# Chapter 6 Event Driven Programming

## Concepts Covered

*The NCurses library,*
*Alarms and interval timers*
*Signals revisited, signal-driven I/O*
*Asynchronous I/O*

*AIO Library*
*alarm, pause, nanosleep,*
*setitimer, getitimer, aio_read,*
*aio_return, aio_error*

## 6.1 Introduction

An event-driven program is a program in which the flow of control of the program depends upon the occurrence of external events. The typical event-driven program remains in a state in which it listens for or awaits events, selects which events to respond to next, responds to them, and then returns to its listening state. Event driven programs must have some type of event recognition mechanism and event handling mechanism. Unlike sequential programs, event-driven programs must work correctly in an environment in which unexpected, dynamic, external stimuli come from sources such as users, hardware, or other processes.

## 6.2 Common Features of Event-Driven Programs

Event-driven programs include programs with graphical user interfaces, operating systems, device drivers, control system software, and video games, to name a few. Writing video games is a good means to master event-driven programming, because their requirements include those commonly encountered in other event-driven programs (*EDPs*), and because it is generally fun to write them. Typical video games need to handle the following:

Spatial control Like many other EDPs, video games have to manage the two-dimensional screen image, maintaining information about where all of its objects are located.

Timing Video games, like many EDPs, usually have moving images whose velocities are monitored and controlled by the game. Games may also time the user's inputs. They often have to keep track of clock time and cause certain events to happen at specific times or at specific intervals of time.

Asynchronous inputs and signals Video games, like all EDPs, have to respond to unpredictable user inputs such as mouse clicks, mouse motion, and keystrokes, as well as inputs from other sensors. These events are asynchronous with respect to the execution of the program.

Process synchronization Video games usually have multiple threads of control. One or more objects might be moving independently across the screen while the user independently types or uses a tracking device. The program has to keep track of and synchronize these independent processes and objects.

## 6.3 Terminal-based Games

Early UNIX systems often came bundled with a large variety of terminal-based games, i.e., games that ran in a pseudo-terminal window rather than a console window. The distinction between these is that a terminal window is a character I/O device that treats its display area as a two-dimensional array whose cells can contain characters, whereas a console window is a bit-mapped display device each of whose pixels can be accessed individually. Historically, these games were located in `/usr/games`.

These days, system administrators do not install the games, one reason being because they know that users tend to use up system resources having fun instead of working[1]. Another reason for not installing the terminal-based games is that there are now many free games that run on top of the X Windows system, using bit-mapped displays, making the older games seem less fun to those accustomed to the advanced technology. Perhaps those who appreciate the old terminal-based games are the same people who still appreciate black-and-white movies.

The great advantage of writing a terminal-based game over one that uses a GUI, is that it is easier to concentrate on the principles rather than the details of the windowing system. Although it is more exciting to create a game that runs in a graphics window, that requires an entirely different set of topics to learn and it would be a distraction from the objective of learning how to control and use signals, how to use time and synchronization, and how to control what the user is able to do with the keyboard. If you also had to learn about video cards, X Windows, and widgets and windows, your time would be consumed with that instead.

We will write a game similar to the game of `pong`, which runs in a terminal window. The game of `pong` is a simplification of an arcade game. In `pong`, there are two controlled objects: a ball and a paddle. The ball is a small circle or square that moves across the screen at some fixed speed. The paddle is a vertical line segment that the user can move up and down with keystrokes. The edges of the terminal window are walls off of which the ball bounces.

## 6.4 The Curses (NCurses) Library

Recall that in Chapter 1 we saw that we could configure the terminal by sending various escape sequences to it, such as "\033[7m" to reverse the colors of the video display. We also saw how we could move the cursor around, clear various portions of the screen, and do other things by sending escape sequences to the terminal. With different kinds of terminals requiring different escape sequences, the task of writing a program that controls a terminal becomes complex, if this is the only means of configuring and controlling terminals.

Fortunately, in UNIX, it is relatively easy to write programs that control the terminal, because UNIX systems come bundled with a character-oriented graphics library called Curses, the header file for which is `<curses.h>`. Curses is basically a library that wraps the complexity of terminal capabilities into an easy to use interface. According to Eric Raymond in the September issue of the Linux Journal,

"The first Curses library was hacked together at the University of California at Berkeley in about 1980 to support a screen-oriented dungeon game called `rogue`. It leveraged an earlier facility called `termcap`, the terminal capability library, which was used in the `vi` editor and elsewhere."

[1] In the past, people would spend idle time playing snake, worm, hangman, chess, or even rogue. The first thing one did when given an account on a UNIX system was to check the contents of `/usr/games`.

AT&T Bell Labs saw the virtues of Curses and developed their own version and incorporated it into SVR1. The SVR1 Curses library had many attractive features, but it was proprietary and it was based on a binary file format called `terminfo`, while the BSD version was free and based on the `termcap` file, a plain text file. Programmers were torn between the proprietary, enhanced Curses of SVR1 and the free, but limited feature BSD version. In 1982, Pavel Curtis solved the problem by rewriting a version of Curses based on the SVR1 version, but his was free and text-based. This made it possible for hackers to improve on it. To shorten the story, from Curses eventually came Ncurses (new curses), with more features and multi-terminal capabilities.

We will use the Ncurses library. The Ncurses library is a library of more than one hundred graphics functions for manipulating a character-oriented display device. The functions treat the display device as a two-dimensional array of characters, with coordinate (0,0) in the upper left corner. The library also contains routines for creating and manipulating windows, sub-windows and panels, and menus. We will focus our attention on a small set of features of the library.

### 6.4.1 NCurses Basics

Most window managers, whether on UNIX, the Macintosh operating systems, or the various Windows operating systems, follow the same principle of drawing: they maintain two data structures representing the canvas on which they draw. One, the *visible canvas*, is what is currently in view on the physical display device, and the other, the *hidden canvas*, is a canvas stored in memory, on which drawing operations take place. This terminology is not standardized and goes by various names, depending on the particular system one uses. I will use the term *double-buffering* to describe this method of rendering, which is what it is called in graphics applications.

In double buffering, applications draw on the hidden canvas, and when it is ready to be displayed, it is drawn onto the screen. In effect the hidden canvas becomes the visible canvas, and the memory used for the visible canvas becomes the hidden canvas. The operation of drawing the hidden canvas on the screen is known by various names, but the most common is *screen updating*. In reality, screen updating is optimized to redraw only those portions of the screen that are different than what is on display.

NCurses uses a form of double buffering. Because NCurses manages the content of a terminal window, the concept of a "screen" in this context is not the monitor's full visible area, but the area enclosed within the terminal emulation window. Henceforth, a screen refers to the content area of a terminal window. In NCurses, screen updating is called *refreshing*.

NCurses, like many graphical libraries, uses a coordinate system derived from matrix coordinates rather than Cartesian coordinates. The origin is at the upper left corner of the screen, and the pair (y,x) representing the coordinates of a point (or character in this case) is the row number followed by the column number, as shown in Figure 6.1.

It defines a data structure called a `WINDOW` to represent a *window*. A `WINDOW` structure describes a sub-rectangle of the screen, possibly the entire screen. It includes the window's starting position on the screen (the (y, x) coordinates of the upper left hand corner), its size, and various properties. It is opaque to the programmer; you do not have access to its members. You can write to a window as though it were a miniature screen, scrolling independently of other windows on the physical screen. A window is a purely internal representation. It is used to build and store a potential image of a portion of the terminal. It doesn't bear any necessary relation to what is really on the terminal screen.

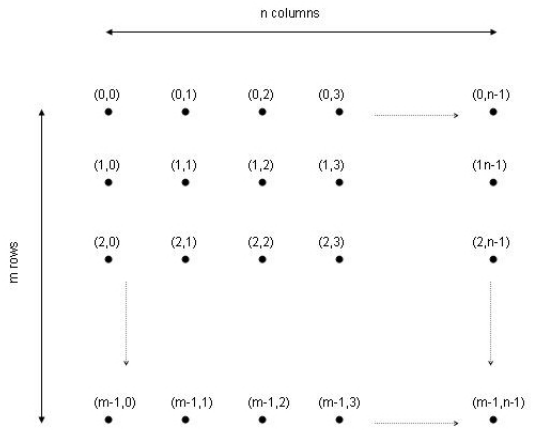


Figure 6.1: NCurses coordinate system

A *screen* is a special window that is the size of the terminal screen, i.e., it starts at the upper left hand corner and extends to the lower right hand corner. There are two predefined screens[2]: `stdscr`, which is automatically provided for the programmer, and `curscr`, for current screen, which is a screen image of what the terminal currently looks like. The programmer can draw on `stdscr`, but not on `curscr`.

### 6.4.2 Screen Updating

Each time that the application makes changes to a window that it wants to become visible on the screen, it needs to refresh the screen. There are two functions that make the section of the terminal screen corresponding to a window reflect the contents of the window structure: `refresh()` and `wrefresh()`. If the application is drawing on `stdscr`, which is the default screen, then it simply calls

```
refresh()
```

If it is drawing on a `WINDOW` named `win`, and it wants to draw that window's content on the screen, it calls

```
wrefresh(win)
```

`refresh()` is equivalent to `wrefresh(stdscr)`. It is actually a macro.

A piece of screen "real estate" may be within the extent of any number of overlapping windows. If two windows, `win1` and win2, overlap, and `wrefresh(win2)` is called, the library determines how to redraw the screen most efficiently, replacing those portions of the screen within the intersection of `win1` and `win2`. It only redraws a window if that window's content has changed in some way. You can call `touchwin(win)` to tell NCurses that the entire window `win` has changed, forcing a redraw when `wrefresh(win)` is called.

[2]There is a third, hidden screen that represents the logical screen on which the hidden drawing takes place. NCurses documentation calls it the *virtual screen*.

### 6.4.3 Building Programs

All programs using NCurses must include the `<ncurses.h>` header file and the standard C I/O library header file `<stdio.h>`. The header file `<ncurses.h>` is often just a symbolic link to `<curses.h>`, so they are often interchangeable. Because the NCurses library is not in the linker's standard set of libraries, you have to build explicitly with `-lncurses` in the command (following all files that reference NCurses symbols):

```
$ gcc -o myprog myprog.c -lncurses
```

### 6.4.4 A Core Repertoire of Functions

This is not intended to be a comprehensive tutorial on NCurses. For that you should consult any of the several on-line reference manuals or tutorials. The objective here is to explain the underlying concepts of the core library and to describe many of the functions that it provides. Below is a collection of the most important, and basic, representative functions for terminal configuration, cursor movement, output and input.

**Configuration Functions**

| | |
|---|---|
| `initscr()` | Initialize the curses library and create a logical screen. |
| `endwin()` | Turn off curses and reset the screen. |
| `wrefresh(win)` | Draw what is in the logical window `win` into the physical display. Remember that `refresh()` is the same as `wrefresh(stdscr)`. |
| `clear()` | Clear the screen. |
| `keypad(stdscr,TRUE)` | Enable use of function and keypad keys (arrows, F1, ...) |

**Output and Cursor Movement**

| | |
|---|---|
| `move(r,c)` | Move the cursor to screen position (r,c). |
| `getyx(win,y,x)` | Get the current cursor position. This is a macro so y and x do not have to be passed by address. |
| `addch(c)` | Draw character `c` on the screen at the current cursor position, advancing the cursor to the end of the character. If `c` is a tab, newline, or backspace, the cursor is moved appropriately within the window. It wraps if it reaches the right margin. |
| `addstr(str)` | Draw the character string `str` on the screen at the current cursor position, advancing the cursor to the end of the string. It is equivalent to calling `addch()` for every character in the string. |
| `addnstr(str, n)` | Like addstr(), except that at most `n` characters of `str` will be written. If `n` is -1, then the entire string will be added, up to the maximum number of characters that will fit on the line, or until a terminating null is reached. Thus, `addstr(str)` is the same as `addnstr(str,-1)`. |

| | |
|---|---|
| `mvaddch(x,y,c)` | Move the cursor to position (x,y) on the screen and draw the character at that position, advancing cursor to the end of the character. |
| `addchstr(str)` | Like `addstr(str)`, except that: the cursor does not advance, it does not perform any kind of checking (such as for the newline, backspace, or carriage return characters), it does not expand other control characters to ^-escapes, and it truncates the string if it crosses the right margin, rather than wrapping it around to the new line. |
| `printw(fmt, ...)` | The same as `printf()` in C but prints to current cursor position. |

**Input**

| | |
|---|---|
| `getch()` | Read a character from the window. |
| `getstr(str)` | Read characters until a newline is received and store (without newline) in `str` (allocated by caller) |
| `scanw(fmt, ...)` | The same as `scanf()` in C – like calling `getstr()`, passing to `sscanf()`. |

**Window Functions**

| | |
|---|---|
| `win = newwin(l,c,y,x)` | Create a new window with `l` lines, `c` columns, whose upper left corner is at (`y,x`). Returns pointer to new window. |
| `mvwin(win,y,x)` | Move window pointed to by `win` to position (`y,x`) |
| `win = dupwin(oldwin)` | Make a duplicate of `oldwin`, returning pointer to the new window. |
| `delwin(win)` | Delete the window `win`, releasing all of its resources. |
| `putwin(win, filep)` | Write all data associated with the window pointed to by `win` into the `FILE` stream to which `filep` points. Returns `ERR` if the underlying write fails. |
| `win = getwin(filep)` | Read all window data stored into the file by `putwin`, and create and initialize a new window with that data. This returns a pointer to the new window. |

**Synopsis.**

The `initscr()` function initializes the terminal in curses mode. It may also clear the screen in certain implementations. This always must be called first. It initializes the NCurses system and allocates memory for the `stdscr` and `curscr` windows and some other data-structures. When a program is finished, it should always call `endwin()` to reset the terminal and release curses resources.

