CSC 357 Programming Assignment 5
`timer` -- An Interactive Timer Program

ISSUED: Monday, 21 May 2007
DUE: On or before 11:59:59PM Tuesday 29 May 2007, via `handin` on falcon/hornet
POINTS POSSIBLE: 100
WEIGHT: 4% of total class grade
READING: Lecture Notes Week 8, Stevens Chapter 10, cited `man` pages

Specification

In this assignment you are implementing a simple timer program, named `timer`. The program takes a single integer command-line argument for the starting time, in seconds. A positive argument means the timer runs in count-down mode, subtracting seconds as it runs. A negative argument means the counter runs in count-up mode, adding seconds as it runs.

As `timer` runs, it continuously displays the time on one line[1] in H:MM:SS format until it reaches `0:00:00`. When it reaches zero, it sounds the bell, prints "`Beeeep! Time's up!`", moves to the next line, and continues displaying the time. If in count-down mode, the continued time is negative; in count-up mode, it's positive.

While running, `timer` responds to the following single-character commands:

Char	Command	Description
`q`	quit	the program exits
`c`	clear	clears the timer (sets it to 0:00:00)
`r`	run/stop	toggles the timer between running and stopped
`d`	display on/off	toggles the timer between display and no-display
`h`	hours	adds an hour to the clock
`H`		subtracts an hour from the clock
`m`	minutes	add a minute to the clock
`M`		subtracts a minute from the clock
`s`	seconds	add a second to the clock
`S`		subtract a second from the clock

The following are specific details of the specification:

- If the argument is missing, not an integer, zero, or out of range, `timer` prints an error message to `stderr` and exits with nonzero status.
- `timer` starts in a running state.
- When the timer is toggled to "stopped" mode, the counter itself stops, as does the display updating. When the timer is re-toggled to "running" mode, the timer resumes counting and displaying at the time it was stopped.
- When the timer is toggled to "no-display" mode, the counter itself remains running, but the display is not updated. When the timer is re-toggled to "display" mode, the display resumes with the current timer value, which has been updated by the number of seconds the display was off.
- The time displayed is the mathematical ceiling of (nearest integer greater than or equal to) the number of seconds.

[1] The trick to single-line output is to use the '`\r`' character to print a carriage-return without a newline. That way each line of output overwrites the previous line.

That is, if the internal time reads 0.5s, it displays as `0:00:01`, while -0.5s displays as `0:00:00`.

- The minutes and seconds fields are always displayed as two digits. The hours field has as many digits as necessary.
- `timer` must set the terminal to use non-canonical I/O to read each character immediately when it is pressed and must not echo them. In addition, it must turn off keyboard signal generation. That is, it does not terminate on a Ctrl-C or suspend on Ctrl-Z.
- `timer` must also set the terminal back to the way it found it when done.
- When the time expires, `timer` sounds the terminal's bell by printing a `bel` character (ASCII character 7), and the message "`Beeep! Time's up!`". It then continues the count on the following line.
- `timer` displays the time in the following format:

Character Position	**Contents**
1	a "twirler" (see below)
2	a space
3	a space or a '-' if the time is negative
4-*n*	the time in H:MM:SS format

- In the first column, the timer displays a "twirler" device, where the displayed character changes every time the clock ticks. This way, it is possible to determine at a glance whether the clock is running or not.

  The device is created by cycling through the characters of the string "`|/-\`". The effect is that of a spinning bar. You may or may not choose to have the twirler change when other actions occur. (Mine does. It was just simpler to update this field every time the clock is displayed, so it also changes on clearing or adjusting the time.)
- `timer` must use `setitimer(2)` to generate the `SIGALRM`s for each clock tick. In order for the twirler to look good, the granularity of the timer must be finer than one tick per second, specifically 1/25 second. This value should be defined as the constant `TICKS_PER_SECOND`.
- `timer` must be careful to block ALRM signals while updating the clock, and to unblock them afterwards.

Sample Execution

Since this program overwrites itself on the screen, it is difficult to show examples on paper. There are a few examples in the table below. To see how things look on the screen, use the executable version of program on hornet in

```
~gfisher/classes/357/programs/5/testing/timer
```

Examples of bad usage:	`% timer`
	`usage: timer <seconds>`
	`% timer 1 2`
	`usage: timer <seconds>`
	`% timer x`
	`"x" is not a number.`
	`% timer 0`
	`Invalid time (0). Must be non-zero.`

The timer one second after expiring:	`% timer 4`
	`Beeeep! Time's up!`
	`\ -0:00:01`

The timer after quitting:	`% timer 4`
	`Beeeep! Time's up!`
	`| -0:00:03`

Implementation Suggestions

The example presented in the Week 8 lecture notes is intended to be a starter for your implementation of `timer`. The code for the lecture example is in `357/examples/simple-timer.c`

The following C library functions may be particularly useful in your implementation. You can read about these in Stevens and the man pages.

Function	**Description**
`ceil`	in the math library; be sure to compile with `-lm`
`setitimer`	setting an interval timer
`sigaction`	setting up a signal handler
`sigemptyset` `sigaddset`	manipulating signal sets used by `sigaction`
`tcgetattr` `tcsetattr`	getting and setting terminal parameters
`fflush`	flushing a stdio stream

Stevens Chapters 10 and 18 cover some key aspects of these functions.

Debugging a program with asynchronous events can be tricky. You may want to develop your program piecewise to be sure you can handle signals and raw IO independently before assembling it into the final timer. You can start with the Lecture 8 example as the base.

I recommend adding another command (say, '`t`') that calls your `tick()` function so you can debug your program without having to run real-time. Remember also that while debugging, you can also generate SIGALRMs via `kill(1)` rather than from the timer.

Also, ***remember that stdio is buffered.*** If you write something to the console with `putchar()` or `printf()`, it may not appear immediately. You should either use unbuffered writes, or flush the output stream with `fflush` after writing.

Deliverables

You must submit a set of `.c` and `.h` files plus a `Makefile` that compiles your program into an executable named `timer`. The files must follow the 357 design and implementation conventions.

Scoring Details

The testing plan file in the 357 `program/5` directory has the precise point breakdown.

Collaboration

NO collaboration is allowed on this assignment. Everyone must do their own individual work.

How to Submit the Deliverable

Submit your deliverable using the `handin` program on hornet. For Section 1 of 357, the command is

```
handin gfisher prog5-s1 ... Makefile
```

For Section 2, it's

```
handin gfisher prog5-s2 ... Makefile
```

where "..." are your program files.