ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUPHA021

EXPERIENCES USING LINUX BASED VME CONTROLLER BOARDS

D. Zimoch*, D. Anicic†, Paul Scherrer Institut, 5232 Villigen PSI, Switzerland

Abstract

For many years, we have used a commercial real-time operating system to run EPICS on VME controller boards. However, with the availability of EPICS on Linux it became more and more charming to use Linux not only for PCs but for VME controller boards as well. With a true multi-process environment, open source software and all standard Linux tools available, development and debugging promised to become much easier. Also the cost factor looked attractive, given that Linux is for free.

However, we had to learn that there is no such thing as a free lunch. While developing EPICS support for the VME bus interface was quite straight forward, pitfalls waited at unexpected places.

We present challenges and solutions encountered while making Linux based real-time VME controllers the main control system component in SwissFEL.

SWISSFEL OVERVIEW

SwissFEL is a 720 m long Free Electron Laser facility at Paul Scherrer Instiut. It provides femtosecond X-ray laser pulses with 100 Hz repetition rate to currently one (later up to three) photon beam lines with up to three experimental stations each [1].

SwissFEL Control System

The control system is based on EPICS [2], currently version 3.14.12.4. An upgrade to 3.16 is planned. The over 300 control nodes, IOCs (Input/Output Controllers) in the EPICS nomenclature, fall into five categories:

1. So called “softIOCs” either not controlling any hardware directly or only controlling IP network accessible devices. These run Scientific Linux 6.8 on vmWare virtual hosts. An upgrade to Red Hat Enterprise Linux 7 is planned.
2. IOxOS IFC1210 VME single board computers [3] running ELDK 5.2 [4] as provided by the manufacturer.
3. Camera servers running Microsoft Windows Server 2008 R2. An upgrade to Windows Server 2016 is planned.
4. DeltaTau Power PMAC motion controllers running ELDK 4.2 as installed by the manufacturer.
5. Moxa DA-661, DA-662 and DA-662A serial servers for controlling devices with RS232 or RS485 serial interface. These run embedded Linux versions installed by the manufacturer.

Scientific Linux 6.8 is used as well on all Consoles and many central servers.

IOXOS IFC1210

The IOxOS IFC1210 is a single board computer in VME 6U form factor. From controls point of view the main components are a Freescale P2020 PowerPC processor and a Xilinx Virtex-6 FPGA which are connected with PCI express. Other on-board components as well as some extension components are accessible though I²C.

The FPGA is used by various real-time applications, for example for low-level RF control. The “TOSCA II” FPGA framework [5] provides a PCIe bus bridge to three different hardware resources: User programmable FPGA functionality (“*USER*”) with access to two on-board FMC slots and to rear transition modules for I/O, the VME bus (“*VME*”) to access other boards in the same crate, and additional “shared” DRAM with dual access from the processor through PCIe and directly from the FPGA user logic (“*SHM*”).

Any of these TOSCA resources can map memory in 1 or 4 MB pages to PCIe and further to Linux user space. All can generate interrupts and all can be used in programmable DMA transfers.

EPICS INTEGRATION

Integrating devices into EPICS means accessing system hardware resources from a user space process using one or more of the following four methods:

1. Exchanging messages with the device. This is the typical access method for devices connected over network or a serial bus. In case of the IFC1210 this applies to the I²C devices.
2. Mapping device memory and registers for direct access by the CPU. Accessing device registers through memory maps is much faster than exchanging messages with the device. This is an important access mode for the VME, USER and SHM resources.
3. Transferring larger data blocks between the device and program memory efficiently. For large data blocks using specific DMA hardware is more efficient than keeping the CPU busy accessing mapped device memory. This is the second important access mode for VME, USER and SHM.
4. Handling device interrupts. Many devices signal when they need attention. This can be seen as a special type of message but deserves special attention because interrupts are asynchronous, they often do not contain all information why the device needs attention and thus require a handler which does additional register access. Interrupts are relevant for the VME and USER resources.

* dirk.zimoch@psi.ch
† damir.anicic@psi.ch

VME Bus Access in EPICS

EPICS has a long tradition of using the VME bus and many EPICS drivers exist to handle a number of VME based devices using the standard VME API built into EPICS since release 3.14. Thus once this API is supported by a new VME controller card, all these drivers can be used without change. The API provides functions to map VME address space to program address space and to register user functions as handlers for VME interrupt vectors. Thus to implement VME access one has to implement those API functions.

