High level Languages and IO Access in Computer Interfacing

Administrivia: HOWTO pages on the web.

Introduction to HLL

Peripherals can be accessed from assembly language and can also be accessed from High Level Languages (HLL) such as C. There are advantages to both approaches and, in fact, we generally use a mixture of C and Assembler in operating system programming and peripheral interfacing.

Background: The C language

C was originally written to help develop operating systems. Dennis Ritchie designed it for UNIX. The UNIX OS, the C compiler itself and most UNIX application programs were (and still are) written in C or its direct descendant C++. You can contemplate the self-referential aspects of an operating system, running an application program which is a compiler which is being used to create the operating system.

C is "*a general purpose programming language which features economy of expression, modern flow control and data structures*" (Kernighan & Ritchie, *The C Programming Language,* 1978, Preface. Commonly known as "K&R". If you believe in reading the original source documents and so choose to read the original K&R note that the Preface also states, "This book is not an introductory programming manual." This statement is correct. The book's style is as terse and to the point as the C language itself. In other words, once you understand C it is interesting and useful to read K&R to explore the fundamental definition of the language.

The C/C++ language is still the dominant language in systems development today. Java is also a direct descendant of C./C++ it uses a very similar syntax and programming style.

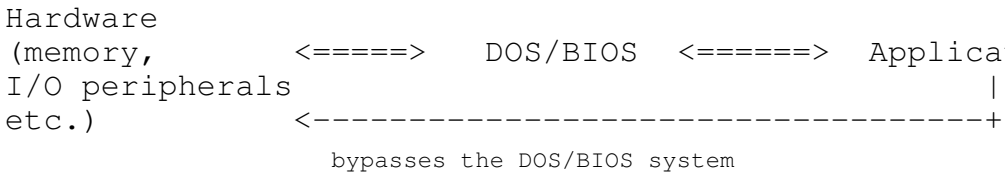
Accessing peripherals in various operating systems

Accessing I/O is, or should be, a function of the operating system. Experts have mixed feelings about I/O access. Stallings (p471) says, "Perhaps the messiest parts of the design of an operating system deal with the I/O facility and the file management system. ... The I/O facility is truly the performance battleground.", while Tannenbaum (p205) says, "One of the main functions of an operating system is to control all the computer's input/output devices." Many programmers would like to ignore hardware completely. Some CS proponents go so far as to claim that hardware can be ignored in system design. This point of view is not supportable in an IT context. Not only is the hardware a major determining factor in system performance, it is also the bottleneck in communications and in Human Computer Interfacing and therefore is certainly an area which requires attention.

It should be noted that there is a significant difference among peripheral IO devices and systems. For example the table in Chapter 11 of Stallings (fig 11.1) shows data rates that vary from 10^2 bps to 10^9 bps. These differences in bandwidth (and related latency differences) lead to significant differences in the approaches we use to accessing the relevant peripherals. However there are both high level and low-level common issues with interfacing devices which can be identified and discussed. We will consider these issues for different operating systems and several different types of device.

MS-DOS device interfacing

Interfacing hardware, OS and applications under DOS follows this simple model.



The OS is designed for applications to access the hardware through DOS/BIOS interrupts and calls. (INT 10/16/21 etc.) It is well-known that this can give poor (slow) performance and so traditionally many SW houses wrote their own code to access the HW (as we have done) and that code by-passes the OS.

This can be taken to extremes. At a time when WordPerfect controlled more than 50% of the word processing market they used to supply drivers for just about every printer and graphics card ever made. The printer and graphics drivers occupied as many disks as the program itself and were constantly being updated. They had complete teams of programmers writing new drivers for new printers and graphics cards as they were manufactured. Thus WP had the best and fastest interface to your monitor and printer but at a very high cost. The skilled programmers developing new driver code were not developing new WP code.

Bypassing the OS to reach the hardware can solve the performance problem but unfortunately it buys us a much bigger problem, especially for multi-tasking systems; namely there can be multiple users contending for the same resources. The OS should manage this process but cannot if it has been by-passed. If there are multiple users contending for resources then deadlock is possible.

========== What is Deadlock? ==========

Deadlock: When more than one task needs resources they can end up waiting indefinitely for them. For example, if task A needs resources R1 and R2 and already has control of R1 and task B also needs resources R1 & R2 and already has control of R2 they are locked in a "deadly embrace" or "deadlocked". This manifests itself to the user as one of the ways your system can hang. See Stallings section 6.1, 6.2 and 6.3 for a discussion of deadlock and how to avoid or prevent it. Also Liu (pages 280-295) below for a detailed discussion of how to handle deadlock in real-time systems.

==========

To return to the WP example. WP wrote their own "print spooler" routines which would allow your printer to run in the background while you continued to work. This was a neat piece of pseudo-multitasking code at a time when multitasking was not readily available on the DOS-based PC. When Windows started to dominate the scene Microsoft wisely unloaded the problem of creating drivers for the myriad peripherals onto the peripheral manufactures and started to provide multitasking capabilities within Windows.

