Teaching Device Drivers Technology in a Real-Time Systems Curriculum

Andrew Kornecki, Hank Wojcicki[*]
Dept. of Computing & Mathematics
Embry-Riddle Aeronautical University
Daytona Beach, FL 32114, USA
korn@db.erau.edu

Loïc Peltier[†], Janusz Zalewski
Dept. ECE
University of Central Florida
Orlando, FL 32816, USA
jza@ece.engr.ucf.edu

Natalia Kruszynska
NIKHEF, POB 41882
1009 DB Amsterdam, The Netherlands
natalia@nikhef.nl

Abstract

The paper describes activities related to development of device driver software - a topic very often left aside in most of the academic programs. With an individualized instruction, access to a well equipped laboratory, and self-motivated students we proved that the device drivers development can be successfully taught. The artifacts of the research and development are posted on the web and thus can serve as easily accessible teaching material for system developers. In the development, we used real-time operating system platforms, LynxOS and VxWorks, and explored Linux.

1. Introduction

One of the critical elements of modern computer technologies is software/hardware interface. The operating system is designed to simplify, for application developers, the access to various external devices. The parts of system software responsible for this are device drivers. For each new device a new driver must be developed and installed. In classic computer science and engineering education, overloaded with conventional material, there is no room for instruction on driver development.

The device drivers are written by developers intimately familiar with hardware operations. Historically, all drivers were written in assembly code being entirely platform specific. Currently, most of the development is done in a high-level language, where C is leading by a high margin. Since the development of hardware interfaces requires a detailed knowledge of the controller architecture, modes of operations, and control/status register contents, the hardware engineers were primarily involved in device driver development. The resulting code was often criptic and poorly documented, without appropriate requirements description, design specs, and adequate testing. The maintenance and modification of any drivers was extremely difficult.

Nowadays, when embedded systems developers are moving towards real-time kernels, there is a greater demand for software engineers that know both how hardware and the data structures of an operating system work. All of the skills a device driver developer needs to master are taught separately. The ability to program at the register level, the basics of processor operation, and operating systems concepts form this set. A course such as device driver development would be focused on the integration of these skills. The professional engineer's college education should include this aspect.

2. Device Driver Instruction

The authors feel that the art of driver development should find its way to university computing programs. However, it requires a significant hands-on experimentation, direct access to the equipment, detailed documentation, and support. For the last two years, both universities the authors are affiliated with offered special topics classes to teach this important element of software development.

2.1 Course Based on LynxOS and PC

This special topics class was designed around the software development cycle for device drivers in general, for

[*]Currently at Honeywell Space Systems, Clearwater, FL 32624, USA
[†]Visiting graduate student from IRESTE, Nantes, France

LynxOS [1] using a data acquisition board PC-LPM-16 from National Instruments [2] (12-bit 16-channel analog-to-digital converter and 8-bit digital I/O ISA module). In the course of one semester, the following tasks got accomplished:

- driver literature review and familiarization with driver structure, functionality and the development concepts
- experiments and in-depth familiarization with the real-time operating system (LynxOS)
- development of a simple test data character driver and simple A/D converter driver (read single channel conversion)
- development of a fully functional driver (multiple channel conversions, interrupts)
- compilation of the lessons learned in the form of handouts posted at ERAU Real-Time Web Server (`http://www.rt.db.erau.edu`).

The work was organized as a self-paced individual study with weekly consultations, progress reports, and frequent interactions in the lab and via email.

2.2 Course Based on VxWorks/VME

The major objective of this course was to write a device driver under VxWorks [3], for a data acquisition card on VMEbus. The VMEbus equipment used was Motorola MVME167 processor board [4] and two data acquisition boards from VMI Corp., VMIVME-2532A digital I/O board and VMEVME-4514A analog/digital I/O board [5]. The particular goals were as follows:

- experience cross development by using development tools on the host SparcStation and downloading executables to the target VME
- get familiarity with and configure the hardware for VMEbus equipment, the most frequently used hardware architecture in real-time applications
- get familiarity with the real-time kernel's I/O system and its data structures
- develop a functioning driver for the above mentioned boards, following an incremental approach
- understand procedures of adding a user-written driver to the kernel
- compare functionality of the driver with bypassing the driver functions to access an I/O device
- develop procedures for testing the driver using module functions
- study issues related to accessing a multifunction data acquisition card.

