

# VxWorks Issues

## *Linac upgrade experience*

Tue, Jun 26, 2001

During the development of the upgraded PowerPC-based replacement software for support of the Linac front-ends, a number of experiences relating to the VxWorks kernel were encountered. Most of the reason for this is that the software being ported to the PowerPC was based upon a different OS kernel, pSOS. Some of these experiences may be useful to other vxWorks kernel users.

### *VxWorks networking*

The pSOS kernel used in the former 68020-based software provided no networking support, so all networking software was written in house. In planning for the new system, we decided to use the widely-supported socket network support that is supported by VxWorks. But much of the network software would still be needed in the new system; only the part that delivers a UDP datagram would have to change. In addition, the ARP support and the multicast support is built-in. Still, we needed to adapt the old logic to the new.

The former ARP support was refined to eliminate the need to wait for an ARP reply; i.e., while waiting for an ARP response, transmissions to other nodes not requiring an ARP request could continue. This was done by queuing network messages destined for a node while continuing to process the network message queue. When an ARP reply was received, the queue would be passed again through the usual network queuing scheme that would this time be successful. None of this logic could be used with VxWorks, because the user does not have little control over ARP communications. So, how can we arrange to send network datagrams without being forced to await ARP replies so that transmissions to other nodes could be made whose physical network addresses are already in the local ARP cache?

We decided to use a higher priority task to execute the `sendto()` call. But this would not solve the problem, except that we hoped `sendto()` would block while waiting for the transmission to complete. We decided that two higher-priority transmit tasks were needed. One would be passed a datagram if the target node was already in the ARP cache; the other would be passed a datagram if an ARP transaction had to be done first.

But this turned out not to be so straightforward as we had hoped. In order to find out whether a node is in the ARP cache, the routine `etherAddrResolve()` is called without delay specified. But if that routine discovers that no entry exists in the ARP cache, it automatically issues an ARP request message. If we thereby find that no ARP entry had been in the cache, we pass the datagram to a high-priority task that first calls `etherAddrResolve()` again, this time specifying a minimal delay, in order to await the ARP reply. But the delay is not shortened upon receipt of the ARP reply message. The delay occurs even if the ARP reply arrives very quickly. Since the units of the delay are in ticks of 60 Hz, we can have unacceptable delays in transmitting the first time to a given node. Such delays can result in errors, since the front-end software is strongly focused on delivering synchronous 15 Hz performance and imposes timeouts to detect missing replies. This problem may not yet be adequately resolved.

It has been very difficult to determine exactly what the VxWorks network software actually does in detail. We can only try to measure its performance characteristics. It is possible that ARP cache entries are removed after some period of time, independent of whether communications are active with a given target node, but we still do not know that for a fact. In such a case, we may find occasional delays in responses to active ongoing requests. We need to make more measurements to nail this down.

Although we hoped that `sendto()` would block during transmission, we have found that it does not. The `sendto()` caller must await the transmission of the datagram. Fortunately, the time of transmission seems fairly brief, as we are using 100 MHz ethernet hardware. The `sendto()` task typically requires about 60 us to execute.

For the reception of network datagrams, the former system used common receiver code that provided for network diagnostics. The task to which control was passed following a network receive interrupt had the job of supporting the UDP/IP protocols. In VxWorks, this part was not needed, but providing support for network diagnostics brought us to the need to organize all network receive activities under a single task. This task is the only one that would include the `recvfrom()` call. The `select()` function was used to enable this organization to work. Any of the expected network activities cause this task to run (after `NetTask`). Following the `select()` call, A reference to the received datagram, copied by `recvfrom()` into the next available space in a circular buffer area, is passed via the appropriate message queue that causes the relevant task that supports that UDP port to run. Again, a data stream called `NETFRAME` is used to hold a record of each received datagram. A page application facilitates capture and listing the contents of this data stream. The times of received (and/or transmitted) datagrams are shown in units of 1 ms within a 15 Hz cycle of a time-of-day.

### *Task activity timing*

In the former system, we were used to operating task timing LEDs to bring out signals indicating each task's activity. This feature is key to analyzing the detailed performance of the system code. In the new system, both PMC slots are in use on the MVME-2401 CPU board. One houses 2 MB of non-volatile memory, used for storing system tables and downloaded local application program files, and a digital interface board known as the Digital PMC board. The latter includes support for timing and two sets of 16 LED signals that are multiplexed so that any 4 of each set of 16 signals are available on a front panel connector. The multiplexing is only for bringing out the signals for viewing on a scope or logic state analyzer; the entire 32 bits are available for the software to control.

In the former system, a set of 16 task signals were driven by code in a task-switch callout procedure that the OS supported. When the kernel switched from one task to another, this optional user-specified callout procedure was invoked, being passed pointers to the two task control blocks in question. From a user-available field in the TCB, we found the mask to use in setting/clearing the appropriate task signal. Using a scope, one can observe the task timing performance.

In the VxWorks kernel, similar task-switch procedure support is provided. But there is one difference. When a task blocks, and there is no other available task to run, the task-switch procedure is not called. (This didn't occur in the former kernel, since a low priority `Idle` task was always available.) We decided that we needed a low priority task to alleviate this problem. This task would wait for a binary semaphore. After "taking" the semaphore, it would simply loop back to wait again. But we needed to be sure that every place in the system code where a block is possible, we should "give" that binary semaphore. This was not as difficult as it sounds. Most tasks wait for a task event. (VxWorks does not support such events, so we emulated them using a binary semaphore and a mask word to hold the event bits.) Other tasks wait for a message queue, so the common code that is called to provide this service was modified to give the low-priority task semaphore before calling the VxWorks message queue routine. Beyond these two cases, the transmitter task that calls `etherAddrResolve()` specifying a delay can block, so the special "give" was inserted there. (Note that an unnecessary "give" is not significant, other than the time taken for making the call; that time is very short.)

