Automating Manufacturing Processes with Linux

Craig Swanson

Ryan Walsh

Abstract

How do you monitor multiple manufacturing lines in real time? We did it ourselves with Linux.

A manufacturing company makes money only when production is running. So, timely information from the production floor is crucial to the business. As our company has grown, so has the complexity. We have outgrown our manual and paper methods for monitoring manufacturing.

Midwest Tool & Die stamps electronic terminals and molds plastic parts for the automotive, electronics and consumer industries. Our manufacturing processes generate a lot of data. Our high-speed presses make up to 1,200 parts per minute, and each part must be correct. We inspect critical dimensions for every part that is produced. Part quality is charted and monitored, and the data is archived for traceability.

Why Automate?

We needed to manage all of this data to improve the manufacturing processes. Our main goals were to improve uptime and understand the causes for downtime. In addition, we hoped to track and control costs, reduce paperwork and avoid human input error.

To meet these goals, we came up with requirements for the new system. The first requirement was to gather data from a variety of machine controls, sensors, automated inspection equipment, programmable logic controllers (PLC) and human operators. The system had to be reliable and be capable of gathering data at our fastest production rate.

Next, the system had to correlate the data that was gathered. The system would need to interact with enterprise PostgreSQL databases. Production data and process status would be passed to PostgreSQL for display and reporting.

The new automation system also had to provide a user interface, so the machine operator and maintenance personnel could log their activities. Process downtime and the reason for the downtime would be logged and passed to the enterprise databases. This requirement would replace a paper log and manual data entry effort.

Finally, the system needed to be flexible and easily upgraded. The solution would be adaptable to new manufacturing lines and changing system inputs.

Why Linux?

We evaluated several solutions to meet the requirements. Industrial PLCs could gather data reliably. However, their approach to networking has been stuck in proprietary nonstandards for decades. Ethernet connectivity has become available, but the systems are expensive. The user interface typically is implemented on vendor-specific display hardware. Each vendor produces its own proprietary development platform. So, vendor lock-in was an issue at every point of the evaluation.

Next, we looked at a PC with a data acquisition (DAQ) board. In the past, we have used a laptop with a DAQ board, Microsoft Windows and Agilent VEE. This combination has worked well for quickly acquiring data with little programming. With that setup, data transfer to our database systems was available only through Windows OLE. We could develop applications, but the proprietary platform would tie us in to the vendor. National Instruments also offers a complete DAQ package for the PC, but at a premium price.

The solution that best met our requirements also used a PC and a DAQ board. The big difference was the use of RTLinux, a real-time OS stack based on Linux. We could limit vendor tie-in and communicate freely with PostgreSQL and TCP/IP networking. The real-time OS offered the reliability of a PLC, without sacrificing communications. Finally, the GUI could be written in the language of our choice. Using open-source tools, we could create flexible, upgradeable applications.
Figure 1. Two SmartPress manufacturing automation systems in use. Each system includes a PC with DAQ board, electrical isolation hardware, battery backup and a barcode printer.
SmartPress System

The computers we chose to perform the task of data acquisition and handling of the data were slow in comparison to computers available today. We were able to recycle old office computers to use on the shop floor. These 400MHz Celeron machines are fast enough to perform the tasks asked of them adequately, without hindering our hard real-time requirements for data acquisition. The system we worked with started off with an installation of the Red Hat 7.3 distribution with the 2.4.18 Linux kernel.

RTLinux and Linux kernels

The kernel separates the user-level tasks from the system hardware. The standard Linux kernel allots slices of time to each user-level task and can suspend the task when the time is up. This can lead to priority tasks being delayed by your system's noncritical tasks. There are commands to control the operations of the Linux scheduler; however, hacking the scheduler's parameters in the 2.4 kernel does not result in a hard real-time system. The 2.6 kernel has enhanced real-time performance but does not fill the needs of a hard real-time system either.

There are many great publications about RTLinux, many of which are written by Michael Barabanov and Victor Yodaiken, who first implemented RTLinux back in 1996 and have been improving it ever since. Finite State Machine Labs, Inc., (FSMLabs) is a privately held software company that maintains the software. Through the years they have produced advancements that are wrapped up into their professional versions of RTLinux. They still do provide RTLinux/Free, which must be used under the terms of the GNU GPL and the Open RTLinux Patent License. For our project, we used the free software, which does not have support from FSMLabs.

The RTLinux HOWTO, by Dinil Divakaran, provided the majority of the information we needed to complete the RTLinux installation and get it up and running on our system.

RTLinux System Flow

In a standard Linux system, if you wrote a function to poll the inputs of a data acquisition card at a set interval, your task would have less than favorable results. Such a system cannot guarantee the scheduling priority that it would receive. In the standard Linux operating system, as illustrated in Figure 2, all of the system processes are isolated from the hardware. This would not be so bad if our data polling was the only process the system performed. Our project, however, required user-space programs and a guarantee that the sensor inputs were polled every 1 millisecond. The hard real-time system would guarantee that sensor inputs would not be missed.

