一个依赖于实现的特性。很多早期的系统将此信息存放在 c_cflag字段中。4.3+BSD则在此结构中有两个分开的字段——一个存放输入速度，另一个则存放输出速度。

11.3 特殊输入字符

POSIX.1定义了11个在输入时作特殊处理的字符。 SVR4另外加了6个特殊字符， 4.3+BSD

则加了7个。表11-3列出了这些特殊字符。

表11-3 终端特殊输入字符

字 符	说 明	c_cc 下标	起作用，由： 字段	起作用，由： 标志	典型值	POSIX.1	SVR4 4.3+BSD 扩充	
CR	回车	不能更改	c_lflag	ICANON	\r	•		
DISCARD	擦除输出	VDISCARD	c_lflag	IEXTEN	^O		•	•
DSUSP	延迟挂起(SIGTSTP)	VDSUSP	c_lflag	ISIG	^Y		•	•
EOF	文件结束	VEOF	c_lflag	ICANON	^D	•		
EOL	行结束	VEOL	c_lflag	ICANON		•		
EOL2	替换的行结束	VEOL2	c_lflag	ICANON			•	•
ERASE	擦除字符	VERASE	c_lflag	ICANON	^H	•		
INTR	中断信号(SIGINT)	VINTR	c_lflag	ISIG	^?, ^C	•		
KILL	擦行	VKILL	c_lflag	ICANON	^U	•		
LNEXT	下一个字列字符	VLNEXT	c_lflag	IEXTEN	^V		•	•
NL	新行	不能更改	c_lflag	ICANON	\n	•		
QUIT	退出信号(SIGQUIT)	VQUIT	c_lflag	ISIG	^\	•		
REPRINT	再打印全部输入	VREPRINT	c_lflag	ICANON	^R		•	•
START	恢复输出	VSTART	c_iflag	IXON/IXOFF	^Q	•		
STATUS	状态要求	VSTATUS	c_lflag	ICANON	^T			•
STOP	停止输出	VSTOP	c_iflag	IXON/IXOFF	^S	•		
SUSP	挂起信号(SIGTSTP)	VSUSP	c_lflag	ISIG	^Z	•		
WERASE	擦除字	VWERASE	c_lflag	ICANON	^W		•	•

在POSIX.1的11个特殊字符中，可将其中9个更改为几乎任何值。不能更改的两个特殊字符是新行符和回车符（\n和\r），有些实施也不允许更改STOP和START字符。为了进行修改，只要更改termios结构中c_cc数组的相应项。该数组中的元素都用名字作为下标进行引用，每个名字都以字母V开头（见表11-3中的第3列）。

POSIX.1可选地允许禁止使用这些字符。若 _POSIX_VDISABLE有效，则 _POSIX_VDISABLE的值可存放在 c_cc数组的相应项中以禁止使用该特殊字符。可以用 pathconf和fpathconf函数查询此特征（见2.5.4节）。

> FIPS151-1要求支持_POSIX_VDISABLE。
>
> SVR4和4.3+BSD也支持此特性。SVR4将_POSIX_VDISABLE定义为0，而4.3+BSD则将其定义为八进制数377。
>
> 某些早期的UNIX系统所用的方法是：若相应的特殊输入字符是 0，则禁止使用该字符。

实例

在详细说明各特殊字符之前，先看一个更改特殊字符的程序。程序 11-1禁用中断字符，并将文件结束符设置为Ctrl-B。

程序11-1 禁止中断字符和更改文件结束字符

```
#include    <termios.h>
#include    "ourhdr.h"

int
main(void)
{
    struct termios  term;
```

```
    long            vdisable;

    if (isatty(STDIN_FILENO) == 0)
        err_quit("standard input is not a terminal device");

    if ( (vdisable = fpathconf(STDIN_FILENO, _PC_VDISABLE)) < 0)
        err_quit("fpathconf error or _POSIX_VDISABLE not in effect");

    if (tcgetattr(STDIN_FILENO, &term) < 0) /* fetch tty state */
        err_sys("tcgetattr error");

    term.c_cc[VINTR] = vdisable;    /* disable INTR character */
    term.c_cc[VEOF]  = 2;           /* EOF is Control-B */

    if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &term) < 0)
        err_sys("tcsetattr error");

    exit(0);
}
```

对此程序要说明下列几点：

(1) 仅当标准输入是终端设备时才修改终端特殊字符。 isatty （见11.9节）用于进行这种检测。

(2) 用fpathconf取_POSIX_VDISABLE值。

(3) 函数tcgetattr（见11.4节）从内核存取termios结构。在修改了此结构后，调用tcsetattr设置属性，这样就进行了我们所希望的修改。

(4) 禁止使用中断键与忽略中断信号是不同的。程序 11-1所做的是禁止使用使终端驱动程序产生SIGINT的特殊字符。但仍可使用kill函数向进程发送该信号。

下面较详细地说明各个特殊字符。我们称这些字符为特殊输入字符，但是其中有两个字符，STOP和START（Ctrl-S和Ctrl-Q）在输出时也对它们进行特殊处理。注意，这些字符中的大多数在被终端驱动程序识别并进行特殊处理后都被丢弃，并不将它们传送给执行读终端操作的进程。例外的字符是新行符（NL，EOL，EOL2）和回车符（CR）。

• CR　POSIX.1的回车符。不能更改此字符。以规范方式进行输入时识别此字符。当设置了ICANON（规范方式）和ICRNL（将CR映照为NL）以及没有设置IGNCR（忽略CR）时，将CR转换成NL，并产生与NL符相同的作用。

此字符返回给读进程（多半是在转换成NL后）。

• DISCARD　SVR4和4.3+BSD的删除符。在扩充方式下（IEXTEN），在输入中识别此字符。在输入另一个删除符之前或删除条件被清除之前（见 FLUSHO选择项）此字符使后续输出都被删除。在处理后此字符即被删除，不送向读进程。

• DSUSP　SVR4和4.3+BSD的延迟-挂起作业控制字符。在扩充方式下，若作业控制被挂起并且ISIG标志被设置，则在输入中识别此字符。与 SUSP字符的相同处是：延迟-挂起字符产生SIGTSTP信号，它被送至前台进程组中的所有进程（见图 9-7）。但是并不是键入此字符时，而是在一个进程读控制终端时，此延迟 -挂起字符才送向进程组。在处理后，此字符即被删除，不送向读进程。

• EOF　POSIX.1的文件结束符。以规范方式进行输入时识别此字符。当键入此字符时，等待被读的所有字节都立即传送给读进程。如果没有字节等待读，则返回　0。在行首输入一个EOF符是向程序指示文件结束的正常方式。在处理后，此字符即被删除，不送向读进程。

• EOL　POSIX.1附加的行定界符，与NL作用相同。以规范方式进行输入时识别此字符。通常不使用此字符。此字符返回给读进程。

• EOL2 SVR4和4.3+BSD的附加行定界符，与NL作用相同。以规范方式输入时识别此字符。通常不使用此字符，此字符返回给读进程。

• ERASE POSIX.1的擦除字符（退格）。以规范方式输入时识别此字符。它擦除行中的前一个字符，但不会超越行首字符擦除上一行中的字符。在处理后此字符即被擦除，不送向读进程。

• INTR POSIX.1的中断字符。若设置了 ISIG标志，则在输入中识别此字符。它产生SIGINT信号，该信号被送至前台进程组中的所有进程（见图 9-7）。在处理后，此字符即被删除，不送向读进程。

• KILL POSIX.1的kill（杀死）字符。（名字“ 杀死 ”在这里又一次被误用，它应被称为行擦除符。）以规范方式输入时识别此字符。它擦除整个 1行。在处理后，此字符即被删除，不送向读进程。

