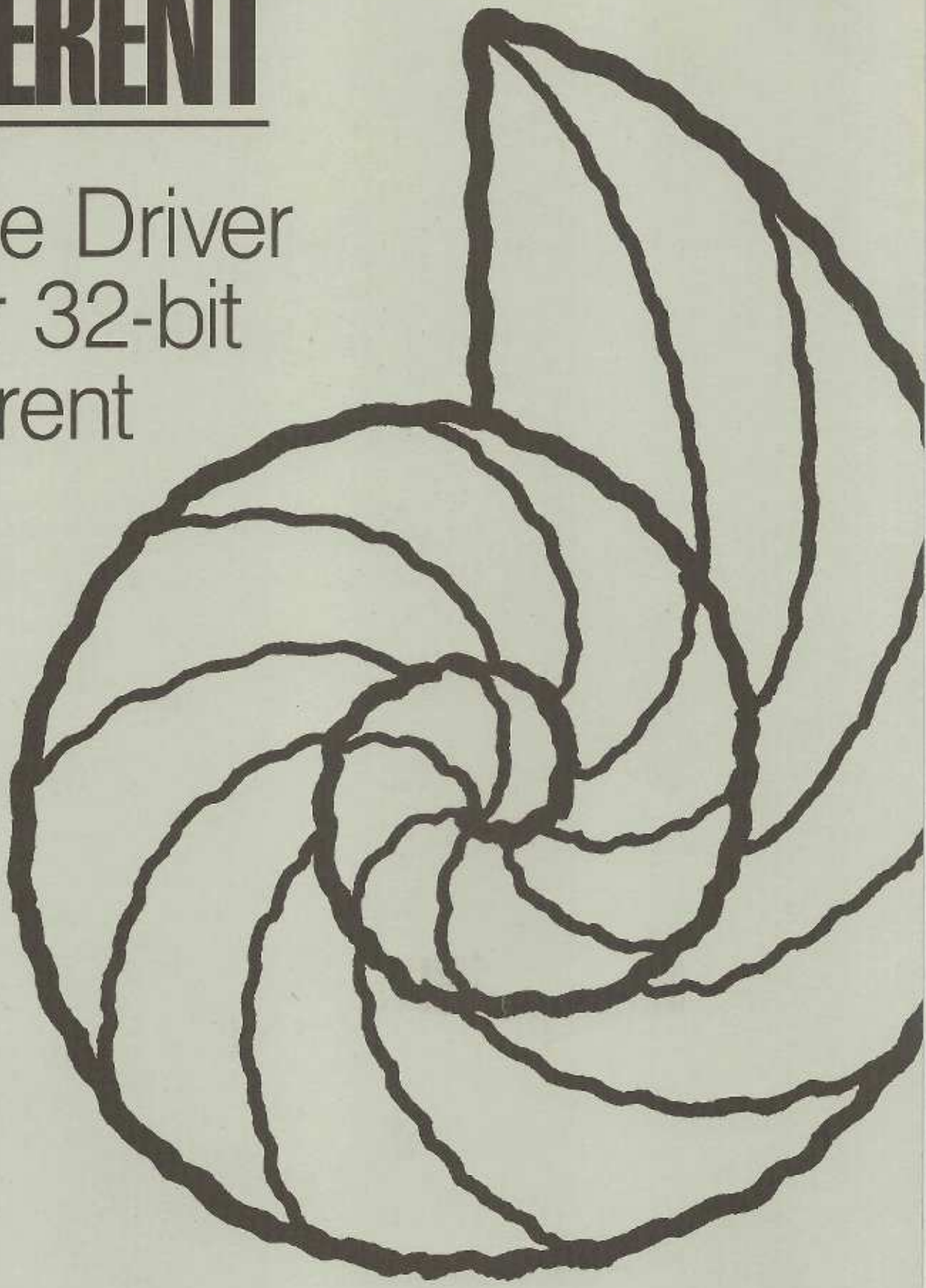
COHERENT™

Device Driver Kit for 32-bit Coherent



Mark Williams Company

COHERENT
Device Driver Kit

Release 4.2

Copyright © 1993

Mark Williams Company
60 Revere Drive
Northbrook, Illinois 60062
Telephone: (708) 291-6700

Mark Williams Company makes no warranty of any kind with respect to this material and disclaims any implied warranties of merchantability or fitness for any particular purpose.

The information contained herein is subject to change without notice.

Printed in U.S.A.

Copyright © 1982, 1993 by Mark Williams Company.
Portions copyright © 1988 by INETCO Systems, Ltd.

All rights reserved.

This publication conveys information that is the property of Mark Williams Company. It shall not be copied, reproduced or duplicated in whole or in part without the express written permission of Mark Williams Company. Mark Williams Company makes no warranty of any kind with respect to this material and disclaims any implied warranties of merchantability or fitness for any particular purpose.

COHERENT is a trademark of Mark Williams Company. UNIX is a trademark of Unix Systems Laboratories. All other products are trademarks or registered trademarks of the respective holders.

Revision 6 Printing 5 4 3 2 1

Published by Mark Williams Company, 60 Revere Drive, Northbrook, IL 60062.

Sales: Phone: (800) MARK-WMS
FAX: (708) 291-6750
E-mail: uunet!mwc!sales
sales@mwc.com
Technical Support:
Phone: (708) 291-6700
FAX: (708) 291-6750
E-mail: uunet!mwc!support
support@mwc.com
BIX: join mwc
CompuServ: 76256,427

Printed in the U.S.A.

Table of Contents

Introduction

This manual documents version 4.2 of the COHERENT Device-Driver Kit. This kit is designed to help you write device drivers for COHERENT release 4.2. It describes the contents of the kit, introduces the COHERENT kernel, gives advice on how to go about writing a device driver, gives examples of device drivers, and documents all of the kernel's accessible functions in Lexicon format.

Before you continue, please read the following carefully:

> *This kit will not teach you how to write a device driver. It is to be used only by persons who are technically knowledgeable. Due to the highly specialized nature of device drivers, this product is not eligible for technical support from Mark Williams Company.*

If you discover a bug in the product or you have a suggestion on how it can be improved, please contact Mark Williams Company. If you run into a difficulty with the hardware for which you are writing the driver, please consult that hardware's technical-reference manual or contact its manufacturer.

Further, a bug in a device driver can inflict great damage on an operating system and its files. You should expect that during development, you will damage the contents of your hard disk at least once. Therefore, we implore you to practice defensive programming in designing and testing your device driver, to protect irreplacable files from damage or destruction. This manual will give you suggestions on how to do this most easily.

Changes From Earlier Releases

Release 4.2 completely changes the COHERENT Device-Driver Kit:

- Tools for modifying and relinking the COHERENT kernel are now included within the COHERENT system itself.
- Release 4.2 introduces the COHERENT implementation of the UNIX Device Driver Interface/Driver-Kernel Interface (DDI/DKI), and STREAMS. We now discourage programmers from using internal kernel calls within their drivers unless absolutely necessary.

The Kit

The COHERENT Device Driver Kit consists of the following:

- Source code for most COHERENT device drivers.
- Configuration files for the device drivers.
- This manual.

Installing the Device Driver Kit

The COHERENT Device Driver Kit requires that you are currently running a production copy of COHERENT release 4.2.

To install the COHERENT Device Driver Kit, log in as the superuser **root**. Then type the command:

```
/etc/install -c Drv_420 /dev/fha0 1
```

for a 5.25-inch floppy-disk drive, or

```
/etc/install -c Drv_420 /dev/fva0 1
```

for a 3.5-inch floppy-disk drive.

The installation program will prompt you to insert the write-protected floppy disk into the device you named on the command line. After the installation completes, place your distribution disk in a safe place, away from heat or magnetic fields.

Using This Manual

This manual consists of six sections:

1. The introduction — the section you are reading now.
2. A sketch of the COHERENT kernel, and how it works.
3. Writing a device driver using internal kernel calls. This chapter describes methods that must be used to write block drivers.
4. Example of a block driver.
5. Example of a character driver.
6. The Lexicon. This gives an entry for each DDI/DKI, STREAMS, or internal-kernel function or macro. It also contains overview articles, which introduce classes of functions or macros, and articles that summarize technical information. Note that you can use the command **man** to view these articles; and you can view their calling conventions by invoking the command **help**.

As noted above, this manual will not teach you how to write a device driver. If you are seeking a tutorial, we suggest you look at one of the volumes listed below. We hope, however, that you will find the tutorials a helpful guide to the COHERENT kernel and resources, and the Lexicon a useful summary.

Bibliography

The following reference manuals discuss the writing of UNIX device drivers, the Intel 80386 microprocessor, STREAMS, and related topics.

Intel Corporation: *386 EX Programmer's Reference Manual.* Santa Clara, Ca.: Intel Corporation, 1990 (part 230985-002).

Campbell, J.: *C Programmers Guide to Serial Communication.* Indianapolis: Howard Sams & Company, 1989 (ISBN 0-67222-584-0).

Crawford, J.; Gelsinger, P.: *Programming the 80386.* SYBEX Incorporated, 1987 (ISBN 0-89588-381-3).

Plauger, P.: Evaluating device controllers. *Embedded Systems Programming,* March 1991, pp 87-92.

Comer, D.: *Operating System Design: The XINU Approach.* Englewood Cliffs, NJ: Prentice Hall, Incorporated, 1984 (ISBN 0-13-637539-1).

Egan, J.; Teixeira, T.: *Writing A UNIX Device Driver.* Englewood Cliffs, NJ: John Wiley and Sons, Incorporated, 1988 (ISBN 0-471-62859-X).

UNIX System Laboratories, Incorporated: *Device-Driver Interface/Driver-Kernel Interface Reference Manual for Intel Processors.* Englewood Cliffs, NJ: Prentice-Hall, Incorportaed, 1992.

AT&T: *UNIX System V STREAMS Primer.* Englewood Cliffs, NJ: Prentice-Hall, Incorporated, 1987.

AT&T: *UNIX System V STEAMS Programmer's Guide.* Englewood Cliffs, NJ: Prentice-Hall, Incorporated, 1987.

UNIX System Laboratories, Incorporated: *UNIX System V Release 4 Programmer's Guide: STREAMS.* Englewood Cliffs, NJ: Prentice-Hall, Incorporated, 1990.

Ritchie, D.: A stream input-output system, in *Unix Research System, Volume II: Papers* (ed. 10). Murray Hill, NJ: Computing Research Center, AT&T Bell Laboratories, 1990.

The COHERENT Kernel

This section sketches the COHERENT kernel. It also discusses defensive programming practices, how to rebuild the kernel to include a newly written driver, and how to interpret a dump from a failed driver.

The kernel is the program that permanently resides in memory to control the moment-to-moment operation of COHERENT. It works by controlling *processes* and *devices*. The following sub-sections introduce each.

Processes and Scheduling

A *process* is a program that is being executed. The kernel "slices" the processor's time amongst all processes, which creates the illusion that COHERENT is running many programs simultaneously.

As you can see, managing processes — that is, seeing that each process receives its share of the processor's time — is one of the kernel's main tasks. To do this, the kernel creates two queues whose entries describe every process that the kernel has been asked to execute. One, the *ready* queue, describes every process that is ready to be processed further by the microprocessor. The other, the *suspended* queue, describes every process that is waiting for something to happen; for example, a word-processing program that is awaiting a keystroke is placed on the suspended queue.

The kernel selects a process from the ready queue, then executes it either until it has reached a stopping point or until it has exhausted the slice of time allotted to it. If the process has exhausted its slice of time, the kernel moves it to the end of the ready queue. If, however, the process is awaiting an event, the kernel moves it to the suspended queue. A process on the suspended queue is said to be *sleeping*. In either case, the kernel saves the current state of the process, then selects the next process from the ready queue and executes it.

When an external event occurs (e.g., the user presses a key), the kernel searches the suspended queue for a process that is awaiting that event. If it finds one, the kernel moves it to the ready queue, where it waits its turn to be executed further. This continues until all processes have run to completion.

Before release 4.2.12 of COHERENT, the kernel selected processes from the ready queue sequentially — that is, it executed each process in turn, regardless of that process's demands upon the system. With this method of scheduling processes, running a computation-intensive program under COHERENT (e.g., a compiler) would degrade the system's response to user input from the keyboard to the point where the system was nearly useless.

Beginning with release 4.2.12 of COHERENT, the scheduler has been redesigned to maximize the number of processes executed. The scheduler now gives processes that make light demands on the system (e.g., a user's input from a keyboard) a higher priority than it gives processes that make heavy demands on the system (e.g., image processing or compiling). The new scheduler now permits a user to compile a program in the background or run a tape backup, yet continue to type input from the console or from a terminal with no noticeable degradation of response to the keyboard. The compile will take slightly longer than it would have otherwise, but the overall usefulness of the system is greatly increased.

In addition to managing processes, the kernel gives timeout functions, deferred functions, and device drivers sequential, uninterrupted access to the CPU. The kernel intersperses the slices of time it assigns to user processes amongst the demands of these lower-level routines. Any time the kernel is about to give the CPU to a user process, such as after completing a system call, it first checks if any timeouts or deferred functions are waiting to be called; if there are, it executes them first.

Devices

A *device* is a piece of hardware with which a process must communicate, e.g., physical memory, a hard-disk drive, a floppy-disk drive, or a serial port. Note that, unlike under MS-DOS, COHERENT does not permit a program to access hardware directly: the COHERENT kernel manages all transfers of data between a process and a device.

Devices come in two types: *character-special* and *block-special*. A *character-special device* is one with which COHERENT exchanges data one character at a time. This class of devices includes serial and parallel ports and the console. A *block-special device* is one with which COHERENT exchanges data one block at a time. COHERENT defines a *block* as being 512 bytes (one-half kilobyte). This class of devices includes the hard disk and the floppy disk. The size of a block is defined by constant **BSIZE** in header **<sys/buf.h>**.

Note that the COHERENT system, unlike most editions of UNIX, allows a device driver to be accessed in either block-special or character-special mode. For example, under COHERENT both hard disks and floppy disks can be accessed in either character or block mode. If a device can be accessed in either mode, its character mode is sometimes called the *raw* device (shown by the fact that its name begins or ends with the letter 'r'), whereas its block mode is sometimes called the *cooked* device. This will be detailed below.

The kernel uses the structure **IO** to manage communication with a device. It is defined in the header file **<sys/io.h>**, and includes the following fields:

io_seek Number of bytes from the beginning of the file or device where the driver should begin to read. This is, of course, is meaningless for devices for devices like serial ports. In the case of disk drives, this number must indicate the block to be read, i.e., the number must be evenly divisible by constant **BSIZE**, which gives the size of a COHERENT block. If this is not true, an error has occurred.

io_ioc Number of bytes to read or write. When the data movement is completed, this should be set to the number of bytes that remain to be read or written. If it is not reset to zero, an error has occurred.

io_base Offset of data to be transferred in the user memory space. This is converted to a physical or virtual memory address before performing the read.

io_flag Flags. See header file **<sys/io.h>** for the flags recognized by COHERENT. **IONDLY** indicates that the request is non-blocking.

Buffer Cache

A buffer cache is an area of RAM that holds data being written to or read from a block-special device. The kernel gives each block-special device its own buffer cache. The kernel, in turn, assigns to each buffer cache a **BUF** structure, which the kernel uses to manipulate that buffer cache. It is defined in the header file **<sys/buf.h>**, and includes the following fields:

b_dev A **dev_t** structure that describes the device being buffered. The internal kernel macros **major()** and **minor()** translate this structure into the device's major and minor numbers.

b_req Type of I/O request, either **BREAD** or **BWRITE**.

b_bno The number of the starting block.

b_paddr The system global (DMA) address for the data.

b_count The number of bytes to read or write.

b_resid The number of bytes remaining to be transferred. A value of zero indicates that all data transferred correctly, i.e., that an error did not occur.

Interrupts

Most devices gain the attention of the kernel by sending an *interrupt*, which is a hardware signal by which the device indicates that it needs attention.

The kernel assigns a unique pointer, or *interrupt vector*, to each device that uses interrupts. A device's interrupt vector points to a function, or *interrupt handler*, that is designed to service the device. The kernel stores its table of interrupt vectors at the beginning of main memory.

When COHERENT receives the interrupt from a device, it saves the state of the process that is being executed. It then reads the table of interrupt vectors and finds the vector for the interrupt it has just received; then jumps to the handler to which the interrupt vector points, and executes it. Executing the interrupt handler may require awakening some sleeping processes.

When the interrupt handler has finished its work, the kernel reloads the saved state of the interrupted process into the CPU, and resumes processing it as if nothing had happened.

Device Drivers

A *device driver* is the software that the kernel uses to communicate with a device. Each device must have its own driver.

A driver stands between the kernel and the physical hardware:

As the above diagram shows, the driver receives input from the hardware and writes output to it; and it receives input from the kernel and writes output to it.