Difficulty: The EPICS VME API does not cover the various VME block transfers modes (BLT, MBLT, 2eVME, and 2eSST) which are translated to DMA by the VME bus bridge. We had to define our own API for DMA in EPICS.

EPICS Driver Model vs. Linux Driver Model

EPICS typically implements drivers for specific devices in the IOC program, only expecting the operating system to provide access to the bus or communication port these devices are connected to. This has worked well with vxWorks [6], the real-time operating system EPICS has originally been developed on, which had no distinction between kernel and user space and allowed direct access to the VME bus for memory mapping, interrupt handling and block transfer.

Difficulty: Real-time systems and Linux have fundamentally different ways to approach device access. In Linux, any access to hardware resources requires a kernel driver which makes those resources available to user space by the means of special device files. This driver is also responsible to handle possible concurrent accesses from multiple user programs and to manage device resources correctly even if a program is not well-behaving.

User Space Interrupt Handlers in Linux

Because of the very real danger to hang up the whole system when doing interrupt handling wrong, user space drivers as used in EPICS are somewhat frowned upon in Linux [7].

Yet user space drivers are possible – under the right circumstances. Good examples are user space drivers for serial (RS232) or network attached (TCP/ UDP) devices as well as for some USB devices. Here, the operating system only provides access to the underlying data transfer hardware (network interface or bus controller) but does not handle the actual device.

The most important prerequisite is that the device can be handled independent of the data transfer hardware. This is indeed the case with TOSCA. A kernel driver can handle the TOSCA bus bridge, while user space drivers can handle VME or USER FPGA devices.

The kernel driver needs to be very careful when handling interrupts because it is so easy to hang up the system if anything goes wrong here. While USER interrupts are edge triggered, and thus are self-acknowledging, VME interrupts are level triggered. Typically a device register needs to be read or written to reset the interrupt.

Because only the user space device driver can acknowledge the interrupt but on the other hand user space cannot execute while interrupts are active, the kernel driver must temporarily disable the active interrupt on the bus bridge until the user space device driver has handled the device. When done, the device driver must tell the kernel to re-enable the interrupt.

Still then there is the danger that a buggy device driver does not handle the device correctly so that the interrupt is not cleared and the system gets trapped in futile interrupt handling. Likewise any interrupt source without a registered user space handler must be disabled or the system may hang up.

Difficulty: Linux has no API for user space interrupts. In Linux, interrupts are usually handled solely in the kernel. A mechanism needs to be defined that passes control to userspace and that allows a user space driver to re-enable the interrupt. This context switch adds latency and jitter to interrupt handling.

User Space DMA in Linux

Reading larger data blocks from devices into EPICS can benefit a lot from DMA because it unburdens the CPU from moving data. While the DMA controllers built into TOSCA perform the data transfer, the CPU can perform other tasks.

The kernel driver must serialize concurrent DMA requests from different user space programs making use of the two available DMA pipelines in TOSCA. After the transfer has finished, the program must be notified and success or failure of the operation must be flagged.

Difficulty: There is no standard Linux API to handle DMA in user space. In Linux, DMA is used only in the kernel. One reason is that apparently contiguous memory in user space is often mapped to several non-contiguous physical memory pages. While this mapping is transparent for memory access by the CPU, it must be taken into account for DMA. Luckily the TOCSA infrastructure allows for scattered DMA, so it can be implemented.

VME Bus Access in Linux

The Linux kernel already includes a rudimentary VME driver framework but with several limitations: Only one VME bus bridge type is currently supported and neither interrupts nor memory maps nor block transfers are available from user space. The only access method is to pass a message with the address and the data to transfer to the driver using *lseek()* and *read()* or *write()* system calls.

Difficulty: Because of the context switch between user space and kernel space involved with any system call, this is by far too slow for a user space device driver with real-time performance. Furthermore it does not match the EPICS VME interface model which expects memory maps, so that all existing VME device drivers would require modification.

I²C Access in Linux

Linux provides a standard I²C user space API. Thus as soon as the I²C controllers on the IFC1210 are supported by a conforming kernel driver, all attached devices are accessible from user space using standard Linux methods.

