Using A Public Domain Real-Time Kernel On A VME/Ethernet Based Control System

David E. EISERT
Synchrotron Radiation Center, University of Wisconsin-Madison, 3731 Schneider Drive, Stoughton, WI 53589-3097, USA

There are many options available when choosing a real-time kernel from full-featured operating systems to writing your own kernel. The cost of these systems also covers a large range from tens of thousands to only a few hundred dollars. Recently there has been added yet another option, well documented real-time kernels that have been submitted to the public domain, these include Chimera (Carnegie Mellon University), RTEMS (U.S. Military), and µC/OS (Jean J. Labrosse). We selected the µC/OS kernel because it is small and has the required capabilities. Although the kernel was ported to the system in a matter of days, several months of development were required to write device drivers for the processor board. This paper will discuss why we selected the µC/OS kernel, the effort required to implement this kernel and our experience with the completed system.

1. INTRODUCTION

The Synchrotron Radiation Center (SRC) designed and built the Aladdin storage ring in the late 1970s. The original Multibus-based controllers were replaced with VME/Ethernet-based systems in 1986 [1]. The VME CPU boards used in that upgrade were based on a 12.5 MHz Motorola 68000 with a total on-board memory of 512 KB. A second VME board was needed for the ethernet interface and was based on the AMD LANCE chip set. The software for the systems was written in assembly language and was made to imitate the original Multibus systems. We wanted to migrate to a high-level language but the compilers available at the time produced inefficient code and consumed a large amount of system resources. Software from the VME CPU board vendor consisted of a monitor ROM and other commercial real-time kernel vendors lacked support for ethernet networking. We had no other choice but to write all the drivers ourselves. Much has changed since then, with both VME CPU boards and commercial software support for networking in real-time kernels. As a result we decided to replace our VME CPU boards sometime in the 1992-1995 budget period, but unfortunately the budget only allowed about $25K for the ten 6U VME systems used on the ring.

In order to understand our choice of a real-time kernel, some background into the selection of a CPU board is necessary. In early 1992 the SRC Optics group was working on a new beamline that required a dedicated control computer to implement a feedback loop for precise positioning of an optical component [2]. The SRC Controls group was called upon to assemble a VME system for this beamline. We decided not to use the older MC68000 VME CPU boards but to evaluate other boards that could later be used on the storage ring. We selected the Heurikon V3D with a MC68030 processor, an optional floating point coprocessor, a 32-bit ethernet coprocessor and up to 16 MB of parity protected RAM. The vendor offers two commercial real-time kernels with network support but the cost of the development system for these kernels exceeded our available budget. Software support was not a consideration in our case due to lack of funds to purchase a real-time kernel and the availability of GNU C++ for software development. In addition, we were purchasing only a single VME CPU board for this project with some prospect of purchasing more boards at a later date. It was very hard to justify the cost of a real-time development system for only a single board.

2. THE KERNEL

Since we had already decided not to purchase a real-time kernel, our original plan was to copy much of the assembly language kernel from the older VME boards. The older kernel simply remained suspended waiting for interrupts from periodic timer events, ethernet messages or other I/O device requests. The interrupt handlers would then process as much as they needed and place the address of a routine that would do further processing into a FIFO queue. After the interrupt returned, the routine would be pull off the queue and executed. This method lacked many desirable features available in a pre-emptive multitasking kernel but proved adequate for the tasks assigned to the VME systems at that time.

2.1 The µC/OS Kernel

As the device drivers for the on-board hardware were being written, *Embedded Systems Journal* published a series of articles describing the µC/OS real-time kernel [3,4]. µC/OS is a preemptive multitasking kernel with semaphores, message mailboxes and dynamic task priorities. This kernel was originally developed for 8-bit microcontrollers and as a result the code was optimized for size and efficiency. The author also tried to make the kernel as portable as possible and reduced the amount of assemble language code to a minimum. μC/OS does have some limitations including a maximum of 63 tasks, each task must have a unique priority and it does not include any of the utility tools normally included with a commercial kernel development package. μC/OS is technically not in the public domain. The license agreement allows distribution of the object code but not the source code.

The process of porting the kernel proved quite simple. The approximately 600 lines of C source code compiled with only a few changes to remove PC style memory declarations. The less than 100 lines of Intel 80186 assembly language were only used during context switches. The context switches occur only after an interrupt. A software interrupt is used to perform a context switch after a new task is added or a task has changed priority. An assembly language interrupt wrapper had already been written so that interrupts could call interrupt handlers written in C. The interrupt wrapper saves the registers, looks up the address of the interrupt handler for the given interrupt vector and calls the handler. When the handler returns, the interrupt wrapper restores the registers and issues a return from interrupt instruction. To handle the context switch a few lines had to be added to the interrupt wrapper. On entry to the interrupt wrapper, registers are saved onto the current task stack and the stack pointer is switched to the interrupt stack. As the interrupt handler executes, calls to μC/OS routines made from the interrupt handler may change the highest priority task ready to execute. When the interrupt handler returns to the interrupt wrapper, the highest priority task is found and the stack pointer is moved to that task's stack. When the interrupt wrapper issues the return from interrupt instruction, the new task starts to execute. The whole process of porting the kernel required only a few days.