Communication With an Application

A user-level application communicates with a driver via a special file called a *device file* or a *special file*. A driver can have any number of special files. Its suite of special files can access different devices of the same type (for example, different floppy-disk drives or different partitions on a hard disk); or can access the same device in different modes (for example, a tape device can be accessed in either rewind or non-rewind mode). A special file is created with the command **mknod**. Most device files are kept in directory **/dev**; if you execute the command **ls -l** on **/dev**, you will see a set of listings that appear something like the following:

```
  1             2     3       4       5       6      7               8           9
brw-------      1     sys     sys     11      0    Fri Apr 27       16:56       at0a
brw-------      2     sys     sys     11      1    Fri Apr 27       16:56       at0b
brw-------      1     sys     sys     11      2    Fri Apr 27       16:56       at0c
brw-------      2     sys     sys     11      3    Fri Apr 27       16:56       at0d
brw-------      1     sys     sys     11      4    Fri Apr 27       16:56       at1a
brw-------      1     sys     sys     11      5    Fri Apr 27       16:56       at1b
brw-------      1     sys     sys     11      6    Fri Apr 27       16:56       at1c
brw-------      1     sys     sys     11      7    Fri Apr 27       16:56       at1d
crw-rw-rw-      1     bin     bin      5      0    Fri Apr 27       16:56       com1r
crw-rw-rw-      3     bin     bin      6      8    Sat Aug 18       12:57       com2
crw-rw-rw-      3     bin     bin      6      8    Sat Aug 18       12:57       com2l
crw-rw-rw-      1     bin     bin      6      0    Fri Apr 27       16:56       com2r
```

The listing consists of nine fields, as follows:

1. Permissions. The first character in the permissions field indicates what type of device this is: **b** indicates a block-special (cooked) device, and **c** indicates a character-special (raw) device.

2 Number of links to the file.

3 The login identifier of the user who owns the file.

4 The group identifier of the user who owns the file.

5 Major-device number. This is a unique number that identifies the device driver to the kernel. The kernel can handle up to 32 devices at any given time, numbered zero through 31. The Lexicon article **device drivers** gives a table of all device drivers current recogized by the COHERENT system, and the major-device number of each.

6 Minor-device number. In addition to a type and a major-device number, each device file has a minor-device number. This allows COHERENT to distinguish among a number of devices of the same type. For example, this table shows that major number 11 indicates the AT hard disk. The above listing shows ten device files with this major-device number 11, five for device **at0** (which supports drive 0), and five for **at1** (which supports drive 1). Files ending in **a** through **d** each support one partition on the drive; the file ending in **x** supports that drive's partition table. Each of these device files has a unique minor device number, to allow the kernel to tell them apart.

7 Date last modified.

8 Time last modified.

9 Name of file.

Communication With the Kernel

The kernel communicates with the driver through *entry points* within the driver. If a user application invokes a system call for a given device (e.g., **open()** or **close()**, the kernel in turn invokes the appropriate entry point within the driver.

The method by which entry points are defined within a driver varies, depending upon whether the driver uses the DDI/DKI interface, or the internal-kernel interface. For details, see the entry for **entry points** within this manual's Lexicon.

Rebuilding the Kernel

The following walks you through the processing of adding a new driver. We will add the driver **foo**, which drives the popular "widget" device.

1. To begin, log in as the superuser **root**.
2. Create a directory to hold the driver's sources and object. Every driver must have its own directory under directory **/etc/conf**; and the sources must be held in directory **src** in that driver's directory. In this case, create directory **/etc/conf/foo**; then create directory **/etc/conf/foo/src**.
3. Copy the sources for the driver into its source directory; in this case, copy them into **/etc/conf/foo/src**.
4. Compile the driver. This should create one object file that has the suffix **.o**. Copy this file in the driver's home directory, and name it **Driver.o**. In this case, the object for the driver should be in file **/etc/conf/foo/Driver.o**.

   In some rare cases, a driver compile into more than one object. Use the command **ar** to store the objects into one archive called **Driver.a** and store the archive in the driver's home directory. The COHERENT commands that build the new kernel know how to handle archives correctly.

   You may wish to add a **Makefile** for your driver, so that it will be recompiled, as needed, whenever the command **idmkcoh** is run. For a sample **Makefile**, see the various **src** subdirectories under **/etc/conf**.
5. Add an entry to file **/etc/conf/mdevice** for the new driver. This file is a little more complex than **sdevice**; in particular, it distinguishes between STREAMS-style drivers and "old-style" COHERENT drivers. In most cases, you can simply copy an entry for an existing driver of the same type, and modify it slightly. In this case, the entry for **foo** should read as follows:

```
# full  func    misc    code    block   char    minor   minor   dmacpu
# name  flags   flags   prefix  major   major   min     max     chanid
foo     -       CGo     foo     15      15      0       255     -1-1
```

   In almost every case, the full name and the code prefix are identical. The code prefix also names the directory that holds the driver's object. Function flags are always always a hyphen, and miscellaneous flags almost always **CGo**. The block- and character-major numbers again are almost always identical. The major number is usually assigned by the creator of the device driver. In future releases of the kernel, these will be assigned dynamically by the kernel itself; poorly written drivers that depend upon the driver having a magic major-device number will no longer work. Finally, the last four columns for non-STREAMS drivers are almost always 0, 255, -1, and -1, respectively.
6. Add an entry to file **/etc/conf/sdevice** for this driver. **sdevice**, names the drivers to be included in the kernel. The entries for practically every entry are identical; you need to note only that the second column marks whether to include the driver in the kernel. In this case, the entry for the driver **foo** should read as follows:

```
foo     Y       0       0       0       0       0x0     0x0     0x00x0
```

   For details on what each column means, read the comments in file **/etc/conf/sdevice**.
7. If the driver has tunable variables, set them in file **Space.c**, which should be stored in the driver's home directory. As it happens, **foo** does not need a **Space.c** file. For examples of such files, look in the various sub-directories of **/etc/conf**. If the driver has tunable variables, you must a line for each to file **/etc/conf/mtune** and (optionally) to **/etc/conf/stune**.

   For example, suppose that the driver **foo** uses an array of **foo_info** structures, and that the number of these used is tunable. Then **Space.c** might contain these lines:

```
#include "conf.h"
#include <sys/foo.h> /* defines struct foo_info */
        ...
extern int num_foo_info = NUM_FOO_INFO;
extern struct foo_info foo_table[NUM_FOO_INFO];
```

/etc/conf/mtune sets the range of legal values and the default value for **NUM_FOO_INFO**. If you want to set **NUM_FOO_INFO** to some value other than the default, then pop an entry for that variable into **/etc/conf/stune**.

In **mtune** and **stune**, the name given is not that of the tunable variable, but of an enumeration constant used to initialize that variable. **idmkcoh** automatically generates the header that defines this constant as needed.

8. Type the command **/etc/conf/bin/idmkcoh** to build a new kernel. If necessary, move the new kernel into the root directory.

9. Save the old kernel and link the newly build kernel to **/autoboot**. You want save the old kernel, just in case the new one doesn't work. For directions on how to boot a kernel other than **/autoboot**, see entry for **booting** in the COHERENT Lexicon.

10. Back up your files! With a new driver in your kernel, it's best to play it safe.

11. Reboot your system to invoke the new kernel. If all goes well, you will now be enjoying the services of the new device driver.

Because of space restrictions in the boot procedure, the newly generated kernel object (e.g., **/testcoh**) does not contain a complete symbol table: values for some symbols instead are stored in the ASCII **.sym** file (e.g., **/testcoh.sym**). The debugger **db** can load symbols from a **.sym** file if you use its command-line option **-a**. To force a specific symbol to be included within the kernel object (notably, to make the symbol patchable within the generated kernel), add the symbol's name to a **-I** option that is output by the file **/etc/conf/install_conf/keeplist**.

Debugging Program Crashes

The following describes the COHERENT system's run-time environment. If your program dies and generates a register dump of the following form on the system console, this information will help you interpret what appears on the console.

When a program dies an untimely death, it dumps information about the contents of registers and memory onto the console device. The registers shown are the 386/486 family registers. All registers that begin with 'e' and that are three characters long are 32-bit registers. Refer to any 386/486 hardware/programming documents for further details on register specifics.

Register **eip** is the instruction pointer. This contains the user program's address at the time that the fault or violation occurred. Register **uesp** is the user-stack pointer. **cmd** specifies the command that faulted.

The following gives a sample dump of registers:

```
eax=32  ebx=408080      ecx=40541E      edx=0
esi=0   edi=8BF84589    ebp=7FFFF694    esp=FFFFFFE4
cs=B    ds=13           es=13           ss=13
err #14 eip=1C34F       uesp=7FFFF664   cmd=my_cmd
efl=13282       cr2=8BF84589    sig=11  trap_ip=FFC0079F
trapcode=4      User Trap
segmentation violation -- core dumped
```

Note that the format differs a little from that shown above. The dump also gives a traceback readout of addresses in the kernel; the addresses shown in the backtrace can be looked up file *kernel***.sym**, where *kernel* names the kernel that you were running when the dump occurred. This is discussed below in more detail.

The following gives the memory map for iBCS2 under COHERENT:

```
       0       -        3fffff       program text   (fixed size)
  400000       -       13fffff       program data   (grows upward)
7FC00000       -       7fffffff      program stack  (grows downward)
FFC00000       -       ffffffff      kernel
```

Examination of this dump uncovers the following clues about what killed your program:

- Error 14 indicates a page fault. The address that generated the fault is in register **cr2**.
- From the fact that **cr2** is an invalid address that matches **edi**, you can surmise that the program attempted to move a block or string to an illegal address.
- The base pointer **ebp** points into user stack, as expected.

- **esp**, the stack pointer, points into the kernel stack (the top four kilobytes), also as expected.
- The lower two bits for all the segment registers (**cs**-**ss**) are set, so the program was in user mode. If all had had their lower bits set, the dump probably would indicate a panic.

  Header file **<sys/reg.h>** defines the error numbers as follows:

SIDIV	0	Divide overflow
SISST	1	Single step
SINMI	2	NMI (parity)
SIBPT	3	Breakpoint
SIOVF	4	Overflow
SIBND	5	Bound
SIOP	6	Invalid opcode
SIXNP	7	Processor extension not available
SIDBL	8	Double exception
SIXS	9	Processor extension segment overrun
SITS	10	Invalid task state segment
SINP	11	Segment not present
SISS	12	Stack segment overrun/not present
SIGP	13	General protection
SIPF	14	Page fault
SISYS	32	System call
SIRAN	33	Random interrupt
SIOSYS	34	System call
SIDEV	64	Device interrupt

  Register **eip** points into user text space. To see what the kernel was doing at the time of the crash, you must check the **eip** address and any backtrace addresses against the known kernel text addresses. Most symbols for any kernel are in its associated **.sym** file. You should sort this before you look up addresses; then you can look up the address with **grep** or a text editor. Some symbols remain within the kernel executable. You can use the command **nm -n** to read these. The entry points within the **CON** structure usually are defined as static, but may be located using the command **/etc/conf/bin/drvldump**.
- Register **uesp** is a valid user-stack value. Note that it is below **ebp**. This is a normal consequence of iBCS2 calling conventions.
- **cmd** matches the program you are debugging, indicating that it is indeed that command and not something it calls.
- You can pick apart the flag register **efl** with your i386 reference manual.
- The signal number **sig** tells you what signal, if any, killed the process. Header file **signal.h** defines signal numbers as follows:

SIGHUP	1	Hangup
SIGINT	2	Interrupt
SIGQUIT	3	Quit
SIGILL	4	Illegal instruction
SIGTRAP	5	Trace trap
SIGIOT	6	IOT instruction
SIGEMT	7	Emulator trap
SIGFPE	8	floating point exception
SIGKILL	9	Kill
SIGBUS	10	Bus error
SIGSEGV	11	Segmentation violation
SIGSYS	12	Bad argument to system call
SIGPIPE	13	Write to pipe with no readers
SIGALRM	14	Alarm
SIGTERM	15	Software termination signal
SIGUSR1	16	User signal 1
SIGUSR2	17	User signal 2
SIGCLD	18	Death of a child
SIGCHLD	18	Death of a child
SIGPWR	19	Restart
SIGWINCH	20	Window change
SIGPOLL	22	Polled event in stream

- **trap_ip** is in kernel text space, as expected. This will not tell you much unless you are debugging the kernel.
- The trap code can safely be ignored.

Where To Go From Here

The next section describes in more detail the structure of a device driver. It emphasizes the internal-kernel interface, which is that used by practically all of the drivers shipped with COHERENT 4.2. The following two sections give examples of drivers for, respectively, a block device and character device.

See the entry **device driver** in this manual's Lexicon for a description of what device drivers are available with the device-driver kit, and of the articles available in the Lexicon.

Writing a Device Driver

This section discusses how to go about writing a device driver.

Types of Device Driver

Beginning with release 4.2, COHERENT offers two methods for interfacing a driver with the kernel: the *internal-kernel (IK) interface* and the *DDI/DKI interface*. Each consists of functions that the driver can call, predefined constants, types, etc. Under COHERENT 4.2, the structures that support the calling functions can be either from the IK interface or the DDI/DKI. This represents a transition from the IK method toward the DDI/DKI and STREAMS, which is the direction of future development.

The following describes the differences between these interfaces.

Internal Kernel Interface

This type of driver uses the routines internal to the COHERENT kernel. Practically all of the drivers included in the Driver Kit use this interface.

DDI/DKI Interface

The source code for drivers **loop** and **dump** use this interface. All new STREAMS drivers must use this interface. This is the preferred interface for new character drivers.

New block drivers must use the basic features of this interface; however, the DDI/DKI interface is preferred for new character drivers.

When you begin to write a driver for COHERENT, you should pick carefully between these strategies:

- The internal-kernel interface is proven and works; however, note that this does not conform to published UNIX definitions, and a driver written in this interface is less portable to another operating system.
- The DDI/DKI interface attempts compatibility with UNIX System V Release 4; however, the COHERENT implementation lacks some features present in true UNIX. We expect that the degree of compatibility between COHERENT's DDI/DKi interface and System V Release 4 will increase with ongoing development of COHERENT internals, including the memory manager and the file system.

STREAMS is a flexible and powerful method of programming character drivers. The COHERENT implementation of STREAMS was developed simultaneously with its implementation of the DDI/DKI, and therefore the two methods of writing drivers show a high degree of overlap. As shipped with COHERENT 4.2, STREAMS does not yet have line discipline or TLI modules available. All STREAMS development should be done with the DDI/DKI interface.

Sets of routines from the DDI/DKI can be combined with those from the internal-kernel interface. In some cases, the DDI/DKI offers the better method of performing a given task; in others, the internal-kernel interface offers the better (or, more likely, the only) method to perform a task. If you are importing a driver from UNIX System V Release 4, then you should use the DDI/DKI routines primarily. Likewise, you should use them primarily if you are writing a driver that you wish to export to UNIX. Note, too, that as COHERENT evolves toward the standard of System V Release 4, the DDI/DKI interface will grow in importance.

The sources included with release 4.2.05 of the device-driver kit are in the internal-kernel format rather than DDI/DKI. It was simply not practical to recast these drivers in the DDI/DKI mold at the present time; however, we are supplying information regarding DDI/DKI interfaces to inform developers of the future direction of COHERENT. In the development of new drivers, DDI/DKI facilities should be used wherever possible for greatest compatibility, e.g., with future releases of COHERENT.

To summarize, all else being equal, the DDI/DKI is preferred over the internal-kernel interface. The Lexicon entries themselves will alert you of the alternate ways of performing a given task, to help you decide which to use.

The best way to judge which interface you should use is to read the sources included with the COHERENT Device Driver Kit:

asy (**/etc/conf/asy/src**)

This driver manipulates non-intelligent serial ports. It is an example of a non-STREAMS character driver.

at (**/etc/conf/at/src**)
This driver manipulates the AT hard disk. It gives the best demonstration of writing a block driver, with regard to compatibility with UNIX System V Release 4.

hai154x (**/etc/conf/hai/src**)
This driver manipulates SCSI devices. It demonstrates how to use first-party DMA.

ss (**/etc/conf/ss/src**)
This driver manipulates the Seagate SCSI disk. It demonstrates how to use memory-mapped I/O.

fd (**/etc/conf/fd/src**)
This driver manipulates the floppy disks. It demonstrates how to perform DMA via the Intel controllers.

Beyond this, you must use your best judgement as you gain experience in working with COHERENT.

Planning the Device Driver

This section discusses how to plan a device driver. We strongly urge you to read this section carefully: it will help you avoid many of the pitfalls that plague developers of device drivers.

Defensive Programming

To begin, you must assume that you will damage your file systems *at least once* during development of your driver. To avoid damaging irreplacable files, we suggest that you do the following.

First, perform a full backup of your system before you begin to test and debug your driver. See the Lexicon entry for **backups** describes how to back up to floppy disk; the entry for **tape** describes how to back up to tape.