2.3 Driver Design Project

The experiences with the concepts led to a full project centered around device driver development. This project focused on the development of an ARINC-429 VME interface card [5] device driver for the VxWorks operating system [3] running on a 68040-based processor board MVME162 [4]. The ARINC-429 is a serial data bus used for point to point communication, that provides the data word format for transmitting navigational information. The VxWorks real-time kernel allows one to construct a device independent interface with the ARINC-429 VME interface card that conforms to the standard system call syntax.

The tasks accomplished in the graduate project resulting in delivery of fully functional driver for ARINC-429 intelligent communications controller were:

- hands-on experiments and familiarization with VxWorks environment and tools
- development of the driver installation module with scaffolding code (Build 0)
- adding transmit/receive functionality in a polled protocol (Buid 1)
- adding mode changes and buffer manipulation (Build 2)
- adding an interrupt-driven protocol (Build 3)
- compilation of lessons learned into the web handouts posted via ERAU Real-Time Web Server
- writing and editing the report and driver documentation.

3. Teaching Materials

When dealing with materials for this type of course, several issues need to be addressed beforehand to avoid unnecessary delays and setbacks. Below, we mention only necessary fundamentals, device driver structure, relationship of a driver to real-time requirements, and the design procedure. Equally important is the prerequisite knowledge or other type of good preparation related to hardware, particularly bus architectures, but this is out of scope of this article.

3.1 Device Drivers Fundamentals

There is an ample literature on developing device drivers, which we quote for convenience (books only) [13]. Depending on the operating system, one can find several kinds of device drivers. In principle, there are four types of device drivers: character drivers, terminal drivers, block drivers, and network drivers. Usually, those different types of drivers are categorized depending on the way the device driver handles the communication between the device itself and the I/O system of the kernel.

Character drivers can handle I/O requests of arbitrary size and can be used to support almost any type of device. Character drivers are mostly used for devices that must transfer data a byte at a time or for devices that work best with blocks of data that are not equal to the standard fixed-size buffers. Character devices deliver or accept a stream of characters (bytes). A subroutine puts a character on the list, or queue, and another subroutine retrieves the character from the list. The I/O procedure is synchronized through hardware completion interrupts: at each interrupt the device driver gets the next character from the queue and sends it to the hardware.

Terminal drivers are simply character drivers specialized to deal with communication terminals. They are responsible not only for sending to and from the user terminals, but also for handling line editing, tab expansion, and many others terminals functions. Because of the additional processing that terminal drivers must perform (and the additional kernel routines and data structures to handle this), it is useful to consider terminal drivers as a separate category.

Block drivers communicate with the operating system through multiple fixed-size buffers, which are usually 512 bytes. The operating system manages a cache of these buffers and attempts to satisfy user requests for data by accessing them. The driver is invoked only when the requested data is not in the cache, or when the buffers in the cache have been changed and must be updated on the device. Because of this, the driver is insulated from many of the details of the users' requests and needs only handle requests from the operating system to fill or empty fixed size buffers. One of the major differences between block and character drivers is that while user processes interact with block drivers only indirectly through the buffer cache, their relationship with character drivers is very direct. Unlike character drivers, data given to block drivers is position addressable. This position is usually established by the application program. Examples of using positional information generally involve seeking to a particular physical location on the device.

Network device drivers manage network interconnections and have a different processing model. While some of their functions are invoked by the kernel, as with others types of devices drivers, their primary interfaces connect to a protocol manager - a part of the network management code. Network device drivers must also be able to process unsolicited requests from the network. In this environment, the device driver may be the target as well as the initiator of I/O operations.

Applications access drivers via special device files that usually reside in /dev directory under Unix. These files are named the same way as regular files and are identified by the device type. Also, the major and minor device numbers associated with the special files can be viewed by listing the /dev directory.

LynxOS, for example, identifies devices using major and minor device numbers. A major device number corresponds to a separate functional unit that often has its own set of control registers, I/O address, and interrupt vector. A floppy disk drive, Ethernet card, or A/D converter card are all physical devices and each has a major device number associated with it. The major number is assigned to the device automatically during installation. A list of the installed major devices can be retrieved using the device utility program.

