Repeater Controller Operating Systems

All but the smallest repeater controllers are now microprocessor based. The software that the processor executes is the controllerõs *operating system*. A properly functioning operating system is as crucial as any component. What does it take to create one?

As a starting place, consider a CW ID program that sends *WA9FBO/R*.

Sending CW involves a time unit called the *element*. A dit is 1 element, a dah is 3, an intracharacter space is 1, an intercharacter space is 3, and a word space is 7. An element time of 60 ms corresponds to a code speed of 20 WPM.

So, you write a simple program that uses timing loops for elements. Based on the execution times of the instructions, you write delay routines of 60 ms, 180 ms, and 420 ms.

The first character, W, is didahdah. To send the first dit, the program turns the tone generator ON and executes a 60 ms delay routine. It turns the tone OFF and does another 60 ms delay. To send the first dah, the program turns the tone ON and delays 180 ms, then turns the tone OFF and delays 60 ms. It then sends another dah, followed by an intercharacter space, and moves on to the A. And so on.

When itõs not sending CW, the program monitors the COR for activity and runs an ID interval timer.

Sounds easy enough, and if thatõs all you have to do, it works.

But note that while sending the callsign, the program must continually run with no interruptions. Exiting a delay routine to do something else ï even momentarily ï disturbs the timing. In other words, all of the processorõs resources are tied up while sending CW. That makes it difficult to add new features.

For example, it would be nice to have a timeout timer with anti-tailgating and penalty attributes; a courtesy beep; even a DTMF decoder. That adds a lot of timing loops to the program. You shouldnõt interrupt a timing loop, yet if you donõt check the DTMF decoder often you may miss a character. Further, a controller like the 7330 must operate three repeaters simultaneously.

Youõd think thereõd be a way to handle multiple, unrelated jobs in a timely way. There is, and itõs called a Real Time Operating System (RTOS).

An RTOS quickly switches tasks, round-robin style, allocating a slice of time for each task. Although the microprocessor is doing only one thing at a time, the RTOS gives the appearance that the controller is doing many things simultaneously.

There are other kinds of operating systems, and if theyõre concerned with balancing a checkbook or word processing, time is not of the essence. But a <u>real time</u> OS must quickly respond to activity on inputs (COR, CTCSS, logic, DTMF, etc.), then update the outputs (PTT, logic, display LEDs, etc.) It must keep track of time and date to run the Scheduler (so your net can be announced on time). It must send CW, beeps, paging tones, CTCSS, and sound file messages. In other words, it must respond quickly to many types of internal and external events.

How does it work?

An RTOS gets its sense of time from ñticksògenerated at precise intervals by a hardware clock. The period of time beginning with a tick is divided among multiple programs. Some programs, like the one that checks logic inputs for transitions, start and finish within a small portion of a time slice. You might want them to run every tick, or every other tick, or every 5th tick depending on the update rate you need. Other programs (like the one that sends CW characters slowly enough for humans to copy) require many ticks to complete.

Ticks not only keep the timing accurate, they also prevent problems by synchronizing things. An error can occur, for example, when a changing input is seen as active by one routine and inactive by the next. So, at the start of a time slice, all hardware inputs are read and saved. Only the frozen states are utilized by the routines. Outputs are updated after all routines have run.

Getting back to the CW programé Using an RTOS as the platform, you revise the program so that each time it starts, its timer is decremented. If the timer is not zero, execution moves on to the next program. If the timer is zero, a state machine steers execution to the proper place so it can pick up where it left off during the previous visit.

Perhaps the program had started the first dit of the W last time, so it had loaded the timer for 60 ms, started the tone, and set up the state machine. This time, the program is directed to stop the tone, load the timer for 60 ms, and set up the state machine for the next visit. No software loops are used, and the timing stays accurate and synchronized.

In addition to using an RTOS to improve efficiency, the 7330 uses hardware to lighten the processorõs workload. Its FPGA (Field Programmable Gate Array) has large data buffers on the serial port, allowing a lower service rate. It also contains specialized hardware that do sine table lookups and D/A updates for tone, voice, and CTCSS generators without processor intervention.

To sum up, using an RTOS is very beneficial in a controller. It avoids time-wasting delay loops and allows tasks to have independent timing requirements via an accurate hardware clock. And with large programs divided into modular, more manageable tasks, the quality of the software is improved.