In later incarnations of DOS, such as WinXX, and also in other systems such as UNIX/Linux, protection of the system was provided for by allowing programs only to run in a protected space. The OS cannot be bypassed. If any program tries to access anything outside its dynamically allocated memory space an exception (error) is generated. The program cannot access the hardware directly at all.

Insert diagram showing concentric circles. from outside to in Users / Applications / Utilities and services / Kernel / hardware.

The problem with this diagram is it gives the impression that hardware is encapsulated and static (could turn it inside out?). It also ignores the fact that Users interface with the system through hardware. As pervasive computing develops users interface in increasingly different ways with the system - Voice, handwriting, gestures, eye-tracking, etc. Twenty years ago we had keyboard screen. The mouse, trackball, glidepoint, torsion-stick, joystick, touch-screen, lightpen, barcode, sketchpen-tablet interfaces were considered radical

and far fetched. they are now common.

Modern CPUs have architectures that support systems like this with functions like *protected memory.* In Wintel architectures these are called "Rings". Ring 0 has the highest (Kernel) access, Ring 3 has the lowest (user) access. Rings 1 and 2 are not commonly used. In the DEC Alpha CPU architecture there two modes simply called "kernel" and "user"

When applications need to access a peripheral device they access a device driver under control of the OS. The device driver can access the hardware. The OS and devices driver are written to prevent hardware conflicts.

Device drivers allow for separation of the operating system and the hardware. Thus you can have an operating system which can run on many different types of hardware. Similarly the operating system (kernel) and the device drivers together present a uniform interface to the applications. The applications do not have to adapt for different hardware or devices, they merely send and receive data from the OS/device driver. Application programmers design applications with little or no knowledge of hardware. On the other hand peripheral designers don't have to design applications, they merely provide a standardized interface (device driver) for the OS.

So writing code to access hardware is a process of writing device drivers that will be loaded and will run under control of the OS.

In Linux for example we have

```
Applications  <===> KERNEL <===> (Virtual      <===> Drivers w.
in allocated                      File System)       HW access
memory
```

Linux/UNIX sees all devices as (virtual) files.

How to design I/O access code

Three concepts of access:

1. Programmed: Issue a command and wait for a response.
2. Interrupt-driven: Issue a command and an interrupt from the I/O device will signal completion.
3. DMA: Send a request for data. The block is DMA transferred and then an interrupt occurs.

1. Programmed: In this approach every time something is needed from a peripheral the software sends a command to the peripheral and waits for a response. Programming is explicit. Timing is controlled directly by the code. Flags may be used to allow multitasking but interrupts are not used.

2. Interrupt driven: The peripheral is set up to generate interrupts when it needs attention. Thus the peripheral rather than the main program code is in control of the I/O function. If data is being sent out then usually data is put in a buffer and the peripheral is started., When the peripheral is ready for more data it interrupts the main program. Software is more difficult to write and debug because the interrupts are asynchronous with respect to the main program.

3. DMA is usually used for high speed data transfer. the transfer from peripheral to memory (ram) takes place without the intervention of the CPU. Care needs to be taken to ensure that data is appropriately synchronized. Since the CPU is not controlling when the data arrives in RAM some mechanism is required to guarantee the validity of the data in the memory when the CPU needs to use it.

Appendices

- Re-entrant routines
- HLL in tiny systems
- Mixing C and ASM
- Borland C/C++ IDE notes
- Linux device drivers
- Windows Device drivers

Reentrant Routines: An additional consideration that becomes important, especially with multithreaded systems is reentrancy. A ***reentrant routine*** is one which can be called by multiple processes at the same time (before it has finished executing) without getting confused. The routine allocates a separate data area for each user or process that calls it. It only changes data in that area and never changes data or flags within the program which could confuse a subsequent call to the routine. Thus if the routine is interrupted and called by a subsequent user before it is finished with one user it saves all the first user's data and creates a new data space for the interrupting user. When permitted to do so (by the OS) it ***reenters*** the first user's data area and continues executing its code.

DOS and BIOS interrupts are not reentrant. This is one reason why DOS cannot be used for multitasking. UNIX/Linux and WinNT having been designed as multitasking systems support re-entrancy much better.

HLL in Tiny Systems

Most interface programming in current IT practice is done in C. Originally computers and memory were expensive and programmers were cheap. The pendulum swung the other way, hardware became cheap and large memories on small devices became more practical. HLL's became very cost effective. The situation is now more complex. Either computers or programmers could be expensive in different situations. *Discuss.* C, however, is a very efficient language, much more structured than ASM and suitable for all but the tiniest of embedded systems.

Relationship between C and machine level access. Discuss.