After initializing curses, there are several functions that can be used to configure the terminal. It is usual to clear the screen with `clear()`, and if the program wants to receive key-presses from the keypad and function keys, then it should call `keypad(stdscr, TRUE)`. Other functions not shown above include functions that affect the terminal driver processing modes – `raw()` and `cbreak()`, `echo()` and `noecho()`, and `halfdelay()`. We will discuss these later.

Output functions can be divided into three families:

`addch()` Print a character at the cursor position, advancing cursor

`addstr()` Print a string at the cursor position, advancing cursor

`printw()` Print formatted output similar to `printf()`, advancing cursor

Thus, `addch()` adds a character, `addstr()` adds a string, and `printw()` prints formatted text. For each of these there are many variants, which are described below.

The cursor can be moved without output using the `move()` function. Its current position can be retrieved using `getyx()`. The `mvaddch()` function is a representative of a class of functions that perform a cursor movement prior to an output operation. Generally speaking, for each output function such as `addch()`, there is a corresponding function of the form `mvaddch()`. For example, there is a `mvaddstr()` function and a `mvprintw()` function.

The basic input functions are `getch()` and the string counterpart, `getstr()`, and the C-like `scanw()`, which is like `scanf()`. Notice that `getch()` has no argument, but `getstr()` expects a pointer to an allocated buffer in which to store the entered text.

Finally, all of the above functions work on the standard screen, `stdscr`. *They are all macros.* For each of them, there is a function that operates on arbitrary windows, and a naming convention that makes it pretty easy to guess what they are. For example, `wgetch(win)` is an input function that reads a character from the current window, `win`, and `getch()` is defined as `wgetch(stdscr)`. Similarly, `waddch(win,ch)` puts a character at the cursor position in `win` and `addch(ch)` is defined as `waddch(stdscr,ch)`.

One can create windows using `newwin()`, which allocates the memory on the heap and returns a pointer to it. If `newwin()` is passed 0 for either lines or columns, that dimension is set to the maximum it can be and fit within the terminal window. The function makes sure that the new window does not extend beyond the bounds of the terminal screen in all cases. A window can be moved using `mvwin()`; you have to refresh to see the change. This does not erase the old window from the screen, which you have to do yourself. You can make a copy of a window with `dupwin()`, and delete a window with `delwin()`.

### 6.4.5 Important Points About Windows and Refreshing

- It is a good idea to call `refresh()` or `wrfresh()` whenever you make changes to the screen, but you should bear in mind several important points.
- The functions of the `addch()` and `addstr()` families that write strings and characters to the screen always call `wrefresh()` themselves, so that it is not necessary to refresh when adding strings or characters. This is not true of the `printw()` functions.
- When drawing many windows to the screen, if `wrefresh()` is called for each window, it can cause bursty output and poor performance. The `wrefresh()` function actually calls two functions, `wnoutrefresh()` and `doupdate()`. A call to `wnoutrefresh(win)` copies the `WINDOW` pointed to by `win` onto the logical screen, and `doupdate()` copies the logical screen to the physical screen. Therefore, it is better to call `wnoutrefresh(win)` for each window to be written to the screen, followed by a single call to `doupdate()`.

- The input functions of the `getch()` and `getstr()` families will call `wrefresh()` on the given window if it has been moved or modified since its last refresh. If echo is on, then automatically a refresh will take place, since this is a modification. To be clear, `getch()` will call `refresh()`, and `wgetch(win)` will call `wrefresh(win)`. This can have serious consequences on the behavior of your program, since the cursor will move into the window on which `wgetch(win)` is being called, and refreshes may have unexpected consequences as well.

- Lastly, as a general rule, you should never write NCurses programs that mix the use of the standard screen, `stdscr`, and other windows. The functions that perform input and output and refreshing on the standard screen interact in unexpected ways with other windows. If you want to write simple programs, do not use windows in them, and conversely, if you feel that the program would benefit from using windows, then do not use any functions that operate on the standard screen.

### 6.4.6 A Few Simple Programs

In keeping with the tradition, we start with a hello-world program.

Listing 6.1: helloworld.c

```
Listing. helloworld.c
#include <ncurses.h>

int main()
{
    initscr();                    /* initialize the library */
    printw("Hello World !!!\n"); /* print at cursor */
    refresh();                    /* update screen (unnecesssary) */
    getch();                      /* wait for a keypress */
    endwin();                     /* clean up and quit curses */
    return 0;

}
```

The input call `getch()` is used so that the screen does not disappear before we can see it. The next program is a bit more interesting.

Listing 6.2: drawpattern.c

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

int main()
{
    char   pattern[] = "1234567890";
    int    i;

    /* Initialize NCurses and clear the screen */
    initscr();
    clear();
```

```
    /* This will wrap across the screen */
    move(LINES/2,0);
    for ( i = 1; i <= 8; i++ ) {
        addstr( pattern );
        addch(' ');
    }

    /* Park the cursor at bottom */
    move(LINES-1,0);
    addstr("Type any char to quit:");
    refresh(); /* not needed */

    /* Wait for the user to type something, otherwise
       the screen will clear.  */
    getch();
    endwin();
    return 0;
}
```

### Comments

1. NCurses has a predefined constant, `LINES`, that contains the number of rows in the terminal window, and a constant `COLS` that stores the number of columns.

2. Notice too that the program calls `refresh()` each time it changes the screen. This is unnecessary, because `addstr()` forces the refresh automatically.

3. If you delete the call to `getch()`, you will not see anything, and if you delete the call to `endwin()`, the screen will not be restored.

The next program draws a grid of periods centered on the screen.

Listing 6.3: drawgrid.c

```
#define  CENTERY    (LINES/2 -2)     /* The middle line in terminal */
#define  CENTERX    (COLS/2 -2)      /* The middle column in terminal */
#define  NUMROWS    (LINES/2)
#define  NUMCOLS    (COLS/2)

int main()
{
    int r, c;
    char  MESSAGE[] = "Press any character to exit:";
    int   length, i, j;
    length = strlen(MESSAGE);

    initscr();      /* Initialize screen */
    clear();        /* Clear the screen */
    noecho();       /* turn off character echo */

    char grid[NUMROWS][NUMCOLS];
```

```
    for ( i = 0; i < NUMROWS; i++ ) {
        for ( j = 0; j < NUMCOLS-1; j++ )
            grid[i][j] = '.';
        grid[i][NUMCOLS-1] = '\0';
    }

    /* move to center to draw grid */
    r = CENTERY - (NUMROWS/2);
    c = CENTERX - (NUMCOLS/2);
    move(r,c);

    /* Draw each row of grid as a string    */
    for ( i = 0; i < NUMROWS; i++ ) {
        mvaddstr(r+i,c,grid[i]);
    }

    /* Move to bottom of screen, post message to display */
    move(LINES-1,0);
    addstr(MESSAGE);

    getch();            /* wait for the user to type something */
    clear();            /* clear the screen */
    endwin();           /* delete curses window and quit */;
    return 0;
}
```

NCurses makes it easy to save any window to a file. The `putwin()` function will write the contents of a window to a `FILE` stream, and this can be read back into a program using `getwin()`. The next two listings show how to do both. The first is a program that draws a face in a window and also saves it to a file specified on the command line.

Listing 6.4: Saving a window: drawface2.c

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

#define  CENTERY    (LINES/2 -2)      /* The middle line in terminal     */
#define  CENTERX    (COLS/2 -2)       /* The middle column in terminal   */

void addhappyface(int * y, int * x)
{
    int orig_y = *y;
    addstr(" ^ ^ "); move(++(*y),*x);
    addstr(" o o "); move(++(*y),*x);
    addstr("  ^  "); move(++(*y),*x);
    addstr("\\___/");move(++(*y),*x);
    addstr("      ");
    *y = orig_y;
    *x = (*x) + 5;
    move( *y, *x);
}
```

```
int main(int argc, char *argv[])
{
    int   r, c;
    char  MESSAGE[] = "Press any character to exit:";
    int   length;
    FILE *fp;       /* for writing window contents */

    length = strlen(MESSAGE);
    if ( argc < 2 ) {
        printf("usage: %s window-file\n", argv[0] );
        return 0;
    }

    fp = fopen(argv[1], "w");
    if ( NULL == fp ) {
        printf("Error opening %s for writing.\n", argv[1]);
        return 0;
    }

    initscr();  /* Initialize curses library and the drawing screen */
    clear();    /* Clear the screen */

    /* Move to bottom of screen and post  message to display */
    move(LINES-1,0);
    addstr(MESSAGE);

    /* move to center of screen - width of face */
    r = CENTERY;
    c = CENTERX - 5;
    move(r,c);
    addhappyface(&r, &c);
    addhappyface(&r, &c);
    addhappyface(&r, &c);

    /* Park cursor at bottom at the right side of the message */
    move(LINES-1,length);
    refresh();

    /* Write the standard screen to a file */
    if ( ERR == putwin(stdscr, fp) ) {
        printw("Error saving window.\n");
    }
    fclose(fp);

    getch();          /* wait for the user to type something */
    clear();          /* clear the screen */
    endwin();         /* delete curses window and quit */
    return 0;
}
```

The next listing is of a program that can read any file created by an NCurses program that saved data using `putwin()`. It tries to open the file and display the window stored there. As `getwin()` returns a `NULL` pointer on failure, it checks that the returned pointer is not `NULL` before displaying the data.