Second, you should create a COHERENT system that can be run from a floppy disk. One attractive feature of the COHERENT system is that a stripped down version is small enough to be run from a high-density floppy disk drive. You can then incorporate your device driver into the kernel that is run from your floppy-disk version of COHERENT; if something goes wrong, the files on your hard disk should be protected from damage. See the Lexicon entry for **booting** for directions on how to build a bootable floppy disk. Also, you can run the script **/etc/conf/bin/Floppy** (which is included with the Driver Kit) to build a bootable floppy disk.

Finally, as you plan your driver, you must bear in mind the fact that the driver stands between the kernel and the device. It receives input from and writes output to the kernel; it receives input from and writes output to the device. Because the driver is linked into the kernel, it in effect is part of the kernel. Therefore, when you write a driver you must think like a kernel programmer: you must *always* bear in mind the context in which a driver operates. Always ask yourself: *Is this function or structure available to me in this context? If it is not, how can I make it available?*

Testing the Hardware

Before you begin to write a driver, be sure to test the hardware. This will involve writing a program at the user level that lets you access the hardware via a device driver. When this is done, you should take the user manual and, as thoroughly as you have time and patience for, test *every* feature described in the manual and confirm that the hardware works as documented. Our experience in both writing and using technical documentation leads us to conclude that, try as one might, it is practically impossible to write an error-free manual.

You will save yourself much time and agony in the debugging phase if you test the hardware ahead of time. We also suggest that you alert the manufacturer to any errors you discover in the manual: this will earn you the gratitude of the manufacturer and of your fellow users.

Major Device Number

Once you have tested and confirmed that the hardware works as described (or noted all the places where the hardware's behavior varies from the documentation), you can begin to write your driver.

The first step is to select a major device number for the device you will be supporting. The entry for *device drivers* in this manual's Lexicon lists the major device numbers that are currently used under the COHERENT system. In addition, header file **<sys/devices.h>** contains symbolic constants for all assigned major numbers. Select one that is unused and assign it to your driver.

Naming Conventions

The next step is to devise some naming conventions for your driver. The conventions will govern both how you structure your driver, and how you name it to the COHERENT system. It is common practice to use the first two letters of the name of the configuration table to indicate the device. You can, however, use a prefix of up to eight characters; it is best to be brief but unambituous.

To create a device file for a file, append the minor device number to the device name. If a driver can support more than one device, they can be distinguished by an alphabetic suffix. For example, COHERENT's hard-disk driver is called **at**. The COHERENT system supports two drives, so there are two minor numbers, **at0** and **at1**. Finally, each drive can have four partitions, each of which is accessed via a different device file, plus one for the partition table. Thus, each drive has five device files: **at0a**, **at0b**, **at0c**, **at0d**, **at0x**, **at1a**, **at1b**, **at1c**, **at1d**, and **at1x**.

To avoid inadvertent name-space collisions, prefix with the name of the device the names of functions, variables, and arrays within your device driver, and the constants, types, etc., defined within associated header files.

Errors

To report an error to the calling process, call function **set_user_error()** with the appropriate error code. For a list of legal error codes, see the entry for the header file **<errno.h>** in the COHERENT manual's Lexicon.

Coding Requirements

The following summarizes the coding requirements for device drivers that use the internal-kernel or DDI/DKI interfaces.

To begin, the coding requirements for the internal-kernel interface:

1. Put 'C' in the miscellaneous flags in the file **/etc/conf/mdevice**.
2. Do not define symbol **_DDI_DKI** in the driver's source file.
3. Place driver's entry points in a **CON** structure. This structure is described below. The functions themselves may be declared as **static**.
4. There is no distinction between internal and external major- and minor-device numbers. A device number (**dev_t**) is a 16-bit object. Use internal-kernel routine **minor()**, *q.v.*, to obtain the minor-device number.
5. Either include **<sys/coherent.h>**, or explicitly define symbol **_KERNEL** to be one, before any other #include directives in the driver source.

The coding requirements for the DDI/DKI interface are as follows:

1. Do not put a 'C' into the miscellaneous-flags field in file **/etc/conf/mdevice** (*q.v.*).
2. Insert the directive

   ```
   #define _DDI_DKI 1
   ```

   in the driver's source file, before any **#include** directives.
3. Put an entry into the function-flags field in **/etc/conf/mdevice** for each of the driver's entry points; do not put them into a **CON** structure.
4. A device number (**dev_t**) is a 32-bit object. There is some discussion in the literature of internal *vs.* external numbering for device numbers and for the major and minor parts of the device number as well. As of COHERENT 4.2.05, only external numbers are of interest to the writer of device drivers. Thus, when a **dev_t** is passed to a driver's entry point, it is an external device number. When major numbers are entered into file **/etc/conf/mdevice**, they are external major numbers. Unit numbers and device features are decoded from the external minor number, which is obtained from the external device number by calling the DDI/DKI routine **geteminor()**.
5. Define symbol **_KERNEL** to be one in the driver source, before any **#include** directives.

The rest of this section emphasizes the internal-kernel interface to the COHERENT, as this interface is used by practically every driver that is now shipped with COHERENT 4.2. The bibliography given at the beginning of this manual lists several excellent books that describe the DDI/DKI and STREAMS, and give extended examples of how to code uses these interfaces. Note that these interfaces will become increasingly important as COHERENT evolves toward the standard set by UNIX System V Release 4.

The Internal-Kernel Interface

This section describes the internal-kernel interface between a device driver and the COHERENT kernel.

The COHERENT kernel contains numerous functions that perform the basic work of driving a device. Some are ordinary kernel system calls, which a driver can call just like any user-level application. Others are internal to the kernel and can be called only by drivers and other internal kernel processes. The following introduces some of the more frequently used calls and routines. Each is described in more detail either in this manual's Lexicon, or in the Lexicon that comes in the manual for the COHERENT system.

Interface to System Calls

Each driver that uses the internal-kernel interface contains a **CON** structure. This structure contains pointers to the functions that the kernel is to execute when an application invokes the system calls **open()**, **close()**, **read()**, **write()**, **ioctl()**, or **poll()**. **CON** includes the following fields:

c_flag This field's bits give the ways in which this device can be accessed, as follows:

DFBLK	Block-special device
DFCHR	Character-special device
DFTAP	Tape device
DFPOL	Accessible via COHERENT system call **poll()**

c_mind
This field gives the device's major-device number. This number is an index to the driver's place in the kernel's table of drivers. This number must be in the range of zero through 31, and must be a symbolic constant found in file **<sys/devices.h>**.

c_open This field points to the routine within the device driver that is executed whenever COHERENT opens the device. This function is always called with two arguments: the first is a **dev_t** that indicates the device being accessed, and the second is an integer that indicates the mode in which it is being opened. The mode can be **IPW** (write mode), **IPR** (read mode), or **IRW | IRP**. If an error occurs during execution of this function, it should set field **u_error** within the process's **UPROC** structure to an appropriate value.

c_close This field points to the routine that is executed whenever COHERENT closes the device. This function takes the same arguments as the **open** function.

c_block
This field points to the routine within the device driver that is executed when the kernel reads a file in block mode. This function is called with a pointer to the structure **BUF**, which contains the following fields:

b_dev	This field is of type **dev_t**, which is a structure that describes the device being buffered. The kernel macros **major()** and **minor()** translate this structure into the device's major and minor numbers.
b_req	Type of I/O request, either **BREAD** or **BWRITE**.
b_bno	This givesn number of the starting block.
b_paddr	This field gives the system-global (DMA) address for the data.
b_count	This field gives number of bytes to read or write.
b_resid	This field gives the number of bytes that remain to be transferred. A value of zero indicates that all data transferred correctly, i.e., that an error did not occur.

The driver function that performs block transfers of data should first perform the I/O transfer, then set field **b_resid** to the appropriate number and call kernel function **bdone()** to clean up after itself.

Note that the function that performs block transfer must *never* sleep or access a process's **uproc** structure. This is because this function is asynchronous and therefore not pegged to a particular process.

c_read This field points to the driver's routine that is called when the kernel wishes to read data from that driver's device. It takes two arguments: the first argument is a **dev_t** that indicates the device to read; the second points to the **IO** structure for that device. This structure contains the following fields:

io_seek This field gives the number of bytes from the beginning of the file or device whence reading should begin. This is, of course, is meaningless for devices for devices like serial ports. In the case of disk drives, this number must indicate the block to be read, i.e., the number must be evenly divisible by constant **BSIZE**, which gives the size of a COHERENT block. If

this is not true, an error has occurred.

io_ioc The number of bytes to read or write. When the read is completed, this should be set to the number of bytes that remain to be read or written. If it is not reset to zero, then an error has occurred.

io_base The offset of data to be transferred in the user memory space. This is converted to a physical or virtual memory address before performing the read.

io_flag Flags. See header file **<sys/io.h>** for the flags that COHERENT recognizes. **IONDLY** indicates that the request is non-blocking.

Unlike a block transfer, the read function does not return until I/O is complete. Your driver can use the kernel functions **x_sleep()** and **wakeup()** to surrender the processor to another process while the read is being performed. It can also use the kernel function **ioputc()** to send characters to the user process and to update counter **io_ioc**.

c_write This field points to the function that the kernel invokes when it wishes to write to this device. It behaves exactly the same as the function pointed to by field **c_read**, except that the direction of data transfer is reversed. Your driver can use kernel function **iogetc()** is used to fetch characters from the user process and to update counter **io_ioc**.

c_ioctl This field points to the function that the kernel executes when it wishes to exert I/O control over a device. This function is called to perform non-standard manipulations of a device, e.g., format a disk, rewind a tape, or change the speed of a serial port.

The kernel always calls this function with three arguments: the first argument is a **dev_t** that identifies the device to be manipulated; the second is an integer that indicates the command to be executed; the third points to a character array that can hold additional information, if any, that the command may need.

This function, by its nature, uses a considerable amount of device-specific information. The header files **<sys/tty.h>**, **<sys/mtioctl.h>**, and **<sys/lpioctl.h>** define codes for, respectively, teletypewriter devices (i.e., terminals), magnetic-tape devices, and line printers.

c_power
This field points to the routine to be executed should power fail on the system. This field is not yet used by COHERENT.

c_timer
This field points to the routine that the kernel executes when a device driver requests periodic scheduling. To request that the timeout routine for device *dev* be called once per second, set **drvl[major(***dev***).d_time** to a nonzero value. The external variable **drvl** is declared in header file **con.h**; macro **major()** is defined in header file **stat.h**.

The value in field **d_time** is not altered by the kernel's clock routines. To stop invocations of the timeout routine, store zero in **drvl[major(***dev***)].d_time**. *dev* is a **dev_t** that indicates which device is being timed out.

c_load This field points to the routine that is executed when this device driver is loaded. This performs all tasks necessary to prepare the device and the driver to exchange information. If the driver is linked into the kernel, then this routine is executed when COHERENT is booted.

Although COHERENT does not currently support loadable drivers, you should still write a load routine for driver startup and driver detachment from the kernel. This will spare you the labor of modifying your driver once loadable drivers are enabled.

c_uload
This field points to the driver's function that the kernel invokes when the driver is unloaded from memory. Although COHERENT does not support loadable drivers, you should still write an unload routine for driver startup and driver detachment from the kernel. This will spare you the labor of modifying your driver once loadable drivers are enabled.

c_poll This field points to a function that can be accessed by commands or functions that poll the device. The driver's polling function is always called with three arguments. The first argument is a **dev_t** that indicates the device to be polled. The second is an integer whose bits flag which polling tasks are to be performed, as follows:

POLLIN	Input data is available
POLLPRI	Priority message is available
POLLOUT	Output can be sent
POLLERR	A fatal error has occurred
POLLHUP	A hangup condition exists
POLLNVAL	**fd** does not access an open stream

These are defined in the header file **<sys/poll.h>**. The third argument is an integer that gives the number of millseconds by which the response should be delayed. Note that the COHERENT clock timer runs at 100 Hz rather than the approximately 18 Hz clock used by MS-DOS.

The kernel functions **pollopen()** and **pollwake()**, respectively, initiate and terminate a polling event. For more information on these functions, see their entries in this manual's Lexicon.

Timing

Sometimes a driver must delay for a period of time while doing something in the kernel. For example, you may want to wait within a device driver for no more than given period of time; if that period of time elapses, you assume that a timeout has occurred in the chain of desired events. If the delay is a given number of clock ticks, use kernel routine **timeout()** to call a given function after the specified number of ticks. Interrupts must be enabled for this to work; granularity is 0.01 sec (10 milliseconds) because that is the current length of a clock tick.

Kernel function **read_t0()** helps you to compute intervals of less than one tick. This function, which takes no arguments and returns an **int**, reads channel 0 (t0) of the programmable interval timer (**PIT**), which drives the system clock. A system clock tick is the time it takes timer t0 to decrease from 11,932 to zero. You can read the timer any time, whether interrupts are masked or not, and get a number between 11,932 and zero. Each unit therefore represents a little less than a microsecond. Overhead per call to **read_t0()** is about five to ten microseconds, depending on your CPU and clock. The kernel functions **busyWait()** and **busyWait2()** are useful for waiting briefly. They can return early from a wait if an awaited condition has been specified and is met. Their timing is independent of CPU speed.

Because CPU speeds among supported equipment vary by at least an order of magnitude, we *strongly* encourage all fine-timing loops to use **read_t0()** rather than simply counting down to zero using some empirically chosen loop count.

Sleeping and Waking

The driver will constantly call the kernel functions **x_sleep()** and **wakeup()** to synchronize your device driver with events in the operating system. **x_sleep()** moves the driver process to the suspended queue and sets a unique condition under which the process will awaken; **wakeup()** wakes up the process associated with that event.

For example, when a driver attempts to read a floppy disk, it may take several seconds for the floppy disk to begin to spin fast enough to be read. This may be a relatively brief period in real time, but the machine may be able to do much work during those few seconds. Thus, the floppy disk driver's **read** routine will begin to spin up the disk, then sleep until the floppy-disk drive signals that the disk is spinning fast enough to be read. The process will then awaken and begin to read; in the meantime, the COHERENT system will have been able to work productively. When you write you driver, you should look out for such situations and use **x_sleep()** and **wakeup()** to exploit them.

Note, however, that calling **x_sleep()** at the wrong time will trigger a "race condition", which under the wrong conditions could cause the device to hang. The entries for **x_sleep()** and **race condition** in this manual's Lexicon discuss when you should use the sleep mechanism, and when you should not. In brief, **x_sleep()** is available only when the active process can be associated with entry into the driver — that is, during the functions that are invoked via system calls **open()**, **close()**, **read()**, **write()**, **ioctl()**, and **poll()**. If a driver sleeps when it is not associated with a process, the kernel will never be able to awaken it. Thus, a driver must not sleep sleep during the routines invoked by the kernel routines **load()**, **unload()**, or **block()** or during the functions invoked by kernel routines **defer()** or **timeout()**.

Error Handling

When the kernel needs to tell a process that an error condition has occurred, it calls function **set_user_error()** with the appropriate error code.

Please note that like sleeping and some other situation, your driver can set the user error status only when user control is valid. A driver can sleep or call **set_user_error()** only from within driver functions invoked through the system calls **open()**, **close()**, **read()**, **write()**, **ioctl()**, and **poll()**, as described above. In particular, information

specific to the calling process is *not* available to the driver functions invoked via the kernel routines **load()**, **unload()**, **block()**, **defer()**, or **timeout()**.

Memory Management

This subsection describes how the kernel manages memory, and suggest how your driver should use memory-management to best advantage.

COHERENT divides all of the memory that is available to the kernel into several pools. The two most commonly used of these are the **kalloc()** pool and the **sysmem** pool.

The **kalloc()** pool acts as the kernel's heap, from which the kernel allocates and de-allocates memory for most small, temporary structures. Use the functions **kalloc()** and **kfree()** to allocate and free memory from this pool. These functions in the Lexicon to this manual.

The **sysmem** pool is a large collection of four-kilobyte "clicks" (memory pages). When new processes are created, the memory they need comes from this pool. Applications can use the functions **malloc()** and **free()** to obtain and free memory.

The kernel uses three modes of memory access: *physical memory*, *virtual memory*, and *system-global memory*.

Physical-memory addresses are the addresses set directly onto the address bus to access memory. Some peripheral equipment, such as video boards, communicate with the CPU through reads and writes within some specific range of physical-memory addresses. This arrangement is also called *memory-mapped I/O*. Devices that perform *direct-memory access* (DMA), including the usual AT-compatible floppy-disk controller, also use physical-memory addresses to obtain information on the source and destination of a DMA transfer. A physical memory address occupies four bytes, and is properly kept in an object of type **paddr_t**.

Virtual-memory addresses are those used by programs that run under COHERENT. Paging hardware allows the operating system to remap each four-kilobyte page of physical memory to any region of virtual memory that is on a four-kilobyte boundary. The kernel maintains tables that give the correspondences between pages of physical and virtual memory. When a context switch occurs (i.e., when one user process is shifted to the ready queue and another is selected from the ready queue for execution), the kernel updates these tables to unmap pages of memory for the old process, and to map pages of memory into virtual addresses for the new process. Virtual-memory addresses occupy four bytes, and are properly stored in type **vaddr_t**.

