FEATURE

Techies' Guide to C Part 17

This month we'll look at the potentially contentious... and confusing... issue of time as it appears in the C language.

Steve Rimmer

Human beings have evolved a fairly simple system of dealing with the time. One can simply look at a watch or ask someone and move on to greater things. Computers have clocks, but no watches. As such, they have come to use a rather more complex system of timekeeping.

In asking for the correct time, we make all sorts of assumptions which a computer cannot. It's unspoken that one usually wants the correct time for the current time zone, and with whatever daylight savings time adjustments as are in use at the moment added to it. Likewise, the date is usually given in terms of the popular Gregorian calendar, as opposed to, say, the less frequently used Mayan one. These are all things that computers need to be told.

In addition to these purely human concerns, computers have a few of their own. For example, our system of dates refers to the number of years between the arbitrary year one and now. Considering that the middle ages saw a fairly sparse use of personal computers, most computer operating systems have seen fit to place the beginning of time at some rather more contemporary point. Computer time and dates are calculated as a number of seconds relative to an arbitrary start of the universe. Naturally, this start time varies between systems, as we'll see.

The format of the time and date values for computers vary a lot too. If you plan to work with files or other things having time and date stamps from varying systems, this will probably cause your brain to hurt after a while.

This month, let's see how the sands of time can be comprehended by your C programs without getting to your disk drives and voiding your system warrantee. These examples pertain to Borland's Turbo C implementation, although they should work with minor fiddling under most PC based C compilers.

Double Time

On a PC under C, time is figured as being the number of seconds from January 1, 1970 GMT and the present. As this will be a fairly large number... there are about thirty-one million seconds in a year... time is stored in long integers.

Under C, the time and the date are derived from this single long integer. The easiest way to have this long integer passed to your program is like this.

```
long t;

time(&t);
```

There are a few things to note about this. First of all, you must include the *time.h* header in any program you want to work with time functions in. Secondly, note that for reasons which aren't terribly obvious, the *time* function does not usually return a long integer, but rather stores the time value in a long integer whose address is passed to it.

You can actually make it return a long value instead if you use it like this.

```
t=time(NULL);
```

This number, while containing the exact time, is not in a terribly useful form. There are numerous C functions to change this, although the one you use will no doubt require a bit of forethought. Several of them seem to have identical uses.

The easiest way to see what time it is involves the *ctime* function. This bit of code illustrates its use.

```
long t;
time(&t);
printf(ctime(&t));
```

The result of this call would be something like

```
Mon Apr 02 12:03:33 1991
```

Now, there are a number of important things to note about the string returned by *ctime*. The first is that you must have the *time.h* header in your program so that your compiler will know that *ctime* does, indeed, return a pointer to a string rather than an integer. Secondly, the string it returns is always exactly twenty-six characters long. The twenty-fifth of these will be a newline character, which is often inconvenient.

You might want to dispense with this.

```
long t;
char b[26];

time(&t);
strcpy(b,ctime(&t));
b[24]=0;
```

The *ctime* function will work with two global variables which are present whenever the *time.h* header is present in your program. You should set *timezone* to represent the number of seconds between GMT and your local time zone. Eastern standard time is five hours removed from GMT, so we would set *timezone* like this.

```
timezone=5*3600;
```

The value 3600 is sixty minutes multiplied by sixty seconds. Note that you don't have to declare *timezone* explicitly... it is provided for you by the compiler.

You can also adjust for daylight savings time if you like. Set *daylight* to a

non-zero value if you're presently using daylight savings time.

Now, bear in mind that the *time* function retrieves the time from your PC's clock. If you have set the clock in your computer to local time, you won't want to use the *timezone* and *daylight* values. They're only used if you keep the clock in your system set to GMT, in which case the displayed time can reflect the geographical location of the computer using your program.

If you write a program which requires that its time be adjusted by its users to reflect their displacement from GMT, you'll be interested in a function called *tzset*. When it's called, it searches the DOS environment for an environment variable called TZ and, if it discovers one, adjusts the *timezone* and *daylight* values accordingly.

