Programming for Humans Terminal Control and Signals

Prof. Seokin Hong

Kyungpook National University

Fall 2018

Objectives

- Ideas and Skills
  - Reading and changing settings of the terminal driver
  - Modes of the terminal driver
  - Nonblocking input
  - Timeouts on user input
  - Introduction to signals: How Ctrl-C works
- System Calls
  - fcntl
  - signal

Device-Specific Programs

- **Device-Specific Programs** : interact with specific devices
  - Ex: Programs to control scanners, record compact disks, operate tape drives, and take digital photographs.

- In this chapter, we explore the ideas and techniques of writing device-specific programs by looking at programs that interact with terminals.

User Programs: A Common Type of Device-Specific Program

- Ex:
  - vi, emacs, pine, more, and many of the games

- These programs adjust settings in the terminal driver to control how keystrokes are handled and output is produced.

- Common concerns of user programs using the terminal include:

  (a) immediate response to keys
  (b) limited input set
  (c) timeout on input
  (d) resistance to Ctrl-C

Contents

- 6.1 Software Tools vs. Device-Specific Programs
- 6.2 Modes of the Terminal Driver
- 6.3 Writing a User Program: play-again.c
- 6.4 Signals
- 6.5 Prepared for Signals: play_again4.c

- A short translation program

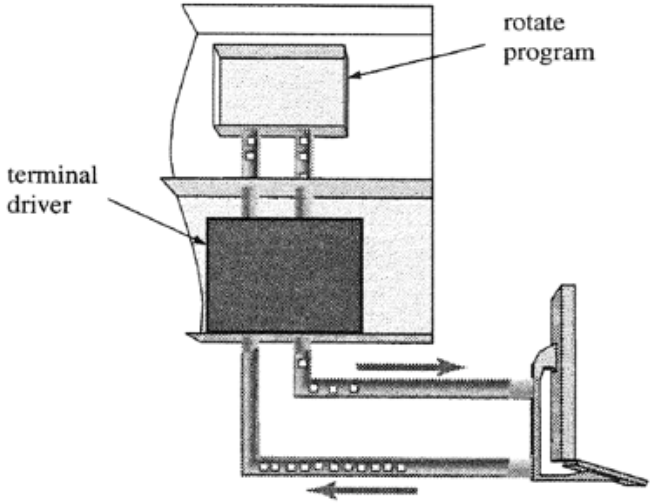
```
/* rotate.c : map a->b, b->c, .. z->a
 *   purpose: useful for showing tty modes
 */

#include   <stdio.h>
#include   <ctype.h>
int main()
{
    int c;
    while ( ( c=getchar() ) != EOF ){
        if ( c == 'z' )
            c = 'a';
        else if (islower(c))
            c++;
        putchar(c);
    }
}
```

Canonical Mode: Buffering and Editing

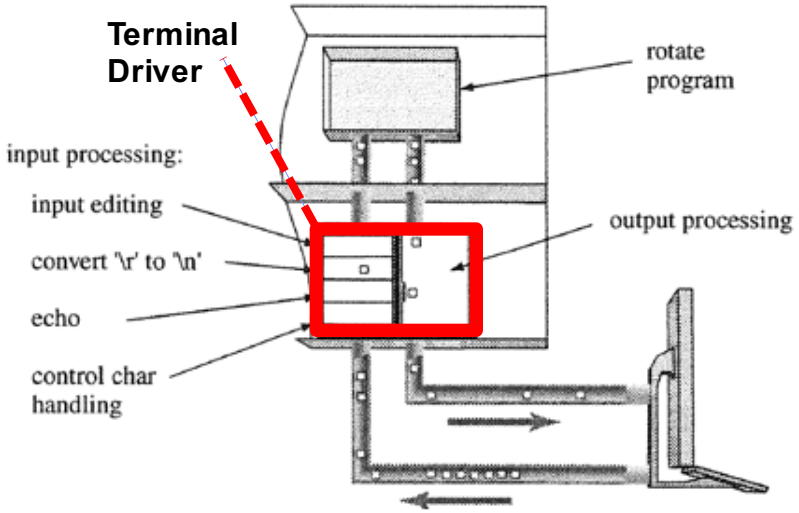
- Run the program using the default settings:

```
$ cc rotate.c -o rotate
$ ./rotate
abx<-cd
bcde
efgCtrl-C
$
```

back space



- The rotate program does none of these operation:
  buffering, echoing, editing, control key processing are all done by the terminal driver



Noncanonical Processing

```
$ stty -icanon ; ./rotate
abbcxy^?cdde
effggh^C
$ stty icanon
```

Turning off canonical mode processing in the driver

Turning off canonical mode processing and echo mode in the driver

```
$ stty -icanon -echo ; ./rotate
bcy^?de
fgh
$ stty icanon echo
```

