Title: Implementing Software Timers

By: Don Libes

Originally appeared in the Nov. 1990 "C User's Journal" and is also

reprinted as Chapter 35 of "Obfuscated C and Other Mysteries",

John Wiley & Sons, 1993, ISBN 0-471-57805-3.

http://www.wiley.com/compbooks/m3.html.

---------------------------------------------------------------------

This column will describe a set of functions to implement software

timers. What are software timers and why do you need them? Software

timers make up for inherent limitations in hardware timers. For

example, while most computers have clock hardware, you can typically

only have the clock trigger an interrupt for one time in the future.

When running multiple tasks, you will want to have the clock keep

track of multiple timers concurrently so that interrupts can be

generated correctly even if the time periods overlap. Operating

systems do this all the time.

Robert Ward discussed the related problem of building a general

purpose scheduler ("Practical Schedulers for Real-Time Applications")

in the April 1990 CUJ. In the "Additional Ideas" section, Robert

described the usefulness of a timer scheduling queue. "Events can

specify the timing of other events by putting a timer programming

request in a special queue." That is exactly what the software in

this column will do. (Thanks for the lead in, Robert.) You may want

to reread at least the beginning of his column right now, although it

isn't really necessary.

The code in this column has other uses as well. For example, you can

use it to simulate multiple timers in environments such as a UNIX

process which only allows the user one software timer. Even if you

aren't interested in software timers, I think you will find this an

intriguing column. Using simple techniques and data structures, this

C code produces very powerful results. The code was very tricky to

get right, and my commentary should be interesting if only as some

more practice in reading and writing C code.

Timers

By implementing the timers as a separate piece of software, we can

reduce the complexity of the scheduler. Some people like this kind of

modularization, and some don't. Similarly some operating systems do

this, and some don't. I like it. It makes the code easier to write,

to read, and to correct (oops).

The basic idea of a timer is that they allow tasks to be run at some

time in the future. When their time arrives, they are scheduled to be

run. The responsibility of actually running them is then turned over

to someone else, such as the scheduler. In order to communicate with

the scheduler, we'll set up a common data structure called a timer

(listing 1). I've also included a few other miscellaneous definitions

that will be needed later on. For instance, the TIME typedef is used

to declare all relative time variables. You can complete this

definition based on what your needs are.

#include <stdio.h>

#define TRUE 1

#define FALSE 0

#define MAX\_TIMERS ... /\* number of timers \*/

typedef ... TIME; /\* how time is actually stored \*/

#define VERY\_LONG\_TIME ... /\* longest time possible \*/

struct timer {

int inuse; /\* TRUE if in use \*/

TIME time; /\* relative time to wait \*/

char \*event; /\* set to TRUE at timeout \*/

} timers[MAX\_TIMERS]; /\* set of timers \*/

listing 1

Each timer will be represented by a timer struct. The set of timers

will be maintained in an array, timers. The first element of each

timer declares whether the timer is in use. The second element of a

timer is the amount of time being waited for. As time passes, this

will be periodically updated. event is a pointer to a value that is

initially set to 0. When it is time to run the task, \*event is set to

1. We can imagine that the scheduler also keeps an event pointer.

Every so often, it reexamines it. When it finds it has been set to 1,

it knows that the timer has expired and the associated task can be

run.

[Notice how simple this is. Other schedulers or other scheduler data

structures could enable runnability, without worrying or even knowing

about timers.]

The code in listing 2 initializes the timers. It runs through the

array setting each inuse flag to FALSE. This for loop will become

idiomatic to you by the end of this column.

void

timers\_init() {

struct timer \*t;

for (t=timers;t<&timers[MAX\_TIMERS];t++)

t->inuse = FALSE;

}

listing 2

Now we can write the routines to schedule the timers. First, I'll

show timer\_undeclare, which is a little simpler than its counterpart,

timer\_declare.

There are a variety of ways to keep track of the timers. Machines

which don't have sophisticated clock hardware usually call an

interrupt handler at every clock tick. The software then maintains

the system time in a register, as well as checking for timer entries

that have expired.

More intelligent machines can maintain the clock in hardware, only

interrupting the CPU after a given time period has expired. By having

the clock interrupt for when an event is waiting, you can get a

tremendous speedup. This technique is also common in software

simulations and thread implementation.