• LNEXT SVR4和4.3+BSD的“ 字面上-下一个 ”字符。以规范方式输入时识别此字符，它使下一个字符的任何特殊含意都被忽略。这对本节提及的所有特殊字符都起作用。使用这一字符可向程序键入任何字符。在处理后， LNEXT字符即被删除，但输入的下一个字符则被传送给读进程。

• NL POSIX.1的新行字符，它也被称为行定界符。不能更改此字符。以规范方式输入时识别此字符。此字符返回给读进程。

• QUIT POSIX.1的退出字符。若设置了 ISIG标志，则在输入中识别此字符。它产生SIGQUIT信号，该信号又被送至前台进程组中的所有进程（见图 9-7）。在处理后，此字符即被删除，不送向读进程。

回忆表10-1，INTR和QUIT之间的区别是： QUIT字符不仅按默认终止进程，而且也产生core文件。

• REPRINT SVR4和4.3+BSD的再打印字符。以扩充规范方式（设置了 IEXTEN和ICANON标志）进行输入时识别此字符。它使所有未读的输入被输出（再回送）。在处理后，此字符即被删除，不送向读进程。

• START POSIX.1的起动字符。若设置了 IXON标志则在输入中识别此字符；若设置IXOFF标志，则作为输出自动产生此字符。在 IXON已设置时接收到的START字符使停止的输出（由以前输入的STOP字符造成）重新起动。在此情形下，在处理后，此字符即被删除，不送向读进程。

在IXOFF标志设置时，若输入不会使输入缓存溢出，则终端驱动程序自动地产生一 START字符以恢复以前被停止的输入。

• STATUS 4.3+BSD的状态-要求字符。以扩充、规范方式进行输入时识别此字符。它产生SIGINFO信号，该信号又被送至前台进程组中的所有进程（见图 9-7）。另外，如果没有设置NOKERNINFO标志，则有关前台进程组的状态信息也显示在终端上。在处理后，此字符即被删除，不送向读进程。

• STOP POSIX.1的停止字符。若设置了IXON标志，则在输入中识别此字符；若IXOFF标志已设置则作为输出自动产生此字符。在IXON已设置时接收到STOP字符则停止输出。在此情形下，在处理后删除此字符，不送向读进程。当输入一个START字符后，停止的输出重新起动。

在IXOFF设置时，终端驱动程序自动地产生一个STOP字符以防止输入缓存溢出。

• SUSP POSIX.1的挂起作业控制字符。若支持作业控制并且 ISIG标志已设置，则在输入中识别此字符。它产生SIGTSTP信号，该信号又被送至前台进程组的所有进程（见图 9-7）。在处理后，此字符即被删除，不送向读进程。

• WERASE　SVR4和4.3+BSD的字擦除字符。以扩充、规范方式进行输入时识别此字符。它使前一个字被擦除。首先，它向后跳过任一白空字符（空格或制表符），然后向后越过前一记号，使光标处在前一个记号的第一个字符位置上。通常，前一个记号在碰到一个白空字符时即终止。但是，可用设置ALTWERASE标志来改变这一点。

此标志使前一个记号在碰到第一个非字母、数字符时即终止。在处理后，此字符即被删除，不送向读进程。

需要为终端设备定义的另一个“字符”是BREAK。BREAK实际上并不是一个字符，而是在异步串行数据传送时发生的一个条件。依赖于串行界面，可以有多种方式通知设备驱动程序发生了BREAK条件。大多数终端有一个标记为BREAK的键，用其可以产生BREAK条件，这就使得很多人认为BREAK就是一个字符。对于异步串行数据传送，BREAK是一个0值的位序列，其持续时间长于要求发送一个字节的时间。整个0值位序列被视为是一个BREAK。11.8节将说明如何发送一个BREAK。

11.4 获得和设置终端属性

使用函数tcgetattr和tcsetattr可以获得或设置termios。这样也就可以检测和修改各种终端选择标志和特殊字符，以使终端按我们所希望的方式进行操作。

```
#include <termios.h>

int tcgetattr(int filedes, struct termios *termptr);

int tcsetattr(int filedes, int opt, const struct termios *termptr);
```

两个函数返回：若成功则为0，若出错则为-1

这两个函数都有一个指向termios结构的指针作为其参数，它们返回当前终端的属性，或者设置该终端的属性。因为这两个函数只对终端设备进行操作，所以若*filedes*并不引用一个终端设备则出错返回，errno设置为ENOTTY。

tcsetattr的参数*opt*使我们可以指定在什么时候新的终端属性才起作用。*opt*可以指定为下列常数中的一个：

• TCSANOW　更改立即发生。

• TCSADRAIN　发送了所有输出后更改才发生。若更改输出参数则应使用此选择项。

• TCSAFLUSH　发送了所有输出后更改才发生。更进一步，在更改发生时未读的所有输入数据都被删除（刷清）。

tcsetattr函数的返回值易于产生混淆。如果它执行了任意一种所要求的动作，即使未能执行所有要求的动作，它也返回0（表示成功）。如果该函数返回0，则我们有责任检查该函数是否执行了所有要求的动作。这就意味着，在调用tcsetattr设置所希望的属性后，需调用tcgetattr，然后将实际终端属性与所希望的属性相比较，以检测两者是否有区别。

11.5 终端选择标志

本节对表11-7中列出的各个终端选择标志按字母顺序作进一步说明，也指出该选择项出现在四个终端标志字段中的哪一个，以及该选择项是否是POSIX.1定义的，或是受到SVR4或4.3+BSD支持的。

所有列出的选择标志（除屏蔽标志外）都用一或多位表示，而屏蔽标志则定义多位。屏蔽

标志有一个定义名，每个值也有一个名字。例如，为了设置字符长度，首先用字符长度屏蔽标志CSIZE将表示字符长度的位清0，然后设置下列值之一：CS5、CS6、CS7或CS8。

由SVR4支持的6个延迟值也有屏蔽标志：BSDLY、CRDLY、FFDLY、NLDLY、TABDLY和VTDLY。对于每个延迟值的长度请参阅termio(7)手册页（AT&T〔1991〕）。如果指定了一个延迟，则OFILL和OFDEL标志决定是驱动器进行实际延迟还是只是传输填充字符。

实例

程序11-2例示了使用屏蔽标志取或设置一个值。

程序11-2 tcgetattr实例

```
#include    <termios.h>
#include    "ourhdr.h"

int
main(void)
{
    struct termios  term;
    int             size;

    if (tcgetattr(STDIN_FILENO, &term) < 0)
        err_sys("tcgetattr error");

    size = term.c_cflag & CSIZE;
    if      (size == CS5)   printf("5 bits/byte\n");
    else if (size == CS6)   printf("6 bits/byte\n");
    else if (size == CS7)   printf("7 bits/byte\n");
    else if (size == CS8)   printf("8 bits/byte\n");
    else                    printf("unknown bits/byte\n");

    term.c_cflag &= ~CSIZE;      /* zero out the bits */
    term.c_cflag |= CS8;         /* set 8 bits/byte */

    if (tcsetattr(STDIN_FILENO, TCSANOW, &term) < 0)
        err_sys("tcsetattr error");

    exit(0);
}
```

下面说明各选择标志：

• ALTWERASE （c_lflag,4.3+BSD）此标志设置时，若输入了WERASE字符，则使用一个替换的字擦除算法。它不是向后移动到前一个白空字符为止，而是向后移动到第一个非字母、数字符为止。

• BRKINT （c_iflag,POSIX.1）若此标志设置，而IGNBRK未设置，则在接到BREAK时，输入、输出队列被刷清，并产生一个 SIGINT信号。如果此终端设备是一个控制终端，则将此信号送给前台进程组各进程。

