A GENERAL DEVICE DRIVER FOR ULTRIX OR LEAVE THE DRIVER TO /DEV/BUS

Gaylord Holder

MS-CIS-89-10
GRASP LAB 174

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104

February 1989

Acknowledgements: This research was supported in part by NSF grants DMC-8512838, MCS-8219196-CER, IRI84-10413-AO2, INT85-14199, DMC85-17315, NIH NS-10939-11 Air Force AFOSR F49620-85-K-0018, U.S. Army grants DAA29-84-K-0061, DAA29-84-9-0027, NOO14-85-K-0807, as part of the Cerebro Vascular Research Center, NIH 1-RO1-NS-23636-01, NATO grant 0224/85, NASA NAG5-1045, ONR SB-35923-0, DARPA grant NOOO14-85-K-0018, and by DEC Corporation, IBM Corporation and LORD Corporation.

A General Device Driver for Ultrix or Leave the Driver to /dev/bus[1]

Gaylord Holder
General Robotics and Active Sensory Perception Laboratory
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389
(holder@grasp.cis.upenn.edu)

Abstract

New hardware is the bane and the boon of the research laboratory: the boon because it brings new power, new capabilities, and new solutions; the bane because it means someone has to sit down and write the interface code for the board. /dev/bus attempts to simplify the process on the MicroVAX IIs running Ultrix 2.0 by allowing user processes direct access to the board's control status registers and Q-Bus memory. Unlike similar drivers for Suns, RTs and Masscomp, /dev/bus provides a means of establishing a user function as an interrupt handler. The delays and variability of the interrupt delivery are analyzed. Problems with the implementation are also described.

Introduction

Inspired by device drivers for the IBM RT and Sun which allowed user processes direct access to the bus, /dev/bus brings this capability to Ultrix 2.0 on the MicroVAX. After opening /dev/bus, a process can request access to both the Q-Bus I/O space and the Q-Bus memory.[2] In addition, the driver can signal a process when an interrupt comes in on a vector.

The General Robotics and Active Sensory Perception Laboratory (GRASP) at the University of Pennsylvania has been using /dev/bus for the past two years to provide user processes with access to:

- frame buffers.
- parallel I/O.
- analog to digital data acquisition.
- real-time clock for timing analysis.

/dev/bus has proved to be reliable, easy to use and versatile.

---

[1] This work was funded in part by National Science Foundation grant number DMC-8512838. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation. The author wishes to thank Robert King and Filip Fuma for their help and support in this work.

[2] The MicroVAX II uses a private bus for system memory which /dev/bus doesn't touch. Memory on the Q-Bus is usually provided by hardware such as frame buffers. This is the Q-Bus memory that /dev/bus provides access to.

The rest of this paper describes the user interface and implementation of /dev/bus.

User interface

After the user process opens /dev/bus, it may request a pointer to the Q-Bus I/O space, Q-Bus memory, or install an interrupt handler by calling the functions: `bus_getio()`, `bus_getqmem()`, and `bus_sethand()` respectively.

bus_getio()

The function `bus_getio()` is used to obtain a pointer the Q-Bus I/O space. The bus address of a device is added to the pointer to reference the device's CSRs (Control Status Registers). `NULL` is returned on error.

```
caddr_t    bus_getio(fd);
int        fd;
```

The CSRs for the DRV-11C 16-bit parallel communications board from Digital Equipment Corporation consist of a control word, output buffer, and and input buffer. A structure similar to the following may be used to interface with the DRV-11C:

```
typedef struct {
     u_short    csr;       /* Configuration word     */
     u_short    obuf;      /* Output buffer          */
     u_short    ibuf;      /* Input buffer           */
} drv_11c;
```

If the DRV-11C lives at `DRV_ADDR` on the Q-Bus, it may be accessed by:

```
drv_11c    *drv_p;    /* Pointer to DRV-11C's CSRs */

drv_p = bus_getio(fd);
drv_p = (drv_11c*)((u_int)(drv_p) + DRV_ADDR);
```

where `fd` is file descriptor returned from a call to open "/dev/bus". If `bus_getio()` should fail, it returns `NULL`.

bus_getqmem()

The function `bus_getqmem()` is used to reference memory on the MicroVAX's Q-Bus such a memory in a frame-buffer. A pointer to the Q-Bus memory is obtained by passing the /dev/bus file descriptor, the beginning address of the desired Q-Bus memory, and the size of the memory in bytes to `bus_getqmem()`.

```
caddr_t    bus_getqmem(fd,qbus_addr,nbytes);
int        fd;
caddr_t    qbus_addr;
int        nbytes;
```

The argument, `nbytes`, must be a multiple of 512. `NULL` is returned on error.