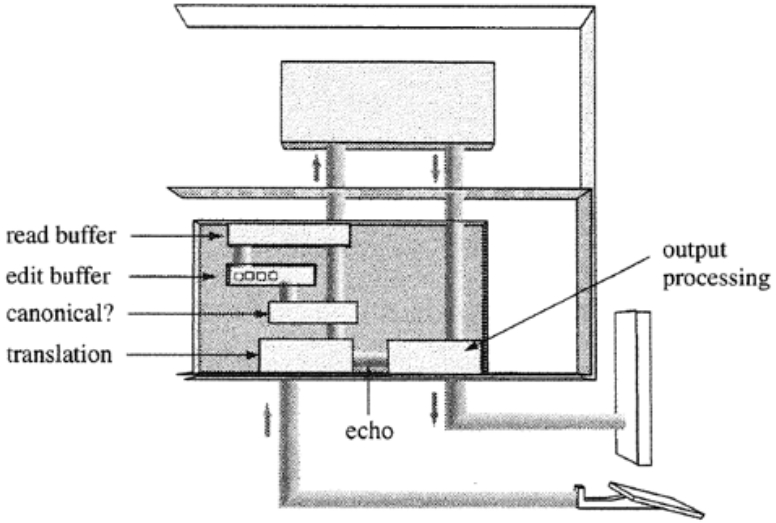
Output comes only from the program

(Note: You won't see this. Why?)

In **canonical** input processing **mode**, **terminal** input is processed in lines terminated by newline ('\n'), EOF, or EOL characters.

Summary of Terminal Modes

- canonical mode : cooked mode
  - **terminal** input is processed in lines terminated by newline ('\n'), EOF, or EOL characters.
  - Buffering and editing is turned on.
- noncanonical mode : crmode
  - Buffering and editing is turned off
  - But, terminal driver still does specific processing :
    ^C, \n, \r
- non-anything mode : raw mode
  - When all processing is turned off, the driver passes input directly to the program;



Contents

- 6.1 Software Tools vs. Device-Specific Programs
- 6.2 Modes of the Terminal Driver
- 6.3 Writing a User Program: play-again.c
- 6.4 Signals
- 6.5 Prepared for Signals: play_again4.c

A shell script for a bank machine(ATM)

```
#!/bin/sh
#
# atm.sh - a wrapper for two programs
#

while true
do
     do_a_transaction      # run a program
     if play_again         # run our program
     then
          continue         # if "y" loop back
     fi
     break                 # if "n" break
done
```

does the work of the ATM

obtains a yes or no answer from the user

play_again.c

- It obtains a yes or no answer from the user

- The logic:

```
prompt user with question
accept input
if "y", return 0
if "n", return 1
```

Ex: play_again0.c

- It obtains a yes or no answer from the user
- The logic:

```
prompt user with question
accept input
if "y", return 0
if "n", return 1
```

```
/* play_again0.c
 *      purpose: ask if user wants another transaction
 *       method: ask a question, wait for yes/no answer
 *      returns: 0=>yes, 1=>no
 *       better: eliminate need to press return
 */
#include        <stdio.h>

#define QUESTION        "Do you want another transaction"
int get_response( char * );
int main(void)
{
        int     response;

        response = get_response(QUESTION);      /* get some answer      */

        return response;
}
```

Ex: play_again0.c

```
int get_response(char *question)
/*
 * purpose: ask a question and wait for a y/n answer
 *  method: use getchar and ignore non y/n answers
 * returns: 0=>yes, 1=>no
 */
{
        printf("%s (y/n)?", question);
        while(1){
                switch( getchar() ){
                        case 'y':
                        case 'Y': return 0;
                        case 'n':
                        case 'N':
                        case EOF: return 1;
                }
        }
}
```

Ex: play_again0.c

- Two Problems caused by running in canonical mode
  1) User has to press the Enter key
  2) Program receives and processes an entire line of data when the user presses Enter.

```
$ play_again0
Do you want another transaction (y/n)? sure thing!
```

- First improvement: Turning off canonical input

Ex: play_again1.c – immediate response

```
/* play_again1.c
 *      purpose: ask if user wants another transaction
 *       method: set tty into char-by-char mode, read char, return result
 *      returns: 0=>yes, 1=>no
 *       better: do no echo inappropriate input
 */
#include        <stdio.h>
#include        <termios.h>

#define QUESTION        "Do you want another transaction"

int get_response(char *);
void set_crmode(void);
void tty_mode(int);

int main(void)
{
        int     response;

        tty_mode(0);                            /* save tty mode        */
        set_crmode();                           /* set chr-by-chr mode  */
        response = get_response(QUESTION);      /* get some answer      */
        tty_mode(1);                            /* restore tty mode     */
        return response;
}
```

