TITLE: Writing Device Drivers - Interrupts
IBM Developer Connection News Volume 3
by Steve Mastrianni

I get a lot of calls asking "What is the interrupt system and how do I program for it?" To answer those questions, this article provides a beginner's view of the interrupt system.

In the traditional Intel-based PC, the interrupt system simply is a collection of electronics that let external events interrupt the program currently being executed. One of the electrical components, the Programmable Interrupt Controller (PIC) is notified of the external event, and in turn notifies the CPU that an interrupt has been received. The CPU saves the address of the code it was executing, then calls a special address assigned to that interrupt level, or IRQ, to perform the interrupt processing. When the interrupt processing is complete, the CPU resumes executing at the next instruction in the thread that was executing at the time of the interrupt. Of course, it's more complicated than that, but those are the basics.

Today, most PCs have 15 different interrupt levels, provided by two 8-channel PICs. One IRQ, IRQ 2, is unavailable because it is used to cascade or piggyback the two PICs. Older PCs, such as the IBM XT, had only one PIC. Therefore, they were limited as to the number of interrupt adapters that could be installed. With larger systems, even 15 interrupts can sometimes be a limitation. ISA bus systems use edge triggered interrupts, a limitation which precludes the sharing of interrupt levels. MicroChannel and EISA bus machines use level triggered interrupts that let adapters share interrupt levels. OS/2 currently supports up to four Micro Channel or EISA adapters for each IRQ.

In DOS, you could hook interrupts by replacing a pointer in the interrupt jump table in the base page. In OS/2, however, you must register for the interrupt notification by calling the Device Helper routine, SetIRQ . One of the parameters to SetIRQ is the address of your driver's interrupt handler. The OS/2 kernel fields all interrupts, and notifies your driver that an interrupt has been received by calling your interrupt handler.

The interrupt handler should perform it's operations and reenable interrupts quickly. The last thing the interrupt handler should do before exiting the interrupt handler is to call DevHlp EOI . This resets the interrupt logic, letting another interrupt occur. If you don't issue the EOI, your driver won't receive any more interrupts. If you leave the interrupts disabled when you exit your interrupt handler, the OS/2 kernel reenables them.

Always try to eliminate or limit the nesting of interrupts. Interrupt nesting will be necessary if your driver receives an interrupt while already processing an interrupt in the interrupt handler. The logic for nesting can be complicated and difficult to debug. The interrupt handler should never nest more than two interrupts. Use caution when designing interrupt handlers for Micro Channel

and EISA bus systems because the interrupt handler is entered with interrupts enabled.

A question that I get most often from first-time driver writers is "I was debugging my device driver, and my driver received a IOCtl packet, category 5, function 0x48. My application never issued this. Where did it come from, and what does it mean?

The kernel issues the Category 5, Function 0x48 Code Page IOCtl, with the purpose of finding out if your driver is a printer driver. If it is not, you can just return ERROR_BAD_COMMAND. Or, you can ignore it by returning DONE.

Steve Mastrianni is an industry consultant specializing in device drivers and real-time applications for OS/2. The author of "Writing OS/2 2.1 Device Drivers in C", Steve is regarded as one of the industry's leading experts in OS/2 and OS/2 device drivers. Steve Mastrianni, an OS/2 Evangelist for OS/2 Device Driver Development and Support, can be reached on Compuserve @ 73354,746 or Internet @ stevemas@vnet.ibm.com.

TIPS

Tip: For device driver development, use a machine with FAT partitions. Crashing an HPFS machine during driver development can result in files being damaged and lengthy reboots.

Tip: For debugging, instead of using an ASCII terminal, use an old PC with a communications program, like Procomm or Telix. This allows you to enable data capture and keep a record or your debugging sessions. Keep a running record of the contents of all the registers by placing a breakpoint at the desired locations. and change the kernel debugger's default command to r;g. The r causes all the registers to appear and the g continues execution.

Tip: Place your Init section at the end of your code. Return a pointer to the beginning of the Init section at the end of Init. This releases the memory occupied by the Init section. Don't try to save space or time during Init, neither one is an issue since Init is called only once during system boot, and the memory can be given back to OS/2 at the end of Init processing.