2.2 Utility Routines

Many real-time kernels supply utility routines that include memory allocation, string handling, and software downloading. Unfortunately μC/OS lacks these routines but source code to many of these routines is readily available in the public domain. In fact, the Free Software Foundation supplies a very complete set of the standard C library and a C++ library. This library was considered, but since the library calls many kernel routines a complete port of this library would have been difficult.

Three basic functions were desired from the standard C library. The string handling functions such as *strcat* and *strlen*, the string formatting functions *printf* and *scanf* and the memory allocation functions. String handling and formatting functions were obtained from a public domain C library that was not as complete a the Free Software Foundation's C library. These routines compiled easily and did not contain any kernel calls. The memory allocation routines were acquired from an article published in the *Embedded Systems Journal* [5]. The routines presented there are well documented and integrated easily with the existing kernel and library routines.

The final utility routines consisted of downloading and debug monitor routines. In our case we had been using the ethernet for software downloading for many years and software downloading would be handled later in the network software. The debug routines that include examining memory and other system operations proved simple to write once the standard library routines for string handling and formatting were available.

3. WRITING DRIVERS

We had been using the network interface for downloading code into the older VME systems almost from the start of the 1986 upgrade. Thus one of the early goals was to use our ethernet based monitor, written for the older boards, to download code into the new boards. The serial port still remains a vital tool for debugging purposes and the serial port driver was implemented after the ethernet driver. A small version of the serial driver was used to dump out messages during the development of the ethernet driver. The last system driver needed was a timer driver. The timer driver is used for preemptive multitasking and for precise delays required by some tasks.

3.1 Ethernet Driver

The ethernet coprocessor used on the Heurikon V3D is the Intel 82596. We had previous experience with the Advanced Micro Devices LANCE ethernet chip set so many of the functions were similar. The main drawback to the Intel coprocessor is that it was designed primarily for little-endian processors with enhancements for big-endian processors. The big-endian mode handles only some of the data ordering problems and close attention to byte ordering is critical during driver development. One advantage of the coprocessor is that it can be programmed to ignore ethernet broadcast messages. At present our VME systems are still connected to the campus wide network through several bridges that eliminate all unnecessary traffic except for ethernet multicasts and broadcasts. The internet protocol uses a great deal of broadcast messages during normal operation and very infrequently "broadcast storms" on the campus wide network would swamp the older VME systems. We did not use a standard network protocol at the time this driver was being developed and only recently have we started to use the internet protocol.

The ethernet driver is composed of three main parts, the initialization of the coprocessor, the interrupt handler and the packet transmission routines. During the initialization phase the ethernet coprocessor is given its ethernet address, any multicast addresses it may accept and a link list of memory buffers to place received packet data. The coprocessor has many different memory organizations available for received packet data, the simplest of which is a link list of buffers. The interrupt handler deals with processing the received packets, cleaning up data structures after a packet has been transmitted and handling any errors generated by the coprocessor. When the interrupt handler is notified that a new packet has been received, it first checks the packet destination address to make sure it is either addressed directly to this system or it is a multicast packet that this system has been enabled to receive. The coprocessor does not completely filter all multicast addresses and other multicast packets can be received by the coprocessor. Next, the header is checked for the proper format and improperly formatted messages are discarded. Finally the packet is placed in a FIFO queue to be processed by a network command handler task. The last set of functions performed by the driver allow other tasks to get a properly formatted network message buffer and then pass it to the coprocessor to be sent out.

3.2 Serial Port Driver

The main task of the serial port driver is to stream bytes into and out of the serial port chip. This is implementated by having two circular queues per serial port, one queue for the received characters and one for the transmitted characters. Counting semaphores are used for both queues with the receive semaphore set to zero and the transmit semaphore set to the transmit queue size. When a character is received, the driver increments the receive semaphore. Thus when a task asks for a character string from the serial port with a call to *gets,* the task is suspended waiting for a non-zero count in the receive semaphore. When a character is received this task is made ready and the *gets* routine places the new character in the task's buffer. This process continues until the end of line character is received. When a task wishes to send a character string out of the serial port, the characters are placed in the transmit queue and the transmit semaphore is decremented by one for each character in the string. If the transmit queue fills before all the characters can be placed into the transmit queue, the task is suspended waiting for some characters to be sent and space to be made available in the transmit queue.