Ex: play_again1.c – immediate response

```
int get_response(char *question)
/*
 * purpose: ask a question and wait for a y/n answer
 *  method: use getchar and complain about non y/n answers
 * returns: 0=>yes, 1=>no
 */
{
        int input;
        printf("%s (y/n)?", question);
        while(1){
                switch( input = getchar() ){
                        case 'y':
                        case 'Y': return 0;
                        case 'n':
                        case 'N':
                        case EOF: return 1;
                        default:
                                printf("\ncannot understand %c, ", input);
                                printf("Please type y or no\n");
                }
        }
}
```

Ex: play_again1.c – immediate response

```
/* how == 0 => save current mode.  how == 1 => restore mode */
void tty_mode(int how)
{
        static struct termios original_mode;
        if ( how == 0 )
                tcgetattr(0, &original_mode);
        else
                tcsetattr(0, TCSANOW, &original_mode);
}
```

```
void set_crmode(void)
/*
 * purpose: put file descriptor 0 (i.e. stdin) into chr-by-chr mode
 *  method: use bits in termios
 */
{
        struct  termios ttystate;

        tcgetattr( 0, &ttystate);               /* read curr. setting   */
        ttystate.c_lflag        &= ~ICANON;     /* no buffering         */
        ttystate.c_cc[VMIN]     =  1;           /* get 1 char at a time */
        tcsetattr( 0 , TCSANOW, &ttystate);     /* install settings     */
}
```

// “c_cc” member : the control character array
// VMIN : tells the driver how many characters at a time we are reading

Ex: play_again1.c – immediate response

$ gcc play_again1.c –o play_again1

```
$ ./play_again1
Do you want another transaction (y/n)?s
cannot understand s, Please type y or no
u
cannot understand u, Please type y or no
r
cannot understand r, Please type y or no
e
cannot understand e, Please type y or no
y$
```

`play_again1` receives and processes characters as they are typed **without** waiting for the **Enter key!**

Ex: play_again2.c – ignore illegal keys

```
/* play_again2.c
 *      purpose: ask if user wants another transaction
 *       method: set tty into char-by-char mode and no-echo mode
 *               read char, return result
 *      returns: 0=>yes, 1=>no
 *       better: timeout if user walks away
 *
 */
#include        <stdio.h>
#include        <termios.h>

#define QUESTION        "Do you want another transaction"

int get_response(char *);
void set_cr_noecho_mode(void);
void ttv mode(int);
int main(void)
{
        int     response;

        tty_mode(0);                            /* save mode */
        set_cr_noecho_mode();                   /* set -icanon, -echo   */
        response = get_response(QUESTION);      /* get some answer      */
        tty_mode(1);                            /* restore tty state    */
        return response;
}
```

Ex: play_again2.c – ignore illegal keys

```
int get_response(char *question)
/*
 * purpose: ask a question and wait for a y/n answer
 *  method: use getchar and ignore non y/n answers
 * returns: 0=>yes, 1=>no
 */
{
        printf("%s (y/n)?", question);
        while(1){
                switch( getchar() ){
                        case 'y':
                        case 'Y': return 0;
                        case 'n':
                        case 'N':
                        case EOF: return 1;
                }
        }
}
```

※ No error reports for illegal input.
Nothing shows up!

Ex: play_again2.c – ignore illegal keys

```
void set_cr_noecho_mode(void)
/*
 * purpose: put file descriptor 0 into chr-by-chr mode and noecho mode
 *  method: use bits in termios
 */
{
        struct  termios ttystate;

        tcgetattr( 0, &ttystate);               /* read curr. setting   */
        ttystate.c_lflag        &= ~ICANON;     /* no buffering         */
        ttystate.c_lflag        &= ~ECHO;       /* no echo either       */
        ttystate.c_cc[VMIN]     =   1;          /* get 1 char at a time */
        tcsetattr( 0 , TCSANOW, &ttystate);     /* install settings     */
}
```

Ex: play_again2.c – ignore illegal keys

```
/* how == 0 => save current mode,  how == 1 => restore mode */
void tty_mode(int how)
{
        static struct termios original_mode;
        if ( how == 0 )
                tcgetattr(0, &original_mode);
        else
                tcsetattr(0, TCSANOW, &original_mode);
}
```

Ex: play_again2.c – ignore illegal keys

- Compile and try this program: (no echo)
  - If you type sure to it, nothing shows up.
  - Only when you press y or n does the program return.

- It needs one more feature;
  - What if this program were used at a real ATM and a customer wandered away without pressing y or n?
  - User programs are more secure when they include a timeout feature.