如果IGNBRK和BRKINT都没有设置，但是设置了PARMRK，则BREAK被读作为三个字节序列\377，\0和\0，如果PARMRK也没有设置，则BREAK被读作为单个字符\0。

• BSDLY （c_oflag, SVR4）退格延迟屏蔽，此屏蔽的值是BS0或BS1。

• CCTS_OFLOW （c_cflag, 4.3+BSD）输出的CTS流控制（见习题11.4）。

• CIGNORE （c_cflag, 4.3+BSD）忽略控制标志。

• CLOCAL （c_cflag, POSIX.1）如若设置，则忽略调制解调器状态线。这通常意味着该设备是本地连接的。若此标志未设置，则打开一个终端设备常常会阻塞到调制解调器回应。

• CRDLY （c_oflag, SVR4）回车延迟屏蔽。此屏蔽的值是CR0、CR1、CR2和CR3。

• CREAD （c_cflag, POSIX.1）如若设置，则接收装置被启用，可以接收字符。

• CRTS_IFLOW （c_cflag, 4.3+BSD）输入的RTS流控制（见习题11.4）。

• CSIZE （c_cflag, POSIX.1）此字段是一个屏蔽标志，它指明发送和接收的每个字节的位数。此长度不包括可能有的奇偶校验位。由此屏蔽定义的字段值是 CS5、CS6、CS7和CS8，分别表示每个字节包含5、6、7和8位。

• CSTOPB （c_cflag, POSIX.1）如若设置，则使用两位作为停止位，否则只使用一位作为停止位。

• ECHO （c_lflag, POSIX.1）如若设置，则将输入字符回送到终端设备。在规范方式和非规范方式下都可以回送字符。

• ECHOCTL （c_lflag, SVR4和4.3+BSD）,如若设置并且ECHO也设置，则除ASCII TAB、ASCII NL、START和STOP字符外，其他ASCII控制符（ASCII字符集中的0～037）都被回送为^X，其中，X是相应控制字符代码值加0100所构成的字符。这就意味着ASCII Ctrl-A字符（01）被回送为^A。ASCII DELETE字符（0177）则回送为^?。如若此标志未设置，则ASCII控制字符按其原样回送。如同ECHO标志，在规范方式和非规范方式下此标志对控制字符回送都起作用。

应当了解的是：某些系统回送 EOF字符产生的作用有所不同，其原因是 EOF的典型值是Ctrl-D，而这是ASCII EOT字符，它可能使某些终端挂断。请查看有关手册。

• ECHOE （c_lflag, POSIX.1）如若设置并且ICANON也设置，则ERASE字符从显示中擦除当前行中的最后一个字符。这通常是在终端驱动程序中写三个字符序列：退格，空格，退格实现的。

如若支持WERASE字符，则ECHOE用一个或若干个上述三字符序列擦除前一个字。

如若支持ECHOPRT标志，则在这里所说明的ECHOE动作假定ECHOPRT标志没有设置。

• ECHOK （c_lflag, POSIX.1）如若设置并且ICANON也设置，则KILL字符从显示中擦除当前行，或者输出NL字符（用以强调已擦除整个行）。如若支持ECHOKE标志，则这里的说明假定ECHOKE标志没有设置。

• ECHOKE （c_lflag, SVR4和4.3+BSD）如若设置并且ICANON也设置，则回送KILL字符的方式是擦去行中的每一个字符。擦除每个字符的方法则由 ECHOE和ECHOPRT标志选择。

• ECHONL （c_lflag, POSIX.1）如若设置并且ICANON也设置，即使没有设置ECHO也回送NL字符。

• ECHOPRT （c_lflag, SVR4和4.3+BSD）如若设置并且ICANON和ECHO也都设置，则ERASE字符（以及WERASE字符，若受到支持）使所有正被擦除的字符按它们被擦除的方式打印。在硬拷贝终端上这常常是有用的，这样可以确切地看到哪些字符正被擦去。

• FFDLY （c_oflag, SVR4）换页延迟屏蔽。此屏蔽标志值是FF0或FF1。

• FLUSHO （c_lflag, SVR4和4.3+BSD）如若设置，则刷清输出。当键入DISCARD字符时设置此标志，当键入另一个 DISCARD字符时，此标志被清除。设置或清除此终端标志也可设置或清除此条件。

• HUPCL （c_cflag, POSIX.1）如若设置，则当最后一个进程关闭此设备时，调制解调器控制线降至低电平（也就是调制解调器的连接断开）。

• ICANON （c_lflag, POSIX.1）如若设置，则按规范方式工作（见11.10节）。这使下列字符起作用：EOF、EOL、EOL2、ERASE、KILL、REPRINT、STATUS和WERASE。输入字符被装配成行。

如果不以规范方式工作，则读请求直接从输入队列取字符。在至少接到 MIN个字节或已超过TIME值之前，read将不返回。详细情况见11.11节。

• ICRNL （c_iflag, POSIX.1）如若设置并且IGNCR未设置，即将接收到的CR字符转换成一个NL字符。

• IEXTEN （c_lflag, POSIX.1）如若设置，则识别并处理扩充的、实现定义的特殊字符。

• IGNBRK （c_iflag, POSIX.1）在设置时，忽略输入中的BREAK条件。关于BREAK条件是产生信号还是被读作为数据，请见BRKINT。

• IGNCR （c_iflag, POSIX.1）如若设置，忽略接收到的CR字符。若此标志未设置，而设置了ICRNL标志则将接收到的CR字符转换成一个NL字符。

• IGNPAR （c_iflag, POSIX.1）在设置时，忽略带有结构错误（非BREAK）或奇偶错的输入字节。

• IMAXBEL （c_iflag, SVR4和4.3+BSD）当输入队列满时响铃。

• INLCR （c_iflag, POSIX.1）如若设置，则接收到的NL字符转换成CR字符。

• INPCK （c_iflag, POSIX.1）当设置时，使输入奇偶校验起作用。如若未设置 INPCK，则使输入奇偶校验不起作用。

奇偶“产生和检测”和“输入奇偶性检验”是不同的两件事。奇偶位的产生和检测是由PARENB标志控制的。设置该标志后使串行界面的设备驱动程序对输出字符产生奇偶位，对输入字符则验证其奇偶性。标志PARODD决定该奇偶性应当是奇还是偶。如果一个其奇偶性为错的字符已经来到，则检查INPCK标志的状态。若此标志已设置，则检查 IGNPAR标志（以决定是否应忽略带奇偶错的输入字节），若不应忽略此输入字节，则检查 PARMRK标志以决定向读进程应传送那种字符。

• ISIG （c_lflag, POSIX.1）如若设置，则判别输入字符是否是要产生终端信号的特殊字符（INTR，QUIT，SUSP和DSUSP），若是，则产生相应信号。

• ISTRIP （c_iflag, POSIX.1）当设置时，有效输入字节被剥离为7位。当此标志未设置时，则保留全部8位。

• IUCLC （c_iflag, SVR4）将输入的大写字符映射为小写字符。

• IXANY （c_iflag, SVR4和4.3+BSD）使任一字符都能重新起动输出。

• IXOFF （c_iflag, POSIX.1）如若设置，则使起动-停止输入控制起作用。当终端驱动程序发现输入队列将要填满时，输出一个 STOP字符。此字符应当由发送数据的设备识别，并使该设备暂停。此后，当已对输入队列中的字符进行了处理后，该终端驱动程序将输出一个START字符，使该设备恢复发送数据。