A minor device number identifies subunits or subfunctions of a major device. Minor devices of the same major device may share some resource(s), such as device registers or the interrupt vector. For example, a 16-channel A/D converter card will have one major number and 16 minor numbers, one for each channel. Minor devices are only necessary if there are multiple minor devices for a major device. For example, if the ADC only had one channel, we would not be concerned with assigning minor numbers. The kernel does not attach any special meaning to the minor number. The meaning is interpreted only by the device driver.

3.2 Device Driver Structure

A device driver code consists of a number of entry point routines and data structures. Every driver has two important data structures, the device information structure and the statics structure. These are used to install the driver and to share information among the entry point routines. The device information structure is a static file which is passed to the install entry point. The purpose of this structure is to pass the information required to install a major device into the install entry point where it is used to initialize the statics structure. The statics structure is used to pass information between the different entry points and is initialized with the information stored in the info structure. The kernel communicates with the driver via its entry point routines. The driver entry point routines, structures, and files related to the driver are named by placing an alphanumeric prefix in front of the routine or file name.

In addition to the routines that implement the basic I/O functions, driver's code also includes an init, device creation, and interrupt handling routines.

When the user calls one of the basic I/O functions, the I/O system responds by routing the call directly to the appropriate routine of the appropriate driver. The I/O system does this by maintaining a table that contains the address of each routine for each driver. Drivers are installed dynamically by calling the I/O system internal function, with arguments being the addresses of the I/O functions for the new driver. The installation routine enters the addresses of the I/O functions into a free slot in the driver table and returns the index of the slot. The index is known as the driver number and is used subsequently to associate particular devices with the driver.

A driver may be capable of servicing many instances of a particular kind of device (e.g. device with many separate channels that differ only in a few parameters). In the VxWorks I/O system, devices are defined by a data structure called a device header. This data structure contains the device name string and the driver number for the driver that services this device. The device headers for all the devices in the system are kept in a memory-resident linked list called the device list. The device header is the initial part of a larger structure determined by the individual drivers. This larger structure, called the device descriptor, contains additional device-specific data such as device addresses, buffers, and semaphores.

Character devices are added to the I/O system dynamically by calling the internal I/O function, iosDevAdd. The arguments to this function are the address of the device descriptor for the new device, the device name, and the driver number of the driver for this device. The device descriptor specified by the driver can contain any device-dependent information, as long as the structure begins with the device header. The driver does not need to fill in the header, only the device dependent information. The iosDevAdd routine enters the specified device name and driver number in the device header and adds it to the I/O system device list.

3.3 Device Drivers and Real Time

In many systems, the device driver supplies a few functions to perform low-level I/O operations such as input or output a sequence of bytes to a character-oriented device. The higher level protocols, such as communications protocols on character-oriented devices, are implemented in the device-independent part of the I/O system. The user requests are heavily processed by the I/O system before the driver functions get control.

While this approach is designed to make it easy to implement drivers and to ensure that devices behave as much alike as possible, it has several drawbacks. The driver writer is often seriously hampered in implementing alternative protocols that are not provided by the existing I/O system. In a real-time system, it may be desirable to bypass the standard protocols altogether for certain devices where the throughput is critical, or where the device does not fit the standard model.

In a VxWorks I/O system, user I/O requests are apportioned between the device-independent I/O system and the device driver itself. Since VxWorks is an operating system designed for real-time applications, minimal processing is done on user I/O requests before control is given to the device driver. The VxWorks I/O system acts as a switch to route user requests to the appropriate driver-supplied routine. Each driver can then process the raw user request as appropriate to its device. In addition, however, several high-level libraries are available. Thus the VxWorks I/O system makes it easy to write a standard driver for most devices and driver developers are free to execute user requests in nonstandard ways where appropriate.

3.4 Device Driver Design Procedure

While developing all drivers mentioned in Section 2, an iterative development process was followed. Such a process breaks the development cycle into more manageable phases (or builds), building a complete functional driver of increasing functionality at each build.