Nonblocking Input: play_again3.c

- We create this timeout feature by telling the terminal driver not to wait for input.
  - If we find no input, we sleep a few seconds then look again for input.
  - After three tries, we give up

Blocking vs. Nonblocking Input

- When a program call getchar() or read() to read data from a file descriptor, the call waits for input: blocked

- How do we turn off input blocking?
  - Use fcntl() or open() to enable nonblocking input for a file descriptor.

- Internally, nonblocking operation is pretty simple.
  - Each file has a space to hold available unread data, inside the driver.
  - If the fd has the O_NDELAY bit set and that space is empty, the read() returns 0.

Ex: play_again3.c – use nonblocking mode for timeouts

```
/* play_again3.c
 *      purpose: ask if user wants another transaction
 *       method: set tty into chr-by-chr, no-echo mode
 *               set tty into no-delay mode
 *               read char, return result
 *      returns: 0=>yes, 1=>no, 2=>timeout
 *       better: reset terminal mode on Interrupt
 */
#include        <stdio.h>
#include        <termios.h>
#include        <fcntl.h>
#include        <string.h>

#define ASK             "Do you want another transaction"
#define TRIES       3                                   /* max tries */
#define SLEEPTIME   2                                   /* time per try */
#define BEEP        putchar('\a')                       /* alert user */

main()
{
        int     response;

        tty_mode(0);                            /* save current mode    */
        set_cr_noecho_mode();                   /* set -icanon, -echo   */
        set_nodelay_mode();                     /* noinput => EOF       */
        response = get_response(ASK, TRIES);    /* get some answer      */
        tty_mode(1);                            /* restore orig mode    */
        return response;
}
```

Ex: play_again3.c – use nonblocking mode for timeouts

```
get_response( char *question , int maxtries)
/*
 * purpose: ask a question and wait for a y/n answer or maxtries
 *  method: use getchar and complain about non-y/n input
 * returns: 0=>yes, 1=>no, 2=>timeout
 */
{
        int     input;

       printf("%s (y/n)?", question);                    /* ask          */
       fflush(stdout);                                   /* force output */
        while ( 1 ){
              sleep(SLEEPTIME);                          /* wait a bit   */
              input = tolower(get_ok_char());            /* get next chr */
               if ( input == 'y' )
                       return 0;
               if ( input == 'n' )
                       return 1;
               if ( maxtries-- == 0 )                    /* outatime?    */
                      return 2;                          /* sayso        */
               BEEP;
        }
}
```

Ex: play_again3.c – use nonblocking mode for timeouts

```
/*
 *  skip over non-legal chars and return y,Y,n,N or EOF
 */
get_ok_char()
{
        int c;
        while( ( c = getchar() ) != EOF && strchr("yYnN",c) == NULL )
                ;
        return c;
}
```

※ returns a pointer to the first occurrence of the character c in the string s

Ex: play_again3.c – use nonblocking mode for timeouts

```
set_cr_noecho_mode()
/*
 * purpose: put file descriptor 0 into chr-by-chr mode and noecho mode
 *  method: use bits in termios
 */
{
        struct  termios ttystate;

       tcgetattr( 0, &ttystate);               /* read curr. setting   */
       ttystate.c_lflag        &= ~ICANON;     /* no buffering         */
       ttystate.c_lflag        &= ~ECHO;       /* no echo either       */
       ttystate.c_cc[VMIN]      =  1;          /* get 1 char at a time */
       tcsetattr( 0 , TCSANOW, &ttystate);     /* install settings     */
}
```

Ex: play_again3.c – use nonblocking mode for timeouts

```
set_nodelay_mode()
/*
 * purpose: put file descriptor 0 into no-delay mode
 *  method: use fcntl to set bits
 *   notes: tcsetattr() will do something similar, but it is complicated
 */
{
        int     termflags;
        termflags = fcntl(0, F_GETFL);          /* read curr. settings  */
        termflags |= O_NDELAY;                  /* flip on nodelay bit  */
        fcntl(0, F_SETFL, termflags);           /* and install 'em      */
}
```

Ex: play_again3.c – use nonblocking mode for timeouts

```
/* how == 0 => save current mode,  how == 1 => restore mode */
/* this version handles termios and fcntl flags             */

tty_mode(int how)
{
        static struct termios original_mode;
        static int            original_flags;
        if ( how == 0 ){
                tcgetattr(0, &original_mode);
                original_flags = fcntl(0, F_GETFL);
        }
        else {
                tcsetattr(0, TCSANOW, &original_mode);
                fcntl( 0, F_SETFL, original_flags);
        }
}
```

Ex: play_again3.c – use nonblocking mode for timeouts

- Small Problems with play_again3:
  - The program sleeps for two seconds before calling getchar() to give the user a change to type something.
    - If the user types within one second,
      - The program does not get the character until two seconds pass
  - Without the call to fflush() after printing the prompt, the prompt does not appear until the program calls getchar();
    - Why?
      - The driver buffers output until it receives a newline or until the program tries to read from the terminal

Ex: play_again3.c – use nonblocking mode for timeouts

- A Big Problem:
  What happens if the user presses Ctrl-C?

```
$ make play_again3
cc     play_again3.c   -o play_again3
$ ./play_again3
Do you want another transaction (y/n)?  press Ctrl-C now
$ logout
Connection to host closed.
bash$
```

※ The program is killed and
the entire login session is also killed!