• IXON （c_iflag, POSIX.1）如若设置，则使起动-停止输出控制起作用。当终端驱动程序接收到一个STOP字符时，输出暂停。在输出暂停时，下一个 START字符恢复输出。如若未设置此标志，则START和STOP字符由进程读作为一般字符。

• MDMBUF （c_cflag, 4.3+BSD）按照调制解调器的载波标志进行输出流控制。

• NLDLY （c_oflag, SVR4）新行延迟屏蔽。此屏蔽的值是NL0和NL1。

• NOFLSH （c_lflag, POSIX.1）按系统默认，当终端驱动程序产生SIGINT和SIGQUIT信号时，输入、出队列都被刷新。另外，当它产生 SIGSUSP信号时，输入队列被刷新。如若设置了NOFLSH标志，则在这些信号产生时，不对输入、出队列进行刷新。

• NOKERNINFO （c_lflag, 4.3+BSD）当设置时，此标志阻止STATUS字符使前台进程组的状态信息显示在终端上。但是不论本标志是否设置， STATUS字符使SIGINFO信号送至前台

进程组中的所有进程。

• OCRNL （c_oflag, SVR4）如若设置，将输出的CR字符映照为NL。

• OFDEL （c_oflag, SVR4）如若设置，则输出填充字符是ASCII DEL，否则它是ASCII NUL，见OFILL标志。

• OFILL （c_oflag, SVR4）如若设置，则为实现延迟，发送填充字符（ ASCII DEL或ASCII NUL，见OFDEL标志），而不使用时间延迟。见 6个延迟屏蔽： BSDLY，CRDLY，FFDLY，NLDLY，TABDLY以及VTDLY。

• OLCUC （c_oflag, SVR4）如若设置，将小写字符映射为大写。

• ONLCR （c_oflag, SVR4和4.3+BSD）如若设置，将输出的NL字符映照为CR-NL。

• ONLRET （c_oflag, SVR4）如若设置，则输出的NL字符将执行回车功能。

• ONOCR （c_oflag, SVR4）如若设置，则在0列不输出CR。

• ONOEOT （c_oflag, 4.3+BSD）如若设置，则在输出中删除EOT字符（^D）。在将Ctrl-D解释为挂断的终端上这可能是需要的。

• OPOST （c_oflag, POSIX.1）如若设置，则进行实现定义的输出处理。关于c_oflag字的各种实现定义标志，见表11-1。

• OXTABS （c_oflag, 4.3+BSD）如若设置，制表符在输出中被扩展为空格。这与将水平制表延迟（TABDLY）设置为XTABS或TAB3产生同样效果。

• PARENB （c_cflag, POSIX.1）如若设置，则对输出字符产生奇偶位，对输入字符则执行奇偶性检验。若 PARODD已设置，则奇偶校验是奇校验，否则是偶校验。也见 INPCK、IGNPAR和PARMRK标志部分。

• PARMRK （c_iflag, POSIX.1），当设置时，并且 IGNPAR未设置，则结构性错（非BREAK）和奇偶错的字节由进程读作为三个字符序列 \377,\0和X，其中X是接收到的具有错误的字节。如若ISTRIP未设置，则一个有效的\377被传送给进程时为\377，\377。如若IGNPAR和PARMRK都未设置，则结构性错和奇偶错的字节都被读作为一个字符\0。

• PARODD （c_cflag, POSIX.1）如若设置，则输出和输入字符的奇偶性都是奇，否则为偶。注意，PARENB标志控制奇偶性的产生和检测。

• PENDIN （c_lflag, SVR4和4.3+BSD）如若设置，则在下一个字符输入时，尚未读的任何输入都由系统重新打印。这一动作与键入REPRINT字符时的作用相类似。

• TABDLY （c_oflag, SVR4）水平制表延迟屏蔽。此屏蔽的值是TAB0、TAB1、TAB2或TAB3。

XTABS的值等于TAB3。此值使系统将制表符扩展成空格。系统假定制表符所扩展的空格数到屏幕上最近一个8的倍数处为止。不能更改此假定。

• TOSTOP （c_lflag,POSIX.1）如若设置，并且该实现支持作业控制，则将信号SIGTTOU送到试图与控制终端的一个后台进程的进程组。按默认，此信号暂停该进程组中所有进程。如果写控制终端的进程忽略或阻塞此信号，则终端驱动程序不产生此信号。

• VTDLY （c_oflag, SVR4）垂直制表延迟屏蔽。此屏蔽的值是VT0或VT1。

• XCASE （c_lflag, SVR4）如若设置，并且ICANON也设置，则认为终端是大写终端，所以输入都变换为小写。为了输入一个大写字符，在其前加一个 \。与之类似，输出一个大写字符也在其前加一个\（这一标志已经过时，现在几乎所有终端都支持大、小写字符）。

11.6 stty命令

上节说明的所有选择项，在程序中都可用 tcgetattr和tcsetattr函数（见11.4节）进行检查和

更改。在命令行中则用stty(1)命令进行检查和更改。stty(1)命令是表11-2中所列的头6个函数的界面。如果以-a选择项执行此命令，则显示终端的所有选择项：

```
$ stty -a
speed 9600 baud; 34 rows; 80 columns;
lflags: icanon isig iexten echo echoe echok echoke -echonl echoctl
        -echoprt -altwerase -noflsh -tostop -mdmbuf -flusho -pendin
        -nokerninfo -extproc
iflags: istrip icrnl -inlcr -igncr ixon -ixoff ixany imaxbel -ignbrk
        brkint -inpck -ignpar -parmrk
oflags: opost onlcr -oxtabs
cflags: cread cs7 parenb -parodd hupcl -clocal -cstopb -crtscts
cchars: discard = ^O; dsusp = ^Y; eof = ^D; eol = <undef>;
        eol2 = <undef>; erase = ^H; intr = ^?; kill = ^U; lnext = ^V;
        quit = ^\; reprint = ^R; start = ^Q; status = ^T; stop = ^S;
        susp = ^Z; werase = ^W;
```

若在选择项名前有一个连字符，表示该选择项禁用。最后四行显示各终端特殊字符的设置（见11.3节）。第1行显示当前终端窗口的行数和列数，11.12节将对此进行讨论。

因为stty命令是一条用户命令，而不是一个操作系统函数，所以它由POSIX.2说明。

> 系统V的stty在标准输入进行操作，将输出写到标准输出上。V7和BSD系统则在标准输出上进行操作，将输出写到标准出错文件上。POSIX.2的最近草案采用系统V的方法，4.3+BSD也这样做。
>
> V7的stty手册页只有1页，SVR4版本的stty手册页则有6页。终端驱动程序趋向于使用愈来愈多的选择项。

11.7 波特率函数

波特率（baud rate）是一个历史沿用的术语，现在它指的是“位/每秒”。虽然大多数终端设备对输入和输出使用同一波特率，但是只要硬件许可，可以将它们设置为两个不同值。

```
#include <termios.h>

speed_t cfgetispeed(const struct termios *termptr);
speed_t cfgetospeed(const struct termios *termptr);
```

两个函数返回：波特率值

```
int cfsetispeed(struct termios *termptr, speed_t speed);
int cfsetospeed(struct termios *termptr, speed_t speed);
```

两个函数返回：若成功为0，出错为-1。

两个cfget函数的返回值，以及两个cfset函数的*speed*参数都是下列常数之一：B50、B75、B110、B134、B150、B200、B300、B600、B1200、B1800、B2400、B4800、B9600、B19200或B38400。常数B0表示“挂断”。在调用tcsetattr时将输出波特率指定为B0，则调制解调器的控制线就不再起作用。