Listing 6.5: Retrieving a saved window: getdrawing.c

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

int main(int argc, char *argv[])
{
    FILE   *fp;
    WINDOW *win;

    if ( argc < 2 ) {
        printf("usage: %s window-file\n", argv[0] );
        return 0;
    }

    fp = fopen(argv[1], "r");
    if ( NULL == fp ) {
        printf("Error opening %s.\n", argv[1]);
        return 0;
    }

    initscr();      /* Initialize curses library and the drawing screen */
    cbreak();       /* So that the character is available immediately */
    noecho();       /* Turn off echo */
    clear();        /* Clear the screen */

    move(LINES-1,0);
    addstr("Enter a character to see the faces:");
    getch();

    win = getwin(fp);
    if ( NULL == win ) {
        clear();
        move(LINES-2,0);
        printw("The file %s was not created using putwin()."
               " Type any character to exit.\n",
                argv[1]);
    }
    wrefresh(win);
    fclose(fp);

    getch();          /* wait for the user to type something */
    clear();          /* clear the screen */
    endwin();         /* delete curses window and quit */
    return 0;
}
```

## 6.5 User Input in NCurses

NCurses has functions to put the terminal into a few different input modes. The following table summarizes the different models of input.

| Function Call | Line Buffering | Erase/Kill Processing | Signal Interpretation | Blocking |
|---|---|---|---|---|
| `raw()` | No | No | No | Yes |
| `cbreak()` | No | No | Yes | Yes |
| `halfdelay(n)` | No | No | Yes | Timed (0.1*n seconds) |
| `nodelay(stdscr, TRUE)` | No | No | Yes | No |

Raw mode, established with the `raw()` function, is similar to non-canonical mode in which, in addition, keyboard signal processing is disabled. Note though that it is a blocking input mode. Cbreak mode, established with `cbreak()`, is like raw mode except that keyboard signals are processed. Cbreak mode is also blocking.

The `halfdelay()` function and the `nodelay()` function both turn off blocking mode, but the `halfdelay()` function has a timeout whereas `nodelay()` is mercilessly unforgiving and does not. Neither is line-buffered nor allows editing functions. The `halfdelay()` function takes a single integer argument that represents the number of tenths of a second to block for terminal input. If no input arrives within that time, then it returns the `ERR` value, which is an integer value. In our demos directory, in chapter07, you can find demo programs named `raw_demo.c`, `cbreak_demo.c`, `halfdelay_demo.c`, and `nodelay_demo.c`, that show how these input modes work.

There are two other functions worth remembering: `noraw()` and `nocbreak()`. If the terminal has been put into raw, cbreak, or halfdelay mode, `noraw()` undoes that effect, turning on line buffering, line editing, blocking, and signal processing. It will not undo the effect of `nodelay()`, which can only be undone by calling

```
nodelay(stdscr, FALSE);
```

The `nocbreak()` function restores line-buffering and line-editing, but does not restore signal processing if it had been disabled by raw mode previously. For that you need to call `noraw()`, which, turns signal processing back on. `nocbreak()` also ends halfdelay mode.

### Example

We will begin with a relatively simple program that puts the terminal into cbreak mode. The program will go into a user-controlled loop that terminates only when the user enters a specific character. To make it a bit more interesting, and realistic, we will use the F1 function key to terminate the program. The program will also show how user input can be used to modify the current window state other than by displaying text. It will let the user move the cursor around on the screen with the arrow keys. Finally, it will create a status bar at the bottom of the screen and write the current cursor position into it as the cursor moves, as well as the user's instructions for what to do.

The listing follows. The comments explain the logic within the program.

Listing 6.6: cursortrack.c

```
/* LINES and COLS are NCurses variables that get initialized when     */
/* initscr() is called. */
```

```
#define  CENTERY   (LINES/2 -2)      /* The middle line in terminal   */
#define  CENTERX   (COLS/2 -2)       /* The middle column in terminal */

int main(int argc, char *argv[])
{
    int x,y;      /* to retrieve coordinates of cursor */
    int ch;       /* to receive user input character */
    int r, c;     /* to store coordinates of cursor */

    /* A string to display in the "status bar" at the bottom of the screen */
    char  MESSAGE[] = "Use the arrow keys to move the cursor. "
                      "Press F1 to exit";
    int   length;
    length = strlen(MESSAGE);  /* compute this once. */

    initscr();              /* Initialize screen */
    clear();                /* Clear the screen */
    noecho();               /* turn off character echo */
    cbreak();               /* disable line buffering */
    keypad(stdscr, TRUE); /* Turn on function keys */

    /* Move to bottom left corner of screen, write message there */
    move(LINES-1,0);
    addstr(MESSAGE);

    /* Start the cursor at the screen center  */
    r = CENTERY;
    c = CENTERX;
    move(r,c);

    /* Print the cursor's coordinates at the lower right */
    move(LINES-1,COLS-8);
    printw("(%02d,%02d)",r,c);
    refresh();

    /* Then move the cursor back to the center */
    move(r,c);

    /* Repeatedly wait for user input using getch(). Because we turned off */
    /* echo and put curses into cbreak mode, getch() will return without   */
    /* needing to get a newline char and will not echo the character.      */
    /* When the user presses the F1 key, the program quits.                */
    while((ch = getch()) != KEY_F(1)) {
        switch(ch) {

        /* When keypad() turns on function keys, the arrow keys are enabled */
        /* and are named KEY_X, where X is LEFT, RIGHT, etc.                 */
        /* This switch updates the row or column as needed, modulo COLS      */
        /* horizontally to wrap, and LINES-1 to wrap vertically without      */
        /* entering the sanctity of the status bar. */
            case KEY_LEFT:
                c = ( 0 == c )? COLS-1:c-1;
                break;
            case KEY_RIGHT:
```

```
                c = ( c == COLS-1 )? 0 : c+1;
                break;
            case KEY_UP:
                r = ( 0 == r )? LINES-2 : r-1;
                break;
            case KEY_DOWN:
                r = ( r == LINES-2 )? 0 : r+1;
                break;
            }

        /* Now we move the cursor to the new position, get its coordinates */
        /* and then move to the lower right to print the new position      */
        move(r,c);
        getyx(stdscr,y,x);
        move(LINES-1,COLS-8);
        printw("(%02d,%02d)",y,x);
        refresh();
        /* Now we have to move back to where we were ince the cursor was   */
        /* in the lower right after the printw(). */
        move(r,c);
      }

      endwin();         /* exit curses  */
      return 0;
}
```

## 6.6 Multiple Windows in NCurses

The next example program demonstrates how to use multiple windows. Note that this program does not use the standard screen.

Listing 6.7: drawmanygrids.c

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

#define  CENTERY    (LINES/2 -2)        /* The middle line in terminal   */
#define  CENTERX    (COLS/2 -2)         /* The middle column in terminal */
#define  NUMROWS    (LINES/2)           /* number of rows we use         */
#define  NUMCOLS    (COLS/2)            /* number of columns we use      */
#define  REPEATS    5
#define  GRIDCHARS ".*@+#"              /* should have REPEATS many chars */

int main(int argc, char *argv[])
{
    char     MESSAGE[] =
             "Type the character of the grid to bring it forward, 'q' to exit:";
    int      length, i, j, k;
    WINDOW *mssge_win;
    WINDOW *windows[REPEATS];
```

```
    char     gridchar[REPEATS] = GRIDCHARS;
    int      rowshift, colshift;
    int      ch;

    initscr();     /* Initialize screen */
    noecho();      /* turn off character echo */

    /* Make sure that the window is wide enough for message at the bottom.*/
    length = strlen(MESSAGE);
    if ( length > COLS - 2 ) {
        endwin();
        printf("This program needs a wider window.\n");
        exit(1);
    }

    /* Calculate the amount by which we shift each window when we draw it */
    rowshift = (LINES - NUMROWS)/5;
    colshift = (COLS - NUMCOLS)/5;

    /* In this loop, we create a new window, fill it with a grid of a unique
       characters
    */
    for ( j = 0; j < REPEATS; j++ ) {
        /* Create a new window at an offset from (0,0) determined by the
           row and column shift. */
        windows[j] = newwin(NUMROWS, NUMCOLS, rowshift*j, colshift*j);
        if ( NULL == windows[j] ) {
            endwin();
            fprintf(stderr, "Error creating window\n");
            exit(1);
        }

        /* Draw each grid row  as a string into windows[j] */
        for ( i = 0; i < NUMROWS; i++ ) {
            for ( k = 0; k < NUMCOLS; k++ ) {
                wmove(windows[j], i,k );
                if ( ERR == waddch(windows[j],  gridchar[j]) )
                    /* Ignore the error; it means we are in the
                       bottom right corner of the window and the
                       cursor was advance to a non-window position
                    */
                    ;
            }
        }
        /* Update the virtual screen with this window's content */
        wnoutrefresh(windows[j]);
    }
    /* Now send the virtual screen to the physical screen */
    doupdate();

    /*Create a window to hold a message and put it in the bottom row */
    mssge_win = newwin(1, COLS, LINES-1, 0);

    /* Write the message into the window; mvwaddstr positions the cursor */
```

```
    mvwaddstr(mssge_win,0,0, MESSAGE);
    wrefresh(mssge_win);

    while ( 1 ) {
        /*
           Read a character from the message window, not from stdscr. The
           call to wgetch forces a refresh on its window argument. If we
           refresh stdscr, our  grids will disappear.
        */
        ch = wgetch(mssge_win);  /* wait for the user to type something */
        if ( ch == 'q' )  /* time to quit */
            break;
        /* Check if they typed a grid character */
        for ( j = 0; j < REPEATS; j++ ) {
            if ( ch == gridchar[j]  ) {
                wmove(mssge_win,0,length);    /* move cursor to bottom */
                touchwin(windows[j]);         /* force the update */
                wrefresh(windows[j]);         /* refresh, bringing it forward */
                break;
            }
        }
    }
    clear();              /* clear the screen */
    endwin();             /* delete curses window and quit */
    return 0;
}
```

## 6.7 Adding Timing to Programs: Sleeps

To make images move or animate on the screen, the program has to control the rate at which images are changed or displayed, which implies their being able to access a time-of-day clock or a timer. You have already seen the `sleep()` system call. It is one method of controlling time. The problem with `sleep()` is that its base unit is a one-second interval, which is too coarse for most video. An alternative is the `usleep()` system call; `usleep()` has a granularity of one microsecond[3]. The problem with `usleep()` though is that it uses the real timer of the process, of which there is just one, so multiple simultaneous calls to `usleep()` will have unexpected results. Both `sleep()` and `usleep()` suffer from the fact that they may share the same timer as the `alarm()` system call. POSIX requires a call named `nanosleep()`, which has even finer granularity and is guaranteed not to interact with any other timers. Therefore, we will use `nanosleep()`:

```
#include <time.h>
int nanosleep(const struct timespec *req, struct timespec *rem);
```

The `timespec` structure is defined by

```
struct timespec {
```

[3]This does not mean that it will be implemented accurately to within a microsecond. The implementation of the timer may be inaccurate for small intervals because of context-switching.

```
        time_t tv_sec;      /* seconds */
        long   tv_nsec;     /* nanoseconds */
    };
```

The first argument specifies the amount of time that the caller should be suspended. The second argument can be used to store the amount of time remaining in case the caller is awakened by a signal. If it is a `NULL` pointer, it is ignored. For now we will pass a `NULL` pointer as the second argument.

The first example of animation alternates two images on the screen at regular intervals determined by the `nanosleep()` timer. It uses three functions defined in file `faces.c` :

```
void addsadface(int * y, int * x);       // Draws a "sad" face at (y,x)
void addhappyface(int * y, int * x);     // Draws a "happy" face at (y,x)
void eraseface(int * y, int * x);        // Erases face at (y,x)
```

that draw, respectively, a "sad face", a "happy face", and a blank face. The coordinates are initially the upper left corner of the rectangle enclosing the face. On return they store the upper right hand corner.

The main loop will repeatedly draw a face, park the cursor in the lower left-hand corner of the screen, call `refresh()`, and then sleep a bit. The sad and happy faces will alternate. The first version of the program, whose listing follows, uses a loop that runs forever and must be killed by the user's entering a `Ctrl-C`.

Listing 6.8: animateface0.c

```
#include <stdio.h>
#include <curses.h>
#include <string.h>
#include <time.h>      /* for struct timespec */
#include "faces.h"     /* The set of face drawing functions */


int main(int argc, char* argv[] )
{
    int r, c;
    int i = 0;
    char  MESSAGE[] = "Type Ctrl-C to exit:";
    char  BLANKS[] = "                              ";
    int   length;
    struct timespec sleeptime = {0,500000000};     /* 1/2 second */


    initscr();     /* Initialize curses library and the drawing screen */
    clear();       /* Clear the screen */

    /* Move to bottom of screen and post  message to display */
    move(LINES-1,0);
    addstr(MESSAGE);

    /* Loop repeatedly until user types any character */
```

```
    while (1) {
        /* move to center of screen */
        r = CENTERY;
        c = CENTERX;
        move(r,c);     /* move to that position to draw */

        /* Draw either a happy face or sad face at (r,c) */
        if (0 == i ) {
            addsadface(&r, &c);
            i = 1;
        }
        else {
            addhappyface(&r, &c);
            i = 0;
        }

        /* Park cursor at bottom */
        move(LINES-1,length);
        refresh();
        nanosleep(&sleeptime, NULL);     /* sleep  1/2 second */
    }

    /* Cleanup -- erase the face first */
    r = CENTERY;
    c = CENTERX;
    move(r,c);
    eraseface(&r, &c);
    /* erase the message at the bottom of the screen */
    move(LINES-1,0);
    addstr(BLANKS);
    refresh();

    endwin();          /* Delete NCurses window and quit */
    return 0;
}
```

## 6.8 Combining User Input and Timing

We can use the `halfdelay()` function in combination with timed sleeps to animate the face and also let the user enter input. Our program can call `halfdelay(1)` to cause reads to wait one-tenth of a second and use a controlled loop whose entry condition is simply `(ERR == getch())` to allow the user to type a character to stop the loop. As soon as the user types, the character will be buffered, and the next time the `getch()` is executed, the character will be removed and returned, and the condition will be false, breaking the loop.