To make this work, you must start at the DOS prompt and type in a SET command for the TZ variable. You can add this to your AUTOEXEC.BAT file if you want to make it permanent. Type the following:

```
SET TZ=EST5EDT
```

This is a fairly complex string. The first three characters represent the local time zone, in this case EST for Eastern Standard time. The number represents the number of hours between GMT and the local time zone. This would be 5 or +5 for Eastern Standard Time, 8 or +8 for Pacific time or -1 for the time in France, for example. Finally, the last three characters are optional, and represent the daylight savings time zone code... you'd only include EDT, in this case, if daylight savings time were in effect. This would cause the *daylight* variable to be set to a non-zero value.

By using an environment variable rather than hard coding these values into your program, you make it possible for your users to adjust the time zone values externally. All you have to do is to make sure your place the *tzset* function early in your program, before any time calculations occur.

Bear in mind that most people do not set their system clocks to GMT. This doesn't really matter, however. If you call *tzset* and it fails to find an environment string called TZ, it will do nothing. Thus, you can have your program work with the time zone values if your users want it this way... all they have to do is to install the TZ environment variable... or with a system clock set to local time.

Other Clocks

There are a number of other time related functions available under C. The *gmtime* will fetch the system time, but it stores it in a struct rather than in a long integer. This is the struct.

```
struct tm {
int tm_sec;
int tm_min;
int tm_hour;
int tm_mday;
int tm_mon;
int tm_year;
int tm_wday;
int tm_yday;
int tm_isdst;
};
```

This allows you to work conveniently with the individual elements of the time and date without having to figure out how many seconds have elapsed since the dawn of computer time.

You can also get the time and date values separately in their own structs. The *time* struct looks like this.

```
struct time {
unsigned char ti_min; /* Minutes */
unsigned char ti_hour; /* Hours */
unsigned char ti_hund; /* Hundredths of
seconds */
unsigned char ti_sec; /* Seconds */
};
```

This is the date struct.

```
struct date {
int da_year; /* Year - 1980 */
char da_day; /* Day of the month */
char da_mon; /* Month (1 = Jan) */
};
```

Note that as these structs pertain specifically to the DOS structure of time and date, they are defined in the *dos.h* header rather than in *time.h*

There are specific functions which use these structures. The *gettime* and *getdate* functions will load the appropriate structure with the system date. The *settime* and *setdate* functions will set the system time based on the contents of the structures passed to them.

As I mentioned at the beginning of this feature, there are numerous incompatible time and date formats about. We've had a look at time as seen by a PC, but if you encounter files from a Macintosh, for example, you'll find that they're date stamped using a wholly different format. Time on the Mac starts in 1904.

In order to convert a Macintosh long integer time value into a PC long integer time value, you must subtract a constant from it. That constant is:

```
#define mac2pc_date 2082830400L
```

Assuming that *t* contained a Macintosh format date stamp, this would display the correct time of that date stamp on a PC.

```
t+=mac2pc_date;
puts(ctime(&t));
```

Well, almost. There are a few catches here. If the date stamp from the Mac represents a date prior to January 1, 1970, the *ctime* function will return something meaningless. Granted, there were no Macintoshes before 1970, but this does not mean that Mac users might not have set their clocks incorrectly, or just changed the date stamps on their files. Secondly, a long integer on a Macintosh is stored differently than it is on a PC.

The Macintosh is based on a Motorola microprocessor, while a PC is based on an Intel chip. Motorola stores its multiple byte numbers with the bytes in the reverse order to that of Intel. Thus, if *t* is a raw Macintosh date stamp, you must send it through the following function before you can do anything with it.

```
long motr2intel(l)
long l;
{
return(((l & 0xff000000L) > 24) +
((l & 0x00ff0000L) > 8) +
((l & 0x0000ff00L) < 8) +
((l & 0x000000ffL) < 24));
}
```

This works both ways, of course. It will also transform an Intel style long integer into a Motorola style long integer, the one being simply the compliment of the other.

The unfortunate thing about time and date values... on any computer... is that they've evolved gradually over time, and every new variation has been added to the previous accretion of formats to ensure backwards compatibility. As such, there are a lot of ways to tell the time on a PC.

If you're interested in properly understanding how the time functions work under C, you might want to warm up your compiler and try a few of the ones we've discussed herein. They do take a bit of getting used to. ■