使用这些函数时，应当理解输入、输出波特率是存放在图11-3所示的设备termios结构中的。在调用任一cfget函数之前，先要用tcgetattr获得设备的termios结构。与此类似，在调用任一

cfset函数后，应将波特率设置到termios结构中。为使这种更改影响到设备，应当调用 tcsetattr函数。

如果所设置的波特率有错，则在调用tcsetattr之前，不会发现这种错误。

11.8 行控制函数

下列四个函数提供了终端设备的行控制能力。其中，参数 *filedes*引用一个终端设备，否则出错返回，errno设置为ENOTTY。

```
#include <termios.h>
int tcdrain(int filedes);
int tcflow(int filedes, int action);
int tcflush(int filedes, int queue);
int tcsendbreak(int filedes, int duration);
```

四个函数返回：若成功则为0，若出错则为-1

tcdrain函数等待所有输出都被发送。 tcflow用于对输入和输出流控制进行控制。 *action*参数应当是下列四个值之一。

- TCOOFF 输出被挂起。
- TCOON 以前被挂起的输出被重新起动。
- TCIOFF 系统发送一个STOP字符。这将使终端设备暂停发送数据。
- TCION 系统发送一个START字符。这将使终端恢复发送数据。

tcflush函数刷清（抛弃）输入缓存（终端驱动程序已接收到，但用户程序尚未读）或输出缓存（用户程序已经写，但尚未发送）。*queue*参数应当是下列三个常数之一：

- TCIFLUSH 刷清输入队列。
- TCOFLUSH 刷清输出队列。
- TCIOFLUSH 刷清输入、输出队列。

tcsendbreak函数在一个指定的时间区间内发送连续的 0位流。若*duration*参数为0，则此种发送延续0.25~0.5秒之间。POSIX.1说明若*duration*非0，则发送时间依赖于实现。

> SVR4 SVID说明若*duration*非0，则不发送0位。但是，SVR4手册页中说，若*duration*非0，则tcsendbreak的行为与tcdrain一样。另一个系统手册页则说，若*duration*非0，则传送0位的时间是*duration*×N，其中N在0.25~0.5秒之间。从中可见，如何处理这种条件还没有统一样式。

11.9 终端标识

历史沿袭至今，在大多数 UNIX系统中，控制终端的名字是 /dev/tty。POSIX.1提供了一个运行时函数，可被调用来决定控制终端的名字。

```
#include <stdio.h>
```

```
char * ctermid(char *ptr);
```

返回：见下

如果*ptr*是非空，则它被认为是一个指针，指向长度至少为 L_ctermid字节的数组，进程的控制终端名存放在该数组中。常数 L_ctermid定义在<stdio.h>中。若*ptr*是一个空指针，则该函数为数组（通常作为静态变量）分配空间。同样，进程的控制终端名存放在该数组中。

在这两种情况中，该数组的起始地址被作为函数值返回。因为大多数 UNIX系统都使用/dev/tty作为控制终端名，所以此函数的主要作用是帮助提高向其他操作系统的可移植性。

实例——ctermid函数

程序11-3是POSIX.1 ctermid函数的一个实现。

程序11-3 POSIX.1 ctermid函数的实现

```
#include    <stdio.h>
#include    <string.h>

static char ctermid_name[L_ctermid];

char *
ctermid(char *str)
{
    if (str == NULL)
        str = ctermid_name;
    return(strcpy(str, "/dev/tty"));    /* strcpy() returns str */
}
```

另外两个与终端标识有关的函数是 isatty和ttyname。如果文件描述符引用一个终端设备，则isatty返回真，而ttyname则返回在该文件描述符上打开的终端设备的路径名。

```
#include <unistd.h>

int isatty(int filedes);
```

返回：若为终端设备则为1（真），否则为0（假）

```
char *ttyname(int filedes);
```

返回：指向终端路径名的指针，若出错则为 NULL

实例——isatty函数

如程序11-4所示，isatty函数很容易实现。其中只使用了一个终端专用的函数 tcgetattr，并取其返回值。

程序11-4 POSIX.1 isatty函数的实现

```
#include    <termios.h>

int
isatty(int fd)
{
    struct termios  term;
```

```
    return(tcgetattr(fd, &term) != -1); /* true if no error (is a tty) */
}
```

程序11-5 测试isatty函数

```
#include    "ourhdr.h"

int
main(void)
{
    printf("fd 0: %s\n", isatty(0) ? "tty" : "not a tty");
    printf("fd 1: %s\n", isatty(1) ? "tty" : "not a tty");
    printf("fd 2: %s\n", isatty(2) ? "tty" : "not a tty");
    exit(0);
}
```

用程序11-5测试isatty函数，得到：

```
$ a.out
fd 0: tty
fd 1: tty
fd 2: tty
$ a.out </etc/passwd 2>/dev/null
fd 0: not a tty
fd 1: tty
fd 2: not a tty
```

实例——ttyname函数

ttyname函数（见程序11-6）稍长一点，因为它要搜索所有设备表项，寻找匹配项。其方法是读/dev目录，寻找具有相同设备号和i节点编号的表项。回忆4.23节，每个文件系统有一个唯一的设备号（stat结构中的st_dev字段，见4.2节），文件系统中的每个目录项有一个唯一的i节点号（stat结构中的st_ino字段）。在此函数中假定当找到一个匹配的设备号和匹配的i节点号时，就找到了所希望的目录项。也可验证这两个表项与st_rdev字段（终端设备的主、次设备号）相匹配，以及该目录项是一个字符特殊文件。但是，因为已经验证了文件描述符参数是一个终端设备以及一个字符特殊设备，而且在UNIX系统中，匹配的设备号和i节点号是唯一的，所以不再需要作另外的比较。

用程序11-7测试这一实现。运行程序11-7得到：

```
$ a.out </dev/console 2> /dev/null
fd 0: /dev/console
fd 1: /dev/ttyp3
fd 2: not a tty
```

程序11-6 POSIX.1 ttyname函数的实现

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <dirent.h>
#include    <limits.h>
#include    <string.h>
#include    <termios.h>
#include    <unistd.h>

#define DEV     "/dev/"         /* device directory */
```

```
#define DEVLEN  sizeof(DEV)-1   /* sizeof includes null at end */

char *
ttyname(int fd)
{
    struct stat     fdstat, devstat;
    DIR             *dp;
    struct dirent   *dirp;
    static char     pathname[_POSIX_PATH_MAX + 1];
    char            *rval;

    if (isatty(fd) == 0)
        return(NULL);
    if (fstat(fd, &fdstat) < 0)
        return(NULL);
    if (S_ISCHR(fdstat.st_mode) == 0)
        return(NULL);

    strcpy(pathname, DEV);
    if ( (dp = opendir(DEV)) == NULL)
        return(NULL);
    rval = NULL;
    while ( (dirp = readdir(dp)) != NULL) {
        if (dirp->d_ino != fdstat.st_ino)
            continue;       /* fast test to skip most entries */

        strncpy(pathname + DEVLEN, dirp->d_name, _POSIX_PATH_MAX - DEVLEN);
        if (stat(pathname, &devstat) < 0)
            continue;
        if (devstat.st_ino == fdstat.st_ino &&
            devstat.st_dev == fdstat.st_dev) {  /* found a match */
                rval = pathname;
                break;
        }
    }
    closedir(dp);
    return(rval);
}
```

程序11-7 测试ttyname函数