Building a hardware device driver and successfully integrating it into a system requires the following steps:

1. Learn and understand the hardware
   - capabilities of the peripheral and its register map
   - configuring modes of operation
   - sequence of events and timing relationships
2. Design the driver
   - use incremental development
   - add functionality one at a time (testing each step)
   - for each step perform rigorous requirements analysis, design, implementation, and testing
3. Code the driver
   - know well the implementation language and bit manipulations
   - document the calling sequence for the application
   - document carefully the entire code
   - document the process of creating executable and installation
4. Debug
   - the driver and the OS kernel are closely tied, so single stepping or breakpoint set-up is not feasible
   - simple mistake can corrupt the entire system
   - watch for intermittent and time-dependent bugs

- schedule extra time for contingencies

5. Integrate
   - install and test the driver during regular system operation
   - test the application-driven sequence of driver calls
6. Document the following
   - functional description and interface description
   - restrictions and examples of use
   - source code, compilation, linking, and installation process.

4. Comparison with Industrial Setting

All the above mentioned efforts, course structure and materials may still be not enough for a professional to begin device driver development in a production oriented industrial environment. In such an environment, what counts first of all is engineer's familiarity with a variety of bus structures, including not only VMEbus but also SBus, PCI, SCSI, etc. All those have to be taught separately and included in the developer's background [10].

Below, we overview major characteristics of a professional driver for PCI/VMEbus adaptor [6] running under Linux [7], `vmehb` (VMEbus Host Bridge). The `vmehb` driver is a loadable device driver that gives access to the VMEbus address spaces in a general way. It is suitable for data triggering, slow control and a variety of other data acquisition functions in any experiment equipped with VME hardware. The distribution [8] contains all necessary information, readme and user manual to install and use the driver on PCI architectures with Linux.

4.1 PCI/VMEbus Adaptor Driver for Linux/Unix

Several issues have to be resolved up-front in the specification to assure optimal design.

Assumption. VME address spaces should each have its own node (minor device number) assigned and its parameters should be constant during the use of the driver. The parameters are, at least, as follows:

- node name
- VME addressing mode (16, 24, 32)
- VME data mode (8, 16, 32)
- VME data swap style
- VME address modifier (fixed to the first two parameters)
- one of the parameters should include the minimal length allowed in DMA, if DMA feature exists.

Proposed implementation. In general, one should always try to implement six base VME address spaces: vme16d16 (compatible with Sun & NIKHEF VME access), vme16d32, vme24d16, vme24d32, vme32d16, and vme32d32. These address spaces should have the most popular VME address modifiers fixing the addressing mode (the first number) and the data access mode (the last number).

The standard structure should be easilly extendable. For devices using block VME access there should be separate nodes named: vmf24d16, vmf24d32, vmf32d16, vmf32d32. We can also extend the node repertoire adding more nodes whenever necessary. For example, if there is a need to make vmexd8 space, or when one device has atypical swap, keep the names familiar and in the style demonstrated above. Adhering to the standard will allow to use applications and libraries previously developed.

4.2 General Features of the Driver

The following are the basic requirements:

- The driver shall be loadable on all not VME based systems allowing dynamic loading.
- The driver shall cover to its best the possibilities of the hardware. If, e.g., the device has DMA feature, i.e., a possibility of releasing CPU from mastering the transfer, it shall be used.
- The transfers shall treat the VME part of the adaptor as a slave only. Let the VME processor driver developer worry about the mastering on VME and accessing Linux/Unix side as its master (many adaptors can serve both sides, but the driver shall serve only one side).
- The driver shall receive VME interrupt, whenever hardware allows and handle it correctly.
- Upon user request, the VME interrupt request shall be implemented, if allowed by the device.
- The driver shall implement the check-vme mode, using memory mapped access and putting extra check delay between the accesses, so the superuser can set it and check the whole configuration in the VME crate for the accessibility of all requested addresses.
- The driver shall be easilly switchable to normal mode by superuser.
- The hardware confilict between DMA and mmapp'ed access shall be avoided. If this is not possible, the two modes, run-time switchable by the superuser, shall be added.

- The driver is totally responsible for proper synchronization of multiuser accesses throughout all its nodes. The way of synchronization depends on the system used, needs of the device and style of the developer.
- The driver shall supply the facilities to allow development of drivers for particular VME devices.