We can also turn off echo within NCurses with the `noecho()` function. Putting this all together, we have the makings of `animateface.c` below.

Listing 6.9: animateface.c

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

```
#include <string.h>
#include <time.h>      /* for struct timespec */
#include "faces.h"     /* The set of face drawing functions */


int main(int argc, char* argv[] )
{
    int r, c;
    int i = 0;
    char  MESSAGE[] = "Press any character to exit:";
    char  BLANKS[] = "                              ";
    int   length;
    length = strlen(MESSAGE);
    struct timespec sleeptime = {0,500000000};     /* 1/2 second */


    initscr();     /* Initialize curses library and the drawing screen */
    clear();       /* Clear the screen */
    noecho();      /* Turn  off character echo */
    halfdelay(1); /* Turn on timed delay of 0.1 second -- if no char  */
                   /* within 0.1 sec,  getch() returns ERR */

    /* Move to bottom of screen and post  message to display */
    move(LINES-1,0);
    addstr(MESSAGE);

    /* Loop repeatedly until user types any character */
    while (ERR == getch()) {
        /* move to center of screen */
        r = CENTERY;
        c = CENTERX;
        move(r,c);     /* move to that position to draw */

        /* Draw either a happy face or sad face at (r,c) */
        if (0 == i ) {
            addsadface(&r, &c);
            i = 1;
        }
        else {
            addhappyface(&r, &c);
            i = 0;
        }

        /* Park cursor at bottom */
        move(LINES-1,length);
        refresh();
        nanosleep(&sleeptime, NULL);     /* sleep  1/2 second */
    }

    /* Cleanup -- erase the face first */
    r = CENTERY;
    c = CENTERX;
    move(r,c);
    eraseface(&r, &c);
```

```
    /* erase the message at the bottom of the screen */
    move(LINES-1,0);
    addstr(BLANKS);
    refresh();

    endwin();          /* Delete NCurses window and quit */
    return 0;
}
```

## 6.9 Timing with the alarm() and pause() system calls

The `sleep()` system call is based upon the use of *alarms*. An alarm in UNIX is essentially the software equivalent of a timer. (A timer goes off after a designated time interval; an alarm clock goes off at a designated clock time; in UNIX alarms are like timers.) When you want to snooze for an hour, you set a timer to wake you in an hour. In UNIX, a process can set an alarm to send itself a signal at some future time. It does this by calling `alarm()`, whose prototype is

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds)
```

`alarm()` sets a timer to expire in the number of seconds specified as its argument and returns immediately. If there is no pending alarm, the return value is 0. Otherwise the return value is the number of seconds remaining in the pending alarm. An alarm is *pending* if `alarm()` was called previously but the time period for which it was set has not yet elapsed. For example, suppose that at time 0 an alarm is set for 10 seconds:

```
alarm(10);
```

and that 4 seconds later, the same process calls `alarm()` again, this time asking for 20 seconds:

```
seconds_left = alarm(20); // called with 6 seconds remaining
```

The value returned by this call to `alarm()`, which is stored in `seconds_left`, would be 6. The alarm is reset to 20, and the alarm will signal the process 20 seconds later. The demo program `snoozealot.c` demonstrates how this works and how to use that return value. Before you look at `snoozealot.c`, take a look at the simpler program, `snooze.c`, which will be described shortly.

When an alarm's timer expires, a `SIGALRM` (there is no "A" between the L and R) signal is sent to the process that set the alarm. If the process does not provide a signal handler for the `SIGALRM`, or if for some other reason, the signal is not caught, then the `SIGALRM` will kill the process. The "other reason" can be that the process is in a system call that cannot be interrupted, or that it is handling some other signal at the time the `SIGALRM` hits it, and the particular handler is not designed to allow multiple signals to be received.

From this discussion you should realize that the `alarm()` call can be used to maintain at most one alarm at a time. If you want the effect of multiple alarms, then you have to code this into the `SIGALRM` handler; i.e. you have to reset the alarm for the new time.

In case it is not yet apparent, the `alarm()` system call has several different uses. A process can set an alarm prior to starting a long task that might not complete if the input data is unexpectedly large. The alarm will prevent the process from spending too much time on potentially endless tasks. It can also set an alarm to do a task asynchronously after a specific amount of time, perhaps based upon the state of its data.

A system call that is often used with alarms is the `pause()` call. When a process calls `pause()`, it is suspended and remains suspended until it receives a signal. Any signal will do to waken it. The prototype of `pause()` is:

```
#include <unistd.h>
int pause(void);
```

If a process calls `pause()` without having scheduled an alarm that will expire after the call to `pause()`, it will most likely never run again[4]. For example:

```
#include <unistd.h>
#include <stdio.h>
int main()
{
    pause();
    printf("You will never see this message!\n");
    return 0;
}
```

This program, when run, will stay blocked until either the kernel sends it a signal or a user does, and because there is no handler, it will take the default action on receiving the signal, which is usually to terminate, not ever reaching the `printf()` statement.

The demos directory contains several different examples to demonstrate the `alarm()`, `signal()`, and `pause()` calls. The `snooze.c` demo is similar to the UNIX `sleep` command. The `snoozealot.c` demo demonstrates how the alarm can be reused, how a signal handler for `SIGALRM` and for `SIGINT` can do program cleanup, and how to allow non-blocking user input while in a programmed loop that is counting down an alarm. The following demo is another example that focuses only on alarms but also records the times that they occur.

The program in the listing below, `alarmdemo1.c`, uses the `signal()` system call to install signal handlers. There is a second version of this program in the demos directory that does the exact same thing using `sigaction()` instead. It is useful to compare them.

Listing 6.10: alarmdemo1.c

```
#include <unistd.h>
#include <stdio.h>
#include <signal.h>
#include <time.h>
#include <stdlib.h>

/* This is the SIGALRM handler. When the SIGALRM is delivered to this */
```

[4]It might run if some other signal is delivered to it, for which it has a handler.

```
/* process, it resets the handler and displays the current time.      */
void catchalarm(int signo )
{
    signal(SIGALRM, catchalarm); /* install handler again */
    time_t t;                    /* time in seconds since the Epoch */
    struct tm *tp;               /* time struct with years, months, days,...*/

    time(&t);                    /* get current time as a time_t in t */
    tp = localtime(&t);          /* convert time to a tm struct */
    printf("Caught alarm at %d:%d:%d\n", tp->tm_hour, tp->tm_min ,tp->tm_sec );
}

int main(int argc, char * argv[])
{
    int k, sec;
    struct tm *tp;      /* time struct with years, months, days, etc */
    time_t t;           /* time in seconds since the Epoch */

    /* check proper usage */
    if (2 > argc) {
        printf("Usage: %s n\n",argv[0]);
        return -1;
    }

    k = atoi(argv[1]);             /* convert argv[1] to int (no error check) */
    signal(SIGALRM, catchalarm);   /* install catchalarm as the handler */

    time (&t);                     /* store current time in t */
    tp = localtime(&t);            /* store t as day,hours,minutes, etc. */

    /* print time at which alarm is set an how long it is set for */
    printf("Time is %d:%d:%d\n", tp->tm_hour, tp->tm_min ,tp->tm_sec );
    printf( "Alarm is set for %d seconds.\n",  k);
    sec = alarm(k);                /* set alarm  */

    pause();                       /* wait for a signal to arrive */
    return 0;
}
```

**Explanation**

The main program begins by installing a `SIGALRM` handler using the `signal()` call:

```
signal(SIGALRM, catchalarm);
```

The `catchalarm()` handler is unlike the earlier examples. Before it does anything else, it calls `signal()` to reinstall the handler. This is because signals of the same type will be lost while the process is handling a signal. The only way to catch a `SIGALRM` while in the handler for `SIGALRM` is to reissue the `signal()`. Although this particular program cannot issue another alarm, in general, signal handlers should be designed so that if a second signal of the same type arrives while they

are processing the first, they are not caught "by surprise" and possibly killed by the second signal. This handler is just demonstrating that technique.

The handler uses the `time()` and `localtime()` system calls to get the current time, convert it to a human readable format, and display it on the console. The main program displays the current time and immediately turns on the alarm by calling

```
alarm(k)
```

where `k` is the command line argument's numeric value. It then calls `pause()` to wait for the `SIGALRM` to be received. If a `SIGALRM` arrives before any other signal, it will cause `catchalarm()` to run, which will display the time. If another signal arrives first, the process will probably be killed.

## 6.10 Interval Timers

The time granularity, or resolution, of the `alarm()` system call is too coarse to be useful for many applications. Furthermore, `alarm()` must be called repeatedly if an alarm is to go off at regular intervals, such as when a process is timing the progress of some task. (Suppose you wanted to display some sort of speed indicator on the console, where instantaneous speed was measured by the amount of data written in a fixed time interval. You would need a timer of fine resolution and a `SIGALRM` catcher that would measure the amount of data processed and reinstall itself, but this would be slightly inaccurate because of time lapsed between the start of the handler and the time it took to reinstall itself.)

*Interval timers* were introduced in later Berkeley distributions of UNIX (4.2BSD) as well as in the SVR4(1170) versions of UNIX as a solution to this problem. An interval timer has two components: an *initial delay* and a *repeat interval*. The value of the initial delay is the amount of time the kernel should delay before sending the first signal to the process. The value of the repeat interval is the amount of time the kernel should wait between successive signals sent to the process. In other words, if an interval timer is started at time $t_0$, with initial delay $= x$ and repeat interval $y$, then it will generate signals at times $t_0 + x$, $t_0 + x + y$, $t_0 + x + 2y$, $t_0 + x + 3y$, $t_0 + x + 4y$, ... until the process terminates.

### 6.10.1 Three kinds of timers: Real, Virtual, and Profile

There are three different types of interval timers. One type of timer ticks during all elapsed time (like the clock on the wall); this is the *real timer*. The second ticks only when the process is in user mode (like the timer in a sporting event, which stops when play is paused for various reasons); this is the *virtual timer*. The last ticks when the process is in user mode or in system calls (like the timer in a professional chess game, which is stopped when one person has stopped it and the other has not yet started it[5]); it is called the *prof timer*. The constants used to define these timers, as you will shortly see in the documentation are:

`ITIMER_REAL` ticks always and sends a `SIGALRM` when it expires

[5] If the two people decide to take a coffee break, the timer is in the off state. You can think of user mode as your time and kernel mode as your opponent's time. Then this analogy fits.

`ITIMER_VIRTUAL` only ticks when the process is in user mode, i.e., not in system calls. It sends a `SIGVTALRM` when it expires.

`ITIMER_PROF` ticks during user mode and in system calls; on expiration sends a `SIGPROF` signal

The "`PROF`" in `SIGPROF` and `ITIMER_PROF` is short for *profile*, which is a snapshot of a process's time usage across all user mode and kernel mode activities. From these definitions, it follows that the time the process spends sleeping is its real time less its profile time, and that the time it spends in kernel mode is profile time less virtual time. Our interest is in real interval timers, those that tick like an ordinary alarm clock.

### 6.10.2 The Initial and Repeating Values

An `itimerval` structure contains two members: the value of the initial delay, and the value of the repeat interval:

```
struct itimerval {
    struct timeval it_interval; /* next value */
    struct timeval it_value;    /* current value */
};
```

The initial delay is stored in the `it_value` element and the repeat interval is stored in `it_interval`. As the timer ticks, the `it_value` element is decremented; when it reaches zero, a signal is sent to the process and the value of `it_interval` is copied into `it_value`.

Each member is of type `timeval`. A `timeval` structure represents a time interval using two elements: the number of *seconds* and the number of *microseconds* in the interval. There is no milliseconds field:

```
struct timeval {
    long tv_sec;            /* seconds */
    long tv_usec;           /* microseconds */
};
```

As a long integer is usually either 32 or 64 bits, depending upon the implementation, the `tv_usec` member is large enough to represent any number of microseconds from 0 to one million. Since it is common to work with time in milliseconds, you need to convert a time measured in milliseconds to a `timeval` with seconds and microseconds units. Mathematically, if $t$ is a time expressed in milliseconds, then

- $\lfloor t/1000 \rfloor$ is the number of whole seconds in $t$, and
- $1000 \cdot (t \mod 1000)$ is the number of microseconds in $t - \lfloor t/1000 \rfloor$

Therefore, the following C code fragment sets a `timeval` structure's fields, given an integer number `m` of milliseconds

```
timeval t;
t.tv_sec = m / 1000;
t.tv_usec = ( m - t.tv_sec * 1000 ) * 1000;
```

This avoids a second division using the modulo operator.

The `getitimer()` and `setitimer()` system calls work with interval timers. The former gets a timer's current value and the latter sets a timer's value.