System global memory provides a third way of viewing memory resources. Consider a device driver that supports raw I/O to a block device. The term *raw* implies that the user process supplies an I/O buffer of arbitrary length. This introduces a problem: when the device driver is ready to transfer data between the user buffer and the I/O device, the user buffer may not be mapped into virtual memory. A *system global* address is a 32-bit object that retains access to the buffer, even when it is paged out of virtual memory.

Only a few operations are valid on system-global addresses:

- Add or subtract an offset.
- Apply macro **P2P()** to obtain the corresponding physical address.
- Appear as the second argument of kernel routine **dmain()** or **dmaout()**.
- Appear as the second argument (destination) in a call to kernel routine **xpcopy()**
- Appear as the first argument (source) in a call to kernel routine **pxcopy()**.

In the last two cases, the final argument in the function call must be manifest constant **SEG_386_KD|SEG_VIRT**. For example:

```
char src[BSIZE];
char dest[BSIZE];

/* copy from kernel data to system global buffer */
xpcopy(src, bp->b_paddr, BSIZE, SEG_386_KD|SEG_VIRT);

/* copy from system global buffer to kernel data */
pxcopy(bp->b_paddr, dest, BSIZE, SEG_386_KD|SEG_VIRT);
```

To get a system global address, use macro **MAPIO()**. This function takes two arguments, the segment table address (obtained from the segment-reference structure [SR]), and the offset within the segment.

For objects in the kernel kalloc pool, use **allocp()** as the SR. For objects in user space for the *current* process, the SR is **u.u_segl[***xxx***]**, where *xxx* is the segment index, e.g., **SIPDATA** or **SISTACK**. The user stack segment grows down, so the lower limit of the stack segment is computed differently than for the other user segments. See the example, below.

If your buffer has virtual address *v*, do the following. If *v* is in kalloc pool, use the following:

```
sgAddr = MAPIO(allocp.sr_segp->s_vmem, v - allocp.sr_base)
```

If *v* is in user data, then:

```
sgAddr = MAPIO(u.u_segl[SIPDATA].sr_segp->s_vmem,
       v - u.u_segl[SIPDATA].sr_base)
```

If *v* is in user stack:

```
sgAddr = MAPIO(u.u_segl[SISTACK].sr_segp->s_vmem,
       v - u.u_segl[SISTACK].sr_base - u.u_segl[SISTACK].sr_size)
```

For devices that support block I/O, a block routine takes a single argument, a pointer to a **BUF** structure. The field **b_paddr** of this **BUF** structure is a system global address.

Finally, please note that drivers that use the DDI/DDK interface will not use **kalloc()** or **kfree()**. They will use statically allocated structures sized by tunable parameters, or they will use the specialized resource-allocation routines that the DDK provides.

Where To Go From Here

The next two sections give, respectively, an example driver for a block device and an example driver for a character device. Each uses the internal-kernel interface. We suggest that you study these examples carefully before you attempt to write a driver on your own. Finally, the Lexicon in this manual describes the functions and macros introduced in this section.

Example of a Block Driver

This section gives an example driver for a block device: the COHERENT driver for the AT hard disk. This driver is described in the article **at** in the COHERENT Lexicon. The source is kept in directory **/etc/conf/at/src/at.c**.

In the following, code appears in monospaced font; comments appear in Roman.

Preliminaries

The following code prefaces the driver.

Header Files

at uses the following header files. Because header files have changed drastically for COHERENT 4.2, you should note carefully the suite included here:

```
#include <sys/cmn_err.h>
#include <sys/inline.h>
#include <sys/stat.h>
#include <sys/errno.h>
#include <stdlib.h>

#include <kernel/typed.h>
#include <sys/coherent.h>
#include <sys/uproc.h>
#include <sys/fdisk.h>
#include <sys/hdioctl.h>
#include <sys/buf.h>
#include <sys/con.h>
#include <sys/devices.h>
```

Manifest Constants

at uses the following gives manifest constants and macros:

```
#define LOCAL

#define HDBASE 0x01F0  /* Port base */
#define SOFTLIM 6      /*  (7) num of retries before diag */
#define HARDLIM 8      /* number of retries before fail */
#define BADLIM 100     /* num to stop recov if flagged bad */

#define BIT(n)  (1 << (n))

#define CMOSA 0x70     /* write cmos address to this port */
#define CMOSD 0x71     /* read cmos data through this port */

/*
 * I/O Port Addresses
 */
#define DATA_REG              (HDBASE + 0)          /* data (r/w) */
#define AUX_REG               (HDBASE + 1)          /* error(r), write precomp cyl/4 (w) */
#define NSEC_REG              (HDBASE + 2)          /* sector count (r/w) */
#define SEC_REG               (HDBASE + 3)          /* sector number (r/w) */
#define LCYL_REG              (HDBASE + 4)          /* low cylinder (r/w) */
#define HCYL_REG              (HDBASE + 5)          /* high cylinder (r/w) */
#define HDRV_REG              (HDBASE + 6)          /* drive/head (r/w) (D <<4)+(1 << H) */
#define CSR_REG               (HDBASE + 7)          /* status (r), command (w) */
#define HF_REG                (HDBASE + 0x206)      /* Usually 0x3F6 */
```

```
/*
 * Error from AUX_REG (r)
 */
#define DAM_ERR              BIT(0)              /* data address mark not found */
#define TR0_ERR              BIT(1)              /* track 000 not found */
#define ABT_ERR              BIT(2)              /* aborted command */
#define ID_ERR               BIT(4)              /* id not found */
#define ECC_ERR              BIT(6)              /* data ecc error */
#define BAD_ERR              BIT(7)              /* bad block detect */

/*
 * Status from CSR_REG (r)
 */
#define ERR_ST               BIT(0)              /* error occurred */
#define INDEX_ST             BIT(1)              /* index pulse */
#define SOFT_ST              BIT(2)              /* soft (corrected) ECC error */
#define DRQ_ST               BIT(3)              /* data request */
#define SKC_ST               BIT(4)              /* seek complete */
#define WFLT_ST              BIT(5)              /* improper drive operation */
#define RDY_ST               BIT(6)              /* drive is ready */
#define BSY_ST               BIT(7)              /* controller is busy */

/*
 * Commands to CSR_REG (w)
 */
#define RESTORE(rate)        (0x10 +(rate))      /* X */
#define SEEK(rate)           (0x70 +(rate))      /* X */
#define READ_CMD             (0x20)              /* X */
#define WRITE_CMD            (0x30)              /* X */
#define FORMAT_CMD           (0x50)              /* X */
#define VERIFY_CMD           (0x40)              /* X */
#define DIAGNOSE_CMD         (0x90)              /* X */
#define SETPARM_CMD          (0x91)              /* X */

/*
 * Device States.
 */
#define SIDLE                0                   /* controller idle */
#define SRETRY               1                   /* seeking */
#define SREAD                2                   /* reading */
#define SWRITE               3                   /* writing */
```

Function Declarations

The following declares the functions used in the driver.

```
/*
 * Forward Referenced Functions.
 */
LOCAL void                   atreset ();
LOCAL int                    atdequeue ();
LOCAL void                   atstart ();
LOCAL void                   atdefer ();
LOCAL int                    aterror ();
LOCAL void                   atrecov ();
LOCAL void                   atdone ();
```

Macros

at uses the following macros:

```
#define NOTBUSY()            ((inb (ATSREG) & BSY_ST) == 0)
#define DATAREQUESTED()      ((inb (ATSREG) & DRQ_ST) != 0)
#define ATDRQ()              (DATAREQUESTED () ? 1 : atdrq ())
#define ATBSYW(u)            (NOTBUSY () ? 1 : myatbsyw (u))
```

TUTORIALS

Global and Static Variables

at uses the following global and static variables:

```
extern typed_space     boot_gift;
extern short   at_drive_ct;
```

The following are used throughout the driver:

ATSECS
This is number of seconds to wait for an expected interrupt.

ATSREG
This must be 3F6 for most new IDE drives; or 1F7 for Perstor controllers and some old IDE drives. Either value works with most drives.

atparm This holds drive parameters. If initialized to zero, the driver will try to use values it read from the BIOS during real-mode startup.

```
extern unsigned ATSECS;
extern unsigned ATSREG;
extern struct hdparm_s atparm [];
```

The next line gives the global variable that holds the partition parameters, as copied from the disk. There are **N_ATDRV** * **NPARTN** positions for the user partitions, plus **N_ATDRV** additional partitions to span each drive.

When aligning partitions on cylinder boundaries, the optimal partition size is 14,280 blocks (2 * 3 * 4 * 5 * 7 * 17); whereas an acceptable partition size is 7,140 blocks (3 * 4 * 5 * 7 * 17).

```
static struct fdisk_s pparm [N_ATDRV * NPARTN + N_ATDRV];
```

The following structure **at** holds information about the disk controller. There exists one copy of this structure for each AT controller.

```
static struct at {
        BUF *at_actf;                           /* Link to first */
        BUF *at_actl;                           /* Link to last */
        paddr_t at_addr;                        /* Source/Dest virtual address */
        daddr_t at_bno;                         /* Block # on disk */
        unsigned at_nsec;                       /* # of sectors on current transfer */
        unsigned at_drv;
        unsigned at_head;
        unsigned at_cyl;
        unsigned at_sec;
        unsigned at_partn;
        unsigned char at_dtype [N_ATDRV];       /* drive type, 0 if unused */
        unsigned char at_tries;
        unsigned char at_state;
        unsigned at_totalsec;
} at;
```

Finally, this last variable holds the template of the message to be displayed when an AT drive times out.

```
static char timeout_msg [] = "at%d: TO\n";
```

Load Routine

atload() is the routine that the kernel executes when this driver is loaded. Under COHERENT 4.2, it is executed once, when the kernel is booted.

This function resets the controller, grabs the interrupt vector, and sets up the drive characteristics.

```
LOCAL void
atload ()
{
        unsigned int u;
        struct hdparm_s * dp;
        struct { unsigned short off, seg; } p;
```

```
        if (at_drive_ct <= 0)
                return;

        /* Flag drives 0, 1 as present or not. */
        at.at_dtype [0] = 1;
        at.at_dtype [1] = at_drive_ct > 1 ? 1 : 0;



        /* Obtain Drive Characteristics. */
        for (u = 0, dp = atparm; u < at_drive_ct; ++ dp, ++ u) {
                struct hdparm_s int_dp;
                unsigned short ncyl = _CHAR2_TO_USHORT (dp->ncyl);

                if (ncyl == 0) {
                        /*
                         * Not patched.
                         *
                         * If tertiary boot sent us parameters,
                         *   Use "fifo" routines to fetch them.
                         *   This only gives us ncyl, nhead, and nspt.
                         *   Make educated guesses for other parameters:
                         *   Set landc to ncyl, wpcc to -1.
                         *   Set ctrl to 0 or 8 depending on head count.
                         *
                         * Follow INT 0x41/46 to get drive static BIOS drive
                         * parameters, if any.
                         *
                         * If there were no parameters from tertiary boot,
                         * or if INT 0x4? nhead and nspt match tboot parms,
                         *   use "INT" parameters (will give better match on
                         *   wpcc, landc, and ctrl fields, which tboot can't
                         *   give us).
                         */

                        FIFO * ffp;
                        typed_space * tp;
                        int found, parm_int;

if (F_NULL != (ffp = fifo_open (& boot_gift, 0))) {
                        for (found = 0; ! found && (tp = fifo_read (ffp)); ) {
                                BIOS_DISK * bdp = (BIOS_DISK *)tp->ts_data;
                                if ((T_BIOS_DISK == tp->ts_type) &&
                                    (u == bdp->dp_drive) ) {
                                        found = 1;
                                        _NUM_TO_CHAR2(dp->ncyl,
                                          bdp->dp_cylinders);
                                        dp->nhead = bdp->dp_heads;
                                        dp->nspt = bdp->dp_sectors;
                                        _NUM_TO_CHAR2(dp->wpcc, 0xffff);
                                        _NUM_TO_CHAR2(dp->landc,
                                          bdp->dp_cylinders);
                                        if (dp->nhead > 8)
                                                dp->ctrl |= 8;
                                }
                        }
                        fifo_close (ffp);
}

                        if (u == 0)
                                parm_int = 0x41;
                        else /* (u == 1) */
                                parm_int = 0x46;
                        pxcopy ((paddr_t)(parm_int * 4), & p, sizeof p, SEL_386_KD);
                        pxcopy ((paddr_t)(p.seg <<4L)+ p.off,
                                & int_dp, sizeof (int_dp), SEL_386_KD);
```

```
                        if (! found || (dp->nhead == int_dp.nhead &&
                                        dp->nspt == int_dp.nspt)) {
                                * dp = int_dp;
                                printf ("Using INT 0x%x", parm_int);
                        } else
                                printf ("Using INT 0x13(08)");

                } else {
                        printf ("Using patched");
                        /*
                         * Avoid incomplete patching.
                         */
                        if (at.at_dtype [u] == 0)
                                at.at_dtype [u] = 1;
                        if (dp->nspt == 0)
                                dp->nspt = 17;
                }

#if VERBOSE > 0
        printf (" drive %d parameters\n", u);

        cmn_err (CE_CONT,
                 "at%d: ncyl=%d nhead=%d wpcc=%d eccl=%d ctrl=%d landc=%d "
                 "nspt=%d\n", u, _CHAR2_TO_USHORT (dp->ncyl), dp->nhead,
                 _CHAR2_TO_USHORT (dp->wpcc), dp->eccl, dp->ctrl,
                 _CHAR2_TO_USHORT (dp->landc), dp->nspt);
#endif
        }

        /* Initialize Drive Size. */
        for (u = 0, dp = atparm; u < at_drive_ct; ++ dp, ++ u) {
                if (at.at_dtype [u] == 0)
                        continue;


                pparm [N_ATDRV * NPARTN + u].p_size =
                        (long) _CHAR2_TO_USHORT (dp->ncyl) * dp->nhead *
                                dp->nspt;
        }

        /* Initialize Drive Controller.       */
        atreset ();
}
```

Unload Routine

Function **atunload()** is called when this driver is unloaded from memory — or would be, if COHERENT 4.2 supported loadable drivers.

```
LOCAL void
atunload ()
{
}
```

Reset the Controller

Function **atreset()** resets the hard-disk controller and defines drive characteristics.

```
LOCAL void
atreset ()
{
        int u;
        struct hdparm_s * dp;
```

```
        /* Reset controller for a minimum of 4.8 microseconds. */
        outb (HF_REG, 4);
        for (u = 100; -- u != 0;)
                /* DO NOTHING */ ;
        outb (HF_REG, atparm [0].ctrl & 0x0F);
        ATBSYW (0);

                /*
                 * Some IDE drives always timeout on initial reset.
                 * So don't report first timeout.
                 */
                static one_bad;

                if (one_bad)
                        printf ("at: hd controller reset timeout\n");
                else
                        one_bad = 1;
        }

        /* Initialize drive parameters. */
        for (u = 0, dp = atparm; u < at_drive_ct; ++ dp, ++ u) {
                if (at.at_dtype [u] == 0)
                        continue;
                ATBSYW (u);

                /* Set drive characteristics. */
                outb (HF_REG, dp->ctrl);
                outb (HDRV_REG, 0xA0 + (u << 4) + dp->nhead - 1);

                outb (AUX_REG, _CHAR2_TO_USHORT (dp->wpcc) / 4);
                outb (NSEC_REG, dp->nspt);
                outb (SEC_REG, 0x01);
                outb (LCYL_REG, dp->ncyl [0]);
                outb (HCYL_REG, dp->ncyl [1]);
                outb (CSR_REG, SETPARM_CMD);
                ATBSYW (u);

                /* Restore heads. */
                outb (CSR_REG, RESTORE (0));
                ATBSYW (u);
        }
}
```

Open Routine

Function **atopen()** is called when a user's application invokes the system call **open()** for an AT device. A pointer to this function appears in field **c_open** of the **CON** structure at the end of this driver.