Ex: play_again3.c – use nonblocking mode for timeouts

- How did it happen?

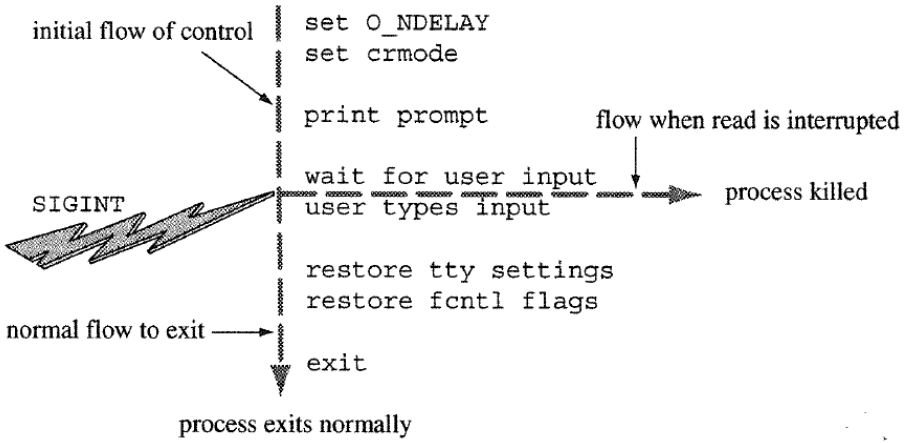


FIGURE 6.5

Ctrl-C kills a program. It leaves terminal unrestored.

  - The program is killed without restoring the state of the terminal driver
    - *The terminals still in nonblocking mode* when the shell returns to print its prompt and get a command line from the user.
    - *The shell calls* `read()` *to get the command line*, but `read()`, operating in *nonblocking* mode, returns 0 immediately

How to protect our program from Ctrl-C?

Contents

- 6.1 Software Tools vs. Device-Specific Programs
- 6.2 Modes of the Terminal Driver
- 6.3 Writing a User Program: play-again.c
- 6.4 Signals
- 6.5 Prepared for Signals: play_again4.c

What Does Ctrl-C Do?

- The Ctrl-C key interrupts the currently running program
- This interruption is produced by a kernel mechanism called a signal