4.3 Driver System Calls

The driver shall have the following functions:

- open() shall supply the user with the access to read()/write() and lseek()
- lseek() shall supply the start VME address of a given VME space; all the following read()/write() will start there till the next lseek() (on systems with 32-bit file count the standard system lseek() is enough)
- read()/write(), depending on the presence of DMA, shall have quick DACQ access (data acquisition mostly with the DMA access) and the last resource virutal sharing (when no DMA - seldom used) and mapping shared between the users
- mmap() shall give the user the access
- ioctl() shall serve all the rest, mostly for use by the superuser, including the following features:
  - stop VME crate allowing exchange of modules
  - start VME crate
  - choose the delay on check mode user gets if the crate is really started/stopped
  - get the node parameters (everybody and everything)
  - set the node parameters (superuser only, the user can get them as well)
  - interrupt request to VME (optional).

In addition, the diagnostic options shall include:

- the driver shall be able to clean its debugging messages by estabilishing proper compile time flag; the driver, when ready shall be mum, except in the case of trouble
- there shall be an option to make internal systrace.

4.4 Programming Practices

Developing bus bridge drivers needs an extra care in design, so does writing other bus adaptor drivers. These devices, in the same hardware set, usually cover the needs of two drivers, each on the other side of the connection. It is critically important to split the functionality and to decide which driver do we need. For VME driver, the PC side should use the PCI mastered access only, leaving the rest of hardware functionality to the drivers of embedded VME processors. It is important to make this clear, to avoid "driver" giving access to multiple registers on both sides, say, PCI regular and DMA as well as VME regular and DMA registers.

Other general practices should be observed as well:

- architecture dependence and the general configuration data should be all kept in one file, avoiding use of `#ifdef` in the code
- make one structure of the driver data and find a uniform way to access it so your code is readable; avoid separate global variables
- avoid leading underscores and other special characters, as they could be used by an operating system
- a good driver is the one that extends neatly; if you have to use complicated debugging tools it mostly means thast your programming or design is sloppy.

For the implementation, it is strongly recommended to fix and keep the general names of IOCTL flags. Example of a `vme_dev` structure containing the parameters described in the section on VME address spaces is attached in Appendix 1.

5. Conclusions

The art of device driver development is a time consuming but very rewarding and practical activity. It is in most cases an individual effort and therefore too often the developers are used to "hack the code". It appears that the critical factor to produce high quality device drivers is, in addition to technical competence in a form of knowledge of hardware and programming, an orderly development process. The spiral development with mutliple builds of increasing functionality is the recommended approach [9]. It is imperative to use, in each build, all four basic phases of software development (requirements, design, implementation, test). It is also advised to have a test plan prepared while developing the requirements. The proper documentation of the code, identification of all programming modules and the detailed instructions on compilation, linking, and installation is also critical. The documentation should also include detailed information about the hardware registers, binary codes, modes of operation, timing characteristics, etc. In the development, particular care must be taken to provide proper atomicity of code elements, mutual exclusion, and reentrancy.

As a direct result of the device driver training, students involved have been working on their job assignments in future product development in R&D. Assembly level tests on the bare machine, ROM monitor ports, and VxWorks BSP and device driver porting/writing are just a few of the tasks they have been doing almost immediately after graduation. Practical knowledge of system architectures, such as VME, allowed them to get involved in similar boards from different manufacturers and facilitated transition to other bus architectures, such as PCI, or even different processor architectures. In addition to all that, knowledge of new directions and technologies, to which industry is or will be soon moving, such as I2O [11] and Windows CE [12] was highly desirable. So is familiarity with the use of make and driver utilities, plus deep insight into driver performance is indispensible.

One comment which one of the authors heard a lot when he first started working at the industry was: "It's easier to teach a EE grad to program data structures in C than to teach a CS grad how hardware works." We strongly disagree with this comment: if one has the proper education, there should be no difference.

References

[1] Lynx Real-Time Systems, *LynxOS Real-Time Kernel Documentation*, San Jose, Calif., 1997

[2] National Instruments, *PC-LPM-16 Laboratory Data Acquisition Board Users Manual*, Austin, Texas, 1996

[3] Wind River Systems, *VxWorks Programmer's Guide*, Alameda, Calif., 1995

[4] Motorola Computer Group, *MVME 162* and *MVME 167 Single Board Computer User Manuals*, Tempe, Ariz., 1994

[5] VME Microsystems International Corporation, *VMIVME-2532A Instruction Manual*, *VMIVME-4514A Product Manual*, and *VMIC-6005 ARINC-429 Intelligent Communications Controller Instruction Manual*, Huntsville, Ala., 1994-96