This function validating the minor device (that is, ensures that the user is attempting to open a devices that exists), and updates the paritition table if necessary.

```
LOCAL void
atopen (dev, mode)
dev_t dev;
{
        int d;          /* drive */
        int p;          /* partition */

        p = minor (dev) % (N_ATDRV * NPARTN);
        if (minor (dev) & SDEV) {
                d = minor (dev) % N_ATDRV;
                p += N_ATDRV * NPARTN;
        } else
                d = minor (dev) / NPARTN;
```

```
        if (d >= N_ATDRV || at.at_dtype [d] == 0) {
                printf ("atopen: drive %d not present ", d);
                set_user_error (ENXIO);
                return;
        }
        if (minor (dev) & SDEV)
                return;

        /* If partition not defined read partition characteristics. */
        if (pparm [p].p_size == 0)
                fdisk (makedev (major (dev), SDEV + d), & pparm [d * NPARTN]);

        /* Ensure partition lies within drive boundaries and is non-zero size. */
        if (pparm [p].p_base + pparm [p].p_size >
            pparm [d + N_ATDRV * NPARTN].p_size) {
                printf ("atopen: p_size too big ");
                set_user_error (EINVAL);
        } else if (pparm [p].p_size == 0) {
                printf ("atopen: p_size zero ");
                set_user_error (ENODEV);
        }
}
```

Read Routine

Function **atread()** is called when a user's application invokes the system call **read()** for an AT device. A pointer to this function appears in field **c_read** of the **CON** structure at the end of this driver. This function simply invokes the common code for processing raw I/O.

```
LOCAL void
atread (dev, iop)
dev_t dev;
IO *iop;
{
        ioreq (NULL, iop, dev, BREAD, BFRAW | BFBLK | BFIOC);
}
```

Write Routine

Function **atwrite()** is called when a user's application invokes the system call **write()** for an AT device. A pointer to this function appears in field **c_write** of the **CON** structure at the end of this driver. This function simply invokes the common code for processing raw I/O.

```
LOCAL void
atwrite (dev, iop)
dev_t dev;
IO *iop;
{
        ioreq (NULL, iop, dev, BWRITE, BFRAW | BFBLK | BFIOC);
}
```

ioctl Routine

Function **atioctl()** is called when a user's application invokes the system call **ioctl()** for an AT device. A pointer to this function appears in field **c_ioctl** of the **CON** structure at the end of this driver. This function validates the minor device and updates the paritition table if necessary.

```
LOCAL void
atioctl (dev, cmd, vec)
dev_t dev;
int cmd;
char *vec;
{
        int d;
```

```
        /* Identify drive number. */
        if (minor (dev) & SDEV)
                d = minor (dev) % N_ATDRV;
        else
                d = minor (dev) / NPARTN;

        /* Identify input / output request. */
        switch (cmd) {
        case HDGETA:
                /* Get hard disk attributes. */
                kucopy (atparm + d, vec, sizeof (atparm [0]));
                break;

        case HDSETA:
                /* Set hard disk attributes. */
                ukcopy (vec, atparm + d, sizeof (atparm [0]));
                at.at_dtype [d] = 1;   /* set drive type nonzero */
                pparm [N_ATDRV * NPARTN + d].p_size =
                          (long) _CHAR2_TO_USHORT (atparm [d].ncyl) *
                          atparm [d].nhead * atparm [d].nspt;
                atreset ();
                break;

        default:
                set_user_error (EINVAL);
                break;
        }
}
```

Watch for Interrupts

Function **atwatch()** watches for interrupts. If **drvl[AT_MAJOR]** is greater than zero, this function decrements it. If it decrements to zero, it simulates a hardware interrupt.

```
LOCAL void
atwatch()
{
        BUF * bp = at.at_actf;
        int s;

        s = sphi ();
        if (-- drvl [AT_MAJOR].d_time > 0) {
                spl (s);
                return;
        }

        /* Reset hard disk controller, cancel request. */
        atreset ();
        if (at.at_tries ++ < SOFTLIM) {
                atstart ();
        } else {
                printf ("at%d%c: bno=%lu head=%u cyl=%u nsec=%u tsec=%d "
                        "dsec=%d <Watchdog Timeout>\n", at.at_drv,
                        (bp->b_dev & SDEV) ? 'x' : at.at_partn % NPARTN + 'a',
                        bp->b_bno, at.at_head, at.at_cyl, at.at_nsec,
                        at.at_totalsec, inb (NSEC_REG));

                at.at_actf->b_flag |= BFERR;
                atdone (at.at_actf);
        }
        spl (s);
}
```

Block Function

Function **atblock()** queues a block to the disk. It also ensures that the transfer is within the disk partition.

```
LOCAL void
atblock (bp)
BUF     * bp;
{
        struct fdisk_s * pp;
        int partn = minor (bp->b_dev) % (N_ATDRV * NPARTN);
        int s;

        bp->b_resid = bp->b_count;
        if (minor (bp->b_dev) & SDEV)
                partn += N_ATDRV * NPARTN;
        pp = pparm + partn;

        /* Check for read at end of partition. */
        if (bp->b_req == BREAD && bp->b_bno == pp->p_size) {
                bdone (bp);
                return;
        }

        /* Range check disk region. */
        if (bp->b_bno + (bp->b_count / BSIZE) > pp->p_size ||
            bp->b_count % BSIZE != 0 || bp->b_count == 0) {
                bp->b_flag |= BFERR;
                bdone (bp);
                return;
        }

        s = sphi ();
        bp->b_actf = NULL;
        if (at.at_actf == NULL)
                at.at_actf = bp;
        else
                at.at_actl->b_actf = bp;
        at.at_actl = bp;
        spl (s);

        if (at.at_state == SIDLE)
                if (atdequeue ())
                        atstart ();
}
```

Dequeue a Request

Function **atdequeue()** obtains the next request for disk I/O.

```
LOCAL int
atdequeue ()
{
        BUF * bp;
        struct fdisk_s * pp;
        struct hdparm_s * dp;
        unsigned int nspt;
        ldiv_t         addr;
        unsigned short secs;
        unsigned short newsec;

        at.at_tries = 0;
        if ((bp = at.at_actf) == NULL)
                return 0;
```

```
        at.at_partn = minor (bp->b_dev) % (N_ATDRV * NPARTN);
        if (minor (bp->b_dev) & SDEV) {
                at.at_partn += N_ATDRV * NPARTN;
                at.at_drv = minor (bp->b_dev) % N_ATDRV;
        } else
                at.at_drv = minor (bp->b_dev) / NPARTN;

        nspt = atparm [at.at_drv].nspt;
        at.at_addr = bp->b_paddr;
        pp = pparm + at.at_partn;
        at.at_bno = pp->p_base + bp->b_bno;

        dp = atparm + at.at_drv;
        addr = ldiv (at.at_bno, dp->nspt);
        at.at_sec = addr.rem + 1;
        addr = ldiv (addr.quot, dp->nhead);
        at.at_cyl = addr.quot;
        at.at_head = addr.rem;
```

The following code was added to the driver for COHERENT 4.2, to speed I/O on the AT disk. The following explains how this speed-up works.

It is unclear why, but IDE writes appear always to lose a revolution, even though reads work comfortably. This may be caused by IDE drives trying to maintain the synchronous semantics of the write, or it may be due to the COHERENT kernel's not making the read time and the slack being taken up by track-buffering.

In either case, COHERENT gains a vast improvement in throughput for writes and a modest gain for reads by looking ahead in the request chain and coalescing separate requests to consecutive blocks into a single multi-sector request.

```
        newsec = secs = bp->b_count / BSIZE;
        while (bp->b_actf != NULL && bp->b_actf->b_bno == bp->b_bno + secs &&
               bp->b_actf->b_req == bp->b_req &&
               bp->b_actf->b_dev == bp->b_dev) {
                /*
                 * Take care to bound the length of the combined request to a
                 * single byte count of sectors.
                 */
                bp = bp->b_actf;

                if (newsec + (secs = bp->b_count / BSIZE) > 256)
                        break;
                newsec += secs;
        }
        at.at_totalsec = at.at_nsec = newsec;
        return 1;
}
```

Send Data to the Disk

Function **atsend()** actually moves data onto the disk.

```
LOCAL void
atsend (addr)
paddr_t addr;
{
        addr = P2P (addr);
        repoutsw (DATA_REG, (unsigned short *) __PTOV (addr), BSIZE / 2);
}
```

Receive Data from the Disk

Function **atrecv()** actually receives data from the disk.

```
LOCAL void
atrecv (addr)
paddr_t addr;
{
        addr = P2P (addr);
        repinsw (DATA_REG, (unsigned short *) __PTOV (addr), BSIZE / 2);
}
```

Abandon a Request

Function **atabandon()** abandons a request for disk I/O.

```
LOCAL void
atabandon ()
{
        buf_t *bp;

        /* Abandon this operation. */
        while ((bp = at.at_actf) != NULL) {
                at.at_actf = bp->b_actf;
                bp->b_flag |= BFERR;
                bdone (bp);
        }
        at.at_state = SIDLE;
}
```

Start a Read/Write Operation

Function **atstart()** starts or restarts the next disk read/write operation.

```
LOCAL void
atstart ()
{
        struct hdparm_s * dp;

        /* Check for repeated access to most recently identified bad track. */
        ATBSYW (at.at_drv);
        dp = atparm + at.at_drv;
        outb (HF_REG, dp->ctrl);
        outb (HDRV_REG, (at.at_drv << 4) + at.at_head + 0xA0);

        outb (NSEC_REG, at.at_nsec);
        outb (SEC_REG, at.at_sec);
        outb (LCYL_REG, at.at_cyl);
        outb (HCYL_REG, at.at_cyl >> 8);

        if (inb (NSEC_REG) != (at.at_nsec & 0xFF)) {
                /*
                 * If we get here, things are confused. We should reset the
                 * controller and retry whatever operation we want to start
                 * now.
                 */
                drvl [AT_MAJOR].d_time = 1;
                return;
        }
```

```
        if (at.at_actf->b_req == BWRITE) {
                outb (CSR_REG, WRITE_CMD);
                while (ATDRQ () == 0) {
                        atabandon ();
                        return;
                }
                atsend (at.at_addr);
                at.at_state = SWRITE;
        } else {
                outb (CSR_REG, READ_CMD);
                at.at_state = SREAD;
        }

        drvl [AT_MAJOR].d_time = ATSECS;
}
```

Interrupt Handler

Function **atintr()** handles interrupts. It clears the interrupt, and defers its processing until a more suitable time.

```
void
atintr ()
{
        (void) inb (CSR_REG);  /* clears controller interrupt */
        atdefer ();
}
```

Defer Service of an Interrupt

Function **atdefer()** actually services the hard-disk interrupt. It transfers the required data, and updates the state of the device.

```
LOCAL void
atdefer ()
{
        BUF * bp = at.at_actf;
        switch (at.at_state) {
        case SRETRY:
                atstart ();
                break;

        case SREAD:
                /* Check for I/O error before waiting for data. */
                if (aterror ()) {
                        atrecov ();
                        break;
                }

                /* Wait for data, or forever. */
                if (ATDRQ () == 0) {
                        atabandon ();
                        break;
                }

                /* Read data block.*/
                atrecv (at.at_addr);

                /* Check for I/O error after reading data. */
                if (aterror ()) {
                        atrecov ();
                        break;
                }
```

```
                /*
                 * Every time we transfer a block, bump the timeout to prevent
                 * very large multisector transfers from timing out due to
                 * sheer considerations of volume.
                 */
                drvl [AT_MAJOR].d_time = ATSECS * 2;

                at.at_addr += BSIZE;
                bp->b_resid -= BSIZE;
                at.at_tries = 0;
                at.at_bno ++;

                /*
                 * Check for end of transfer (total, or simply part of a large
                 * combined request).
                 */
                if (-- at.at_nsec == 0)
                        atdone (bp);
                else if (bp->b_resid == 0) {
                        at.at_addr = (at.at_actf = bp->b_actf)->b_paddr;
                        bdone (bp);
                }
                break;

        case SWRITE:
                /* Check for I/O error. */
                if (aterror ()) {
                        atrecov ();
                        break;
                }

                /* bump timeout again, for reasons given above. */
                drvl [AT_MAJOR].d_time = ATSECS * 2;

                at.at_addr += BSIZE;
                bp->b_resid -= BSIZE;
                at.at_tries = 0;
                at.at_bno ++;

                /*
                 * Check for end of transfer, either the real end or the end
                 * of a block boundary in a combined transfer.
                 */
                if (-- at.at_nsec == 0) {
                        atdone (bp);
                        break;
                } else if (bp->b_resid == 0)
                        at.at_addr = bp->b_actf->b_paddr;

                /* Wait for ability to send data, or forever. */
                while (ATDRQ () == 0) {
                        atabandon ();
                        break;
                }

                /* Send data block. */
                atsend (at.at_addr);
                if (bp->b_resid == 0) {
                        at.at_actf = bp->b_actf;
                        bdone (bp);
                }
        }
}
```

Check for an Error

Function **aterror()** checks for drive error. If it finds an error, it increments the error count and prints a message that reports the error. It returns zero if it did not find an error, and one if it did.

```
LOCAL int
aterror ()
{
        BUF * bp = at.at_actf;
        int csr;
        int aux;

        if ((csr = inb (ATSREG)) & (ERR_ST | WFLT_ST)) {
                aux = inb (AUX_REG);

                if (aux & BAD_ERR) {
                        at.at_tries    = BADLIM;
                } else if (++ at.at_tries < SOFTLIM)
                        return 1;

                printf ("at%d%c: bno =%lu head =%u cyl =%u",
                        at.at_drv,
                        (bp->b_dev & SDEV) ? 'x' : at.at_partn % NPARTN + 'a',
                        (bp->b_count / BSIZE) + bp->b_bno - at.at_nsec,
                        at.at_head, at.at_cyl);

                if ((csr & (RDY_ST | WFLT_ST)) != RDY_ST)
                        printf (" csr =%x", csr);
                if (aux & (DAM_ERR | TR0_ERR | ID_ERR | ECC_ERR | ABT_ERR))
                        printf (" aux =%x", aux);

                if (aux & BAD_ERR)
                        printf (" <Block Flagged Bad>");
                if (at.at_tries < HARDLIM)
                        printf (" retrying...");
                printf ("\n");
                return 1;
        }
        return 0;
}
```

Attempt to Recover from an Error

Function **atrecov()** attempts to recover from a reported error.

```
LOCAL void
atrecov ()
{
        BUF * bp = at.at_actf;
        int cmd = SEEK (0);
        int cyl = at.at_cyl;

        switch (at.at_tries) {
        case 1:
        case 2:
                /* Move in 1 cylinder, then retry operation */
                if (--cyl < 0)
                        cyl += 2;
                break;

        case 3:
        case 4:
                /* Move out 1 cylinder, then retry operation */
                if (++ cyl >= _CHAR2_TO_USHORT (atparm [at.at_drv].ncyl))
                        cyl -= 2;
                break;
```

```
        case 5:
        case 6:
                /* Seek to cylinder 0, then retry operation */
                cyl = 0;
                break;

        default:
                /* Restore drive, then retry operation */
                cmd = RESTORE (0);
                cyl = 0;
                break;
        }

        /* Retry operation [after repositioning head] */
        if (at.at_tries < HARDLIM) {
                drvl [AT_MAJOR].d_time = cmd == RESTORE (0) ? ATSECS * 2 :
                                                              ATSECS;
                outb (LCYL_REG, cyl);
                outb (HCYL_REG, cyl >> 8);
                outb (HDRV_REG, (at.at_drv << 4) + 0xA0);
                outb (CSR_REG, cmd);
                at.at_state = SRETRY;
        } else {

                /* Give up on block. */
                bp->b_flag |= BFERR;
                atdone (bp);
        }
}
```

Release the Current I/O Buffer

Function **atdone()** releases the current I/O buffer.

```
LOCAL void
atdone (bp)
BUF * bp;
{
        at.at_actf = bp->b_actf;
        drvl [AT_MAJOR].d_time = 0;
        at.at_state = SIDLE;

        if (atdequeue ())
                atstart ();
        bdone (bp);
}
```

Indicate the Drive Is Not Busy

Function **notBusy()** indicates that the drive is not busy. See macro **NOTBUSY()**, defined above.

```
int
notBusy ()
{
        return NOTBUSY ();
}
```

Indicate Whether Data Have Been Requested

Function **dataRequested()** indicates whether data have been requested. See macro **DATAREQUESTED()**, defined above.

```
int
dataRequested ()
{
        return DATAREQUESTED ();
}
```

Report a Timeout, First Version

Function **_report_timeout()** actually prints the message that reports that an I/O operation has timed out.

```
static  int report_scheduled;
static  int report_drv;
LOCAL void
_report_timeout ()
{
        printf (timeout_msg, report_drv);
        report_scheduled = 0;
}
```

Report a Timeout, Second Version

Function **report_timeout()** manages the task of reporting that an I/O request has timed out.

```
LOCAL void
report_timeout (unit)
int unit;
{
        short s = sphi();
        if (report_scheduled == 0) {
                report_scheduled = 1;
                spl(s);

                report_drv = unit;
                defer (_report_timeout);
        } else
                spl (s);
}
```

Wait Until the Controller Is Freed

Function **myatbsyw()** waits while the controller is busy. It returns zero if the driver timed out while executing this I/O task; or a non-zero value if it did not.

```
int
myatbsyw (unit)
int unit;
{
        if (busyWait (notBusy, ATSECS * HZ))
                return 1;
        report_timeout (unit);
        return 0;
}
```

Wait for Controller to Initiate Request

Function **atdrq()** waits for the controller to initiate a request. It returns zero if the driver timed out while waiting; or one if it did not.

```
int
atdrq ()
{
        if (busyWait (dataRequested, ATSECS /* * HZ */))
                return 1;
        report_timeout (at.at_drv);
        return 0;
}
```

The CON Structure

Finally, the following gives the **CON** structure for this driver. This structure contains pointers to the functions to be invoked by the kernel's system calls. For details on this structure, see the entry for **CON** in this manual's Lexicon.

```
CON     atcon   = {
        DFBLK | DFCHR,              /* Flags */
        AT_MAJOR,                   /* Major index */
        atopen,                     /* Open */
        NULL,                       /* Close */
        atblock,                    /* Block */
        atread,                     /* Read */
        atwrite,                    /* Write */
        atioctl,                    /* Ioctl */
        NULL,                       /* Powerfail */
        atwatch,                    /* Timeout */
        atload,                     /* Load */
        atunload                    /* Unload */
};
```

Where To Go From Here

The following section gives an example of a driver for a character device. The kernel functions invoked in this driver are described in this manual's Lexicon.

