Running PLCs with Linux: Challenges and Solutions

By Peter Wurmsdobler, Director, Open Control Laboratory, Control.com Inc., peter@control.com

Off−the−shelf computers provide an inexpensive platform for many of today's automation needs. GNU/Linux is an open source operating system that is adaptable to these platforms and many computing problems. The question arises, why not use the combination of off−the−shelf computers and the Linux OS to run a PLC? This article attempts to answer that question and addresses some of the issues that arise with such an approach. First, however, let's be clear in how we characterize both a PLC and GNU/Linux.

When talking about a programmable logic controller (PLC), we typically refer to the entire controller−hardware components, systems, and application software. In general, the hardware consists of a central processing unit, some direct analog or digital I/O, network and fieldbus I/O, as well as at least one serial communications port. Firmware guarantees that, once booted, the PLC loops eternally through a sequence of control instructions (its application software) which can be changed online or off−line by a programming system to which it is connected. From a computing point of view, a PLC is a single process running on the core computer hardware.

PLC Architecture

During the PLC's loop cycle, also referred to as a 'scan', the PLC carries out an input scan, executes the control and timer tasks in a round robin manner, and updates the output image. Either at the end of such a scan or somewhere in−between, the communications port is served for upload of data or download of PLC software. However, at no time during the scan do we see such external events as timer interrupt−driven schedulers launching tasks as independent computing entities. The PLC is designed to run through the loop cycle as fast it can and the worst case loop time determines the response time of the system (Figure 1. PLC−classic).

The reason behind the PLC's scan architecture is that the system was designed to come as close as possible to analog hardware, albeit with relays. This constitutes the PLC operating paradigm, which is focused on predictability, determinism, and reliability.

Enter GNU/Linux, a POSIX−compliant, UNIX−like, general purpose operating system. GNU/Linux runs on general purpose computer hardware, like off−the−shelf desktop systems, as well as on specialized computers like CompactPCI or PC104 systems, all based on the same computer architecture. What all these computers have in common is that they utilize modern hardware techniques that push for throughput and average performance across a broad range of applications. Techniques employed include fast, pipelined CPUs with many registers and interrupt facilities, several levels of cache for memory, buffers for all devices, and all kinds of buses like ISA, PCI, and USB. Unfortunately, such a design, which seeks to be something useful to nearly everyone, decreases the predictability of the system's behavior and has a tremendous impact on real time and control processes.

The Linux Paradigm

GNU/Linux is a multi−processing, multi−user operating system. Similar to the underlying hardware, it is designed for maximum throughput and average performance. Several users and processes which compete at one time for resources are served on a fair scheduling policy. In order to accomplish this task, the kernel of the GNU/Linux operating system, Linux, uses such modern techniques as I/O buffering, virtual memory management, and other tricks to improve average performance. For example, Linux tries to regroup the hard disk block requests of several processes in a sequential order and, at a certain threshold, will deliver a series of disk blocks in one chunk. During this time no other process can be executed.

The crucial question is, how can a PLC be implemented on Linux given the opposite operating paradigms of the GNU/Linux operating system and the programmable logic controller? At the Open Control Laboratory of Control.com, we are working on just such issues. We started with an open source PLC run−time system called SmartPLC by Infoteam GmbH, Germany (www.infoteam.de). The run time system was ported to Linux and implemented as a standard Linux process. It was combined with OpenPCS, a PLC programming environment that runs on a Windows host. Under Windows, control software can be written in any IEC language, then compiled and downloaded to the run time system using any network connection. The Linux process executing the run time system then loops through an infinite PLC scan and carries out communication work every Nth cycle. Such an approach synthesizes both the off−the−shelf PC and PLC paradigms, as depicted in Figure 2, PLC−smart.

Porting to Linux

Porting the run−time system itself, which does not include any graphics, was straight−forward. Only system call specific code had to be adapted from Windows C−code to Linux and hence POSIX compliant code. Specifically, this involved file creation and handling as well as serial port communication system calls.

An issue arose in the addressing of I/O in the PLC world, such as %I0.0 and %Q0.0 for input and outputs, respectively. Since in UNIX everything is treated as a file, the analog I/O is carried out by ioctl, read and write functions to the corresponding driver. The input and output values are copied to and from the run time system's process image and the input scan and the output update, respectively.