```
#include    "ourhdr.h"

int
main(void)
{
    printf("fd 0: %s\n", isatty(0) ? ttyname(0) : "not a tty");
    printf("fd 1: %s\n", isatty(1) ? ttyname(1) : "not a tty");
    printf("fd 2: %s\n", isatty(2) ? ttyname(2) : "not a tty");
    exit(0);
}
```

11.10 规范方式

规范方式很简单——发一个读请求，当一行已经输入后，终端驱动程序即返回。许多条件造成读返回。

- 所要求的字节数已读到时读即返回。无需读一个完整的行。如果读了部分行，那么也不会丢失任何信息——下一次读从前一次读的停止处开始。
- 当读到一个行定界符时，读返回。回忆11.3节，在规范方式中，下列字符被解释为“行

结束”：NL、EOL、EOL2和EOF。另外，在11.5节中也曾说明，如若已设置ICRNL，但未设置IGNCR，则CR字符的作用与NL字符一样，所以它也终止一行。

在这五个行定界符中，其中只有一个EOF符在终端驱动程序对其进行处理后即被删除。其他四个字符则作为该行的最后一个字符返回调用者。

• 如果捕捉到信号而且该函数并不自动再起动（见10.5节），则读也返回。

实例——getpass函数

下面说明getpass函数，它读入用户在终端上键入的口令。此函数由 UNIX login(1)和crypt(1)程序调用。为了读口令，该函数必须禁止回送，但仍可使终端以规范方式进行工作，因为用户在键入口令后，一定要键入回车，这样也就构成了一个完整行。程序 11-8是一个典型的UNIX实现。

程序11-8 getpass函数的实现

```
#include    <signal.h>
#include    <stdio.h>
#include    <termios.h>

#define MAX_PASS_LEN    8       /* max #chars for user to enter */

char *
getpass(const char *prompt)
{
    static char     buf[MAX_PASS_LEN + 1];  /* null byte at end */
    char            *ptr;
    sigset_t        sig, sigsave;
    struct termios  term, termsave;
    FILE            *fp;
    int             c;

    if ( (fp = fopen(ctermid(NULL), "r+")) == NULL)
        return(NULL);
    setbuf(fp, NULL);

    sigemptyset(&sig);  /* block SIGINT & SIGTSTP, save signal mask */
    sigaddset(&sig, SIGINT);
    sigaddset(&sig, SIGTSTP);
    sigprocmask(SIG_BLOCK, &sig, &sigsave);

    tcgetattr(fileno(fp), &termsave);   /* save tty state */
    term = termsave;                    /* structure copy */
    term.c_lflag &= ~(ECHO | ECHOE | ECHOK | ECHONL);
    tcsetattr(fileno(fp), TCSAFLUSH, &term);

    fputs(prompt, fp);

    ptr = buf;
    while ( (c = getc(fp)) != EOF && c != '\n') {
        if (ptr < &buf[MAX_PASS_LEN])
            *ptr++ = c;
    }
    *ptr = 0;           /* null terminate */
    putc('\n', fp);     /* we echo a newline */

                        /* restore tty state */
    tcsetattr(fileno(fp), TCSAFLUSH, &termsave);

                        /* restore signal mask */
    sigprocmask(SIG_SETMASK, &sigsave, NULL);
```

```
    fclose(fp);          /* done with /dev/tty */

    return(buf);
}
```

在此例中，有很多方面应当考虑：

• 调用函数ctermid打开控制终端，而不是直接将/dev/tty写在程序中。

• 只是读、写控制终端，如果不能以读、写方式打开此设备则出错返回。在有些系统中也使用一些其他约定。在4.3+BSD中，如果不能以读、写方式打开控制终端，则 getpass从标准输入读，写到标准出错文件中。SVR4则总是写到标准出错文件中，但只从控制终端读。

• 阻塞两个信号SIGINT和SIGTSTP。如果不这样做，则在输入 INTR字符时就会使程序终止，并使终端仍处于禁止回送状态。与此相类似，输入 SUSP字符时将使程序暂停，并且在禁止回送状态下返回到shell。在禁止回送时，选择了阻塞这两个信号。在读口令期间如果发生了这两个信号，则它们被保持，直到getpass返回前才解除对它们的阻塞。也有其他方法来处理这些信号。某些getpass版本忽略SIGINT（保存它以前的动作），在返回前则将其动作恢复为以前的值。这就意味着在该信号被忽略期间所发生的这种信号都丢失。其他版本捕捉 SIGINT（保存它以前的动作），如果捕捉到此信号，则在复置终端状态和信号动作后，用 kill函数发送此信号。没有一个getpass版本捕捉、忽略或阻塞 SIGQUIT，所以键入QUIT字符就会使程序夭折，并且极可能终端仍处于禁止回送状态。

• 要了解某些shell，例如KornShell在以交互方式读输入时都使终端处于回送状态。这些shell是提供命令行编辑的shell，因此在每次输入一条交互命令时都处理终端状态。所以如果在这种shell下调用此程序，并且用 QUIT字符使其夭折，则这种 shell可以恢复回送状态。不提供命令行编辑的shell，例如Bourne shell和C shell将使程序夭折，并使终端仍处于不回送状态。如果对终端做了这种操作，则stty命令能使终端回复到回送状态。

• 使用标准I/O读、写控制终端。我们特地将流设置为不带缓存的，否则在流的读、写之间可能会有某些相互作用（这样就需调用 fflush）。也可使用不带缓存的I/O（见第3章），但是在这种情况下就要用read来实现getc。

• 最多只可取8个字符作为口令。输入的多余字符则被忽略。

程序11-9调用getpass并且打印我们所输入的。这只是为了验证 ERASE和KILL字符在正常工作（如同它们在规范方式下应该的那样）。

程序11-9 调用getpass函数

```
#include    "ourhdr.h"

char    *getpass(const char *);

int
main(void)
{
    char    *ptr;

    if ( (ptr = getpass("Enter password:")) == NULL)
        err_sys("getpass error");
    printf("password: %s\n", ptr);

    /* now use password (probably encrypt it) ... */

    while (*ptr != 0)
        *ptr++ = 0;      /* zero it out when we're done with it */