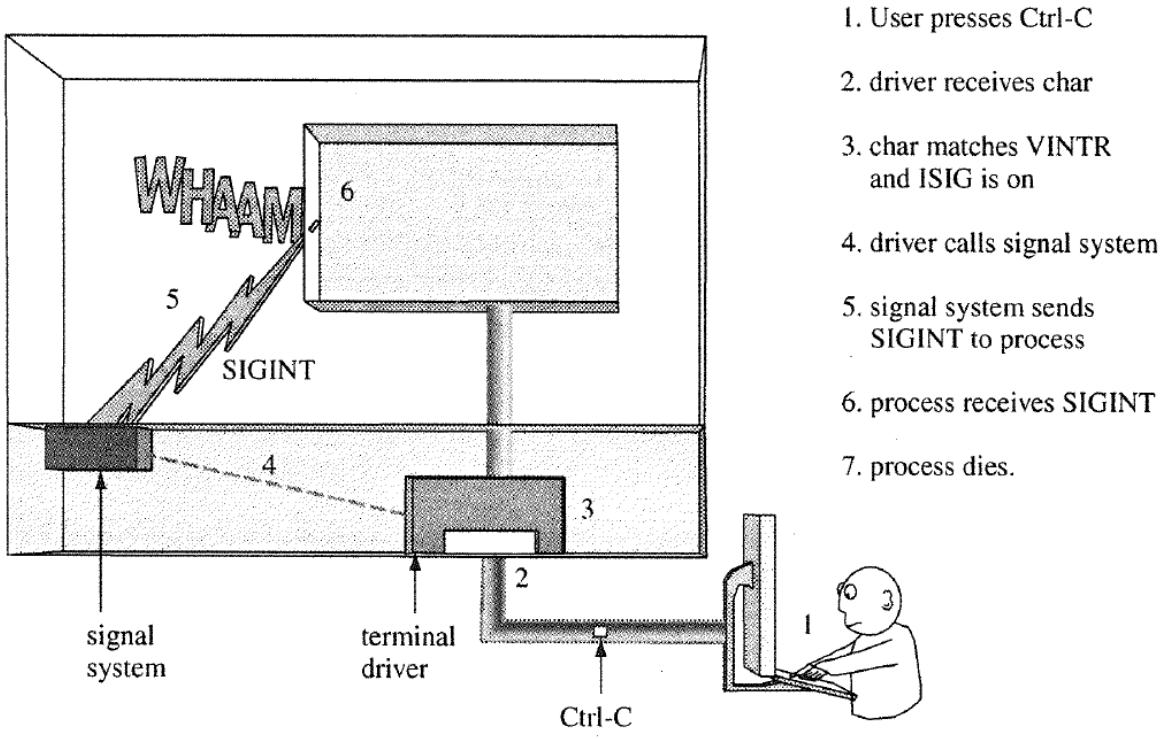


FIGURE 6.6
How Ctrl-C works.

What is a Signal?

- A signal is a one-word message from kernel to process
  - Each signal has a numerical code (e.g., interrupt signal is code number 2)
  - When pressing the Ctrl-C key, kernel send the interrupt signal to the currently running process
- Requests for signals come from three sources :
  - **Signals from user,** ex) when user press Ctrl-C key
  - **Signals from kernel,** ex) when the process does something woring such as process exception and segmentation violation
  - **Signals from other processes,** ex) a process can send a signal to communicate with other processes

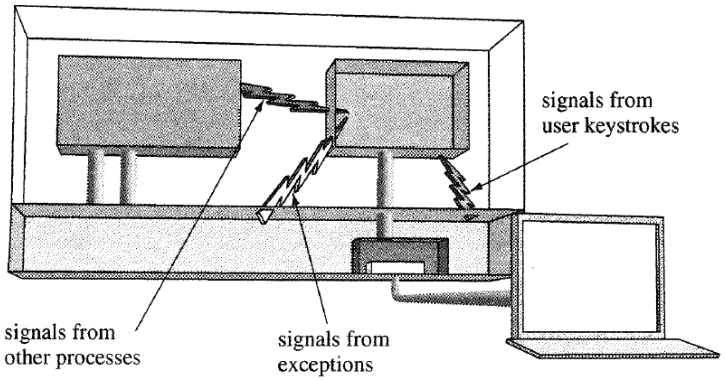


FIGURE 6.7
Three sources of signals.

Where Can I Find a List of Signals?

- /usr/include/signal.h

※ $ man 7 signal

```
#define SIGHUP    1    /* hangup, generated when terminal disconnects */
#define SIGINT    2    /* interrupt, generated from terminal special char */
#define SIGQUIT   3    /* (*) quit, generated from terminal special char */
#define SIGILL    4    /* (*) illegal instruction (not reset when caught)*/
#define SIGTRAP   5    /* (*) trace trap (not reset when caught) */
#define SIGABRT   6    /* (*) abort process */
#define SIGEMT    7    /* (*) EMT instruction */
#define SIGFPE    8    /* (*) floating point exception */
#define SIGKILL   9    /* kill (cannot be caught or ignored) */
#define SIGBUS    10   /* (*) bus error (specification exception) */
#define SIGSEGV   11   /* (*) segmentation violation */
#define SIGSYS    12   /* (*) bad argument to system call */
#define SIGPIPE   13   /* write on a pipe with no one to read it */
#define SIGALRM   14   /* alarm clock timeout */
#define SIGTERM   15   /* software termination signal */
```

What Can a Process Do about a Signal?

- A process does not have to die when it receives SIGINT

- A process can tell the kernel by using the signal(), how it wants to respond to a signal

- A process has three choices:
  - Accept the default action (usually death)
    signal(SIGINT, SIG_DFL);
  - Ignore the signal
    signal(SIGINT, SIG_IGN);
  - Call a function
    signal(signum, functionname);

Signal Handling

signal	
PURPOSE	Simple signal handling
INCLUDE	`#include <signal.h>`
USAGE	`result = signal (int signum, void (*action)(int))`
ARGS	signum the signal to respond to action how to respond
RETURNS	-1 if error prevaction if success

Example of Signal Handling

- Ex 1: Catching a Signal

```
/* sigdemo1.c - shows how a signal handler works.
 *            - run this and press Ctrl-C a few times
 */

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

main()
{
        void    f(int);                 /* declare the handler  */
        int     i;
        signal( SIGINT, f );            /* install the handler  */
        for(i=0; i<5; i++ ){            /* do something else    */
                printf("hello\n");
                sleep(1);
        }
}

void f(int signum)                      /* this function is called */
{
        printf("OUCH!\n");
}
```