Difficulty: The first implementation of the I²C controllers was not generic enough to be used with the Linux I²C API. It made too strict assumptions about connected devices and did not clearly distinguish device from bus controller as needed for a generic bus driver.

Manufacturer Provided Kernel Driver

IOxOS, the manufacturer of the IFC1210, has provided us with a kernel driver together with a user space API. This driver did not use the Linux VME API nor the Linux I²C API. Instead it was completely vendor and board specific. But it gave full user space access to all features of the TOSCA framework. Our first implementation of EPICS access to the IFC1210 used this API.

Difficulty: Unfortunately it turned out that driver and API had serious issues which could not be solved easily. In particular concurrent access to the memory map configurations, to DMA and to interrupts was a problem. But as EPICS is heavily multi threaded, proper locking and serialization is essential. For example, we need concurrent access from different processes, not only threads, to DMA channels. Thus implementing access locking in the program was no solution.

Furthermore resources like memory map windows needed to be released explicitly at program termination. This has caused problems when a program terminates abnormally due to errors. Often the whole system then needed to be rebooted to recover.

"Tosca" Kernel Driver

Lacking the necessary expertise to write or fix a Linux kernel driver for a complex device like TOSCA, we decided to outsource that task to a company specialized in user specific Linux drivers. The result is the "Tosca" kernel driver.

Special attention had been paid to proper resource management, including concurrency and automatic cleanup, and a good real-time performance, while at the same time trying to follow Linux kernel coding styles as closely as possible.

This driver is based on the Linux VME API but needed to modify it heavily in order to implement the required memory maps and in particular for user space interrupts and DMA transfers from and to user space program memory.

Difficulty: The driver developer needs a great amount of knowledge not only about the Linux kernel and drivers but as well about the device to implement, in this case the TOSCA framework. It is not easy to find both in the same person. This made the development very time consuming due to misunderstanding, misconceptions and a lot of communication overhead.

Interfacing the Tosca Kernel Driver

The Tosca kernel driver provides a character device file (/dev/bus/vme/m0) for memory maps. These include maps to VME address spaces as well as maps to other TOSCA resources (USER and SHM). A program can configure a window into those address spaces with *ioctl()* and then *mmap()* the window to program address space.

The kernel also provides a method to map program memory or USER or SHM address space to the VME bus. This allows other VME boards to access resources on the IFC1210. This "slave" map is configured in a similar way using a different character device (/dev/bus/vme/s0).

VME and USER interrupts are handled using individual character device files for each interrupt source. For VME there is one file for each combination of the 7 interrupt levels and 256 vectors (/dev/toscavmeeventL.V) while for USER interrupts there is one file for each of the 16 interrupt lines (/dev/toscausereventL). The file becomes readable when an interrupt has happened. Thus an interrupt handler thread can wait for one or more interrupts using functions from the *select()* family. Writing to the file acknowledges and re-enables the interrupt.

DMA between any TOSCA resources or program memory is performed using *ioctl()* on yet another character device (/dev/dmaproxy0). While many devices allow DMA only with contiguous physical memory and thus are often limited to kernel memory, we preferred to be able to use program memory directly, for example allocated from the heap with *malloc()*. This allows to transfer data efficiently into and out of EPICS records without any additional copy.

EMBEDDED LINUX

The manufacturer of the board had provided us with a Linux system based on ELDK (Embedded Linux Development Kit). This provides the cross-development environment and allowed us to easily build a boot loader (U-Boot), a kernel and a root file system. Also we could easily install all software packages we needed to run EPICS and other programs. However, kernel and boot loader needed several modifications from what ELDK provides in order to support all system components.

Boot Loader

The boot loader "U-Boot" [8] reads a system configuration (host name, boot server, …) from flash memory, configures the network port with DHCP and then programs the FPGA with a downloaded configuration file. It also downloads and starts the Linux kernel.

The boot loader had already been modified by the board manufacturer to support the board features like programming the FPGA. However further modifications were necessary to allow loading the FPGA configuration from the network.

Difficulty: Before having access to the network port, the PCI bus had to be initialized. But the loaded FPGA configuration file adds a PCI device to the system. It was necessary to add a mechanism to restart the boot loader after the FPGA has been configured.

Root File System

For easier software development we decided to use an NFS mounted root file system shared among all IFC1210 systems. Thus the systems do not need any local storage besides the boot loader configuration in flash memory.