[6] Bit3 Computer Corporation, *Model 617 PCI to A32/D32 VMEbus Adaptor Instruction Manual*, St. Paul, Minnesota, 1996

[7] Linux Operating System Documentation, `http://www.sunsite.unc.edu/pub/Linux`

[8] N. Kruszynska, VMEbus Host Bridge Driver, `http://sunsite.unc.edu/pub/Linux/drivers/char` and `http://nikhef.nl/pub/projects/vmehb`

[9] A. Kornecki, Spiral Software Development as a Methodology for Teaching Object-Oriented Simulation, Proceedings of Object-Oriented Simulation Conference, pp. 151-156, The Society for Computer Simulation, San Diego, Calif., 1996

[10] J. Zalewski (Ed.), *Advanced Multimicroprocessor Bus Architectures*, IEEE Computer Society Press, Los Alamitos, Calif., 1995

[11] L. Mittag, An Introduction to I2O, Embedded Systems Programming, Vol.10 , No. 10, pp. 44-50, October 1997

[12] S. Liming, S. Quintanilla, Device Driver Development for Windows CE, Windows CE Tech Journal, Vol. 1, No. 1, pp. 18-25, 1998

[13] P.M. Adams, C.L. Tondo, *Writing DOS Device Drivers in C*, Prentice Hall, Englewood Cliffs, NJ, 1990

A. Baker, *The Windows NT Device Driver Book*, Prentice Hall, Upper Saddle River, 1997

T. Burke, M.A. Parenti, A. Wojtas. *Writing Device Drivers: Tutorial and Reference*, Digital Press, Boston, 1995

E.N. Dekker, J.M. Newcomer, *Developing Windows NT Device Drivers: A Programmer's Handbook*, Addison-Wesley, Reading, Mass., 1999

J.I. Egan, T.J. Teixeira, *Writing a Unix Device Driver. Second Edition*, John Wiley and SOns, New York, 1992

J.E. Hanrahan, L. Leahy, *VMS Advanced Device Driver Techniques*, Digital Press, Bedford, Mass., 1988

K. Hazzah, *Writing Windows VxDs and Device Drivers. Second Edition*, R&D Books, Lawrence, Kansas, 1997

R.M. Hines, S. Wilcox (Eds.), *Device Driver Programming: Unix SVR4.2*, Unix Press, Englewood Cliffs, NJ, 1992

R.M. Hines, S. Wilcox (Eds.), *Device Driver Reference: Unix SVR4.2*, Unix Press, Englewood Cliffs, NJ, 1992

P. Kettle, S. Statler, *Writing Device Drivers for SCO Unix: A Practical Approach*, Addison-Wesley, Wokingham, England, 1993

R.S. Lai, *Writing DOS Device Drivers. Second Edition*, Addison-Wesley, Reading, Mass., 1992

S.J. Mastriani, *Writing OS/2 2.1 Device Drivers in C. Second Edition*, Van Nostrand Reinhold, New York, 1993

D.A. Norton, *Writing Windows Device Drivers*, Addison-Wesley, Reading, Mass., 1992

G. Pajari, *Writing Unix Device Drivers*, Addison-Wesley, Reading, Mass., 1992

A. Rubini, *Linux Device Drivers*, O'Reilly & Associates, Sebastopol, Calif., 1998

Appendix 1

```
/* get/set parameters of a minor */
#define VME_GET_SPACE  _IOR(VMEIOC,0,struct vme_dev)
#define VME_SET_SPACE  _IOW(VMEIOC,1,struct vme_dev)
/* driver modes of work          */
#define VME_CHECK_ACCESS _IOW(VMEIOC,2,int)
#define VME_SLOW_CONTROL _IOW(VMEIOC,3,int)
#define VME_DMA_CONTROL  _IOW(VMEIOC,4,int)
/* VME AM set/get                */
#define VME_SET_AM       _IOW(VMEIOC,5,int)
/* crate on/off                  */
#define VME_CRATE_OFF    _IO(VMEIOC,10)
#define VME_CRATE_ON     _IO(VMEIOC,11)
/* superuser  clear all maps    */
#define VME_CLEAR_MAPS   _IO(VMEIOC,61)
/* trigger the trace info        */
#define VME_TRACE_INFO   _IO(VMEIOC,63)
```