Example of Signal Handling

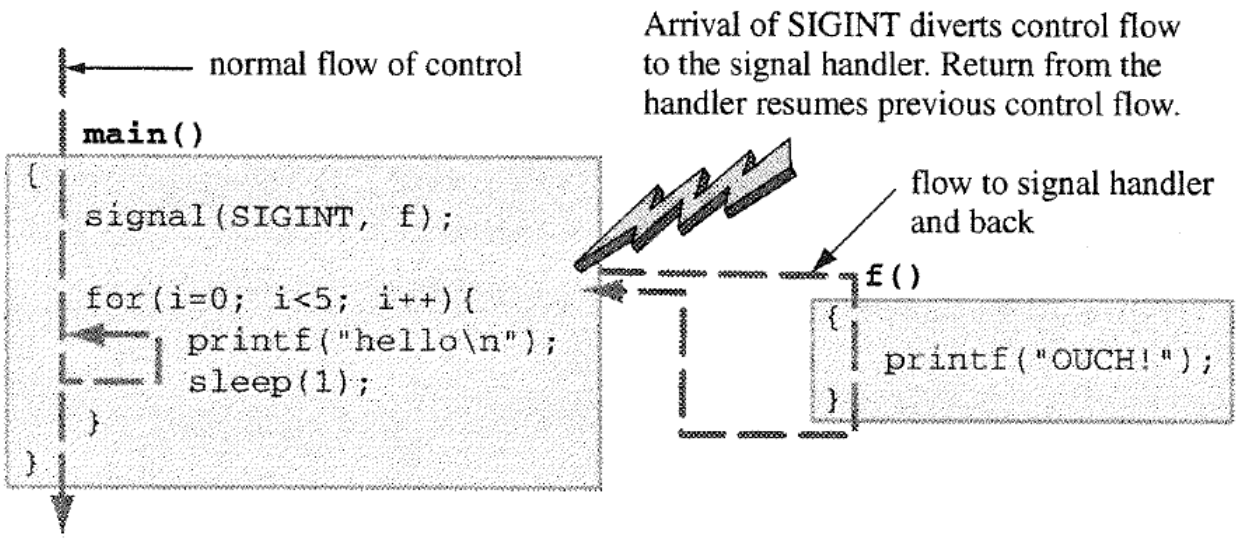


FIGURE 6.8

A signal causes a subroutine call.

```
$ ./sigdemo1
hello
hello      press Ctrl-C now
OUCH!
hello      press Ctrl-C now
OUCH!
hello
hello
$
```

Example of Signal Handling

- Ex 2: Ignoring a Signal

```
/* sigdemo2.c - shows how to ignore a signal
 *            - press Ctrl-\ to kill this one
 */

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

main()
{
        signal( SIGINT, SIG_IGN );

        printf("you can't stop me!\n");
        while( 1 )
        {
                sleep(1);
                printf("haha\n");
        }
}
```

Example of Signal Handling

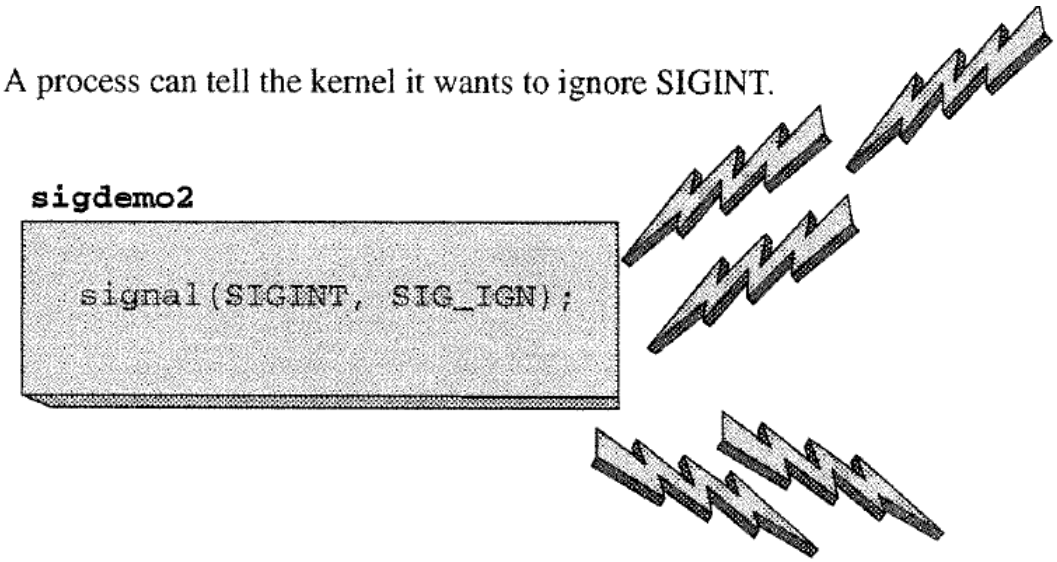


FIGURE 6.9

The effect of `signal(SIGINT, SIG_IGN)`.

```
$ ./sigdemo2
you can't stop me!
haha
haha
haha      press Ctrl-C now
haha      press Ctrl-C nowpress Ctrl-C now
haha
haha
haha      press ^\ now
Quit
$
```