3.3 Timer Driver

The timer driver is designed to have very fast response and as a result bypasses most of the kernel. The timer driver builds a list of time events ordered from the shortest time delays to the longest. The driver adds a new timing event into the queue by subtracting all the time from events that occur before the new timing event in the queue. When the time has elapsed for the time event in the head of the list, the timer driver calls the routine associated with the time event at the timer interrupt level. This means that the routine must function very quickly or signal a task to complete the work outside of the interrupt level. One of the main timer events is the call to the kernel routine *OSTimeTick.* This routine decrements the delay counter of any task that voluntarily suspends itself a number of kernel time ticks. When the task's delay counter goes to zero, the task is made available to execute. When the timer interrupt returns, the context switch is made to the highest priority task available to execute. Another timer event is used for ramping digital to analog converters (DAC). The fast response of the timer driver allows for a time delay resolution in the tens of microseconds. With this resolution the DACs can be ramped very smoothly.

The timer driver uses a timer chip that derives its clock from a crystal oscillator. Crystal oscillators are reasonably good for short time periods but they display long term drifts. In addition, the timer driver may have difficulty obtaining the exact elapsed time between removing the time event from the head of the queue to starting the next time event in the queue. Therefore the timer driver makes small adjustments to the kernel time tick delay by using a battery-backed real-time clock. Every 30 seconds the real-time clock is read and compared to the elapsed time from the time tick. If a discrepancy exists, an adjustment is made to realign the clocks over the next 30 seconds. The small adjustments do not effect the timing of any other timing event but long term times that are obtained from the time tick are accurate to about 2 seconds per day.

4. SYSTEM TASKS

All systems contain a few common tasks needed for normal operation. These tasks include processing network messages, serial port messages and periodic I/O driver tasks. The network task parses the received ethernet packets and performs the operations detailed in those messages. The monitor task waits for any commands typed into a serial port and executes them. Finally, the driver task calls all the I/O drivers at a periodic rate so the I/O drivers can perform any operations off this event without creating its own timer event.

4.1 Network Task

The network task suspends waiting on the ethernet driver's receive packet semaphore. When a packet is received by the ethernet driver it places the packet into a FIFO queue and sets the receive packet semaphore. The network task removes the packet and inspects the header of the packet for a proper network protocol. Two protocols are currently supported, a private protocol developed in 1986 and the internet protocol. The network driver then removes the first data word in the packet. This data word is used in a *switch* statement to call a routine to process the rest of the information in the packet. The routine may load software, read or write information from a driver, or perform some other task.

We used our own packet format during the 1986 upgrade because we found a simple method on our VAX/VMS systems to send and receive raw ethernet packets. VAX/VMS has supported this raw packet I/O for many years and even after several VAX/VMS operating systems revisions it has remained unchanged. Back at that time we were considering DECnet as the network protocol but we never implemented DECnet because only minor software maintenance was required to support the raw packet I/O. We did not consider TCP/IP because there was only third party support for TCP/IP under VAX/VMS.

Recently it has become obvious that the internet protocol is the primary network protocol in use and we have implemented a subset of that protocol on our VME systems. Although formal implementations of TCP/IP require a minimum set of functions, it is possible to implement a much smaller set of functions. The minimum subset of TCP/IP needed for communication are the internet protocol (IP), user data protocol (UDP) and the address resolution protocol (ARP). The IP protocol is actually just 20 bytes of information inserted at the beginning of the network packet. Eight of those bytes are the internet addresses of the sending and receiving systems, two bytes are used for the total length of the packet and another two bytes are used for a checksum of the IP part of the packet. The other bytes select various options for packet fragmenting, data protocol and routing. These options can normally be set to a fixed value for all transmitted packets. The UDP protocol is inserted after the IP section and consists of an eight byte header plus the actual user supplied data. Four of those bytes specify the source and destination UDP ports, two are used for the length of the UDP part of the packet and the last two bytes are used for an optional checksum of the UDP part of the packet. The actual implementation of these two protocols required less than 50 lines of code in the VME systems. The ARP protocol defines a method for ethernet based TCP/IP systems to obtain the physical ethernet address of another system on the same ethernet segment. Basically a system will broadcast a simple packet that contains the IP address of the system for which it wants to find the physical ethernet address. Then all systems on the ethernet segment inspect the packet and compare their IP address to the address given in the packet. If their IP address matches the one in the packet, that system replies to the requesting system with a packet containing its physical ethernet address. Although the ARP protocol could be implemented very easily, we did not want to inspect all the broadcast packets on the ethernet segment. We avoided this problem by using a VAX/VMS system to handle the ARP requests for the VME systems. The VMS TCP/IP software can be set up to watch for other system's IP addresses on the ethernet segment and reply with the other system's physical ethernet address.