The Data Translation 2651 is a frame-grabber with two 512 by 512 by 8 bit frame buffers.

```
typedef struct  {
     u_char     fba[512][512];  /* Frame buffer A */
     u_char     fbb[512][512];  /* Frame buffer B */
} dt_fb;
```

Given that it is at DTQMEMADDR, the following code can be used to access the DT2651's memory.

```
dt_fb   *fbp;     /* Ptr to DT2651 frame buffer memory */

fbp = bus_getqmem(fd,DTQMEMADDR,sizeof(dt_fb));
```

Again, fd is the /dev/bus file descriptor.

bus_sethand()

A user process may attach a function to an interrupt vector with bus_sethand(). The /dev/bus descriptor, pointer to the function, and interrupt vector are all passed to bus_sethand().

```
int   bus_sethand(fd,fn,vec)·
int   fd;
int   (*fn)();
int   vec;
```

bus_sethand() returns -1 on error.

After a interrupt hander has been installed with bus_sethand(), /dev/bus will send a SIGINT signal to the user process whenever an interrupt is asserted. bus_sethand() takes care of calling signal() so (*fn)() will be executed with the driver delivers the signal.

The DRV-11C can be configured to interrupt the CPU whenever a word is sent and/or received. A simple(-minded) scheme to count the number of times a device at the vector VEC_ADDR interrupts is shown below.

```
int   interrupt_counter = 0;

interrupt_handler()
{
     ++interrupt_counter;
}

main()
{
     .
     .
     .
     bus_sethand(fd,interrupt_handler,VEC_ADDR);
```

```
        /* Don't exit until we're finished */
        for(;;)
             sigpause();
}
```

Because of the time required to process signals, this is not a highly reliable scheme if the device can interrupt more frequently than 500 times a second.

Implementation

This section provides a brief outline of the /dev/bus driver.

Initialization

The first thing the driver does when /dev/bus is opened is to lock the process in memory. This keeps the kernel from trying to do something stupid like swap frame buffer memory a process is accessing out to disk. In addition, `open()` allocates a data structure the driver will use to keep track of Q-Bus memory the process is accessing, the process id, and interrupt vectors the process is handling, so the process's state can be restored when it exits or closes /dev/bus.

Q-Bus Access

`bus_getio()` and `bus_getqmem()` work in pretty much the same way. The major difference is that `bus_getio()` returns a pointer to the Q-Bus I/O space, addresses `0x20000000` to `0x20001FFF`, while `bus_getqmem()` works with the address from `0x30000000` to `0x303FFFFF`.[3] A single common driver routine, `bus_qmem()`, is called by both `bus_getio()` and `bus_getqmem()`. It takes a pointer to the process structure, the beginning address of the the Q-Bus memory to be mapped in, and the number of bytes to be mapped.

`bus_qmem()` calls `expand()` to add PTEs (Page Table Entries) to the user process. The original value of the PTEs are saved so they can be restored later. The new PTEs are changed to reference the appropriate Q-Bus page frames and the pointer to the Q-Bus memory is returned to the user process. The only funny business about this whole thing is that it requires the swap space associated with the process also be expanded, otherwise, the operating system panics, thinking it somehow grew a process without remembering to adjust the swap space.

Interrupt Handling

When a user process establishes a signal handler, the /dev/bus driver records the vector in the data structure that it allocated the process when /dev/bus was opened. A small change to the assembler routine `_stray` in `locore.s` allows /dev/bus to pass device interrupts to interested processes.

When a device for which the operating system is not configured interrupts, `_stray` picks off the interrupt vector and interrupt priority. `_stray` was changed to call the /dev/bus

---

[3] MicroVAX Handbook, pp 5-34 to 5-36, Digital Equipment Corporation, Nauhua, N.H.

function, `bus_sigintr()` with the interrupt vector as an argument. This function runs though the /dev/bus data structures looking for a process which wants to handle it. If none is found, the `bus_sigintr()` returns and `_stray` logs the stray interrupt. If a process has established an interrupt handler for the vector, `bus_sigintr()` calls `psignal()` to deliver a `SIGINT` signal to the process.

Interrupt Latency

One of the original uses planned for /dev/bus was to provide a real time capability for user programs. The idea was that a device could interrupt the user process which would do its thing. This would be great for robot control since all the code would be running in user space on a single machine. Standard debuggers could be used to make sure the code worked and life would be just peachy. The big question was how long it took the interrupt to wend its way through Ultrix and kick in the user's interrupt handler and how variable the times where. An experiment was set up to learn how long it took until the kernel got a-hold of the interrupt, and then how long it took to pass to the user process.

A KWV-11 real time clock board was set to run at 2 MHz. It was then set to wait 2 milliseconds and then interrupt. The time between the interrupt and resetting the KWV-11 was recorded. The interrupt routine would set the KWV-11 to interrupt in 2 milliseconds to give the interrupt handler to return and restores the process's normal context. A total of 10 trials of 10,000 samples were run with the interrupt handler in the kernel and from the user process. The trials were run on a MicroVAX II with 5 Mb of memory, a quiet network, standard user priority and a version of Ultrix 2.0 modified for /dev/bus.

