Notes on porting TR log to linux

Kevin E. Schmidt, W9CF
6510 S. Roosevelt St.
Tempe, AZ 85283 USA

1 Original program structure

TR was written for DOS on 286 and later 386 class machines. This had major consequences that deeply affected the way I have implemented the linux port.

- DOS was limited to 640 Kbytes of memory. Later extended/expanded memory was available, but is was more difficult to use, and DOS TR did not use it.
- In DOS only a single program ran at a time. Since no other process can interrupt you, the timing is not a problem.
- DOS didn't have the concept of different users having different permissions or any security. All DOS programs can directly manipulate any and all hardware on the computer.

With the very limited memory available information needed by more than one routine in the code was almost always stored in global public variables in the Pascal units. Many, units are included by most other units, which essentially makes all of these variables global to the entire program. The major consequence of this is that these variables cannot be manipulated or changed without a more or less global understanding of the entire program. Without this understanding, it is very easy to make a local change that inadvertently breaks another part of the code.

Changing to linux where essentially any system will have at a Gigabyte of real memory or more (i.e. at least 2000 times more than the machines TR was written for), and much more virtual memory, memory use becomes a nonissue. It is then possible to encapsulate different parts of the code to make modification easier. This however, needs to be done in a way that doesn't break the code. That means it will be done slowly.

With DOS being a single process operating system, a second consequence is that TR under DOS could have completely well defined timing. TR under DOS took over the timer interrupt, and used this to send CW, communicate with the rigs, packet, etc. While there are real time kernels for linux, these are not used in the main linux distributions. However, the standard kernels do have many of the soft real-time timers etc., so that while not guaranteed, timing is good enough that on a reasonably lightly loaded machine, things like sending CW from a serial or parallel port sounds fine. Further since TR was the only thing running on the DOS machine, whenever TR needed to wait for input, or anything else, it would busy wait. Under linux this code would cause TR to use nearly 100 percent of the CPU cycles.

The natural way (at least to me) to implement the various timing requirements would be to run a multithreaded version on linux. The initial port does not do this. It does run a second thread for the sound card output, but the use of a large number of global variables in the original code makes it extremely difficult to write thread-safe code where only one thread at a time can manipulate the values of variables. I tried several different ways of emulating the original DOS timer. Most interacted badly with the display. In the end, I resorted to

- Replacing every busy wait, with calls to a routine that sleeps for 1 millisecond.
- The routine that sleeps for 1 millisecond, checks every 500 microseonds if the time for the DOS interrupt has occured, and if so, it calls the timer routine.

In this way, the timer code is called at more or less the same time that it would have been called under DOS, and it is called in the same thread as the rest of TR, so there are no thread safety issues.

The timer routines have been encapsulated in a new timer unit. It has three procedures

- timerinitialize – this must be called to start the calls to the timer routines.
- millisleep – this needs to be called whenever the code is waiting for something like keyboard input. Before timerinitialize is called, it simply sleeps for 1 millisecond. After timerinitialize is called it calls the timer routines on average every 1680 microseconds with a granularity of about 500 microseconds,
- addtimer – this takes as an argument the a procedure from a class, of the form "procedure timer(caughtup: boolean)" – which will be called every 1680 microseconds. As many of these routines as desired can be added. They are called in turn. They are called with argument caughtup set to true if the timer occurs at the correct time. If the timer has been delayed because the computer has been busy doing other things, caughtup will be false. This allows each timer procedure to only process critical items in an attempt to get better real time performance.

More work needs to be done on the timer routine to be able to remove timers, etc. Right now only the original timer routine is added with addtimer. All other timer routines are called by it. That should be changed. I think that eventually, the timers should be replaced with threads.