How to access hardware in DOS using C and ASM: Summary of mixing machine level access with High-level language (HLL)

- Write ASM modules, assemble to OBJ and link with HLL OBJ modules
- Write single ASM commands or complete ASM subroutines inside C using the ***asm{}*** keyword.
- Access machine level peripherals using special HLL keywords/functions such as ***inpw(), inport(),outp( ), outportb()*** for IN and OUT,
  access memory bytes directly with ***peek()*** and ***poke()***
  int86(), intdosx(), intr() for interrupt calls and your own interrupts,
  and so on. See the Borland C help files for further details.

See attached example of sending a tone via 8254 timer and timer-control PPI

Notes: Borland C/C++ compiler (DOS)

1. Pressing F1 while inside Borland C displays the *Help* menu.

2. Highlighting a function (or part of one) and pressing Ctrl-F1 takes you to the *Help* page for that function. Many *Help* pages include working examples of code.

How to access hardware in Linux - Device Drivers: Different types of devices are defined. UNIX has **character** and **block** devices. See */dev*. Character is byte-by-byte sequential (think of a tape), block is random access and in multiples of the block size (non-multiples are possible in UNIX). E.G. printer=char, HDD=block.

See Stones and Matthew, "Beginning Linux programming" 2nd ed. chapter 21 for more detail.

Writing device drivers for Linux is similar in approach to writing device drivers for DOS but there are a number of conventions and OS considerations to be taken into account. The Kernel does not directly access memory or I/O by address and so the device drivers (DDs) have to translate actual addresses to OS-compatible virtual addresses. The DDs also need to do kernel based operations such as checking whether a particular device is already in use before taking it and then registering the fact that they have it. When the DD is finished it must release the device so other DDs can seize it.

Linux typically recognizes *character* and *block* devices. Character devices are typically things like serial and parallel ports, keyboards etc. Block devices are read in blocks and are typically things like disk drives

Accessing I/O in Linux using existing device drivers

- It is usually necessary to change the permissions on the device (ttyS0) to allow user-written programs to have access to it.
- *ioctl()* allows access to devices. Use *info ioctl* in Linux for more information.
- *outp()* and related functions exist but requires kernel level access (root privileges). This cannot be used for writing general purpose programs.

Example: See Serial programming Guide. in chapter 1, "Accessing the Serial Port." Also see chapter 2 for examples of using bit-identifier to set and clear option bits and chapter 4 for examples or *ioctl(),* general purpose IO control.

Interrupt handlers also negotiate with the kernel. The interrupt handler registers the IRQ it is using, disable interrupts while accessing critical sections and appropriately signal the kernel when it is used.

Windows Device Drivers: In Windows NT systems there are three types of device drivers. They are: Virtual Device drivers, GDI drivers and kernel mode drivers. Virtual device drivers allow 16 bit apps to think they are accessing hardware but all they are doing is passing the hardware request to a special kernel-mode driver. GDI drivers are graphics card specific. Kernel mode drivers are similar in concept to the Linux drivers described above. See the WinNT FAQ in the references below for details. The presentation by Chuck Berg (see below) contains good general advice for driver development.

Summary: In summary, interrupt handlers and device drivers for DOS , UNIX, WinXX and other OS's have many similarities. They all access the hardware and thus require knowledge of the system hardware configuration to write. They are all hardware and OS specific. Thus moving a program from one OS or hardware base to another always requires a new device driver. Run-time spent inside drivers or interrupt handlers is precious as you typically have the undivided attention of the CPU while you are busy. Spend as little time as possible there and off-load processing tasks to application level programs. Device drivers for multithreading/multitasking systems are more complex than those for single tasking systems and must negotiate with their respective OS's to protect against resource contention. C is the commonest language used for writing interrupt handlers and device drivers with assembly code or constructs mixed in.

Problems

1. What is "device independence."

2. What is the relationship between hardware development time and device driver development time? (See Chuck berg presentation)

3. The *outp()* function, used with root level access privileges, allows you to access the hardware very directly.. Describe two problems with using *outp()* for writing device drivers in multitasking systems.

References:

Kernel Mode systems:: NT Driver Resources:: FAQ for writing WinNT drivers. May 1999. http://www.cmkrnl.com/faq01.html

Chuck Berg, *PCI System Design Series Workshop (PCI Hardware/software (driver) design considerations)*, Powerpoint presentation 99/08/04. http://www.cmkrnl.com/files/SwForPciHwDes/index.htm

Liu, Jane W. S. *Real Time Systems* Prentice Hall, 2000.

Michael K. Johnson. *Writing Linux Device Drivers*1995 http://people.redhat.com/johnsonm/devices.html

Stallings, William. *Operating Principles, Internals and Design Principles*.4th Ed. Prentice Hall, 2001.

Stones, Richard and Matthew, Neil. *Beginning Linux Programming* 2nd Ed. Wrox Press, July 2000.

Tannenbaum, Andrew S. *Modern operating Systems* Prentice Hall, 1992