Test Port – A Real-time Embedded Monitor (Keyboard & Glass TTY)

Priority Tasking is illustrated in this design in which the Test Port task is a low priority task. Priority tasking allows CPU load distribution to occur automatically among tasks depending on the demand of each task and task priority. The Test Port task is using what would otherwise be idle capacity. "Time slice" tasking is NOT chosen because that scheme presumes the designer can know present and future CPU loading demands from each task.

Priority tasking can be visualized as a way of controlling what appears on a multi-trace oscilloscope to be chaotic activity as the processor responds to a blizzard of interrupts and where traversing the same paths through the code are seldom of constant duration. High priority tasks (typically interrupts and IO) are time critical. By design, those code segments are kept short. They queue workload for medium or low priority task where time to completion can fluxuate.

A good rule of thumb is to keep average CPU loading below 50%. Momentary loads may well exceed the 50% loading threshold. However, the intent is to smooth processor loading by postponing less time critical work to run when the processor would otherwise be idle.

Priority tasking is illustrated in the diagram below where high priority work is at the top and low priority work is at the bottom:

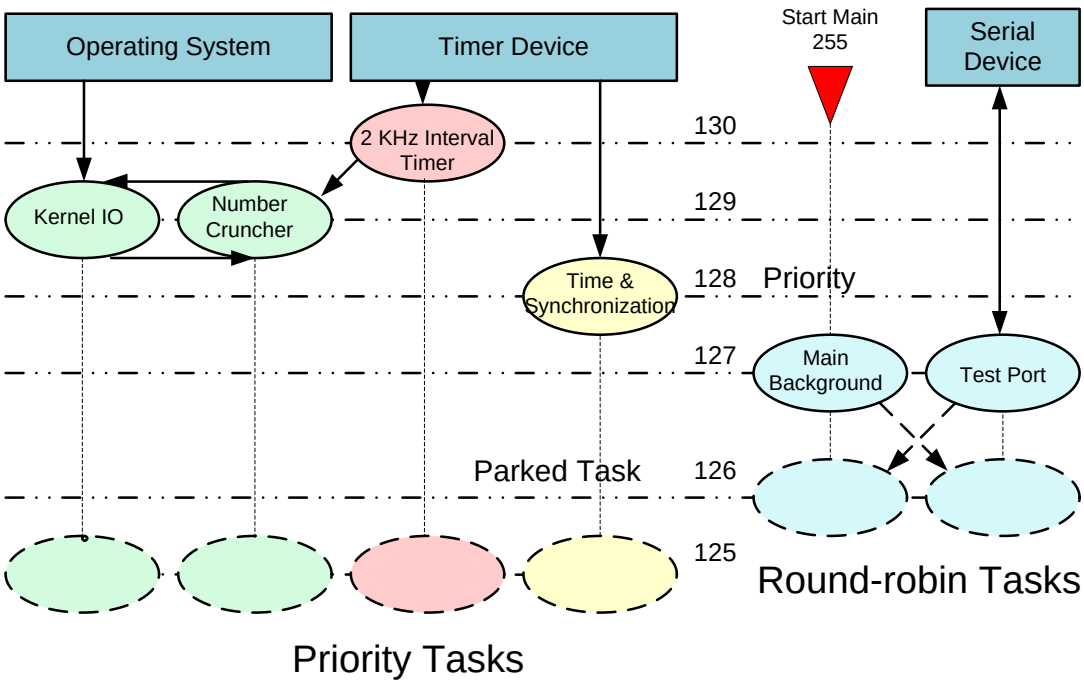

Figure 1

Parked tasks at the bottom are those whose priority is so low (125 or 126) that the OS scheduler will never run them because higher priority tasks are always waiting to run. For example, either the Main Background task or the Test Port Task is always alive and waiting to run at priority 127. A task may raise the priority of a parked task while parking itself (by lowering its own priority) as do the Main Background and Test Port tasks. They each give all remaining processor idle capacity to the other in a "round-robin" hand over.

In this example, the Main Background task launches initially at the highest possible priority (255). That high priority allows it to control the processor while initializing all other task priorities, queues, etc at start up. Once initialization is complete it lowers its priority, thus allowing the OS scheduler to start running the established task and event sequences.

In this example the Number Cruncher task is exchanging Synchronous messages with the Kernel IO task that cause each task at priority 129 to "pend" waiting for a message back from the other, except the Number Cruncher parks itself (priority 125) after receiving back the Kernel synchronous message.

The high priority (130) 2KHz Interval Timer task fires and raises the Number Cruncher task from parked at priority level 125 to running at level 129 so that one more cycle of synchronous messages are exchanged between the two task at level 129 before the number cruncher parks itself once again. The 2KHz task is little more than a heartbeat that keeps the data flow moving. It pends itself waiting for the next 500 usec high resolution timer event to fire. In this example, the 2 KHz task substitutes for what would normally be an IO interrupt handler.