As figure 1 shows, the interrupt latency for the user process averages to about 0.7 milliseconds while the kernel latency, as shown in figure 2, is about 0.12 milliseconds. The MicroVAX's 10 millisecond clock shows up in figure 2 quiet clearly.

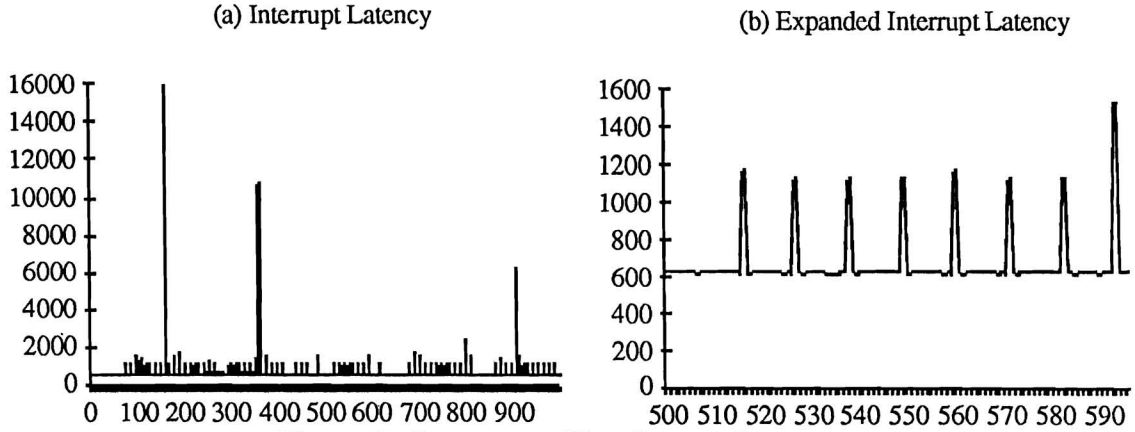

Figure 1: Interrupt to User Process Latency

The kernel interrupt is being called lowest hardware priority. However, the only thing that it do to bring in the user process and still clear the interrupt stack (remember, there isn't a clean way to get back from the user process context to the kernel interrupt stack) is to set up an AST (Asynchronous System Trap), which is essentially what Ultrix's signaling mechanism does, or call `psignal()` as /dev/bus already was.

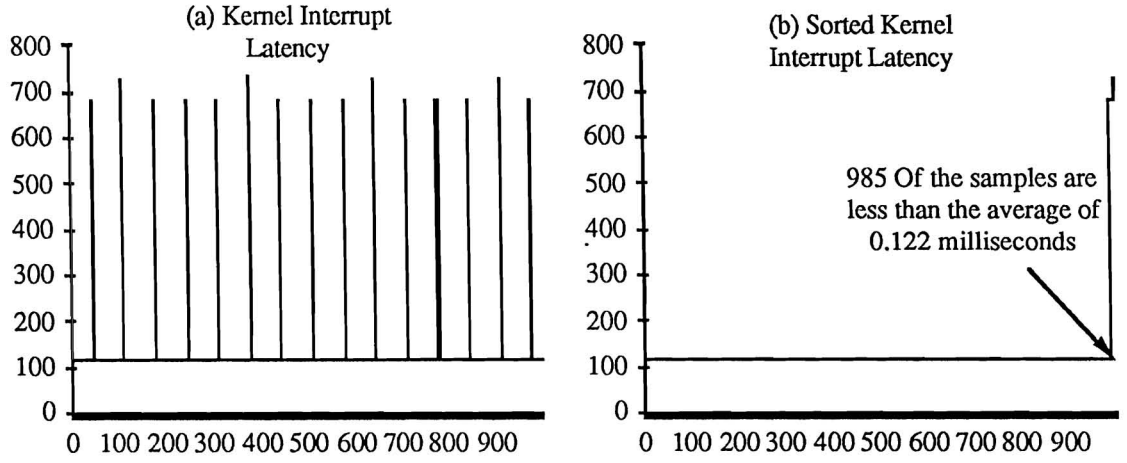

Figure 2: Interrupt to Kernel Latency

From these, and other, experiments clearly showed that the variability of the interrupt latency is coming after the setting up the AST. Other hardware, kernel housekeeping, network traffic and what not was being handled at the expense of the interrupt handler's reliability and responsiveness.

Although the latency from interrupt to user process was less than 7 milliseconds 90% of the time, see figure 3, the enormous variability destroyed any hope of using /dev/bus for real time work.