    exit(0);
}
```

调用getpass函数的程序完成后，为了安全起见，应清除存放过用户键入的文本口令的存储区。如果该程序会产生其他用户能读的 core文件（回忆10.2节，core的系统默认许可权使每个用户都能读它），或者如果某个其他进程能够设法读该进程的存储空间，则它们就能读到口令。

11.11 非规范方式

将termios结构中c_lflag字段的ICANON标志关闭就使终端处于非规范方式。在非规范方式中，输入数据不装配成行，不处理下列特殊字符： ERASE、KILL、EOF、NL、EOL、EOL2、CR、REPRINT、STATUS和WERASE。

如前所述，规范方式很容易——系统每次返回一行。但在非规范方式下，系统怎样才能知道在什么时候将数据返回给我们呢?如果它一次返回一个字节，那么系统开销就很大。（回忆表3-1，从中可以看到每次读一个字节的开销会多大。每次使返回的数据加倍，就使系统调用的开销减半。）在起动读数据之前，往往不知道要读多少数据，所以系统不能总是返回多个字节。

解决方法是：当已读了指定量的数据后，或者已经过了给定量的时间后，即通知系统返回。这种技术使用了termios结构中c_cc数组的两个变量： MIN和TIME。c_cc数组中的这两个元素的下标名为： VMIN和VTIME。

MIN说明一个read返回前的最小字节数。TIME说明等待数据到达的分秒数（秒的1/10为分秒）。有下列四种情形：

情形A：MIN>0,TIME>0

TIME说明一个字节间的计时器，在接到第一个字节时才起动它。在该计时器超时之前，若已接到MIN个字节，则read返回MIN个字节。如果在接到MIN个字节之前，该计时器已超时，则read返回已接收到的字节（因为只有在接到第一个字节时才起动，所以在计时器超时时，至少返回1个字节）。在这种情形中，在接到第一个字节之前，调用者阻塞。如果在调用 read时数据已经可用，则这如同在read后，数据立即被接收到一样。

情形B：MIN>0,TIME==0

已经接到了MIN个字节时， read才返回。这可以造成read无限期的阻塞。

情形C：MIN==0，TIME>0

TIME指定了一个调用read时起动的读计时器。（与情形A相比较，两者是不同的）。在接到1个字节或者该计时器超时时， read即返回。如果是计时器超时，则read返回0。

情形D：MIN==0，TIME==0

如果有数据可用，则read最多返回所要求的字节数。如果无数据可用，则read立即返回0。

在所有这些情形中， MIN只是最小值。如果程序要求的数据多于MIN个字节，那么它可能能接收到所要求的字节数。这也适用于MIN==0的情形A和B。

表11-4列出了非规范方式下的四种不同情形。在表中， *nbytes*是read的第三个参数（返回的最大字节数）。

> POSIX.1允许下标VMIE和VTIME的值分别与VEOF和VEOL相同。确实，SVR4就是这样做的。这样就提供了与系统V早期版本的兼容性。问题是从非规范方式转换为规范方式时，必须恢复 VEOF和VEOL，如果不这样做，那么VMIN等于VEOF，并且它已被设置为典型值1，于是文件结束字符就变成Ctrl-A。解决这一问题最简单的方法是：在转入非规范方式时将整个 termios结构保存起来。在以后再转回规范方式时恢复它。

表11-4 非规范输入的四种情形

	MIN > 0	MIN == 0
TIME > 0	**A:** 在计时器超时前，读返回[MIN, *nbytes*]； 若计时器超时，读返回[1, MIN] (TIME=字节间计时器，调用者可能无限阻塞)	**C:** 在计时器超时前，读返回[1, *nbytes*]； 若计时器超时，读返回0 (TIME=读计时器。)
TIME == 0	**B:** 可用时，读返回[MIN, *nbytes*]， (调用者可能无限阻塞。)	**D:** 立即读返回[0, *nbytes*]。

实例

程序11-10定义了函数tty_cbreak和tty_raw，它们将终端分别设置为*cbreak*和原始方式（术语*cbreak*和*raw*来自于V7的终端驱动程序）。tty_reset函数的功能是将终端恢复为以前的工作方式。其中还提供了另外两个函数：tty_atexit, tty_termios。tty_atexit可被登记为终止处理程序，以保证exit恢复终端工作方式。tty_termios则返回一个指向原先的规范方式termios结构的指针。第18章的调制解调器拨号程序中将使用所有这些函数。

程序11-10 将终端方式设置为原始或cbreak方式

```
#include    <termios.h>
#include    <unistd.h>

static struct termios   save_termios;
static int              ttysavefd = -1;
static enum { RESET, RAW, CBREAK }  ttystate = RESET;

int
tty_cbreak(int fd)  /* put terminal into a cbreak mode */
{
    struct termios  buf;

    if (tcgetattr(fd, &save_termios) < 0)
        return(-1);

    buf = save_termios; /* structure copy */

    buf.c_lflag &= ~(ECHO | ICANON);
                    /* echo off, canonical mode off */

    buf.c_cc[VMIN] = 1; /* Case B: 1 byte at a time, no timer */
    buf.c_cc[VTIME] = 0;

    if (tcsetattr(fd, TCSAFLUSH, &buf) < 0)
        return(-1);
    ttystate = CBREAK;
    ttysavefd = fd;
    return(0);
}

int
tty_raw(int fd)     /* put terminal into a raw mode */
{
    struct termios  buf;

    if (tcgetattr(fd, &save_termios) < 0)
        return(-1);
```

```
    buf = save_termios; /* structure copy */
    buf.c_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);
                    /* echo off, canonical mode off, extended input
                       processing off, signal chars off */
    buf.c_iflag &= ~(BRKINT | ICRNL | INPCK | ISTRIP | IXON);
                    /* no SIGINT on BREAK, CR-to-NL off, input parity
                       check off, don't strip 8th bit on input,
                       output flow control off */
    buf.c_cflag &= ~(CSIZE | PARENB);
                    /* clear size bits, parity checking off */
    buf.c_cflag |= CS8;
                    /* set 8 bits/char */
    buf.c_oflag &= ~(OPOST);
                    /* output processing off */
    buf.c_cc[VMIN] = 1; /* Case B: 1 byte at a time, no timer */
    buf.c_cc[VTIME] = 0;
    if (tcsetattr(fd, TCSAFLUSH, &buf) < 0)
        return(-1);
    ttystate = RAW;
    ttysavefd = fd;
    return(0);
}

int
tty_reset(int fd)       /* restore terminal's mode */
{
    if (ttystate != CBREAK && ttystate != RAW)
        return(0);
    if (tcsetattr(fd, TCSAFLUSH, &save_termios) < 0)
        return(-1);
    ttystate = RESET;
    return(0);
}

void
tty_atexit(void)        /* can be set up by atexit(tty_atexit) */
{
    if (ttysavefd >= 0)
        tty_reset(ttysavefd);
}

struct termios *
tty_termios(void)       /* let caller see original tty state */
{
    return(&save_termios);
}
```

对cbreak方式的定义是：

• 非规范方式。如本节开始处所述，这种方式不对某些输入特殊字符进行处理。这种方式仍对信号进行处理，所以用户可以键入任一终端产生的信号。调用者应当捕捉这些信号，否则这种信号就可能终止程序，并且终端将仍处于cbreak方式。

作为一般规则，在编写更改终端方式的程序时，应当捕捉大多数信号，以便在程序终止前恢复终端方式。

• 关闭回送（ECHO）标志。

• 每次输入一个字节。为此将MIN设置为1，将TIME设置为0。这是表11-4中的情形B。至少有一个字节可用时，read再返回。

对原始方式的定义是：

• 非规范方式。另外，还关闭了对信号产生字符（ ISIG）和扩充输入字符的处理（IEXTEN）。关闭BRKINT，这样就使BREAK字符不再产生信号。

• 关闭回送（ECHO）标志。

• 关闭ICRNL、INPCK、ISTRIP和IXON标志。于是：不再将输入的 CR字符变换为NL（ICRNL）、使输入奇偶校验不起作用（INPCK）、不再剥离输入字节的第8位（ISTRIP）、不进行输出流控制（IXON）。

• 8位字符（CS8），不产生奇偶位，不进行奇偶性检测（PARENB）。

• 禁止所有输出处理（OPOST）。

• 每次输入一个字节（MIN=1，TIME=0）。

程序11-11测试原始和cbreak方式。运行程序11-11可以观察这两种终端工作方式的工作情况。

```
$ a.out
Enter raw mode characters,terminate with DELETE
                                                4
                                                33
                                                133
                                                62
                                                63
                                                60
                                                172
                          键入DELETE
Enter cbreak mode characters,terminate with SIGINT
1                         键入Ctrl-A
10                        键入退格
signal caught             键入中断符
```

程序11-11 测试原始和cbreak工作方式

```
#include    <signal.h>
#include    "ourhdr.h"

static void sig_catch(int);

int
main(void)
{
    int     i;
    char    c;

    if (signal(SIGINT, sig_catch) == SIG_ERR)   /* catch signals */
        err_sys("signal(SIGINT) error");
    if (signal(SIGQUIT, sig_catch) == SIG_ERR)
        err_sys("signal(SIGQUIT) error");
    if (signal(SIGTERM, sig_catch) == SIG_ERR)
        err_sys("signal(SIGTERM) error");

    if (tty_raw(STDIN_FILENO) < 0)
        err_sys("tty_raw error");
    printf("Enter raw mode characters, terminate with DELETE\n");
    while ( (i = read(STDIN_FILENO, &c, 1)) == 1) {
        if ((c &= 255) == 0177)     /* 0177 = ASCII DELETE */
            break;
        printf("%o\n", c);
    }
    if (tty_reset(STDIN_FILENO) < 0)
        err_sys("tty_reset error");
    if (i <= 0)
        err_sys("read error");
```