Example of a Character Driver

This section gives an example driver for a character device: the COHERENT driver for the 8250-style asynchronous ports. It is described in the article **asy** in the COHERENT Lexicon.

In this driver, the minor-device number is a bit map that describes the features of the port, as follows:

0x80 1 for NO modem control, 'l' (lower-case "el")
0x40 1 for polled operation (no IRQ service), 'p'
0x20 1 for RTS/CTS flow control, 'f'
0x1F The channel number - 0 through 31

Character-device line discipline, which includes such operations as processing backspaces entered and echoing input characters, is performed in the **tty** module in COHERENT 4.2. Source code is provided in the 4.2 Device-Driver Kit. Future releases of COHERENT will use a STREAMS-based line discipline.

Preliminaries

The following prefaces the body of the driver.

Header Files

asy begins by including the following header files.

```
#include <sys/errno.h>
#include <sys/stat.h>
#include <termio.h>
#include <poll.h>

#include <sys/coherent.h>
#include <kernel/trace.h>
#include <sys/uproc.h>
#include <sys/proc.h>
#include <sys/tty.h>
#include <sys/con.h>
#include <sys/devices.h>
#include <sys/sched.h>
#include <sys/asy.h>
#include <sys/ins8250.h>
#include <sys/poll_clk.h>
```

Manifest Constants

The following gives manifest constants used throughout the driver.

```
#define IEN_USE_MSI    (IE_RxI | IE_TxI | IE_LSI | IE_MSI)
#define IEN_NO_MSI     (IE_RxI | IE_TxI | IE_LSI)

#define CTLQ   0021
#define CTLS   0023

#define NUM_IRQ              16            /* PC allows irq numbers 0..15 */
#define BPB                  8             /* 8 bits per byte */
#define DTRTMOUT             3             /* DTR seconds for close */
#define LOOP_LIMIT           100           /* safety valve on irq loops */
```

```
/*
 * For rawin silo (see poll_clk.h), use last element of si_buf to count
 * the number of characters in the silo.
 */
#define SILO_CHAR_COUNT              si_buf[SI_BUFSIZ-1]
#define SILO_HIGH_MARK               (SI_BUFSIZ-SI_BUFSIZ/4)
#define SILO_LOW_MARK                (SI_BUFSIZ/4)
#define MAX_SILO_INDEX               (SI_BUFSIZ-2)
#define MAX_SILO_CHARS               (SI_BUFSIZ-1)
```

Macros

asy uses the following macros:

```
#define RAWIN_FLUSH(in_silo) { \
  in_silo->si_ox = in_silo->si_ix; \
  in_silo->SILO_CHAR_COUNT = 0; }
#define RAWOUT_FLUSH(out_silo)       { out_silo->si_ox = out_silo->si_ix; }
#define channel(dev)                 (dev & 0x1F)
#define IEN                          ((a0->a_nms)?IEN_NO_MSI:IEN_USE_MSI)

#define NW_OUTSILO           1             /* bits in need_wake[] entries */

typedef void (* VPTR)();      /* pointer to function returning void */
typedef int (* FPTR)();       /* pointer to function returning int */
```

Local Functions

The following declares the functions used locally.

```
void asy_putchar();

/* Configuration functions (local) */
static void cinit();

/* Support functions (local) */
static void add_irq();
static void asy_irq();
static int asy_send();
static void asybreak();
static void asyclock();
static void asycycle();
static void asydump();
static int asyintr();
static void asyparam();
static void asysph();
static void asyspr();
static void asystart();
static void endbrk();
static void irqdummy();
static void upd_irq1();

static int p1(),p2(),p3(),p4();

extern int albaud[], alp_rate[];
```

Global Variables

asy uses the following global variables: When the command **asypatch** patches the **asy** driver for your system's configuration of ports, it checks whether its internal value for **ASY_VERSION** matches this driver's value. This prevents the patch utility and the driver from getting out of synch.

```
int ASY_VER = ASY_VERSION;
int ASY_HPCL = 1;
int ASY_NUM = 0;
int ASYGP_NUM = 0;
asy0_t asy0[MAX_ASY] = {
        { 0 }
};
asy_gp_t asy_gp[MAX_ASYGP] = {
        { 0 }
};
```

Static Variables

asy uses the following static variables.

```
static asy1_t * asy1;                       /* unused entries have type US_NONE */
static short dummy_port;                    /* used only during driver startup */
static int poll_divisor;                    /* set by asyspr(), read by asyclk() */
static char pptbl [MAX_ASY];                /* channel numbers of polled ports */
static int ppnum;                           /* number of channels in pptbl */
```

Variables **irq0[***x***]** and **irq1[***x***]** are lists for IRQ number *x*. **irq0[]** has nodes that may possibly cause an IRQ. **irq1[]** contains nodes for active devices. Whenever a device becomes active or inactive, **irq1** is rebuilt from **irq0**.

nodespace is an array of the nodes that are available. **nextnode** points to the next free node. Nodes are taken from node space only when the driver is loaded.

```
static FPTR ptbl [PT_MAX] = { asyintr,p1,p2,p3,p4 };
static struct irqnode *irq0[NUM_IRQ], *irq1[NUM_IRQ];
static struct irqnode nodespace[MAX_ASY];
static char need_wake[MAX_ASY];
static char nextnode;
static int initialized;       /* for asy_putchar() */
```

The Load Routine

The first function is **asyload()**. The kernel invokes this function when the driver is loaded into memory. Because COHERENT 4.2 does not support loadable drivers, this function is executed only when the kernel boots.

Field **c_load** in the **CON** structure contains a pointer to this function.

```
static void
asyload()
{
        int s, chan;
        asy0_t *a0;
        asy1_t *a1;
        TTY *tp;
        short port;
        char irq;
        char speed;
        char g;
        char sense_ct = 0;

        /* Allocate space for asy structs.  Possible error return. */
        asy1 = (asy1_t *)kalloc(ASY_NUM * sizeof(asy1_t));
        if (asy1 == 0) {
                printf("asyload: can't allocate space for %d async devices\n",
                  ASY_NUM);
                return;
        }
        kclear(asy1, ASY_NUM*sizeof(asy1_t));
```

```
/*
 * For each non-null port:
 *      sense chip type
 *      write baud rate to sgtty/termio structs
 *      disable port interrupts
 *      hang up port
 *      set default baud rate (also resets UART)
 *      hook "start" function into line discipline module
 *      hook "param" function into line discipline module
 *      hook CS into line discipline module
 *       if port uses irq
 *              if not in a port group
 *                      add to irq list
 */

for (chan = 0; chan < ASY_NUM; chan++) {
        a0 = asy0 + chan;
        a1 = asy1 + chan;
        tp = &a1->a_tty;
        speed = a0->a_speed;
        tp->t_sgttyb.sg_ispeed = tp->t_sgttyb.sg_ospeed = speed;
        tp->t_dispeed = tp->t_dospeed = speed;
        port = a0->a_port;

        /*
         * A port address of zero means a skipped entry in the table.
         * In this case a1->a_ut keeps its initial value of US_NONE.
         */
        if (port) {
                dummy_port = port;

                /*
                 * uart_sense() prints port info.
                 * Do this four times per line.
                 */
                a1->a_ut = uart_sense(port);
                sense_ct++;
                if ((sense_ct & 1) == 0)
                        putchar('\n');
                else
                        putchar('\t');

                s = sphi();
                outb(port+MCR, 0);
                outb(port+LCR, LC_DLAB);
                outb(port+DLL, albaud[speed]);
                outb(port+DLH, albaud[speed] >> 8);
                outb(port+LCR, LC_CS8);
                tp->t_start = asystart;
                /* leave tp->t_param at 0 */
                tp->t_ddp = (char *) chan;
                spl(s);

                if (a0->a_irqno && a0->a_asy_gp == NO_ASYGP)
                        add_irq (a0->a_irqno, asyintr, chan);
        }
}

if (sense_ct & 1)
        putchar('\n');

/* for each port group,       add group to irq list */
for (g = 0 ; g < ASYGP_NUM ; g ++)
        add_irq (asy_gp [g].irq, ptbl [asy_gp [g].gp_type], g);
```

```
        /* Attach irq routines. */
        for (irq = 0 ; irq < NUM_IRQ ; irq ++) {
                if (irq0 [irq])
                        setivec (irq, asy_irq);
        }
}
```

The Unload Routine

The kernel invokes function **asyunload()** when **asy** is unloaded from memory. As COHERENT 4.2 does not support loadable drivers, this function is never invoked.

Field **c_uload** in the **CON** structure contains a pointer to this function.

```
static void
asyunload()
{
        char chan, irq;

        /*
         *  for each channel:
         *      disable UART interrupts
         *      hangup port
         *      cancel timer
         */
        for (chan = 0; chan < ASY_NUM; chan++) {
                asy0_t * a0 = asy0 + chan;
                asy1_t * a1 = asy1 + chan;
                short port = a0->a_port;
                TTY *tp = &a1->a_tty;

                outb(port+IER, 0);
                outb(port+MCR, 0);
                timeout (tp->t_rawtim, 0, NULL, 0);
        }

        /* for each irq,  detach irq routine if one was attached */
        for (irq = 0 ; irq < NUM_IRQ ; irq ++)
                if (irq0 [irq])
                        clrivec(irq);

        /* deallocate dynamic storage */
        if (asy1)
                kfree (asy1);
}
```

The Open Routine

The kernel invokes function **asyopen()** when a user application invokes the system call **open()** to open a serial port.

Field **c_open** in the **CON** structure contains a pointer to this function.

```
static void
asyopen(dev, mode)
dev_t dev;
int mode;
{
        int s;
        char msr, mcr;
        char chan = channel(dev);
        asy0_t *a0 = asy0 + chan;
        asy1_t *a1 = asy1 + chan;
        TTY *tp = &a1->a_tty;
        short port = a0->a_port;
```

```
        /* chip not found */
        if (a1->a_ut == US_NONE) {
                set_user_error (ENXIO);
                goto bad_open;
        }

        if ((tp->t_flags & T_EXCL) != 0 && ! super()) {
                set_user_error (ENODEV);
                goto bad_open;
        }

        /* Can't open for hardware flow control if modem status
         * interrupts are disallowed.
         */
        if (a0->a_nms && (dev & CFLOW) != 0) {
                set_user_error (ENXIO);
                goto bad_open;
        }

        /* Can't open a polled port if another driver is using polling. */
        if (dev & CPOLL && poll_owner & ~ POLL_ASY) {
                set_user_error (EBUSY);
                goto bad_open;
        }

        /* Can't have both com[13] or both com[24] IRQ at once. */
        if (!(dev & CPOLL) && a0->a_ixc) {
                struct irqnode *np = irq1[a0->a_irqno];
                while (np) {
                        if (np->func != ptbl[0] || np->arg != chan) {
                                set_user_error (EBUSY);
                                goto bad_open;
                        }
                        np = np->next_actv;
                }
        }

        /* If port already in use, are new and old open modes compatible? */
        if (a1->a_in_use) {
                int oldmode = 0, newmode = 0; /* mctl:1 irq:2 flow:4 */

                if (a1->a_modc)
                        oldmode += 1;
                if (a1->a_irq)
                        oldmode += 2;
                if (a1->a_flc)
                        oldmode += 4;
                if ((dev & NMODC) == 0)
                        newmode += 1;
                if ((dev & CPOLL) == 0)
                        newmode += 2;
                if (dev & CFLOW)
                        newmode += 4;
                if (oldmode != newmode) {
                        set_user_error (EBUSY);
                        goto bad_open;
                }
        }
```

At this point, sleep if another process is opening or closing the port. This can happen if:

- Another process is trying a first open and awaiting CD.
- Another process is closing the port after losing CD.

- A remote process opened the port, spawned a daemon, and disconnected, yet the daemon ignored **SIGHUP** and is improperly keeping the port open.

Do not try to set **tp->t_flags** before this sleep! During the sleep, **ttclose()** may be called and clear the flags.

```
while (a1->a_in_use &&
       (a1->a_hcls || ((dev & NMODC) == 0 &&
              (inb (port + MSR) & MS_RLSD) == 0))) {

       if (x_sleep ((char *) & tp->t_open, pritty, slpriSigCatch,
                      "asyblk") == PROCESS_SIGNALLED) {
              set_user_error (EINTR);
              goto bad_open;
       }
}

/* If channel not in use, mark it as such. */
if (a1->a_in_use == 0) {
       /* Save modes for this open attempt to avoid future conflicts.
        * Then start asycycle() for this port.
        */
       if (dev & NMODC) {
              tp->t_flags &= ~T_MODC;
              a1->a_modc = 0;
       } else {
              tp->t_flags |= T_MODC;
              a1->a_modc = 1;
       }

       if (dev & CPOLL)
              a1->a_irq = 0;
       else
              a1->a_irq = 1;
       if (dev & CFLOW) {
              tp->t_flags |= T_CFLOW;
              a1->a_flc = 1;
       } else {
              tp->t_flags &= ~T_CFLOW;
              a1->a_flc = 0;
       }
}
a1->a_in_use++;

/* From here, error exit is bad_open_u. */
if (tp->t_open == 0) {        /* not already open */
       silo_t  * in_silo = &a1->a_in;

       if (!(dev & CPOLL)) {
              upd_irq1(a0->a_irqno);
              a1->a_has_irq = 1;
       }

       /* Need to start cycling to scan for CD. */
       asycycle(chan);

       s = sphi();
       /* Raise basic modem control lines even if modem
        * control hasn't been specified.
        * MC_OUT2 turns on NON-open-collector IRQ line from the UART.
        * since we can't have two UART's on same IRQ with MC_OUT2 on
        */
       mcr = MC_RTS | MC_DTR;
```

```
        if (dev & CPOLL) {
                outb(port+MCR, mcr);
        } else {
                outb(port+MCR, mcr | a0->a_outs);
                outb(port+IER, IEN);
        }

        if ((dev & NMODC) == 0) {     /* want modem control? */
                tp->t_flags |= T_HOPEN | T_STOP;
                for (;;) {     /* wait for carrier */
                        msr = inb(port+MSR);
                        /* If carrier detect present
                         *   if port not already open
                         *     break out of loop and finish first open
                         *   else
                         *     do second (or third, etc.) open
                         */
                        if (msr & MS_RLSD)
                                break;

                        /* wait for carrier */
                        if (x_sleep ((char *) & tp->t_open, pritty,
                                     slpriSigCatch, "need CD")
                            == PROCESS_SIGNALLED) {
                                outb(port + MCR, 0);
                                    outb(port + IER, 0);
                                set_user_error (EINTR);
                                tp->t_flags &= ~(T_HOPEN | T_STOP);
                                spl(s);
                                goto bad_open_u;
                        }
                }

                /* Mark that we are no longer hanging in open.
                 * Allow output over the port unless hardware flow
                 * control says not to.
                 */
                tp->t_flags &= ~T_HOPEN;
                tp->t_flags &= ~T_STOP;
                if (!(tp->t_flags & T_CFLOW) || (msr & MS_CTS))
                        a1->a_ohlt = 0;
                else
                        a1->a_ohlt = 1;

                /* Awaken any other opens on same device. */
                wakeup((char *)(&tp->t_open));
        }
        ttopen(tp);                             /* stty inits */
        tp->t_flags |= T_CARR;
        if (ASY_HPCL)
                tp->t_flags |= T_HPCL;

        asyparam(tp);  /* gimmick: do this while t_open is zero */

        /* TO DO: flush UART input register(s) */
        spl(s);

        /* Turn on polling for the port. */
        if (dev & CPOLL) {
                a1->a_poll = 1;
                asyspr();
        }
} /* end of first-open case */
```

```
        tp->t_open++;
        ttsetgrp(tp, dev, mode);
        return;

bad_open_u:
        a1->a_in_use--;
        wakeup((char *)(&tp->t_open));
bad_open:
        return;
}
```

The Close Routine

The kernel invokes function **asyclose()** when a user application invokes the system call **close()** to close a serial port.