Reading the clock may require an operating system call, but for our

purposes we will assume the variable time\_now to be automatically

updated by the hardware for just this purpose. volatile indicates

that the variable should not be cached in a register but read from

storage each time.

volatile TIME time\_now;

We will define several variables for shorthands. timer\_next will

point to the timer entry that we next expect to expire.

time\_timer\_set will contain the system time when the hardware timer

was last set.

struct timer \*timer\_next = NULL;/\* timer we expect to run down next \*/

TIME time\_timer\_set; /\* time when physical timer was set \*/

void timers\_update(); /\* see discussion below \*/

void

timer\_undeclare(t)

struct timer \*t;

{

disable\_interrupts();

if (!t->inuse) {

enable\_interrupts();

return;

}

t->inuse = FALSE;

/\* check if we were waiting on this one \*/

if (t == timer\_next) {

timers\_update(time\_now - time\_timer\_set);

if (timer\_next) {

start\_physical\_timer(timer\_next->time);

time\_timer\_set = time\_now;

}

}

enable\_interrupts();

}

Listing 3

Undeclaring Timers - Why and How?

timer\_undeclare does just what its name implies, it undeclares a

timer. Undeclaring timers is actually an important operation in some

applications. For example, network code sets timers like crazy. In

some protocols, each packet sent generates a timer. If the sender

doesn't receive an acknowledgement after a given interval, the timer

forces it to resend a packet. If the sender does receive an

acknowledgement, it undeclares the timer. If things are going well,

every single timer declared is later undeclared.

timer\_undeclare (listing 3) is performed with interrupts disabled.

This is necessary because we are going to have an interrupt handler

that can access the same data. Because this data is shared, access

must be strictly controlled. I've shown the interrupt manipulation as

a function call, but you must use whatever is appropriate to your

system. This is very system dependent.

timer\_undeclare starts by checking the validity of the argument as a

timer entry. We will see later that the system clock can implicitly

undeclare timer entries. Thus we must make a reasonable attempt to

assure ourselves that a timer to be undeclared is still declared.

Once assured the timer is valid, timer\_undeclare marks the entry

invalid. If the timer happens to be the very one next expected to

expire, the physical timer must be restarted for the next shorter

timer. Before doing that, all the timer entries have to be updated by

the amount of time that has elapsed since the timer was last set.

This is done by timers\_update which also calculates the next shortest

timer. Looking for the shortest timer in that function is a little

obscure but happens to be very convenient since timers\_update has to

look at every timer anyway.

timers\_update (listing 4) goes through the timers, subtracting the

given time from each. If any reach 0 this way, they are triggered by

setting the event flag. Any lag in the difference between when a

timer was requested and timers\_update is called, is accounted for by

basing the latency against time\_now and also collecting timers that

have "gone negative" in timers\_update. (Why might a timer go

negative?) Lastly, we also remember the lowest nonzero timer to wait

for as timer\_next.

timer\_last is just a temporary. It is a permanently non-schedulable

timer that will only show up when all the other timers have been

scheduled.

/\* subtract time from all timers, enabling any that run out along the way \*/

void

timers\_update(time)

TIME time;

{

static struct timer timer\_last = {

FALSE /\* in use \*/,

VERY\_LONG\_TIME /\* time \*/,

NULL /\* event pointer \*/

};

struct timer \*t;

timer\_next = &timer\_last;

for (t=timers;t<&timers[MAX\_TIMERS];t++) {

if (t->inuse) {

if (time < t->time) { /\* unexpired \*/

t->time -= time;

if (t->time < timer\_next->time)

timer\_next = t;

} else { /\* expired \*/

/\* tell scheduler \*/

\*t->event = TRUE;

t->inuse = 0; /\* remove timer \*/

}

}

}

/\* reset timer\_next if no timers found \*/

if (!timer\_next->inuse) timer\_next = 0;

}

listing 4

Declaring Timers

timer\_declare (listing 5) takes a time and an event address as

arguments. When the time expires, the value that event points to will

be set. (This occurs in timers\_update under the comment /\* tell

scheduler \*/.) timer\_declare returns a pointer to a timer. This

pointer is the same one that timer\_undeclare takes as an argument.

struct timer \*

timer\_declare(time,event)

unsigned int time; /\* time to wait in 10msec ticks \*/

char \*event;

{

struct timer \*t;

disable\_interrupts();

for (t=timers;t<&timers[MAX\_TIMERS];t++) {

if (!t->inuse) break;

}

/\* out of timers? \*/

if (t == &timers[MAX\_TIMERS]) {

enable\_interrupts();

return(0);

}

/\* install new timer \*/

t->event = event;

t->time = time;

if (!timer\_next) {

/\* no timers set at all, so this is shortest \*/

time\_timer\_set = time\_now;

start\_physical\_timer((timer\_next = t)->time);

} else if ((time + time\_now) < (timer\_next->time + time\_timer\_set)) {

/\* new timer is shorter than current one, so \*/

timers\_update(time\_now - time\_timer\_set);

time\_timer\_set = time\_now;

start\_physical\_timer((timer\_next = t)->time);

} else {

/\* new timer is longer, than current one \*/

}

t->inuse = TRUE;

enable\_interrupts();

return(t);

}

listing 5

As with its counterpart, interrupts are disabled in timer\_declare to

prevent concurrent access to the shared data structure.