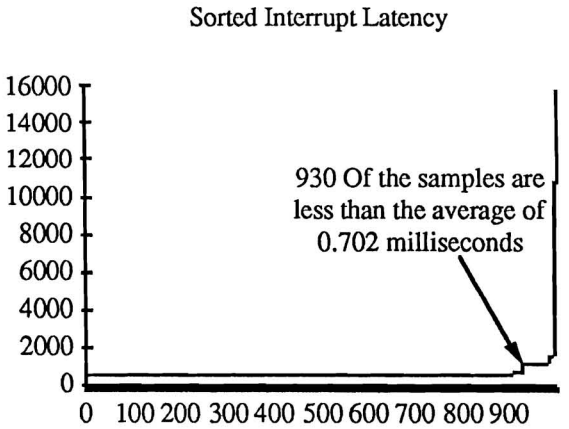

Figure 3: Sorted Interrupt to User Latency

Clean Up

The user interface doesn't provide any means of removing an interrupt handler or restoring the process's original memory map, but the kernel must. When a process closes /dev/bus, or it exits, the kernel function `bus_clear_uinfo()` is called to restore any PTEs that /dev/bus used. Any interrupt handlers the process had installed are also removed.

Other Issues

So far, with the exception of the modification of `_stray`, the implementation of /dev/bus has been completely in the driver module. Unfortunately, there are a few more patches that must be made to get everything working.

More Cleaning Up

The biggest problem /dev/bus has been getting the operating system to call `bus_clear_uinfo()` at the right time. During the initial design of the driver, it had been assumed that driver's close function would be called before any of the process's memory was de-allocated -- or at least there was a single procedure where processes were dismantled. It was a great surprise to find not only is the process's memory returned to the system's memory pool long before the devices are closed, but also that it is done from several routines.

In the current implementation, `bus_clear_uinfo()` is called from `vmemfree()` in `./sys/vm_mem.c`.

Protecting the User from Him/Herself

One of the other modules which had to be modified was `machdep.c`. Without this change, a user process could request memory that wasn't really there, for example the memory address to `bus_getqmem()` could be wrong. When the process went to use the pointer, it would reference non-existent memory, and the system would panic. A check was added to see if the current process was using /dev/bus. If so, it was assumed to be responsible for problem. This way, a process making an invalid reference exited with `SIGBUS` rather than taking the whole system with it.

Memory Allocation

One of the bigger surprises was that the `sbrk()` keeps a local copy of the size of the process in the global assembler variable `curbrk`. When a process called `bus_getio()` or `bus_getqmem()` the size of a process changed without `sbrk` updating `curbrk`. Thus, the next time the process decided to print something with `printf()` (which calls `malloc()` which can in turn call `sbrk()`) `sbrk()` decides to set the process size based on the value in `curbrk` which is usually a good deal smaller than the real size of the process. The kernel then tries to free the newly allocated PTEs which point to the Q-Bus and panics when it realizes that someone has been tampering with its processes.

To fix this problem, `bus_getio()` and `bus_getqmem()` call a routine `fix_curbrk()` with the number of bytes to add to `curbrk`. This way `sbrk()'s` notion of the process's size matches reality.

Problems

There are still some problems.

/dev/bus should provide some support for multi-user access. At the moment, no checking is done to see if the Q-Bus memory or interrupt vector requested by one process is already being used by another.

/dev/bus also ignores problems that might crop up if a process using /dev/bus decides to spawn a child. Since both the interrupt and illegal memory reference features look up a process based on the recorded process id, `fork()` can be a nasty problem. An associated problem has to do with debugging. While a process using /dev/bus can still be run under any of the standard debuggers, care must be exercised when looking a variable values. If the /dev/bus process has a pointer out to Q-Bus memory, it is fine to look at the value of the pointer -- but looking at the contents of what the pointer references sends the whole system into an uproar. This is because the process being run under the debugger had its PTEs mangled by /dev/bus, while the debugger didn't.

The only board with DMA used on the GRASP laboratory's MicroVAXen is the Ethernet Interface. This is not one to use to debug the DMA facilities of /dev/bus, so the DMA support isn't.

Finally, /dev/bus still occasionally will crash the machine. For the most part, the machines are as stable as any other Unix box. But once and a while, especially if there is a system is to be demonstrated, the system will panic in `vrelvm()` or something and away it goes. Sigh.

Conclusion

/dev/bus was originally designed so that user processes could interface with new hardware without the operating system overhead or writing a new device driver. On the whole, that goal has been met. New devices can be installed and a rough set of interface libraries written in only a few days. The best part is that usually, the system only has to be rebooted for the physical installation of the device.

While /dev/bus doesn't make device drivers redundant, it does give the system's programmer a chance to work with the device before having to plunge into the kernel. This pulls more of the development work out of the kernel and shortens the time need to write the device driver. Best of all, it gives knowledgeable users the tools needed to write their own interface software.