Field **c_close** in the **CON** structure contains a pointer to this function.

```
static void
asyclose(dev, mode)
dev_t dev;
int mode;
{
        int chan = channel(dev);
        asy0_t *a0 = asy0 + chan;
        asy1_t *a1 = asy1 + chan;
        TTY *tp = &a1->a_tty;
        silo_t *out_silo = &a1->a_out;
        silo_t *in_silo = &a1->a_in;
        int flags, maj;
        int s;
        short port = a0->a_port;
        char lsr;

        if (--tp->t_open)
                goto not_last_close;
        s = sphi();

        a1->a_hcls = 1; /* disallow reopen till done closing */
        flags = tp->t_flags; /* save flags - ttclose() zeroes them */
        ttclose(tp);

        /* Wait for output silo and UART xmit buffer to empty.
         * Allow signal to break the sleep.
         */
        for (;;) {
                int chipEmpty = 0, siloEmpty = 0;

                lsr = inb(port + LSR);
                chipEmpty = (lsr & LS_TxIDLE);
                siloEmpty = (out_silo->si_ix == out_silo->si_ox);

                if (chipEmpty && siloEmpty)
                        break;
                need_wake[chan] |= NW_OUTSILO;
                if (x_sleep ((char *) out_silo, pritty, slpriSigCatch,
                             "asyclose") == PROCESS_SIGNALLED) {
                        RAWOUT_FLUSH(out_silo);
                        break;
                }
        }
        need_wake[chan] &= ~NW_OUTSILO;
```

```
        /* If not hanging in open */
        if ((flags & T_HOPEN) == 0) {
                /* Disable interrupts. */
                outb(port+IER, 0);
                outb(port+MCR, inb(port+MCR) & ~MC_OUTS);
        }

        /* If hupcls */
        if (flags & T_HPCL) {
                /* Hangup port - drop DTR and RTS. */
                outb(port+MCR, inb(port+MCR) & MC_OUTS);

                /* Hold dtr low for timeout */
                maj = major(dev);
                drvl[maj].d_time = 1;

                x_sleep ((char *) & drvl [maj].d_time, pritty, slpriNoSig,
                         "drop DTR");
                drvl[maj].d_time = 0;
        }

        a1->a_poll = 0;
        asyspr();
        RAWIN_FLUSH(in_silo);
        a1->a_hcls = 0; /* allow reopen - done closing */
        wakeup((char *)(&tp->t_open));
        spl(s);
        a1->a_in_use--;

        if (!(dev & CPOLL))
                upd_irq1(a0->a_irqno);
        return;

not_last_close:
        a1->a_in_use--;
        wakeup((char *)(&tp->t_open));
        return;
}
```

The Read Routine

The kernel invokes function **asyread()** when a user application invokes the system call **read()** to read data from a serial port.

Field **c_read** in the **CON** structure contains a pointer to this function.

```
static void
asyread(dev, iop)
dev_t dev;
register IO * iop;
{
        int chan = channel(dev);
        asy1_t *a1 = asy1 + chan;
        TTY *tp = &a1->a_tty;
        ttread(tp, iop);
}
```

The Timeout Routine

The kernel invokes function **asytimer()** when a timeout occurs.

Field **c_timer** in the **CON** structure contains a pointer to this function.

```
static void
asytimer(dev)
dev_t dev;
{
        if (++drvl[major(dev)].d_time > DTRTMOUT)
                wakeup((char *)&drvl[major(dev)].d_time);
}
```

The Write Routine

The kernel invokes function **asywrite()** when a user application invokes the system call **write()** to write data to this port.

Field **c_write** in the **CON** structure contains a pointer to this function.

```
static void
asywrite(dev, iop)
dev_t dev;
register IO * iop;
{
        int chan = channel(dev);
        asy0_t *a0 = asy0 + chan;
        asy1_t *a1 = asy1 + chan;
        TTY *tp = &a1->a_tty;
        short port = a0->a_port;
        register int c;

        /* Treat user writes through tty driver. */
        if (iop->io_seg != IOSYS) {
                ttwrite(tp, iop);
                return;
        }

        /* Treat kernel writes by blocking on transmit buffer. */
        while ((c = iogetc(iop)) >= 0) {
                /* Wait until transmit buffer is empty.
                 * Check twice to prevent critical race with interrupt handler.
                 */
                for (;;) {
                        if (inb(port+LSR) & LS_TxRDY)
                                if (inb(port+LSR) & LS_TxRDY)
                                        break;
                }

                /* Output the next character. */
                outb(port+DREG, c);
        }
}
```

The ioctl Routine

The kernel invokes function **asyioctl()** when a user application invokes the system call **ioctl()** to manipulate a serial device.

Field **c_open** in the **CON** structure contains a pointer to this function.

```
static void
asyioctl(dev, com, vec)
dev_t dev;
int com;
struct sgttyb *vec;
{
        int chan = channel(dev);
        asy0_t *a0 = asy0 + chan;
        asy1_t *a1 = asy1 + chan;
        TTY *tp = &a1->a_tty;
        int s;
        int temp;
        silo_t *out_silo = &a1->a_out;
        silo_t *in_silo = &a1->a_in;
        short port = a0->a_port;
        unsigned char msr, mcr, lcr, ier;
        char do_ttioctl = 0;
        char do_asyparam = 0;

        s = sphi();
        ier = inb(port+IER);
        mcr = inb(port+MCR);   /* get current MCR register status */
        lcr = inb(port+LCR);   /* get current LCR register status */

        /* If command will drain output, do the drain now
         * before calling ttioctl().
         * Don't do this for 286 kernel:  we're running out of code space.
         */
        switch(com) {

        case TCSETAW:
        case TCSETAF:
        case TCSBRK:
        case TIOCSETP:
                /* Wait for output silo and UART xmit buffer to empty.
                 * Allow signal to break the sleep.
                 */
                for (;;) {
                        if (! ttoutp (tp) &&
                            out_silo->si_ix == out_silo->si_ox &&
                            (inb (port + LSR) & LS_TxIDLE) != 0)
                                break;
                        need_wake[chan] |= NW_OUTSILO;

                        if (x_sleep ((char *) out_silo, pritty, slpriSigCatch,
                                     "asydrain") == PROCESS_SIGNALLED)
                                break;
                }
                need_wake [chan] &= ~NW_OUTSILO;
        }

        switch(com) {
        case TIOCSBRK: /* set BREAK */
                outb (port + LCR, lcr | LC_SBRK);
                break;

        case TIOCCBRK: /* clear BREAK */
                outb (port + LCR, lcr & ~ LC_SBRK);
                break;

        case TIOCSDTR: /* set DTR */
                outb (port + MCR, mcr | MC_DTR);
                break;
```

```
case TIOCCDTR: /* clear DTR */
        outb (port + MCR, mcr & ~ MC_DTR);
        break;

case TIOCSRTS: /* set RTS */
        outb (port + MCR, mcr | MC_RTS);
        break;

case TIOCCRTS: /* clear RTS */
        outb (port + MCR, mcr & ~ MC_RTS);
        break;

case TIOCRSPEED:       /* set "raw" I/O speed divisor */
        outb (port + LCR, lcr | LC_DLAB); /* set speed latch bit */
        outb (port + DLL, (unsigned) vec);
        outb (port + DLH, (unsigned) vec >> 8);
        outb (port + LCR, lcr); /* reset latch bit */
        break;

case TIOCWORDL:               /* set word length and stop bits */
        outb (port + LCR, ((lcr & ~ 0x7) | ((unsigned) vec & 0x7)));
        break;

case TIOCRMSR:         /* get CTS/DSR/RI/RLSD (MSR) */
        msr = inb (port + MSR);
        temp = msr >> 4;
        kucopy (& temp, (unsigned *) vec, sizeof (unsigned));
        break;

case TIOCFLUSH:               /* Flush silos here, queues in tty.c */
        RAWIN_FLUSH (in_silo);
        RAWOUT_FLUSH (out_silo);
        do_ttioctl = 1;
        break;

        /* If port parameters change, plan to call asyparam().
         * Need to check now before structs are updated.
         */
case TCSETA:
case TCSETAW:
case TCSETAF:
        {
                struct  termio  trm;

                ukcopy (vec, & trm, sizeof (struct termio));
                if (trm.c_cflag != tp->t_termio.c_cflag)
                        do_asyparam = 1;
        }
        do_ttioctl = 1;
        break;

case TIOCSETP:
case TIOCSETN:
        {
                struct  sgttyb  sg;

                ukcopy(vec, &sg, sizeof(struct sgttyb));
                if (sg.sg_ispeed != tp->t_sgttyb.sg_ispeed ||
                   ((sg.sg_flags ^ tp->t_sgttyb.sg_flags) & ANYP) != 0)
                        do_asyparam = 1;
        }
        do_ttioctl = 1;
        break;
```

```
        default:
                do_ttioctl = 1;
        }

        outb (port + IER, ier);
        if (do_ttioctl)
                ttioctl (tp, com, vec);
        spl (s);
        if (do_asyparam)
                asyparam (tp);

        /* Things to be done after calling ttioctl(). */
        switch(com) {
        case TCSBRK:
                /* Send 0.25 second break:
                 * 1.  Turn on break level.
                 * 2.  Set timer to turn off break level 0.25 sec later.
                 * 3.  Sleep till timer expires.
                 * 4.  Turn off break level.
                 */
                outb (port + LCR, lcr | LC_SBRK);
                a1->a_brk = 1;
                timeout (& tp->t_sbrk, HZ / 4, endbrk, chan);

                while (a1->a_brk)
                        x_sleep (a1, pritty, slpriNoSig, "asybreak");

                outb (port + LCR, lcr & ~ LC_SBRK);
        }
}
```

Turn Off the Break Level

Function **endbrk()** turns off the break level. Called from a timeout after the function **ioctl(fd, TCSBRK, 0)**.

```
void
endbrk(chan)
int chan;
{
        asy1_t  *a1 = asy1 + chan;
        a1->a_brk = 0;
        wakeup (a1);
}
```

Read Parameters

Function **asyparam()** reads parameters from the port.

```
static void
asyparam(tp)
TTY * tp;
{
        int chan = (int)tp->t_ddp;
        asy0_t *a0 = asy0 + chan;
        asy1_t *a1 = asy1 + chan;
        short port = a0->a_port;
        int s;
        int write_baud=1, write_lcr=1;
        unsigned char mcr, newlcr, speed, oldSpeed;
        unsigned short cflag = tp->t_termio.c_cflag;
```

```
speed = cflag & CBAUD;
switch (cflag & CSIZE) {
case CS5:  newlcr = LC_CS5;  break;
case CS6:  newlcr = LC_CS6;  break;
case CS7:  newlcr = LC_CS7;  break;
case CS8:  newlcr = LC_CS8;  break;
}

if (cflag & CSTOPB)
        newlcr |= LC_STOPB;
if (cflag & PARENB) {
        newlcr |= LC_PARENB;
        if ((cflag & PARODD) == 0)
                newlcr |= LC_PAREVEN;
}

/* Don't bang on the UART needlessly.
 * Writing baud rate resets the port, which loses characters.
 * You want this on first open, NOT on later opens.
 */
oldSpeed = a0->a_speed;

if (speed == oldSpeed && tp->t_open) {
        write_baud = 0;
        if (newlcr == a1->a_lcr) {
                write_lcr = 0;
        }
}
a0->a_speed = speed;
a1->a_lcr = newlcr;

if (write_lcr) {
        char ier_save;
        s = sphi();
        ier_save = inb(port+IER);

        if (write_baud) {
                if (speed) {
                        short divisor = albaud [speed];

                        if (oldSpeed == 0) {
                                /* if previous baud rate was zero,
                                 * need to go off hook. */
                                mcr = inb(port+MCR) | (MC_RTS | MC_DTR);
                                outb(port+MCR, mcr);
                        }

                        outb(port+LCR, LC_DLAB);
                        outb(port+DLL, divisor);
                        outb(port+DLH, divisor >> 8);
                } else {
                        /* Baud rate of zero means hang up. */
                        mcr = inb(port+MCR) & ~(MC_RTS | MC_DTR);
                        outb(port+MCR, mcr);
                }
        }

        outb(port+LCR, newlcr);
        if (a1->a_ut == US_16550A)
                outb(port+FCR, FC_ENABLE | FC_Rx_RST | FC_Rx_08);
        outb(port+IER, ier_save);
        spl(s);
}
```

```
        if (write_baud)
                asyspr ();
}
```

Start Processing

Function **asystart()** starts processing of data.

```
static void
asystart(tp)
TTY * tp;
{
        int chan = (int)tp->t_ddp;
        asy0_t *a0 = asy0 + chan;
        asy1_t *a1 = asy1 + chan;
        short port = a0->a_port;
        int s;
        int need_xmit = 1; /* True if should start sending data now. */
        silo_t *out_silo = &a1->a_out;
        char lsr;

        /* Read line status register AFTER disabling interrupts. */
        s = sphi ();
        lsr = inb (port + LSR);

        /* Process break indication.
         * NOTE: Break indication cleared when line status register was read.
         */
        if (lsr & LS_BREAK)
                defer (asybreak, chan);

        /* If no output data, it may be time to finish closing the port;
         * but won't need another xmit interrupt.
         */
        if (out_silo->si_ix == out_silo->si_ox) {
                if (need_wake[chan] & NW_OUTSILO) {
                        need_wake[chan] &= ~NW_OUTSILO;
                        wakeup((char *)out_silo);
                }
                need_xmit = 0;
        }

        /* Do nothing if output is stopped. */
        if (tp->t_flags & T_STOP)
                need_xmit = 0;
        if (a1->a_ohlt)
                need_xmit = 0;

        /* Start data transmission by writing to UART xmit reg. */
        if ((lsr & LS_TxRDY) && need_xmit) {
                int xmit_count;
                xmit_count = (a1->a_ut == US_16550A)?16:1;
                asy_send(out_silo, port+DREG, xmit_count);
        }
        spl(s);
}
```

The Poll Routine

The kernel invokes function **asypoll()** when a user application invokes the system call **poll()** to poll a serial port.

Field **c_poll** in the **CON** structure contains a pointer to this function.

```
static int
asypoll(dev, ev, msec)
dev_t dev;
int ev;
int msec;
{
        int chan = channel(dev);
        asy1_t *a1 = asy1 + chan;
        TTY *tp = & a1->a_tty;
        return ttpoll(tp, ev, msec);
}
```

Wake Up Sleeping Devices

Function **asycycle()** wakes up of any sleeping ports at regular intervals.

```
static void
asycycle(chan)
int chan;
{
        asy0_t *a0 = asy0 + chan;
        asy1_t *a1 = asy1 + chan;
        TTY *tp = &a1->a_tty;
        short port = a0->a_port;
        int s;
        char msr, mcr;
        silo_t *out_silo = &a1->a_out;
        silo_t *in_silo = &a1->a_in;
        int n, ch;
        int do_start = 1;
        unsigned char iir;

        /* Check Carrier Detect (RLSD).
         * Modem status interrupts were not enabled due to 8250 hardware bug.
         * Enabling modem status and receive interrupts may cause lockup
         * on older parts.
         */
        if (tp->t_flags & T_MODC) {
                /* Get status */
                msr = inb(port+MSR);

                /* Carrier changed. */
                if ((msr & MS_RLSD) && !(tp->t_flags & T_CARR)) {
                        /* Carrier is on - wakeup open. */
                        s = sphi();
                        tp->t_flags |= T_CARR;
                        spl(s);
                        wakeup((char *)(&tp->t_open));
                }

                if (!(msr & MS_RLSD) && (tp->t_flags & T_CARR)) {
                        s = sphi();
                        RAWIN_FLUSH(in_silo);
                        RAWOUT_FLUSH(out_silo);
                        tp->t_flags &= ~T_CARR;
                        spl(s);
                        tthup(tp);
                }
        }
```

```
/* Empty raw input buffer.
 * The line discipline module (tty.c) will set T_ISTOP true when the
 * tt input queue is nearly full (tp->t_iq.cq_cc >= IHILIM), and make
 * T_ISTOP false when it's ready for more input.
 * When T_ISTOP is true, ttin() simply discards the character passed.
 */
if (!(tp->t_flags & T_ISTOP)) {
        while (in_silo->SILO_CHAR_COUNT > 0) {
                s = sphi();
                ttin(tp, in_silo->si_buf[in_silo->si_ox]);
                if (in_silo->si_ox < MAX_SILO_INDEX)
                        in_silo->si_ox++;
                else
                        in_silo->si_ox = 0;
                in_silo->SILO_CHAR_COUNT--;
                spl(s);
        }
}

/* Hardware flow control.
 *      Check CTS to see if we need to halt output.
 *      (MS_INTR should have done this - repeat code here to be sure)
 *      Check input silo to see if we need to raise RTS.
 */
if (tp->t_flags & T_CFLOW) {
        /* Get status */
        msr = inb(port+MSR);
        s = sphi();
        if (msr & MS_CTS)
                a1->a_ohlt = 0;
        else
                a1->a_ohlt = 1;
        spl(s);

        /* If using hardware flow control, see if we need to drop RTS. */
        if ((tp->t_flags & T_CFLOW)
        && (in_silo->SILO_CHAR_COUNT > SILO_HIGH_MARK)) {
                s = sphi();
                mcr = inb(port+MCR);
                if (mcr & MC_RTS) {
                        outb(port+MCR, mcr & ~MC_RTS);
                }
                spl(s);
        }

        /* If input silo below low mark, assert RTS */
        if (in_silo->SILO_CHAR_COUNT <= SILO_LOW_MARK) {
                s = sphi();
                mcr = inb(port+MCR);

                if ((mcr & MC_RTS) == 0) {
                        outb(port+MCR, mcr | MC_RTS);
                }
                spl(s);
        }
}

/* Calculate free output slot count. */
n = sizeof(out_silo->si_buf) - 1;
n += out_silo->si_ox - out_silo->si_ix;
n %= sizeof(out_silo->si_buf);
```

```
/* Fill raw output buffer */
for (;;) {
        if (--n < 0)
                break;
        s = sphi();
        ch = ttout(tp);
        spl(s);
        if (ch < 0)
                break;

        s = sphi();
        out_silo->si_buf[out_silo->si_ix] = ch;
        if (out_silo->si_ix >= sizeof(out_silo->si_buf) - 1)
                out_silo->si_ix = 0;
        else
                out_silo->si_ix++;
        spl(s);
}

/* if port has an interrupt pending (probably missed an irq)
 *      the following two loops should not be merged
 *      - need ALL port irq's inactive at once
 *      for each port on this irq line (use irq1 for this)
 *              disable interrupts (clear IER)
 *      for each port on this irq line
 *              restore interrupts
 */
if (a1->a_has_irq && ((iir = inb (port + IIR)) & 1) == 0) {
        struct irqnode *ip;
        asy_gp_t *gp;
        int s;
        short p;
        char c, slot;

        do_start = 0;
        s = sphi ();
        ip = irq1 [a0->a_irqno];

        while(ip) {
                if (ip->func == asyintr) {
                        p = ip->arg;
                        outb (p + IER, 0);
                } else {
                        gp = asy_gp + ip->arg;
                        for (slot = 0; slot < MAX_SLOTS; slot++) {
                                if ((c = gp->chan_list [slot]) <
                                    MAX_ASY) {
                                        p = asy0 [c].a_port;
                                        outb (p + IER, 0);
                                }
                        }
                }
                ip = ip->next_actv;
        }
```

```
                /* Now, all ports on the offending irq line have irq off. */
                ip = irq1 [a0->a_irqno];
                while (ip) {
                        if (ip->func == asyintr) {
                                p = ip->arg;
                                outb (p + IER, IEN);
                        } else {
                                gp = asy_gp + ip->arg;
                                for (slot = 0; slot < MAX_SLOTS; slot++) {
                                        if ((c = gp->chan_list [slot]) <
                                            MAX_ASY){
                                                p = asy0 [c].a_port;
                                                outb (p + IER, IEN);
                                        }
                                }
                        }
                        ip = ip->next_actv;
                }
                spl (s);
        }

        if (do_start)
                ttstart (tp);

        /* Schedule next cycle. */
        if (a1->a_in_use)
                timeout (& tp->t_rawtim, HZ / 10, asycycle, chan);
}
```