The first thing timer\_declare does is to allocate a timer. If none

are available, a NULL is returned so that the caller can fail or retry

later.

Once a timer is allocated and initialized, we must check if the

physical timer must be changed. There are three cases:

1) There are no other timers;

In this case, we go ahead and start the physical timer with the

time of this timer.

2) There are other timers, but this new one is the shortest of all the others;

In this case, we must restart the physical timer to the new time.

But before we do that, we must update all the other timers by the

amount of time that has elapsed since the physical timer was last

set.

3) There are other timers, and this new one is not the shortest.

There is nothing to do in this case. However, for legibility it is

broken into its own case which contains only a comment. That way

it is clear what is going on when the previous else-if test fails.

Before enabling interrupts and returning, the timer's inuse flag is

set. The reason it is done afterwards rather than with the earlier

timer settings is that this prevents timers\_update from updating it

with a time period that occurred before it was even declared.

Handling Timer Interrupts

The only remaining routine is the interrupt handler (listing 6)

actually called when the physical clock expires. When the interrupt

handler is called, we are guaranteed that the time described by

timer\_next has elapsed.

void

timer\_interrupt\_handler() {

timers\_update(time\_now - time\_timer\_set);

/\* start physical timer for next shortest time if one exists \*/

if (timer\_next) {

time\_timer\_set = time\_now;

start\_physical\_timer(timer\_next->time);

}

}

listing 6

Each time the interrupt handler is called, a timer has expired. By

calling timers\_update, all the timers will be decremented and any

timers that have expired will have their event flags enabled. This

will also set up timer\_next so that the physical timer can be

restarted for the next timer we expect to occur.

Let's examine one special case. Suppose we have only one timer set

up. Now imagine that we have called timer\_undeclare and just as

interrupts are disabled, the physical clock ticks down all the way.

Since interrupts are disabled, the interrupt will be delivered

immediately after interrupts are enabled. But they will be enabled

after the timer has been deleted. So we see a situation where an

interrupt will be delivered for a timer that no longer exists. What

occurs in the interrupt handler?

timers\_update is called. It finds nothing to update. As a

consequence of this, timer\_next is set to 0. The remainder of the

interrupt handler already handles the case of no remaining timers, and

the handler returns normally.

This is an example of the kind of special casing you have to keep in

mind when writing the code. (In fact, my first implementation didn't

handle this right, and it was painful to debug. Debuggers don't work

very well when fooling around with interrupts!)

Conclusion

I have presented an implementation of timers. The code is carefully

designed so that it is relatively free of special demands it places on

a scheduler. For example, it doesn't close off the scheduler from

using a different kind of timer at the same time.

One thought that may have occurred to you while reading this, is why

the timers are maintained as an array rather than say, a linked list.

Using a linked list would avoid the overhead of stepping through

arrays (which can be almost entirely empty). Keeping the list sorted

by time would make the timers\_update function much simpler.

On the other hand, it would complicate the other functions. For

example, timer\_undeclare, would either require you to use a

doubly-linked list, or to search the entire list from the beginning

each time. Another point is that real-time systems typically avoid

dynamic structures to begin with. For example, using malloc/free from

a process-wide heap can cause an indeterminate amount of time that is

difficult to estimate. If I was to recode this using linked lists, I

would use a malloc implementation from a small pool of timer-only

buffers, which in effect is very similar to what I've done here with

arrays. There would be a tradeoff in space and time, which you might

prefer or not depending upon your application.

If you decide to recode or just modify my implementation, be very

careful. Always imagine the worst thing that can happen when two

processes attempt to access the same data structure at the same time.

Happy interruptions!

Thanks

Debugging timing routines is very different than other code, since

unrelated events in the computer can make your programs behave

differently. Even putting in printf statements can change critical

execution paths. It is extremely aggravating when problems disappear

only when you are debugging. Furthermore, most debuggers do not work

well when interrupts are disabled. Ed Barkmeyer was of great help

debugging the timer code and teaching me to persevere when I saw code

behaving in ways that had to be impossible. Thanks to Sarah Wallace

and Randy Miller who debugged this column and also forced me to make

all the explanations much clearer.