```
#include <sys/time.h>
int getitimer(int which, struct itimerval *value);
int setitimer(int which, const struct itimerval *value,
struct itimerval *ovalue);
```

The first parameter to both is an integer constant that specifies the type of timer, one of the constants, `ITIMER_REAL`, `ITIMER_VIRTUAL`, or `ITIMER_PROF`. The second parameter to `getitimer()` is a pointer to the `itimerval` structure to be filled with its current values. The `it_value` element of this structure is given the time remaining on the timer, not the time it was originally set to be.

The `setitimer()` function's second parameter, `value`, is the address of the `itimerval` structure with which to set the timer, and the third, `ovalue`, if it is not `NULL`, is the address of a structure to be filled with its current values.

To stop a timer, set the initial and repeat intervals to 0. If the repeat is 0 but the initial delay is not, the timer sends a single signal and then stops. If the initial value is 0 the timer never starts, no matter what the repeat interval is.

The function `set_timer()`, below, can be used to set the value of an interval timer, given a time value expressed in milliseconds. It has three parameters, the type of timer to set, the number of milliseconds in the initial delay, and the number of milliseconds in the repeat interval.

Listing 6.11: set_timer()

```
int set_timer( int which, long initial, long repeat )
{
    struct itimerval itimer;
    long secs;

    /* initialize initial delay */
    secs = initial / 1000 ;
    itimer.it_value.tv_sec     = secs;
    itimer.it_value.tv_usec    = (initial - secs*1000 ) * 1000 ;

    /* initialize repeat interval */
    secs = repeat / 1000 ;
    itimer.it_interval.tv_sec  = secs;
    itimer.it_interval.tv_usec = (repeat - secs*1000 ) * 1000 ;

    return setitimer(which, &itimer, NULL);
}
```

The demo program, `timerdemo.c`, demonstrates how this function can be used. It accepts command line arguments so that you can control the initial and repeat delays, and the signal handler is designed to simply count how many signals are received and to quit after a pre-specified number of signals.

Listing 6.12: timerdemo.c

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <signal.h>
#include "timers.h"

void    count_alarms(int);

int main(int argc, char* argv[])
{
    int initial = 250;  /* default value */
    int repeat  = 500;  /* default value */

    if (argc >= 3) {
        initial = atoi(argv[1]);
        repeat  = atoi(argv[2]);
    }
    if ( initial == 0 || repeat == 0 ) {
        printf("Setting either interval to 0 hangs the process.\n");
        printf("Bailing out...\n");
        return 0;
    }

    signal(SIGALRM, count_alarms);
    if ( set_timer(ITIMER_REAL, initial, repeat) == -1 )
        perror("set_timer");
    else
        while( 1 )
            pause();
    return 0;
}

void count_alarms(int signum)
{
    int alarmsaccepted = 10;
    static int count = 0;
    printf("alarm %d \n", ++count);
    fflush(stdout);
    if ( alarmsaccepted == count ){
        printf("No more alarms allowed!\n");
        exit(0);
    }
```

```
}
```

If you run this program with various values as arguments, you will see how it works.

### 6.10.3 How Timers Are Implemented in UNIX

There is only one system clock. In contrast, there are many processes, and more than one of these might have active timers. The kernel maintains a data structure containing the timers of all processes. With each tick of the system clock, the kernel decrements each of the currently active per-process timers. If a process's timer reaches 0, the kernel sends the appropriate signal to the process and copies the `it_interval` value into `it_value`, provided that `it_interval` is not 0, effectively resetting the timer. If the `it_interval` is 0, the timer is stopped.

### 6.10.4 Timer Limitations and Precautions

Each process can have one of each kind of timer: a real timer, a virtual timer, and a profile timer, but only one of each. Although both the seconds value and the microseconds value are used to set the timer parameters, most operating systems will not give a process an interval of that exact amount of time because these are not real-time timers and because the operating system typically uses a time resolution on the order of a few milliseconds, not microseconds. UNIX systems that conform to SVR4 and to 4.4BSD specs do guarantee, though, to generate a signal no sooner than the requested time interval. The signal's delivery may be delayed on very heavily loaded systems. In addition, if a system is very heavily loaded, it is even a possibility that a later signal may fail to be delivered because the signal from an earlier timer expiration has not yet been delivered and so the second will be lost.

## 6.11 Timers and Signals in Video Games

So far we have seen how to create the illusion of movement on the screen using the NCurses library by erasing, repositioning, and drawing the same object, with a small time delay between repeated drawing. The `animateface.c` program used the `nanosleep()` function to achieve this time delay because `nanosleep()` provided a small enough time resolution and does not interfere with `SIGALRM` signals. If we want a video game to be interactive, however, then it has to respond to user inputs while creating the illusion that the action on the screen is independent of the user's actions. The method used by `animateface.c` will not work because when the program is waiting in the `nanosleep()` call, it is unable to respond to user inputs.

Instead, we can let a timer run in the background. At regular intervals, it can interrupt the process by sending a `SIGALRM` signal. All of the functionality to update the drawing can be put into the signal handler for the `SIGALRM` signal. However, there are dangers with extrapolating these ideas to programs in general, as is explained below.

### 6.11.1 Cautions About Signal Handler Design

The signal generation and delivery mechanism is a complex system with many nuances, and the programmer must be aware of them and must design the handlers with utmost care. The first issue is with respect to potential race conditions within the handlers themselves.

Because signal handlers cannot have any parameters other than the signal number or the structures passed to it by the kernel in the case of the newer `sa_sigaction()` style handlers, the only way that they can share data with the rest of the program is through global variables. For example, if the `SIGALRM` signal handler has to update the position of an object on the screen, then the handler needs read and write access to a variable that stores the object's current position. This variable must be either a static variable within the handler, or a global variable in the program. It must be a global if the variable needs to be accessed by other parts of the program outside of the handler. In either case, the variable cannot reside on the runtime stack because if it did, it would be destroyed between invocations of the handler. If the variable's contents are destroyed between invocations of the handler, there will be no means of animating the object.

In general, using signal handlers that have to access either global data or data that is not on the stack is a dangerous thing. If handled correctly in our video programs, there is little risk, but if this same strategy is used for programs in general, it can lead to unreliable and insecure programs. It can open up a Pandora's box of problems associated with the possibility of race conditions within the handler itself. This is because the handler might be re-entered as a result of another signal arriving while the handler is active. For example, if the handler is registered with the `SA_NODEFER` flag set, then it can be interrupted in the middle of its execution and variables within the handler might be in an inconsistent state as a result. Still worse, under certain circumstances, intruders could find ways to send the appropriate signal sequences to the program to force it to core dump and could use these dumps to gain root access (see Zalewski [3]).

A second issue pertains to certain system calls and library functions. Certain system calls and library functions are marked as safe, and the rest are unsafe. If a signal handler makes a call to a library function or a system call, and another signal causes it to be re-entered (because the signal was not masked or blocked) during the time the handler is in the call, the second invocation of the signal handler may also enter that same function. If it does, then the function will be re-entered as well, by the same process. If this function is not safe, then the data state of the handler will be corrupted and its execution no longer predictable. For example, in the following handler

```
void sighandler(int signum)
{
    ....
    printf("Running with uid=%d euid=%d\n",getuid(),geteuid());
    ...
}
```

if a second signal arrives while the first is in the `printf()` function, then both invocations will be using the `printf()` code, which is not re-entrant, and hence not safe. This means that the output of `printf()` may be corrupted. Far worse scenarios can result, making a system vulnerable to attack. POSIX.1-2004 requires that the following functions can be safely called within a signal handler:

```
_Exit(), _exit(), abort(), accept(), access(), aio_error(), aio_return(),
aio_suspend(), alarm(), bind(), cfgetispeed(), cfgetospeed(), cfsetispeed(),
cfsetospeed(), chdir(), chmod(), chown(), clock_gettime(), close(), connect(),
creat(), dup(), dup2(), execle(), execve(), fchmod(), fchown(), fcntl(),
fdatasync(), fork(), fpathconf(), fstat(), fsync(), ftruncate(), getegid(),
geteuid(), getgid(), getgroups(), getpeername(), getpgrp(), getpid(),
```

```
getppid(), getsockname(), getsockopt(), getuid(), kill(), link(), listen(),
lseek(), lstat(), mkdir(), mkfifo(), open(), pathconf(), pause(), pipe(),
poll(), posix_trace_event(), pselect(), raise(), read(), readlink(), recv(),
recvfrom(), recvmsg(), rename(), rmdir(), select(), sem_post(), send(),
sendmsg(), sendto(), setgid(), setpgid(), setsid(), setsockopt(), setuid(),
shutdown(), sigaction(), sigaddset(), sigdelset(), sigemptyset(), sigfillset(),
sigismember(), signal(), sigpause(), sigpending(), sigprocmask(), sigqueue(),
sigset(), sigsuspend(), sleep(), socket(), socketpair(), stat(), symlink(),
sysconf(), tcdrain(), tcflow(), tcflush(), tcgetattr(), tcgetpgrp(),
tcsendbreak(),tcsetattr(), tcsetpgrp(), time(), timer_getoverrun(),
timer_gettime(), timer_settime(), times(), umask(), uname(), unlink(),
utime(), wait(), waitpid(), write().
```

POSIX.1-2008 removes `fpathconf()`, `pathconf()`, and `sysconf()` from the preceding list, and adds the following functions to it:

```
execl(), execv(), faccessat(), fchmodat(), fchownat(), fexecve(), fstatat(),
futimens(), linkat(), mkdirat(), mkfifoat(), mknod(), mknodat(), openat(),
eadlinkat(), renameat(), symlinkat(), unlinkat(), utimensat(), utimes()
```

In general, I/O functions are not safe to invoke inside signal handlers.

Non-reentrant functions are functions that cannot safely be called, interrupted, and then recalled before the first call has finished without resulting in memory corruption. An easy way to think of a function being re-entrant is that every single variable used by that function is stored on the run time stack, including any return value. It uses no static variables and no globals. Each time it is invoked, the new invocation has its own set of variables.

A signal handler would have to completely remove all possibility of its being interrupted if it contained a call to an unsafe function within it. This is not realistic. If an unsafe function is in the middle of execution when a signal arrives, and the handler for this signal also calls an unsafe function, then the result of execution becomes undefined, meaning all bets are off about what will happen. This is an even more compelling reason to avoid unsafe functions within handlers.

Three general rules to follow when designing signal handlers, whenever possible, as recommended by Wheeler[2], are:

1. Where possible, have your signal handlers unconditionally set a specific flag and do nothing else.

2. If you must have more complex signal handlers, use only calls specifically designated as being safe for use in signal handlers. In particular, don't use `malloc()` or `free()` in C (which on most systems aren't protected against signals), nor the many functions that depend on them (such as the `printf()` family and `syslog()`). You could try to "wrap" calls to insecure library calls with a check to a global flag (to avoid re-entry), but I wouldn't recommend it.

3. Block signal delivery during all non-atomic operations in the program, and block signal delivery inside signal handlers.

In addition to these recommendations, I would add one more:

- Use the `SA_RESTART` flag when possible, to avoid the possibility of system calls being interrupted and terminated, which may then cause the program to exit abnormally, and check the return value of all system calls when using signal handlers in a program.