```
        if (tty_cbreak(STDIN_FILENO) < 0)
            err_sys("tty_raw error");
        printf("\nEnter cbreak mode characters, terminate with SIGINT\n");
        while ( (i = read(STDIN_FILENO, &c, 1)) == 1) {
            c &= 255;
            printf("%o\n", c);
        }
        tty_reset(STDIN_FILENO);
        if (i <= 0)
            err_sys("read error");
        exit(0);
    }

    static void
    sig_catch(int signo)
    {
        printf("signal caught\n");
        tty_reset(STDIN_FILENO);
        exit(0);
    }
```

在原始方式中，输入的字符是 Ctrl-D(04)和特殊功能键F7。在所用的终端上，此功能键产生6个字符：*ESC* (033)，〔(0133)，*2*(062)，*3*(063)，*0*(060)和z(0172)。注意，在原始方式下关闭了输出处理（~OPOST ），所以在每个字符后没有得到回车符。另外也要注意的是，在 cbreak 方式下，不对输入特殊字符进行处理（所以对 Ctrl-D、文件结束符和退格等不进行特殊处理），但是对终端产生的信号则进行处理。

11.12 终端的窗口大小

SVR4和伯克利系统都提供了一种功能，用其可以对当前终端窗口的大小进行跟踪，在窗口大小发生变化时，使内核通知前台进程组。内核为每个终端和伪终端保存一个 winsize结构。

```
struct winsize {
  unsigned short  ws_row;     /* rows, in characters */
  unsigned short  ws_col;     /* columns, in characters */
  unsigned short  ws_xpixel;  /* horizontal size, pixels (not used) */
  unsigned short  ws_ypixel;  /* vertical size, pixels (not used) */
};
```

此结构的作用是：

(1) 用ioctl（见3.14节）的TIOCGWINSZ命令可以取此结构的当前值。

(2) 用ioctl的TIOCSWINSZ命令可以将此结构的新值存放到内核中。如果此新值与存放在内核中的当前值不同，则向前台进程组发送 SIGWINCH信号。（注意，从表10-1中可以看出，此信号的系统默认动作是忽略。）

(3) 除了存放此结构的当前值以及在此值改变时产生一个信号以外，内核对该结构不进行任何其他操作。对结构中的值进行解释完全是应用程序的工作。

提供这种功能的目的是，当窗口大小发生变化时通知应用程序（例如 vi编辑程序）。应用程序接到此信号后，它可以取得窗口大小的新值，然后重绘屏幕。

实例

程序11-12打印当前窗口大小，然后睡眠。每次窗口大小改变时，就捕捉到 SIGWINCH信号，然后打印新的窗口大小。必须用一个信号终止此程序。

程序11-12 打印窗口大小

```
#include    <signal.h>
#include    <termios.h>
#ifndef TIOCGWINSZ
#include    <sys/ioctl.h>   /* 4.3+BSD requires this too */
#endif
#include    "ourhdr.h"

static void pr_winsize(int), sig_winch(int);

int
main(void)
{
    if (isatty(STDIN_FILENO) == 0)
        exit(1);

    if (signal(SIGWINCH, sig_winch) == SIG_ERR)
        err_sys("signal error");

    pr_winsize(STDIN_FILENO);   /* print initial size */
    for ( ; ; )                 /* and sleep forever */
        pause();
}

static void
pr_winsize(int fd)
{
    struct winsize  size;

    if (ioctl(fd, TIOCGWINSZ, (char *) &size) < 0)
        err_sys("TIOCGWINSZ error");
    printf("%d rows, %d columns\n", size.ws_row, size.ws_col);
}

static void
sig_winch(int signo)
{
    printf("SIGWINCH received\n");
    pr_winsize(STDIN_FILENO);
    return;
}
```

在一个带窗口的终端上运行此程序得到：

```
$ a.out
35 rows, 80 columns          起始长度
SIGWINCH received            更改窗口大小：捕捉到信号
40 rows, 123 columns
SIGWINCH received            再一次
42 rows, 33 columns
^? $                         键入中断以终止
```

11.13 termcap,terminfo和curses

termcap的意思是终端性能（terminal capability），它涉及到文本文件/etc/termcap和一套读此文件的例程。termcap这种技术是在伯克利为了支持vi编辑器而发展起来的。termcap文件包含了对各种终端的说明：终端支持哪些功能（行、列数、是否支持退格等），如何使终端执行某些操作（清屏、将光标移动到指定位置等）。把这些信息从需要编译的程序中取出来并把它们放在易于编辑的文本文件中，这样就使得vi能在很多不同的终端上运行。

然后，支持termcap文件的一套例程也从vi编辑程序中抽取出来，放在一个单独的 curses

（光标）库中。为使这套库可被要进行屏幕处理的任何程序使用，增加了很多功能。

termcap这种技术不是很完善的。当越来越多的终端被加到该数据文件中时，为了找到一个特定的终端就需使用较长的时间扫描此文件。此数据文件也只用两个字符的名字来标识不同的终端属性。这些缺陷导致开发另一种新技术 —— terminfo及与其相关的curses库。在terminfo中，终端说明基本上是文本说明的编译版本，在运行时易于快速定位。 terminfo由SVR2开始使用，此后所有系统V版本都使用它。

> 系统V使用terminfo，而4.3+BSD则使用termcap。

Goodheart〔1991〕对terminfo和curses库进行了详细说明。Strang, Mui和O'Reilly[1991] 则对termcap和terminfo进行了说明。

不论是termcap还是terminfo都致力于本章所述及的问题 —— 更改终端的方式、更改终端特殊字符、处理窗口大小等等。它们所提供的是在各种终端上执行典型操作（清屏、移动光标）的方法。另一方面，在本章所述问题方面 curses能提供更详细的帮助。 curses提供了很多函数，包括：设置原始方式、设置 cbreak方式、打开和关闭回送等等。但是 curses是为字符终端设计的，而当前的趋势则是向以象素为基础的图形终端发展。

11.14 小结

终端有很多特征和选择项，其中大多数都可按需进行改变。本章说明了很多更改终端操作的函数 —— 特殊输入字符和选择标志的函数，介绍了可为终端设备设置的各个终端特殊字符以及很多选择项。

终端的输入方式有两种 —— 规范的（每次一行）和非规范的。本章中包含了若干这两种工作方式的实例，也提供了一些函数，它们在 POSIX.1终端选择项和较早的BSD cbreak及原始方式之间进行变换。本章也说明了如何取用和改变终端的窗口大小。第 17和18章包含了终端I/O的另外一些实例。

习题

11.1 写一个调用tty_raw并且不恢复终端模式就终止的程序。如果系统提供 reset(1)命令（SVR4和4.3+BSD中都提供），使用该命令恢复终端模式。

11.2 c_cflag字段的PARODD标志允许我们设置奇偶校验，而BSD中的tip程序也允许奇偶校验位是0或1，它是如何实现的？

11.3 如果你的系统中stty(1)命令输出MIN和TIME值，做下面的练习。两次登录系统，其中一次登录时打开vi编辑器，在另外一次登录中用 stty命令确定vi设置的MIN和TIME值（终端为非规范模式）。

11.4 随着终端接口的行速度变得越来越快（19 200和38 400在当前是非常一般的），硬件的流控制就越来越重要，它包括取代 XON和XOFF字符的RS-232 RTS和CTS。POSIX.1没有指定硬件流控制。在SVR4和4.3+BSD中，进程是如何开关硬件流控制？