In order not to disrupt the other systems, a shared root file system must not be writeable. On the other hand, Linux routinely writes a number of files, not only in /tmp and /var but as well in /etc. We solved that problem by mounting a ram file system to /var and /tmp and linking files like /etc/resolve.conv to locations on that ram file system.

Another issue related to the NFS mounted root file system was how to reliably pass DHCP leases obtained by the boot loader to the Linux run-time system because the IP address cannot change while the root file system is mounted. This may become a problem on busy networks, for example if EPICS clients flood the network with search broadcasts so much that DHCP times out.

Difficulty: Such a set-up is not one of the standard scenarios provided by ELDK. A considerable amount of work was needed to find a good set-up with a shared NFS mounted root file system.

Kernel

The kernel must allow control system applications to run in real-time with strict deadlines in the millisecond range.

Difficulty: Linux is not designed as a real-time (RT) operating system. There is an RT-patched kernel [9] available to improve the situation, but not for each kernel version and often not for the latest one. Also the RT-patched kernel is not as widely used as the standard one. Thus there can be bugs in the RT-patched kernel that are hard to get fixed. This gets worse if the platform is not wide spread in the Linux community, as it is the case for the P2020 CPU on the IFC1210. We have had such problems with the driver for the network interface build-in to the processor. It needed several kernel upgrades to get the problem fixed. However each kernel upgrade comes with API changes which made it necessary to modify the Tosca driver.

PERFORMANCE MEASUREMENTS

Interrupt performance tests have been made with a USER FPGA logic that generates up to 16 interrupts and starts counters at the same time. A user space interrupt handler triggers EPICS records to read the counter values. Histograms of the counter values show the latency distribution.

Using 16 simultaneous interrupts shows some clear peaks between 50 and 200 microseconds overlaid with a bell-shaped distribution between 50 and 1200 microseconds with a broad maximum around 550 microseconds when using a RT-patched kernel. This is sufficient for SwissFEL with a repetition rate of 100 Hz. Without RT patch a low rate of higher latencies in the range of several milliseconds can be observed which can make reliable 100 Hz operation difficult.

DMA performance depends on transfer direction and TOSCA resources involved. The most commonly used scenario is to read data into program memory. Using the CPU to read 32 bit words reaches 2.5 MB/s from USER or SHM and 2.0 MB/s from VME (accessing VME mapped SHM of the same board). DMA is more than 80 times faster, reaching 210 MB/s from USER, 380 MB/s from SHM and 165 MB/s from VME (using 2eSST mode) when reading 1 MB blocks. The smaller the block size the lower is the overall DMA performance because of an overhead of 133 microseconds for setting up the DMA transfer. Break even is reached when reading a few hundred words.

CONCLUSION

Supporting a new hardware platform in Linux is a difficult and time consuming task which requires up-to-date knowledge of current kernel developments in addition to expert knowledge of the hardware to support. The Linux kernel API changes rapidly and not much help can be expected from the Linux community for devices which are unusual or unknown to the mass market or for non-mainstream CPU types and in particular when not closely following the standard Linux way of implementing drivers. Real-time support is rather sporadic and may cause unexpected problems which can be hard to fix. Thus a lot of time and effort must be spent before a new hardware platform can be used reliably in a real-time control system. So even though Linux does not cost anything, nothing comes for free.

ACKNOWLEDGEMENT

We thank Stefano Babic (Denx) for the implementation of the Tosca kernel driver.

REFERENCES

[1] SwissFEL, https://www.psi.ch/swissfel
[2] EPICS, http://www.aps.anl.gov/epics
[3] IFC1210 Data Sheet, http://www.ioxos.ch/images/pdf/01_datasheet/IFC_1210_DS.pdf
[4] ELDK-5, https://www.denx.de/wiki/ELDK-5
[5] TOSCA II Data Sheet, http://www.ioxos.ch/images/pdf/01_datasheet/TOSCA_II_DS.pdf
[6] VxWorks, https://www.windriver.com/products/vxworks
[7] User-space I/O, http://yarchive.net/comp/linux/userspace_io.html
[8] Das U-Boot, https://www.denx.de/wiki/U-Boot
[9] Real-Time Linux, https://wiki.linuxfoundation.org/realtime