In this example, the Kernel IO task resides in kernel space while the Number Cruncher resides in application space. The Kernel IO task is using the Test Port for debugging platform hardware.

The Test Port Task uses one RS-232 serial port interfacing to a PC keyboard and text display. It allows the developer to write more specific diagnostics than are possible with a general purpose debugger. Custom scanning text displays driven by the test port task allow the developer an organized view of what is happening in specific functions like the Number Cruncher. Test Port task diagram is below:

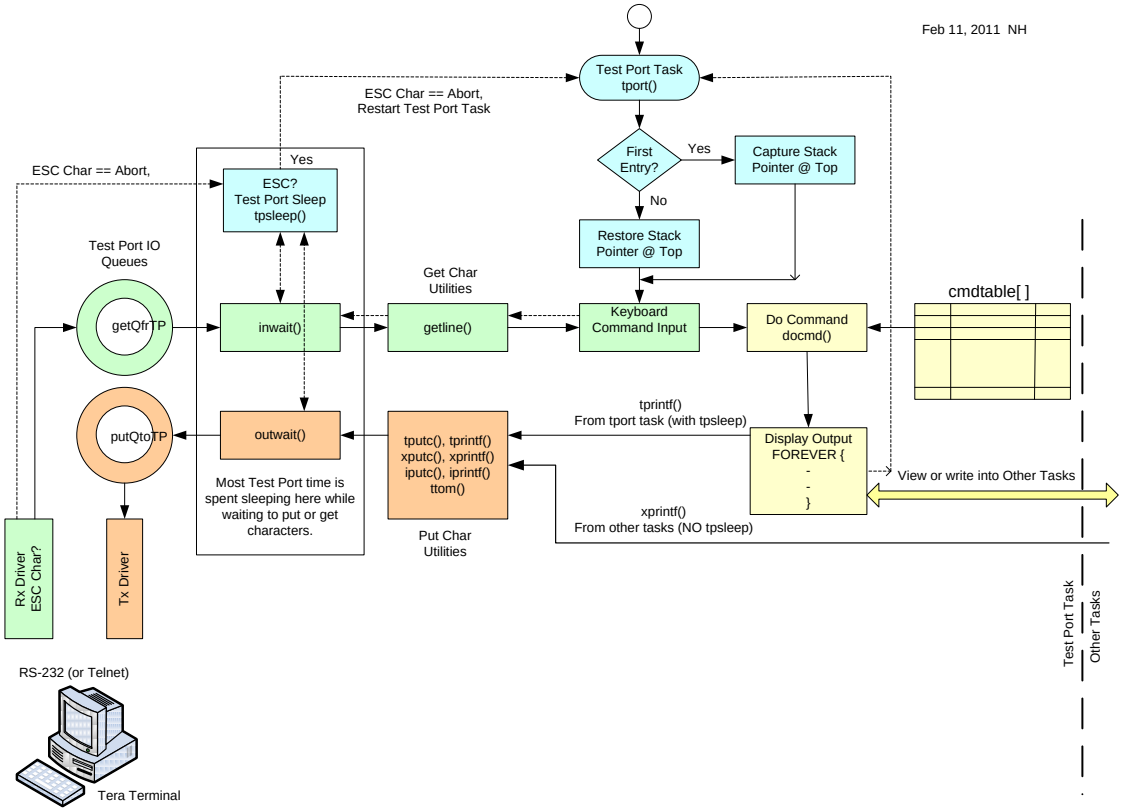

Figure 2

Most tasks enter into a loop where they remain "forever", being parked and un-parked, or pended and un-pended inside the loop. The Test Port Task also uses forever loops except when an ESC keyboard character is received. The ESC character causes the task to completely restart by calling itself at the Test Port function entry point. Such action would eventually cause the Test Port stack to underflow, except the stack pointer is restored to the top of the Test Port stack where its location was captured and remembered on first entry.

Test Port CPU utilization is kept low by the relatively slow IO speed of the serial channel. Most Test Port time is spent waiting to get or put characters at the queues to the RS-232 serial channel.

Test Port history has evolved from when a 16 kbyte Cromemco S-100 bus video card was first placed in the top 1/4th of Z-80 CPU memory space (1977). Program variables assembled for storage in video memory could be watched on the CRT display. Video memory and program variables were scanned and displayed at the CRT refresh rate.

In the generations to follow, the S-100 bus and video card are replaced by serial communications and a computer display terminal. New contributors made it possible for the scanning display to be formatted by a variant of printf() and the Test Port became an embedded task running on any RTOS.