Interrupt signals

quit signal

Contents

- 6.1 Software Tools vs. Device-Specific Programs
- 6.2 Modes of the Terminal Driver
- 6.3 Writing a User Program: play-again.c
- 6.4 Signals
- 6.5 Prepared for Signals: play_again4.c

play_again4.c

```
/* play_again4.c
 *      purpose: ask if user wants another transaction
 *       method: set tty into chr-by-chr, no-echo mode
 *               set tty into no-delay mode
 *               read char, return result
 *               resets terminal modes on SIGINT, ignores SIGQUIT
 *      returns: 0=>yes, 1=>no, 2=>timeout
 *       better: reset terminal mode on Interrupt
 */
#include        <stdio.h>
#include        <termios.h>
#include        <fcntl.h>
#include        <string.h>
#include        <signal.h>

#define ASK           "Do you want another transaction"
#define TRIES      3                                    /* max tries */
#define SLEEPTIME  2                                    /* time per try */
#define BEEP       putchar('\a')                        /* alert user */


main()
{
        int     response;
        void    ctrl_c_handler(int);

        tty_mode(0);                            /* save current mode    */
        set_cr_noecho_mode();                   /* set -icanon, -echo   */
        set_nodelay_mode();                     /* noinput => EOF       */
        signal( SIGINT, ctrl_c_handler );       /* handle INT           */
        signal( SIGQUIT, SIG_IGN );             /* ignore QUIT signals  */
        response = get_response(ASK, TRIES);    /* get some answer      */
        tty_mode(1);                            /* reset orig mode      */
        return response;
}
```

play_again4.c

```
get_response( char *question , int maxtries)
/*
 * purpose: ask a question and wait for a y/n answer or timeout
 *  method: use getchar and complain about non-y/n input
 * returns: 0=>yes, 1=>no
 */
{
        int     input;

        printf("%s (y/n)?", question);                  /* ask          */
        fflush(stdout);                                 /* force output */
        while ( 1 ){
                sleep(SLEEPTIME);                       /* wait a bit   */
                input = tolower(get_ok_char());         /* get next chr */
                if ( input == 'y' )
                        return 0;
                if ( input == 'n' )
                        return 1;
                if ( maxtries-- == 0 )                  /* outatime?    */
                        return 2;                       /* sayso        */
                BEEP;
        }
}
```

play_again4.c

```
/*
 *  skip over non-legal chars and return y,Y,n,N or EOF
 */
get_ok_char()
{
        int c;
        while( ( c = getchar() ) != EOF && strchr("yYnN",c) == NULL )
                ;
        return c;
}



set_cr_noecho_mode()
/*
 * purpose: put file descriptor 0 into chr-by-chr mode and noecho mode
 *  method: use bits in termios
 */
{
        struct  termios ttystate;

        tcgetattr( 0, &ttystate);               /* read curr. setting   */
        ttystate.c_lflag        &= ~ICANON;     /* no buffering         */
        ttystate.c_lflag        &= ~ECHO;       /* no echo either       */
        ttystate.c_cc[VMIN]     =  1;           /* get 1 char at a time */
        tcsetattr( 0 , TCSANOW, &ttystate);     /* install settings     */
}
```

play_again4.c

```
set_nodelay_mode()
/*
 * purpose: put file descriptor 0 into no-delay mode
 *  method: use fcntl to set bits
 *   notes: tcsetattr() will do something similar, but it is complicated
 */
{
        int     termflags;

        termflags = fcntl(0, F_GETFL);          /* read curr. settings  */
        termflags |= O_NDELAY;                  /* flip on nodelay bit  */
        fcntl(0, F_SETFL, termflags);           /* and install 'em      */
}

void ctrl_c_handler(int signum)
/*
 * purpose: called if SIGINT is detected
 *  action: reset tty and scram
 */
{
        tty_mode(1);
        exit(1);
}
```

play_again4.c

```
/* how == 0 => save current mode,  how == 1 => restore mode */
/* this version handles termios and fcntl flags              */

tty_mode(int how)
{
        static struct termios original_mode;
        static int            original_flags;
        static int            stored = 0;

        if ( how == 0 ){
                tcgetattr(0, &original_mode);
                original_flags = fcntl(0, F_GETFL);
                stored = 1;
        }
        else if ( stored ) {
                tcsetattr(0, TCSANOW, &original_mode);
                fcntl( 0, F_SETFL, original_flags);
        }
}
```

Summary

- Some programs process data from specific devices. These device-specific programs have to control the connection to the device
- Terminal-