We will not be able to adhere to rule 1 in some of our demo programs because they are designed to produce output or change program state during the handler calls to illustrate various principles, but we can stick to rules 2 and 3.

### 6.11.2 A Demonstration

We will develop a simple program to illustrate the first method of animation. Our program, called `bouncestr.c`, moves a string, in this case a worm-like fellow, horizontally across the screen, from left to right and then back again. It is called `bouncestr.c` because each time the poor guy hits the "wall", he bounces back in the opposite direction. The program allows the user to control the game with three different keys:

- Typing '`f`' speeds up the motion;
- Typing '`s`' slows down the motion;
- Typing a space character reverses the direction of the worm.

The *speed* is the number of character positions that our object will move each second. For example, a speed of 6 means that it moves 6 columns (i.e., characters) per second. When the user presses the `f` or `s` key, the speed should increase or decrease linearly, up to some reasonable limits. For example, if $v$ is the current speed, then one press of `f` should mean $v = v + 2$. The changes in speed are handled by changing the intervals in the interval timer and resetting it with `setitimer()` whenever the user presses the '`f`' or '`s`' key.

This first method of animation will make the main program in charge of getting user input, and use timers and signal handlers to interrupt the main program and update the worm's position on the screen. Because timers will interrupt the main program loop whenever they occur, there is a good chance they will interrupt the `read()` system call that is invoked within the `getch()` code to get user input. For this reason, they must be established with the `SA_RESTART` flag, to restart these calls and not lose the user's input. There is no need to make the input non-blocking; doing so would waste needless CPU cycles asking the kernel if input is available. But line buffering should be disabled, as well as echo and line-editing. Therefore, the program will turn on `cbreak` mode and turn off echo.

The program needs one global variable:

```
int direction;
```

to store the direction of movement (left or right, by one cell), and the signal handler needs two static variables:

```
int row, col;
```

which store the current position at which to draw. The logic in the main program's loop handles the input events, as follows.

Listing 6.13: Main processing loop of bouncestr.c

```
while ( !done )  {
    is_changed = FALSE;
    c = getch();
    switch (c) {
        case 'Q':
        case 'q':
            done = 1;
            break;
        case ' ':
            direction = ( direction == LEFT)? RIGHT:LEFT;
            break;
        case 'f':
            if ( 1000/speed > 2 ) {          /* if interval > 2 */
                speed = speed + 2;           /* increase        */
                is_changed = TRUE;
            }
            break;
        case 's':
            if (1000/speed <=  500 ) {       /* if interval <= 500 */
                speed = speed - 2 ;          /* decrease           */
                is_changed = TRUE;
            }
            break;
    }
    if ( is_changed ) {
        set_timer( ITIMER_REAL, 1000/speed, 1000/speed );
    }
}
```

**Notes**

1. `is_changed` lets us know whether to reset the timer.
2. Blocking input is on, so the `getch()` can never return without data. The loop just has to check which character was typed.
3. Speed is the reciprocal of the interval length, in the same way that frequency is the inverse of period with a periodic function (like a wave). If we want a frequency of $k$ signals every 1000 milliseconds, then the interval between each signal must be $1000/k$. Similarly, if we want an object to be moved $k$ times each second ( equivalently $k$ times each 1000 ms), then the interval to give to the interval timer must be $1000/k$ ms. Since speed contains the current number of chars per second for moving the object, the interval to give to the timer is $1000/speed$, since the `set_timer()` function (defined in Listing 6.11 ) is expecting the interval expressed in milliseconds.

4. As a concrete example, if *speed* = 10, then the timer must expire every 1000/10 = 100 milliseconds, in order that we can move the object 10 times per second. If speed is increased by 2, to 12 chars/second, then the interval must be 1000/12 ≈ 83 milliseconds, so that the timer will expire every 83 milliseconds.

5. We make sure we avoid a division by zero, by preventing the user from decrementing speed below 1, which is achieved by making sure $1000/speed \leq 500$. We set an upper bound on the speed simply because high speeds are not easy to watch.

The logic of redrawing is now in the signal handler, `move_msg()`, shown below:

Listing 6.14: move_msg()

```
void move_msg(int signum)
{
    static int row = ROW;
    static int col = 0;
    char   mssge[40];
    mvaddstr( row, col, BLANK );  /* erase old string      */
    col += dir;                   /* advance one column    */
    move( row, col );             /* move to new locataion */
    if ( RIGHT == dir ) {
        addstr( MESSAGE );        /* add forward string    */
        if ( col+strlen(MESSAGE) >= COLS-1 )
            dir = LEFT;           /* reverse if hitting edge */
    }
    else {
        addstr( REVMSSGE );       /* add reverse string      */
        if ( col <= 0 )
            dir = RIGHT;          /* reverse if hitting edge */
    }

    move( LINES-1, 0);
    sprintf(mssge, "Current speed: %d (chars/sec)", speed);
    addstr(mssge);
    refresh();
}
```

Note that

- The handler uses static variables, a.k.a globals, making it non-re-entrant.
- It makes calls to several functions that are not safe.

However, because this is a `SIGALRM` handler, and the time intervals are extremely long relative to the length of the code, it is essentially impossible for a `SIGALRM` signal to be delivered while the handler is running. That is why it is effectively safe. Of course you can send it multiple Ctrl-C's and it will be unsafe for them.

If the signal handler were installed using the `signal()` system call, the handler would have to reset itself by calling `signal()` immediately. We use the `sigaction()` call instead. If the user were allowed to speed up the animation enough, the `SIGALRM` signals might arrive so fast that they would arrive before the handler has finished executing. In this case, the handler's behavior would

be unsafe. By using `sigaction()`, we can make sure that signals are blocked while the program is executing inside the handler.

The program without the main processing loop from Listing 6.13 is below.

Listing 6.15: Main program of bouncestr.c

```
#include  <stdio.h>
#include  <string.h>
#include  <curses.h>
#include  <signal.h>
#include  "timers.h"

#define     INITIAL_SPEED    50
#define     RIGHT            1
#define     LEFT             -1
#define     ROW              12
#define     MESSAGE          "ooooooo=>"
#define     REVMSSGE         "<=ooooooo"
#define     BLANK            "         "

int         dir;          /* Global variable to store direction of movement */
int         speed;        /* Current speed in chars/second */

int main()
{
    int    done;
    int    is_changed;
    int    c;

    /* Set up signal handling  */
    struct sigaction newhandler;       /* for installing handlers       */
    sigset_t         blocked;          /* to set mask for handler */

    newhandler.sa_handler = move_msg; /* name of handler  */
    newhandler.sa_flags = SA_RESTART; /* flag is just RESTART */
    sigemptyset(&blocked);            /* clear all bits of blocked set */
    newhandler.sa_mask = blocked;     /* set this empty set to be the mask */

    if ( sigaction(SIGALRM, &newhandler, NULL) == -1 ){  /* try to install */
        perror("sigaction");
        return (1);
    }

    /* Prepare the terminal for the animation */
    initscr();      /* initialize the library and screen */
    cbreak();       /* turn off line buffering and editing */
    noecho();       /* turn off echo */
    clear();        /* clear the screen */
    curs_set(0);    /* hide the cursor */

    /* Initialize the parameters of the program */
    dir    = RIGHT;
    done   = 0;
    speed  = INITIAL_SPEED ;
```

```
    /* Start the real time interval timer with delay interval size */
    set_timer( ITIMER_REAL, 1000/speed, 1000/speed );

    /* main processing loop omitted but would be here */

    endwin();
    return 0;
}
```

## 6.12 Non-polling Input

The `bouncestr.c` program uses a timer to generate interrupts to update the screen, but it obtains the user's input through what is essentially a polling loop: the main program repeatedly polls the terminal for input. This is fine if the CPU is not going to be used by any other process or if the program does not have other tasks to perform in the main loop, but it is an inefficient method of checking for the availability of input, which is extremely infrequent in the life of a processor. We should be unsatisfied with the idea that our program is a CPU hog, stuck in a polled I/O loop, even if the process is blocked each time it calls `getch()` to check for user input. The process basically calls `getch()`, blocks, is awakened when the user types, does a bit of work and blocks again, over and over. It would be more efficient if the input part of the program were also signal-driven, meaning that the program would ask the kernel to notify it when input could be delivered to it, perhaps through the signal-handling mechanism. In this case, the program would be essentially idle, waiting for a signal of any kind, either from the timer to update the screen, or from the kernel because input was available.

There are two different types of non-polling input: *signal-driven* and *asynchronous*. To understand the difference between them, it is important to know that input is first moved from a device to a buffer in the kernel's address space, and from there to the process's address space.

- In signal-driven I/O, the program tells the kernel to notify it when input has been placed into the kernel's address space. Once the process is notified that the input is in the kernel's address space, if it makes a `read()` system call, because the data is immediately available, it will not block. In other words, a `read()` executed after the process is notified is guaranteed to return immediately with data.
- In asynchronous I/O, the process tells the kernel to notify it when input has been moved from the device to the kernel's address space and then into a buffer in the process's address space. In this type of I/O, when the process receives the signal, the `read()` has already been executed, and the user process has the data already, but not necessarily in the memory location into which it must go.

Signal-driven I/O is available in UNIX by setting the `O_ASYNC` flag in the file descriptor and then establishing appropriate signals. Asynchronous I/O is available through the POSIX Asynchronous I/O Interface (AIO). It is a bit confusing that the flag to enable signal-driven I/O is called `O_ASYNC`. We will first explore signal-driven I/O by modifying the `bouncestr.c` program. Then we will create a version of the `bouncestr.c` program that uses asynchronous I/O with the AIO interface.

### 6.12.1 Non-polling I/O Using the `O_ASYNC` Flag

When you set the `O_ASYNC` flag on a file descriptor, it causes input from the descriptor's file connection to be partially delivered asynchronously. To be precise, it means that when input is available on the device, it is copied by the kernel into a location in the kernel's address space, after which the kernel sends a `SIGIO` signal to the process. To set up signal-driven input by this method, the program must do the following:

1. Tell the kernel which process should be sent the `SIGIO` signal when the data is ready to read by calling

   ```
   fcntl(SETOWN, getpid());
   ```

   The `SETOWN` operation makes the process-id in the second argument the owner of the signal to be received. Usually the program wants to receive the signal itself, so it calls this with `getpid()`.

2. Retrieve the existing flags on the standard input device with

   ```
   fcntl(0, F_GETFL);
   ```

3. Set the `O_ASYNC` flag on the connection with

   ```
   fcntl(0,F_SETFL,(fd_flags|O_ASYNC));
   ```

4. Assuming that `on_input()` is the function that will handle the `SIGIO` signal, register that signal handler:

   ```
   struct sigaction newhandler;
   sigset_t         blocked;
   newhandler.sa_handler = on_input;
   newhandler.sa_flags = SA_RESTART;
   sigemptyset(&blocked);
   newhandler.sa_mask = blocked;
   sigaction(SIGIO, &newhandler, NULL);
   ```

   The `on_input()` handler can call the NCurses `getch()` function and will be guaranteed to receive the single character input by the user. This way it does not have to be in a loop doing a blocking read and can instead do other things in the loop.

This is all put together in the program `bouncestr_async.c`. The first three of the above steps can be put into a function called `enable_keybd_signals()`:

```
void enable_keybd_signals()
{
    int   fd_flags;

    fcntl(0, F_SETOWN, getpid());
    fd_flags = fcntl(0, F_GETFL);
    fcntl(0, F_SETFL, (fd_flags|O_ASYNC));
}
```

### 6.12.2 The `bouncestr.c` Program Using `O_ASYNC`: Flawed Version

Sometimes it is worth writing a bad program in order to understand how to write a good program. This is such an exercise. The program (excluding the `#includes` and parts that are identical to the `bouncestr.c` program's ) is shown below. The program is terminated within the `on_input()` handler when it receives the quit input character. This is because, once the program has started, it cannot be terminated by turning off the timer or by setting the control variable of the loop to 1. This will be explained later.

The tasks of the main program are:

1. Establish the signal handlers.
2. Initialize NCurses (in `cbreak` mode with no echo).
3. Initialize the data state of the program (speed, direction, rows, columns, etc)
4. Set up keyboard signals.
5. Start the interval timer.
6. Display the messages on the last line and loop until it is time to quit.