But what can we do about VxWorks tasks that might execute? If such a task exits or blocks, how can we ensure that our low priority task can run? Of course, we can't do this in general, because VxWorks code is proprietary and secret. But we have found that the only VxWorks task that operates normally is the `NetTask`, which runs every time a network message is received. So if we focus on that case, we can assume that the arrival of a network message will cause it to execute. Of course, we have no access to the ethernet interrupt routine, either. But there is another hook available when a network datagram is received that is presumably invoked by `NetTask`. Unlike the case of the task-switch hook, kernel calls are permitted in this network receive hook, so we can "give" the required semaphore. When `NetTask` exits, the low-priority "idle" task will run, so that the task-switch hook will be invoked, and the LED can be turned off.

At least one gap was found in this logic. It appears that when a node receives an ARP request (from another node that would like to send it a datagram), the ethernet interrupt code arranges to return the ARP reply without passing it to `NetTask`. This means we cannot see the reception of ARP request messages. In addition, it seems that there are times when `NetTask` runs, but the user-specified receive hook is not invoked. In such a case, the `NetTask` timing may seem unexpectedly long—until another task needs to run.

Finally, to facilitate detailed study of task timing, data stream support is used. By “data stream,” we refer to the support for circular buffer management provided by the system code. The task-switch hook includes code that writes a record into a data stream, if one called `TASKLOG` exists within the system. Since the time register is accessed only once within the task-switch hook, the elapsed times computed by this method include the time spent in the task, the VxWorks task-switch overhead, and the time used to execute the task-switch code itself. The minimum elapsed time seen so far via this method is 9  $\mu$ s.

A page application is provided that accesses the `TASKLOG` data stream in a given node and encodes it for listing output via the serial port of any node for capture by a terminal emulator. The time typically covered by a complete capture of the data stream contents is about 1 second or less. To get detailed data about some particular happening in a node, one should cause the happening to occur, then request the task timing data within a half second. Sometimes this requires a certain level of choreographic competence.

### *Nonvolatile file system*

The system architecture supports flexible configuration of downloaded programs that run in addition to the basic system code. These programs are of two types: page applications that support a small “little console” display of sixteen lines of 32-character, and local applications that have no such display interface but have a set of parameters that are passed to them each time they are called. Only one page application can run at one time, but many local applications can be active at once. The code for these programs resides in nonvolatile memory, so that they are not lost when power fails. When the system resets, the same complement of local applications that was enabled for execution before the reset is restored automatically. Each application is really a function that is called at 15 Hz and also in response to relevant network activity, if needed. It is assumed that each application will run for only a short time in order to maintain consistent and reliable 15 Hz operation.

For the case of the VxWorks upgrade, 1.5 MB of the 2 MB nonvolatile memory is configured as a RAM Disk and maintained as a DOS-format file system. System code supports access to such files. To that end, a separate file directory is maintained in nonvolatile memory in addition to that maintained by VxWorks file system support. File names are of the form `PAGEXXXX` or `LOOPXXXX`. The latter set of files are local applications; the former are page applications. The `LATBL` system table houses one entry for each local application instance, along with the set of 16-bit parameter values. The `PAGEP` system table holds the names of page application associated with up to 31 pages identified with page numbers in the range 0–9,A–V.

It has been found useful to retain a record of the version of each program file. When a new version is built, the file modification date is of course automatically updated in the development system. When a special TFTP client tool is used on the development system to send the new version to a target node, the version date is obtained from the file system and appended to the file name in the TFTP write request. When the TFTP server (a local application in the target node) receives the write request, it strips off the version date and arranges for it to be stored in both the system file directory as well as the internal one maintained by VxWorks. When system-level support is used to copy a program file from one node to another, the version date is also transferred. In this way, we can compare version dates of the programs in two nodes and determine which ones are more recent. A page application called `PAGEVERS` is available that facilities such comparisons between a target node and a reference node. For each file in the target node whose version differs, the relation to that in the reference node is shown.

The normal sequence of events in modifying a local application is to edit and build the new version in the development system, then execute the TFTP client to send the new version to the library node that keeps the most recent copies of all program files. From there, it is copied into whichever nodes need the new version. As each node receives a new version via its system support, if the same program file is actively in use, that version is closed and the new version automatically activated. The result is that the new version is immediately and automatically brought into execution in the target node. For page application,s the sequence is similar, but it is necessary to call up the page application to bring the new version into execution. If the same page application were active when the new version is received, it would not automatically replace the active one; however, a simple recall of the same page will do it.

In the former system, the program files were simply copies of position-independent executable files. They obtained access to system functions via a TRAP-based scheme. In the new systems, the program files are object files. They are loaded using the `vxWorks ld ( )` function, which establish the required links to any external named entry points in the system code or in VxWorks itself. Because one local application does not reference any functions in other local applications, it is possible to `unld ( )` one local application when it terminates. (In general, one would have to `unld ( )` in the reverse order of the `ld ( )` calls. But since each only links to system routines, this restriction does not apply.)

An attractive feature of the local application scheme is that one can download such programs while the system is running; a reboot is not necessary. In contrast, any changes made to the system code, of course, will require a reboot to bring in the new system version.