The most important resource to be offered to the run−time system, however, is time, i.e., a reliable system call must provide a time stamp which can be used for all timer tasks of the PLC. Fortunately, Linux offers the gettimeofday system call which gives time in microsecond resolution. This time information is retrieved from an internal kernel time data structure which is updated on each kernel timer tick in combination with the time stamp clock from the last timer interrupt on. But, we wondered, is this time stamp reliable and what is its uncertainty?

Benchmarking Performance

Given that Linux is a multi−user operating system, the run time system process could be preempted at any time, even within the gettimeofday system call. The control process will then be rescheduled by the scheduler in an indeterminate amount of time, up to as long as several seconds. To test the predictability of such a PLC run time system on Linux, we looked at a simple loop which only gets a time stamp and calculates the time difference between two consecutive gettimeofday system calls. This time difference should be constant and only depend on the execution speed. Nevertheless, there is a wide spectrum of uncertainty with poor worst case values, if for example a "ls −R /" is executed in parallel on a Pentium 850:

uncertainty less than	counts
1 us	28238601
10 us	71755198

```
      100 us  5701
     1000 us  332
    10000 us  44
   100000 us  101
  1000000 us  23
worst case = 593185 us
```

This shows that in most cases the time uncertainty is less than 1ms, thus meeting the requirement for the Infoteam run time system in 99.9998% of all cases. However, if you look at the worst case of approximately 600ms, this is actually poor performance. A less portable version of this test uses in−line assembler code to read the time stamp clock directly. Here the average performance results are better, but the worst case still remains poor:

```
uncertainty
less than     counts
       1 us  75455007
      10 us  24543293
     100 us  1424
    1000 us  129
   10000 us  46
  100000 us  88
 1000000 us  13
worst case  = 453184 us
```

Both examples show that, in general, two time stamps are retrieved very close to each other. But there is a second peak above 10ms which is the Linux scheduler time slice. Here, another process got rescheduled and ate up at least one such slice. Here's what happened. The scheduler, the part of the Linux kernel that decides which process will be executed on which CPU next, offers three scheduling policies: SCHED_OTHER for all normal processes (the default time−sharing policy with static priority 0), and two real−time scheduling policies (SCHED_FIFO and SCHED_RR with priorities higher than 0, up to 99). This means that as a normal user process, the PLC run time system can be preempted for quite a long time.

One way to cope with this delay is to use real time priority scheduling, instead. And indeed, setting the priority to its maximum value and using the sched_setscheduler and sched_setpriority system calls helps quite a bit. Here is what this show:

uncertainty
less than counts
1 us 28616694
10 us 71378763
100 us 4534
1000 us 2
10000 us 7
worst case 1371 us

What happens here is that no other process gets a chance to run. Only kernel threads and other kernel activities, like reacting on interrupts, can prevent the run time system from carrying out its control task. For example, if the buffer flush daemon decides to flush dirty buffers, the kernel swap daemon frees pages for another process. If the kernel based NFS daemon is waiting for a network connection, time blackouts of the control system can even be longer. By the way, it is a good idea to lock the run time system to memory to prevent it from being swapped out to disk. To test for this scenario, a ping flood to the Ethernet board will generate plenty of interrupts. Then an increase in delays up to 500us can be observed.

If the real time performance of the last approach is not sufficient for your needs, there are other approaches available. One is to apply a low latency patch to the Linux kernel. This introduces preemption points in the kernel where the scheduler looks and decides whether a real time process needs to run. Using this approach, the worst case performance can be brought down to 500usecs. The average performance will not improve, though, as it will stay the same regardless of your approach.

There are currently two real time Linux variants available, RTLinux by FSMlabs (www.rtlinux.org) and RTAI by Politechnico Milano (www.rtai.org), which slide a real time executive underneath the Linux kernel and run it as a low priority task. With these systems, any hard real time control can be built into kernel modules, thus offering much better real time worst case performance. But average performance will decrease as these

systems don’t utilize a number of hardware features that can bolster average performance, such as caching and buffering.

To conclude, some will say that the standard GNU/Linux general purpose operating system is not appropriate for real time control. Through our extensive testing, we have found that with some precautions, Linux can in fact be used, depending on the application’s time constraints and uncertainty requirements. For slow processes, Linux supports hard real time without any changes. For fast processes, some kernel changes are necessary, and, as a last resort, the application can be built as a Linux kernel module.