The program:

Listing 6.16: A flawed bouncestr_async.c

```
...
/*  <---- snip ---->  */
int         row;            /* current row           */
int         col;            /* current column         */
int         dir;            /* Global variable to store direction of movement */
int         speed;          /* Current speed in chars/second */
volatile    sig_atomic_t    finished;

void     on_alarm(int);         /* handler for alarm   */
void     on_input(int);         /* handler for SIGIO   */
void     enable_kbd_signals(); /* setup for SIGIO     */

int main( int argc, char * argv[])
{
    struct sigaction newhandler;       /* for installing handlers       */
    sigset_t         blocked;          /* to set mask for handler */

    /* Set up signal handling  */
    newhandler.sa_handler = on_input; /* name of handler  */
    newhandler.sa_flags = SA_RESTART; /* flag is just RESTART */
    sigemptyset(&blocked);            /* clear all bits of blocked set */
    newhandler.sa_mask = blocked;     /* set this empty set to be the mask */
    if ( sigaction(SIGIO, &newhandler, NULL) == -1 ) {
        perror("sigaction");
        return (1);
    }
```

```
    sigemptyset(&blocked);              /* clear all bits of blocked set */
    sigaddset(&blocked, SIGIO);
    newhandler.sa_mask = blocked;       /* set this empty set to be the mask */
    newhandler.sa_handler = on_alarm; /* SIGALRM handler function */
    if ( sigaction(SIGALRM, &newhandler, NULL) == -1 ){  /* try to install */
        perror("sigaction");
        return (1);
    }

    /* Prepare the terminal for the animation */
    initscr();      /* initialize the library and screen */
    cbreak();       /* put terminal into non-blocking input mode */
    noecho();       /* turn off echo */
    clear();        /* clear the screen */
    curs_set(0);    /* hide the cursor */

    /* Initialize the parameters of the program */
    row         = ROW;
    col         = 0;
    dir         = RIGHT;
    finished    = 0;
    speed       = INITIAL_SPEED ;

    /* Turn on keyboard signals */
    enable_kbd_signals();

    /* Start the real time interval timer with delay interval size */
    set_timer( ITIMER_REAL, 1000/speed, 1000/speed );

    mvaddstr(LINES-1, 0, "Current speed:");
    refresh();

    /* Put the message into the first position and start */
    mvaddstr(row, col,   MESSAGE);

    while( 0 == finished  ) {
        pause();
    }
    endwin();
    return 0;
}

void on_input(int signum)
{
    int     c;
    int     is_changed = 0;
    char    mssge[40];

    c = getch();
    switch (c) {
        case 'Q':
        case 'q':
            finished = 1;                   /* quit program */
            clear();
```

```
            endwin();
            /* exit(0);  UNCOMMENT THIS IF YOU WANT IT TO WORK!!! */
            break;
        case ' ':
            dir = (LEFT == dir)? RIGHT:LEFT;  /* reverse direction */
            break;
        case 'f':
            if ( 1000/speed > 2 ) {           /* if interval > 2 */
                speed = speed + 2;            /* increase  */
                is_changed = 1;
            }
            break;
        case 's':
            if (1000/speed <  500 ) {         /* if interval <= 500 */
                speed = speed - 2 ;           /* decrease */
                is_changed = 1;
            }
            break;
    }
    if ( is_changed ) {
        set_timer( ITIMER_REAL, 1000/speed, 1000/speed );
        sprintf(mssge, "Current speed: %d (chars/sec)", speed);
        mvaddstr(LINES-1, 0, mssge);
    }

    move( LINES-1, COLS-12);
    sprintf(mssge, "Last Char:%c", c);
    addstr(mssge);
    refresh();
}


void on_alarm(int signum)
/* same as in bouncestr.c, and so omitted here  */

void enable_keybd_signals()
{
    int  fd_flags;

    fcntl(0, F_SETOWN, getpid());
    fd_flags = fcntl(0, F_GETFL);
    fcntl(0, F_SETFL, (fd_flags|O_ASYNC));
}
```

**Notes.**

1. The biggest difference between this and the `bouncestr.c` program is that the input handling is entirely inside the `on_input()` handler.

2. The `enable_keybd_signals()` function sets up the asynchronous input on file descriptor 0.

3. This program must call `exit()` from within the handler, otherwise it will never terminate. If you modify the `on_input()` handler so that when a '`q`' is typed, all it does is to set the `finished` flag

to 1, the program will not stop. In fact, the main program will continue to see the value 0 stored in `finished`. You can go one step further and delete the main loop completely, and the program will animate forever. In other words, the signal handler for `SIGALRM` continues to run and the `endwin()` call is never reached. This problem is not related to NCurses, nor to the timers.

The problem is that, as the man page for `fcntl()` notes, " a `SIGIO` signal is sent whenever input or output becomes possible on that file descriptor." From various experiments I have carried out, I have determined that the problem is that, when the terminal is in non-canonical mode and signal-driven input has been set up on the input descriptor of the terminal (file descriptor 0), if the main program or the `SIGIO` handler attempts output on the terminal device, it corrupts the `SIGIO` signal mechanism so that when the `SIGIO` handler terminates, instead of returning to the main program, execution will resume in the handler again, as if the `SIGIO` signal was not cleared from the process's state. So the process continues to execute only in the input handler and the `SIGALRM` handler if it has not been blocked. If one removes all output instructions of any kind from the signal handler and the main program to "slow devices", meaning the screen, then the program will work correctly. The behavior of programs that issue writes within the handler or the main program to the screen is apparently undefined.

4. The signal handler for the `SIGIO` must not block `SIGALRM`, or else the animation will disappear. The `SIGALRM` handler can block `SIGIO` signals though.

What follows is a better version of this same program, also using a `SIGIO` signal handler that adheres to all safety rules noted above, and that works correctly.

### 6.12.3 The `bouncestr.c` Program Using `O_ASYNC` : A Proper Solution

In this version, all code has been removed from the `SIGIO` signal handler except to set the value of a state variable of type `volatile sig_atomic_t` that the main program checks. According to the *CERT Secure Programming Standard, SIG31* [1],

> Accessing or modifying shared objects in signal handlers can result in race conditions that can leave data in an inconsistent state. The exception to this rule is the ability to read and write to variables of `volatile sig_atomic_t`. The need for the `volatile` keyword is described in rule DCL34-C. Use `volatile` for data that cannot be cached. It is important to note that the behavior of a program that accesses an object of any other type from a signal handler is undefined.
>
> The type `sig_atomic_t` is the integer type of an object that can be accessed as an atomic entity, even in the presence of asynchronous interrupts. The type of `sig_atomic_t` is implementation defined, though it provides some guarantees. Integer values ranging from `SIG_ATOMIC_MIN` through `SIG_ATOMIC_MAX`, inclusive, may be safely stored to a variable of the type.

Further details can be found on the CERT website or in the cited reference.

In our program, if the state variable is set, then the main loop calls `getch()` to get the input, and then calls a function to process the input. Otherwise it blocks itself on `pause()`. The listing follows.

Listing 6.17: bouncestr_async2.c: A safe version of bouncestr_async.c

```
// #includes omitted here
```

```
#define     INITIAL_SPEED    30
#define     RIGHT            1
#define     LEFT             -1
#define     ROW              12
#define     MESSAGE          "ooooooo=>"
#define     REVMSSGE         "<=ooooooo"
#define     BLANK            "         "

int         dir;            /* Global variable to store direction of movement */
int         speed;          /* Current speed in chars/second */
volatile    sig_atomic_t    input_ready;

/************************************************************************/
/*                      Signal Handler Prototypes                       */
/************************************************************************/
void  on_alarm(int);                                    /* handler for alarm   */
void  on_input(int);                                    /* handler for keybd   */

/* This is not a signal handler -- it consolidates logic for updating */
int   update_from_input(int c,int *speed,int *dir );

/************************************************************************/
/*                                  Main                                */
/************************************************************************/
int main( int argc, char * argv[])
{

    struct sigaction newhandler;        /* for installing handlers */
    sigset_t         blocked;           /* to set mask for handler */
    int     fd_flags;
    int     c;
    int     finished;

    /* Set up signal handling  */
    newhandler.sa_handler = on_input; /* name of handler                  */
    newhandler.sa_flags = SA_RESTART; /* flag is just RESTART             */
    sigemptyset(&blocked);            /* clear all bits of blocked set    */
    newhandler.sa_mask = blocked;     /* set this empty set to be the mask */
    if ( sigaction(SIGIO, &newhandler, NULL) == -1 ) {
        perror("sigaction");
        return (1);
    }

    sigemptyset(&blocked);            /* clear all bits of blocked set */
    newhandler.sa_mask = blocked;     /* set this empty set to be the mask */
    newhandler.sa_handler = on_alarm; /* SIGALRM handler function */
    if ( sigaction(SIGALRM, &newhandler, NULL) == -1 ){  /* try to install */
        perror("sigaction");
        return (1);
    }

    /* Prepare the terminal for the animation */
    initscr();      /* initialize the library and screen */
```

```
    cbreak();       /* put terminal into non-blocking input mode */
    noecho();       /* turn off echo */
    clear();        /* clear the screen */
    curs_set(0);    /* hide the cursor */

    /* Initialize the parameters of the program */
    dir         = RIGHT;
    finished    = 0;
    speed       = INITIAL_SPEED ;
    input_ready = 0;

    /* Turn on keyboard signals */
    fcntl(0, F_SETOWN, getpid());
    fd_flags = fcntl(0, F_GETFL);
    fcntl(0, F_SETFL, (fd_flags|O_ASYNC));

    /* Start the real time interval timer with delay interval size */
    set_timer( ITIMER_REAL, 1000/speed, 1000/speed );

    /* Put a message in bottom row with current speed. */
    mvaddstr(LINES-1, 0, "Current speed:");

    /* Put the message into the first position and start */
    mvaddstr(ROW, 0,   MESSAGE);

    while( !finished  ) {
        if ( input_ready ) {
            c = getch();
            finished = update_from_input(c, &speed, &dir);
            input_ready = 0;
        }
        else
            pause();
    }
    clear();
    endwin();
    return 0;
}

/*****************************************************************************/

int update_from_input( int c, int *speed, int *dir )
{
    int is_changed = 0;
    char  mssge[40];
    switch (c) {
        case 'Q':
        case 'q':
            return 1;                          /* quit program */
        case ' ':
            *dir = (LEFT == *dir )? RIGHT:LEFT;  /* reverse direction */
            break;
        case 'f':
            if ( 1000/(*speed) > 2 ) {           /* if interval > 2 */
```

```
                *speed = *speed + 2;            /* increase  */
                is_changed = 1;
            }
            break;
        case 's':
            if (1000/(*speed) <  500 ) {        /* if interval <= 500 */
                *speed = *speed - 2 ;           /* decrease */
                is_changed = 1;
            }
            break;
    }
    if ( is_changed ) {
        set_timer( ITIMER_REAL, 1000/(*speed), 1000/(*speed) );
        sprintf(mssge, "Current speed: %d (chars/sec)", (*speed));
        mvaddstr(LINES-1, 0, mssge);
    }

    move( LINES-1, COLS-12);
    sprintf(mssge, "Last Char:%c", c);
    addstr(mssge);
    refresh();
    return 0;
}

/*****************************************************************************/

void on_input(int signum)
{
    input_ready = 1;
}

/*****************************************************************************/

void on_alarm(int signum)
{
    static int row = ROW;
    static int col = 0;

    mvaddstr( row, col, BLANK );  /* erase old string */
    col += dir;                   /* advance one column  */
    move( row, col );             /* move to new locataion */
    if ( RIGHT == dir ) {
        addstr( MESSAGE );        /* add forward string  */
        if ( col+strlen(MESSAGE) >= COLS-1 )
            dir = LEFT;           /* reverse if hitting edge */
    }
    else {
        addstr( REVMSSGE );       /* add reverse string        */
        if ( col <= 0 )
            dir = RIGHT;          /* reverse if hitting edge */
    }
    refresh();
}
```

### 6.12.4 The `bouncestr.c` Program Using AIO

The AIO interface is a POSIX interface that provides asynchronous I/O. Whereas setting the `O_ASYNC` flag on a file descriptor causes a signal to be sent when data is available to be read, using the AIO interface causes a signal to be sent when the data has actually been read and placed into a user buffer. The `aio_read()` call is an asynchronous read. In essence, it places a read request in the I/O device driver's queue and returns immediately, as indicated in the man page:

```
#include <aio.h>
int aio_read(struct aiocb *aiocbp);

The  aio_read()  function  requests  an asynchronous
     "n = read(fd, buf, count)"
with   fd,   buf,   count   given   by    aiocbp->aio_fildes, aiocbp->aio_buf,
aiocbp->aio_nbytes, respectively.  The return status n can be retrieved upon
completion using aio_return(3).

The data is read starting at the absolute file offset aiocbp->aio_offset,
regardless of the current file position.  After this request, the value of
the current file position is unspecified.

The "asynchronous" means that this call returns as soon as the  request has
been enqueued; the read may or may not have completed when the call returns.
One tests for completion using aio_error(3).
```

When the request is satisfied, the driver sends a `SIGIO` signal. The signal handler can process the input and then issue a new `aio_read()` call to get more data.