4.2 Monitor Task

The monitor task is used mainly for debugging software over the serial port. This task waits for character strings to be entered through the serial port, then it parses and executes the commands. The monitor task is not a complete debugger because it lacks features such as break points and watch points. Its main features are to examine/modify memory, display task status and display memory allocation status. All of these features are also available with the ethernet-based monitor software that executes on remote networked PC systems.

4.3 Driver Task

The driver task is used to periodically call all the I/O drivers on the system. When the driver is called, it can perform basic tasks that are relative to the kernel time tick. An example would be for a driver to read all the channels of a multiplexed ADC and log those readings over time. After the driver task has called all the I/O drivers on the system it voluntarily suspends itself for one kernel time tick.

5. FINAL INSTALLATION

As noted earlier, the initial implementation of the kernel and CPU board drivers was on a single storage ring beamline. The beamline system had several types of VME I/O boards but the storage ring uses different I/O boards. Thus drivers for the storage ring VME I/O boards had to be completed before the new CPU boards could be installed on the ring.

5.1 Hardware Drivers

Development of the storage ring VME I/O board drivers proceeded quickly due to many years of familiarity with the storage ring I/O boards. The first CPU board was installed on the ring within two months after they arrived. The rest of the CPU boards were installed over an eight-month interval to allow for normal operation of the storage ring.

5.2 Maintenance

Even though the new kernel had been running in the beamline system for over a year we still discovered two bugs in the ethernet driver. The first problem appeared when one system would generate a Spurious Interrupt after several weeks of operation. It turned out that the ethernet coprocessor would signal an interrupt request but when the processor issued an interrupt acknowledge the coprocessor did not respond. The board was sent back to the manufacturer but they did not find any problems. As more CPU boards were installed on the storage ring we noticed that some boards never demonstrated this problem and others would generate the error almost weekly. The problem was resolved by trapping the Spurious Interrupt in the ethernet driver and checking the status of the ethernet coprocessor. If the coprocessor status word indicated that it was requesting service, the ethernet interrupt service routine would be executed. The second problem was discovered recently that causes an ethernet coprocesssor fault from insufficient memory buffers for received packets. Apparently a burst of unfiltered multicast traffic on the ethernet segment arrived at such a rate that the processor could not discard the packets quickly enough. The ethernet driver should have reset and reinitialized the coprocessor but the code did not clean up some memory structures properly. The conditions that generated this fault are almost impossible to reproduce. They occurred on several occasions during a few month time period and then seemed to have stopped. Code that corrected the problem was placed in a few systems and those systems have shown the ability to recover from one such event.

Another indication of the quality of the kernel can be observed from the ability to modify the system software as hardware changes are needed. Recently we upgraded some of our analog converters from 12-bit converters to 16-bit converters. After the new 16-bit ADC boards were selected and ordered, the driver was written for the boards several weeks prior to the expected delivery. The driver was fairly complex requiring an interrupt service routine to handle the hardware scanning of the ADC. Coding of the driver required 400 lines of C and was completed in a single day.

When the board arrived, the driver functioned without a single problem. The first board was placed into operation on the storage ring two days after the boards had arrived.

6. CONCLUSION

Although the kernel only required a few days to port, the device drivers for the CPU board required more than a month of work. Many more months of work were required for the complete set of VME I/O drivers. Many believe that commercial real-time kernels are well worth their purchase price. One would clearly purchase a commercial kernel for large projects involving many programmers. For smaller facilities such as SRC the decision is not as clear. Three costs have to be considered when writing a kernel, the development cost, the project delay cost and the maintenance cost. We did not have any problems with project delays and maintenance costs have been minimal. The development costs were reduced substantially by using public domain code in the kernel.

This work was supported by the National Science Foundation. The current contract number is DMR-9212658.

References

[1] J. P. Stott and D. E. Eisert, Nucl. Instr. and Meth. **A293**, 107 (1990).

[2] D. E. Eisert, M. Bissen, M. Fisher, R. Reininger, J. P. Stott and H. Höchst, Rev. Sci. Instrum. **66**, 1671 (1995).

[3] Jean J. Labrosse, *Embedded Systems Programming* Vol. 5 No. 5-6, (1992).

[4] Jean J. Labrosse, *μC/OS The Real-Time Kernel,* R&D Publications, (1992).

[5] Leslie Aldridge, *Embedded Systems Programming* Vol. 2 No. 7, 28 (1989).