In the RTLinux operating system, also illustrated in Figure 2, the real-time task is isolated from the rest of the system processes and is implemented as a module. The module gains direct access to the hardware and the DAQ card drivers while receiving the priority that it needs over the rest of the system. The module is written to perform a specific task that produces reliable results and presents them to the user through the FIFO device files. The code for the module is kept simple, performing only tasks that must be held to hard real-time restraints.

Connecting a handler function to one of the FIFO device files can control the module from the operator interface program. This structure, produced by RTLinux, ensures that the kernel cannot delay important module task with secondary priorities.

Data Acquisition Threads

We needed to monitor only digital input status with hard real-time requirements, which made the real-time function fairly small. Polling the data inputs was made easy, because United Electronics Industries provided RTLinux drivers with their DAQ card. A DAQ card from ADLINK Technology, Inc., also was tested with drivers for the RTLinux platform, and it configured easily. Not many companies provided such a service. The Comedi Project, however, offers another option for open-source drivers, tools and libraries for data acquisition.

The real-time task was written in the form of a loadable module, which has to have at least two functions: init_module, called when the module is inserted into the kernel, and cleanup_module, which is called right before it was removed (Listing 1).

Once the base module structure was established, we needed to create a thread for our real-time task in the module. The thread was created inside of the init_module and was set up to run with established RTLinux priorities. Establishing the priority and rate of execution for the thread was an important step to creating our real-time task.

FIFO Device Files

With a task running with predictable timing, real-time memory for data transfer was needed. The user-level task needed to access collected data and to control the real-time task. Real-time FIFOs are queues that can be read from and written to by Linux processes. The real-time FIFO devices are built during the RTLinux installation and created in the initialization of the real-time modules. Now a handler can be created and tied to one of the FIFO Devices. The handler can be set up to execute when 1 is written to the handler FIFO from the user interface program as controlling of the module is needed.
The module was created in order to be installed into the kernel as a real-time task. The tricky part of the real-time module was setting up the framework. Our real-time task code was straightforward, but lengthy. So, we've included only the real-time module skeleton (Listing 1). Notice the simplicity of the code that is implemented with real-time requirements.

Listing 1. Example Real-Time Module Code Skeleton

```c
#include <pthread.h>
#include <rtl_fifo.h>
#include <rtl_core.h>
#include <rtl_time.h>
#include <rtl.h>
#include <rtl_fifo.h>

pthread_t thread_variable;

void thread_name(void)
{
    Struct Sched_param p;
    p.sched_priority = 1
    pthread_setschedparam(pthread_self(),
        SCHED_FIFO, &p);
    pthread_make periodicnp(pthread_self(),
        getrtime(), 1000000);
    while(1) {
        // Real Time Task Code
        // Poll Data input lines, count low to high
        // transitions
        rtf_put() // Counts to be transferred by FIFO
        pthread_wait_np();
    }
}

int handler_function(){
    // Code tied to the handler FIFO
    // Variables for counting above are cleared out
}

int init_module(void)
{
    ififo_status = rtl_create(unsigned int fifo,
        int size)
    pthread_create(&thread_variable, NULL,
        thread_name, NULL);
    rtf_create_handler(FIFO_Number,
        &handler_function)
}

int cleanup_module(void)
{
    rtf_destroy(unsigned int fifo)
    pthread_cancel(thrad_variable)
}
```

User Interface

With the real-time task up and running, the next step was to build the user interface for the project. The GUI was one of many tasks in our program that did not need to be accomplished within hard real-time restraints. We did not have heavy concerns for system resources, because our real-time module would execute within our established hard real-time requirements. We selected KDevelop IDE and Qt Designer GUI builder from Trolltech for programming the interface. Qt Designer allowed for the development of a GUI with signal and slot interaction with functions in the KDevelop program. The result of developing with the combination of the two packages was perfect for our application. We were able to develop a user-friendly interface quickly.

Managing the Data

Gathering the data was only the first step for the system and program. The most important aspect was to make it usable by all of the departments of the company. The scheduling department, material purchasing department and tool service departments are some examples of where the data will be put to good use.

The SmartPress program stored the gathered data in a PostgreSQL database. PostgreSQL also is our enterprise back-end database. So, future application development will make SmartPress data visible in enterprise applications across the company.

Summary—Do-It-Yourself Information Technology

Our SmartPress system has moved from the test bed to the production floor. Looking back, we have created a flexible manufacturing monitoring system. Using Linux, the result is an expandable system that has been customized to the needs of the company. We can adapt the SmartPress for new manufacturing processes. The system also is upgradeable. In the future, we intend to rebuild the project on a newer kernel. For the upgrade, we probably will use Fedora Core as the base Linux platform.

The SmartPress system's low hardware cost is important as we are installing a system for each stamping press line. We cut hardware costs by using personal computers and DAQ boards that were supported under Linux. Our software development also was inexpensive. The time that Ryan Walsh spent on this project was similar to learning and developing in a proprietary controls language. Now, Ryan is fluent with kernel modules, real-time operating systems, PostgreSQL and GUI development. These skills are much more useful than learning one vendor's control programming language. For us, the do-it-yourself option resulted in lower cost, no vendor tie-in and upgraded developer skills.

Resources for this article: http://www.linuxjournal.com/article/7810.