The program must

- create a buffer to store the input data, and
- fill an `aiocb` structure with appropriate values before issuing the first read.

An `aiocb` structure has the following members:

```
int             aio_fildes      //File descriptor.
off_t           aio_offset      // File offset.
volatile void  *aio_buf         //Location of buffer.
size_t          aio_nbytes      //Length of transfer.
int             aio_reqprio     //Request priority offset.
struct sigevent aio_sigevent    // Signal number and value.
int             aio_lio_opcode  //Operation to be performed.
```

A program does not have to assign a value to the `aio_reqprio` member, but all others must be initialized. The following function, `setup_aio_buffer()`, demonstrates how to set up a read of a single character at a time into a buffer named `input`. It is given a pointer to an `aiocb` structure and fills its members with the required data. The main program can then give the address of this structure to the `aio_read()` function.

Listing 6.18: setup_aio_buffer()

```
void setup_aio_buffer(struct aiocb *aio_buf)
{
    static char input[1];                   /* 1 char of input  */

    /* describe what to read  */
    aio_buf->aio_fildes     = 0;      /* file descriptor for I/O */
    aio_buf->aio_buf        = input;  /* address of buffer for I/O          */
    aio_buf->aio_nbytes     = 1;      /* number of bytes to read  each time */
    aio_buf->aio_offset     = 0;      /* offset in file to start reads   */

    /* describe what to do when read is ready  */
    aio_buf->aio_sigevent.sigev_notify = SIGEV_SIGNAL;
    aio_buf->aio_sigevent.sigev_signo  = SIGIO;       /* send SIGIO    */
}
```

The main program should declare the `aiocb` structure so that it is visible to the various functions that must access it. The program follows.

Listing 6.19: bouncestr_aio.c

```
#include  <unistd.h>
#include  <stdio.h>
#include  <curses.h>
#include  <signal.h>
#include  <string.h>
#include  <aio.h>
#include  "timers.h"

#define    INITIAL_SPEED    30
#define    RIGHT            1
#define    LEFT             -1
#define    ROW              12
#define    MESSAGE          "ooooooo=>"
#define    REVMSSGE         "<=ooooooo"
#define    BLANK            "         "

int       dir;            /* Global variable to store direction of movement */
int       speed;          /* Current speed in chars/second */
volatile  sig_atomic_t  input_ready;

struct aiocb kbcbuf;        /* an aio control buf    */

void move_msg(int signum);   /* handler for alarm     */
void on_input(int);          /* handler for keybd     */
int  update_from_input( int *speed, int *dir );


/*****************************************************************************/
/*                        Signal Handler Prototypes                          */
/*****************************************************************************/

/* SIGALRM signal handler -- it is responsible for animating the string */
void    move_msg(int);
```

```
/* SIGIO signal handler -- it is responsible for retrieving user input  */
void    setup_aio_buffer(struct aiocb *aio_buf);

/*************************************************************************/
/*                                  Main                                 */
/*************************************************************************/

int main(int argc, char* argv[])
{
     struct sigaction newhandler;           /* new settings         */
     sigset_t         blocked;              /* set of blocked sigs  */
     int              finished;

     newhandler.sa_handler = on_input;      /* handler function     */
     newhandler.sa_flags = SA_RESTART;      /* options      */

     /* then build the list of blocked signals  */
     sigemptyset(&blocked);                 /* clear all bits       */
     newhandler.sa_mask = blocked;          /* store blockmask      */
     if ( sigaction(SIGIO, &newhandler, NULL) == -1 )
         perror("sigaction");

     newhandler.sa_handler = move_msg;      /* handler function     */
     if ( sigaction(SIGALRM, &newhandler, NULL) == -1 )
         perror("sigaction");

    /* prepare the terminal for the animation */
    initscr();      /* initialize the library and screen */
    cbreak();       /* put terminal into non-blocking input mode */
    noecho();       /* turn off echo */
    clear();        /* clear the screen */
    curs_set(0);    /* hide the cursor */

    /* Initialize the parameters of the program */
    dir        = RIGHT;
    finished   = 0;
    speed      = INITIAL_SPEED ;

    /* initialize aio buffer for the first read and place call */
    setup_aio_buffer(&kbcbuf);
    aio_read(&kbcbuf);

    /* Start the real time interval timer with delay interval size */
    set_timer( ITIMER_REAL, 1000/speed, 1000/speed );

    mvaddstr(LINES-1, 0, "Current speed:");
    refresh();

    /* Put the message into the first position and start */
    mvaddstr(ROW, 0,   MESSAGE);

    while( !finished )
        if ( input_ready ) {
            finished = update_from_input(&speed, &dir);
```

```
            input_ready = 0;
        }
        else
            pause();
    clear();
    endwin();
    return 0;
}

/*****************************************************************************/
/*                              SIGIO Signal Handler                         */
/*****************************************************************************/

void on_input(int signo)
{
    input_ready = 1;
}


/*  Handler called when aio_read() has stuff to read */
/*  First check for any error codes, and if ok, then get the return code */

int update_from_input( int *speed, int *dir )
{
    int  c;
    int is_changed = 0;
    char *cp = (char *) kbcbuf.aio_buf;      /* cast to char *  */
    char   mssge[40];
    int finished=0;

    /* check for errors  */
    if ( aio_error(&kbcbuf) != 0 )
        perror("reading failed");
    else
        /* get number of chars read  */
        if ( aio_return(&kbcbuf) == 1 ) {
            c = *cp;
            /*ndelay = 0; */
            switch (c) {
                case 'Q':
                case 'q':
                    finished = 1;                    /* quit program */
                    break;
                case ' ':
                    *dir = (*dir == LEFT)? RIGHT:LEFT;  /* reverse direction */
                    break;
                case 'f':
                    if ( 1000/(*speed) > 2 ) {         /* if interval > 2 */
                        *speed = *speed + 2;           /* increase  */
                        is_changed = 1;
                    }
                    break;
                case 's':
                    if (1000/(*speed) <  500 ) {       /* if interval < 500 */
```

```
                        *speed = *speed - 2 ;          /* decrease */
                        is_changed = 1;
                    }
                    break;
                }
            if ( is_changed ) {
                set_timer( ITIMER_REAL, 1000/(*speed), 1000/(*speed) );
                sprintf(mssge, "Current speed: %d (chars/sec)", *speed);
                mvaddstr(LINES-1, 0, mssge);
            }
            /* write the status line message */
            move( LINES-1, COLS-12);
            sprintf(mssge, "Last Char:%c", c);
            addstr(mssge);
            refresh();
        }
    /* place a new request  */
    aio_read(&kbcbuf);
    return finished;
}

/*****************************************************************************/
/*                            SIGALRM Signal Handler                         */
/*****************************************************************************/

/* SIGALRM handler -- moves string on the screen when the signal is received */
void move_msg(int signum)
{
    static int row = ROW;
    static int col = 0;
    mvaddstr( row, col, BLANK );  /* erase old string */
    col += dir;                   /* advance one column  */
    move( row, col );             /* move to new locataion */
    if ( RIGHT == dir ) {
        addstr( MESSAGE );        /* add forward string  */
        if ( col+strlen(MESSAGE) >= COLS-1 )
            dir = LEFT;           /* reverse if hitting edge */
    }
    else {
        addstr( REVMSSGE );       /* add reverse string        */
        if ( col <= 0 )
            dir = RIGHT;          /* reverse if hitting edge */
    }
    refresh();
}

/*****************************************************************************/
/*                        Asynchronous I/O Library Setup                     */
/*****************************************************************************/

/*  The following function initializes the AIO structure to enable */
/*  asynchronous I/O through the AIO library. */
void setup_aio_buffer(struct aiocb *aio_buf)
/* Same as in Listing above */
```

### 6.12.5 Simulating Multiple Timers

Even though a process can have only a single timer, it is still possible to animate an unlimited number of independently moving objects. The key is to simulate in the program exactly what the kernel does with its timers relatively to the system clock. The idea is to create an array of the objects to be animated. Each entry of the array can have all of the information needed to animate a single object, and in particular, the length of the interval between movements of that object. In the world of animation, these movable creatures are called *sprites*, so I will call them that here.

In essence, a sprite can be represented by a structure such as the one below.

```
struct  sprite
{
    int      interval;       // number of time units between redraws
    int      counter;        // counter for elapsed time between redraws
    char     shape;          // shape used to draw object
    position display_pos;    // current location on screen (int,int)
    position real_pos;       // current real location  (double, double)
    double   dx;             // current x-coordinate of direction
    double   dy;             // current y-coordinate of direction
};
```

For each tick of the process's single interval timer, it can iterate through an array of sprites, decrementing each counter. If any counter reaches 0, it copies the interval value into it and issues a request to move the sprite in the (`dx,dy`) direction from position `real_pos`. The particular implementation above uses two positions, a real position and a display position. The idea is to keep track of the actual position as a floating point value, and display it in the cell in which its center of mass resides. The real position is updated by the (`dx,dy`) value and then the display position is calculated from that. The (`dx,dy`) pair is a vector of length 1 that is added to the real position of the point. The display cell is obtained by rounding the `x` and `y` values to the nearest integer. Moving a sprite can be accomplished with the following function.

Listing 6.20: move_sprite()

```
void move_sprite(sprite *sp)
{
    erase_sprite(*sp);

    sp->real_pos.y += sp->dy;
    sp->real_pos.x += sp->dx;
    sp->display_pos.r = (int)( sp->real_pos.y + 0.5);
    sp->display_pos.c = (int)( sp->real_pos.x + 0.5);
    draw_sprite(*sp);

    if  ( ( sp->real_pos.y > LINES-0.5 ) && ( sp->dy > 0 ) )
        sp->dy = -sp->dy;
    else if ( ( sp->real_pos.y < 0.0 ) && ( sp->dy < 0 ) )
        sp->dy = -sp->dy;
    if ( ( sp->real_pos.x > COLS-0.5 ) && ( sp->dx > 0 ) )
```

```
        sp->dx = -sp->dx;
    else if ( ( sp->real_pos.x < 0.5 ) && ( sp->dx < 0 ) )
        sp->dx = -sp->dx;
}
```

An unsafe `SIGALRM` signal handler would be as follows:

```
void update_all(int signum)
{
    int k;
    for ( k = 0; k < NUM_OBJS; k++ )
        if ( --object[k].counter == 0 ){
            move_sprite(&(object[k]));
            object[k].counter = object[k].interval;
        }
    move(LINES-1, COLS-1);
    refresh();
}
```

The rest of this program is relatively easy to piece together.

### 6.12.6 Summary

This chapter introduced the NCurses library as a means for controlling the user's terminal in a simpler and more powerful way than was possible by modifying terminal driver attributes. It barely scratched the surface of the library's interface. It also introduced timers as a way of introducing timed events and motion. Finally, it introduced several different models of input/output, including signal-driven and asynchronous I/O, an well as several different models of terminal processing, such as raw, cbreak, and non-canonical mode.

More efficient processing and better control can be achieved by the use of multiple processes. This is the topic of the next chapter.

# Bibliography

[1] Robert C. Seacord and Jason A. Rafail. The cert c secure coding standard, 2008.

[2] David A. Wheeler. Secure programming for linux and unix howto, 2003.

[3] Michal Zalewski. Delivering signals for fun and profit, 2001.