Suppress Interrupts During Chip Sensing

Function **irqdummy()** suppresses interrupts that may occur during chip sensing.

```
static void
irqdummy()
{
        /* Try to clear all pending interrupts. */
        inb(dummy_port+IIR);
        inb(dummy_port+LSR);
        inb(dummy_port+MSR);
        inb(dummy_port+DREG);
}
```

Add a Port Information to IRQ0 List

Function **add_irq()** adds information about a port to the **irq0** list.

```
static void
add_irq(irq, func, arg)
int irq;
int (*func)();
int arg;
{
        struct irqnode * np;

        /* Sanity check */
        if (irq <= 0 || irq >= NUM_IRQ)
                return;
```

```
        if (nextnode < MAX_ASY) {
                np = nodespace + nextnode++;
                np->func = func;
                np->arg = arg;
                np->next = irq0[irq];
                irq0[irq] = np;
        } else {
                printf("asy: too many irq nodes (%d)\n", nextnode);
        }
}
```

Service an Interrupt

Function **asy_irq()** services an async interrupt.

```
static void
asy_irq (level)
int level;
{
        struct irqnode *ip = irq1 [level];
        int doit;

        do {
                struct irqnode * here = ip;

                doit = 0;

                while (here != NULL) {
                        doit |= (* here->func) (here->arg);
                        here = here->next_actv;
                }
        } while (doit);
}
```

Rebuild Links for Active Devices

Function **upd_irq1()** rebuild the links for active devices.

```
static void
upd_irq1(irq)
int irq;
{
        struct irqnode *np;
        asy1_t *a1;
        int chan;
        int s;

        /* Sanity check */
        if (irq <= 0 || irq >= NUM_IRQ)
                return;

        /* For each node in the irq0 list
         *      if node is for irq status port
         *              for each channel using the status port
         *                      if channel in use, in irq mode
         *                              add node to irq1 list
         *                              skip rest of channels for this node
         *      else - node is for simple UART
         *              if channel in use, in irq mode
         *                      add node to irq1 list
         */
```

```
        s = sphi();
        np = irq0[irq];
        irq1[irq] = 0;
        while (np) {
                if (np->func != asyintr) {
                        char ix, loop = 1;
                        asy_gp_t *gp = asy_gp + np->arg;

                        for (ix = 0; ix < MAX_SLOTS && loop; ix++) {
                                if ((chan = gp->chan_list[ix]) < MAX_ASY) {
                                        a1 = asy1 + chan;
                                        if (a1->a_in_use && a1->a_irq) {
                                                np->next_actv = irq1[irq];
                                                irq1[irq] = np;
                                                loop = 0;
                                        }
                                }
                        }

                } else {
                        a1 = asy1 + np->arg;
                        if (a1->a_in_use && a1->a_irq) {
                                np->next_actv = irq1[irq];
                                irq1[irq] = np;
                        }
                }
                np = np->next;
        }
        spl(s);
}
```

The Break Routine

Function **asybreak()** breaks connection with a port.

```
static void
asybreak(chan)
int chan;
{
        int s;
        asy1_t *a1 = asy1 + chan;
        silo_t *out_silo = &a1->a_out;
        silo_t *in_silo = &a1->a_in;
        TTY *tp = &a1->a_tty;

        s = sphi();
        RAWIN_FLUSH(in_silo);
        RAWOUT_FLUSH(out_silo);
        spl(s);
        ttsignal(tp, SIGINT);
}
```

Handle an Interrupt

Function **asyintr()** handles an interrupt for a single channel.

```
static int
asyintr(chan)
int chan;
{
        asy0_t *a0 = asy0 + chan;
        asy1_t *a1 = asy1 + chan;
        TTY *tp = &a1->a_tty;
        silo_t *out_silo = &a1->a_out;
        silo_t *in_silo = &a1->a_in;
        int c, xmit_count;
        int ret = 0;
        short port = a0->a_port;
        unsigned char msr, lsr;

        if (chan >= MAX_ASY) {
                printf("asy: irq on channel %d\n", chan);
                return 0;
        }

rescan:
        switch (inb(port+IIR) & 0x07) {
        case LS_INTR:
                ret = 1;
                lsr = inb(port + LSR);
                if (lsr & LS_BREAK)
                        defer(asybreak, chan);
                goto rescan;

        case Rx_INTR:
                ret = 1;
                c = inb(port+DREG);
                if (tp->t_open == 0)
                        goto rescan;

                /* Must recognize XOFF quickly to avoid transmit overrun.
                 * Recognize XON here as well to avoid race conditions.
                 */
                if (_IS_IXON_MODE (tp)) {
                        /* XON */
                        if (_IS_START_CHAR (tp, c) ||
                            (_IS_IXANY_MODE (tp) &&
                             (tp->t_flags & T_STOP) != 0)) {
                                tp->t_flags &= ~(T_STOP | T_XSTOP);
                                goto rescan;
                        }

                        /* XOFF */
                        if (_IS_STOP_CHAR (tp, c)) {
                                tp->t_flags |= T_STOP;
                                goto rescan;
                        }
                }

                /* Save char in raw input buffer. */
                if (in_silo->SILO_CHAR_COUNT < MAX_SILO_CHARS) {
                        in_silo->si_buf[in_silo->si_ix] = c;
                        if (in_silo->si_ix < MAX_SILO_INDEX)
                                in_silo->si_ix ++;
                        else
                                in_silo->si_ix = 0;
                        in_silo->SILO_CHAR_COUNT ++;
                }
```

```
                /* If using hardware flow control, see if we need to drop RTS. */
                if ((tp->t_flags & T_CFLOW) != 0 &&
                    in_silo->SILO_CHAR_COUNT > SILO_HIGH_MARK) {
                        unsigned char mcr = inb (port + MCR);
                        if (mcr & MC_RTS) {
                                outb(port+MCR, mcr & ~MC_RTS);
                        }
                }
                goto rescan;

        case Tx_INTR:
                ret = 1;
                /* Do nothing if output is stopped. */
                if (tp->t_flags & T_STOP) {
                        goto rescan;
                }
                if (a1->a_ohlt)
                        goto rescan;

                /* Transmit next char in raw output buffer. */
                xmit_count = (a1->a_ut == US_16550A)?16:1;
                asy_send(out_silo, port+DREG, xmit_count);
                goto rescan;

        case MS_INTR:
                ret = 1;
                /* Get status (and clear interrupt). */
                msr = inb(port+MSR);

                /* Hardware flow control.
                 *      Check CTS to see if we need to halt output.
                 */
                if (tp->t_flags & T_CFLOW) {
                        if (msr & MS_CTS)
                                a1->a_ohlt = 0;
                        else
                                a1->a_ohlt = 1;
                }
                goto rescan;

        default:
                return ret;
        } /* endswitch */
}
```

Handle Timer Interrupts

Function **asyclk()** is called every time **TO** interrupts. If it returns zero, **asy** performs the usual system timer routines. It polls all pollable ports.

```
static int
asyclk()
{
        static int count;
        int ix;

        for (ix = 0; ix < ppnum; ix++)
                asysph(pptbl[ix]);
        count++;
        if (count >= poll_divisor)
                count = 0;
        return count;
}
```

Set Polling Rate on a Port

Function **asyspr()** sets the polling rate on a port. **asy** calls it when a port is opened, closed, or changes speed. It sets the polling rate only as fast as needed, and shuts off polling whenever possible. It updates the links in **irq1[0]**, which lists the polled-mode ports.

```
static void
asyspr()
{
        asy0_t *a0;
        asy1_t *a1;
        int chan;
        int s;
        int ix, max_rate, port_rate;

        /* Rebuild table of pollable ports. */
        s = sphi();
        ppnum = 0;
        for (chan = 0; chan < ASY_NUM; chan++) {
                a1 = asy1 + chan;
                if (a1->a_poll)
                        pptbl[ppnum++] = chan;
        }
        spl(s);

        /* If another driver has the polling clock, do nothing. */
        if (poll_owner & ~ POLL_ASY)
                return;

        /* Find highest valid polling rate in units of HZ/10.
         * If using FIFO chip, can poll at 1/16 the usual rate.
         */
        max_rate = 0;
        for (ix = 0; ix < ppnum; ix++) {
                chan = pptbl[ix];
                a0 = asy0 + chan;
                a1 = asy1 + chan;
                port_rate = alp_rate[a0->a_speed];
                if (a1->a_ut == US_16550A) {
                        port_rate /= 16;
                        if (port_rate % HZ)
                                port_rate += HZ - (port_rate % HZ);
                }

                if (max_rate < port_rate)
                        max_rate = port_rate;
        }

        /* if max_rate is not current rate, adjust the system clock */
        if (max_rate != poll_rate) {
                poll_rate = max_rate;
                poll_divisor = poll_rate/HZ; /* used in asyclk() */
                altclk_out(); /* stop previous polling */
                poll_owner &= ~ POLL_ASY;

                if (poll_rate) { /* resume polling at new rate if needed */
                        poll_owner |= POLL_ASY;
                        altclk_in(poll_rate, asyclk);
                }
        }
}
```

Handle Polling

Function **asysph()** handles the polling of serial ports.

```
static void
asysph(chan)
int chan;
{
        asy0_t *a0 = asy0 + chan;
        asy1_t *a1 = asy1 + chan;
        TTY *tp = &a1->a_tty;
        silo_t *out_silo = &a1->a_out;
        silo_t *in_silo = &a1->a_in;
        int c, xmit_count;
        short port = a0->a_port;
        char lsr;

        /* Check for received break first.
         * This status is wiped out on reading the LSR.
         */
        lsr = inb(port + LSR);
        if (lsr & LS_BREAK)
                defer(asybreak, chan);

        /* Handle all incoming characters. */
        for (;;) {
                lsr = inb(port + LSR);
                if ((lsr & LS_RxRDY) == 0)
                        break;
                c = inb(port+DREG);
                if (tp->t_open == 0)
                        continue;

                /* Must recognize XOFF quickly to avoid transmit overrun.
                 * Recognize XON here as well to avoid race conditions.
                 */
                if (_IS_IXON_MODE (tp)) {
                        /* XOFF */
                        if (_IS_STOP_CHAR (tp, c)) {
                                tp->t_flags |= T_STOP;
                                continue;
                        }

                        /* XON */
                        if (_IS_START_CHAR (tp, c)) {
                                tp->t_flags &= ~T_STOP;
                                continue;
                        }
                }

                /* Save char in raw input buffer. */
                if (in_silo->SILO_CHAR_COUNT < MAX_SILO_CHARS) {
                        in_silo->si_buf[in_silo->si_ix] = c;
                        if (in_silo->si_ix < MAX_SILO_INDEX)
                                in_silo->si_ix++;
                        else
                                in_silo->si_ix = 0;
                        in_silo->SILO_CHAR_COUNT++;
                }

                /* If using hardware flow control, see if we need to drop RTS. */
                if ((tp->t_flags & T_CFLOW)
                  && (in_silo->SILO_CHAR_COUNT > SILO_HIGH_MARK)) {
                        unsigned char mcr = inb(port+MCR);
                        if (mcr & MC_RTS) {
                                outb(port+MCR, mcr & ~MC_RTS);
                        }
                }
        }
```

```
        /* Handle outgoing characters. Do nothing if output is stopped. */
        lsr = inb(port + LSR);
        if ((lsr & LS_TxRDY)
          && !(tp->t_flags & T_STOP)
          && !(a1->a_ohlt)) {
                /* Transmit next char in raw output buffer. */
                xmit_count = (a1->a_ut == US_16550A)?16:1;
                asy_send(out_silo, port+DREG, xmit_count);
        }

        /* Hardware flow control.
         *      Check CTS to see if we need to halt output.
         */
        if (tp->t_flags & T_CFLOW) {
                if (inb(port+MSR) & MS_CTS)
                        a1->a_ohlt = 0;
                else
                        a1->a_ohlt = 1;
        }
}
```

Write to UART

Function **asy_send()** write to the **xmit** data register of the UART. Assume all checking about whether it's time to send has been done already. This function is called by time-critical IRQ and polling routines!

Argument *rawout* is the output silo for the TTY structure that supplies data to the port. *dreg* is the I/O address of the UART **xmit** data register. *xmit_count* is the maximum number of characters we can write (16 for FIFO parts).

```
static int
asy_send(rawout, dreg, xmit_count)
register silo_t *rawout;
int dreg, xmit_count;
{
        /* Transmit next chars in raw output buffer. */
        for (;(rawout->si_ix != rawout->si_ox) && xmit_count; xmit_count--) {
                outb(dreg, rawout->si_buf[rawout->si_ox]);
                /* Adjust raw output buffer output index. */
                if (++rawout->si_ox >= sizeof(rawout->si_buf))
                        rawout->si_ox = 0;
        }
        return xmit_count;
}
```

Interrupt Handler for Comtrol-Type Port Groups

Function **p1()** is the interrupt handler for Comtrol-type port groups. The status register has one in bit positions for interrupting ports.

```
static int
p1(g)
int g;
{
        asy_gp_t *gp = asy_gp + g;
        short port = gp->stat_port;
        unsigned char status, index, chan;
        int safety = LOOP_LIMIT;
        int ret = 0;
```

```
        /* while any port is active
         *      call simple interrupt handler for active channel
         */
        while (status = inb(port)) {
                ret = 1;
                index = 0;
                if (status & 0xf0) {
                        status &= 0xf0;
                        index +=4;
                } else
                        status &= 0x0f;

                if (status & 0xcc) {
                        status &= 0xcc;
                        index +=2;
                } else
                        status &= 0x33;

                if (status & 0xaa)
                        index++;
                chan = gp->chan_list[index];
                asyintr(chan);

                if (safety-- == 0) {
                        printf("asy: p1 runaway - status %x\n", status);
                        break;
                }
        }
        return ret;
}
```

Interrupt Handler for Arnet-Type Port Groups

Function **p2()** is the interrupt handler for Arnet-type port groups. The status register has zero in bit positions for interrupting ports.

```
static int
p2(g)
int g;
{
        asy_gp_t *gp = asy_gp + g;
        short port = gp->stat_port;
        unsigned char status, index, chan;
        int safety = LOOP_LIMIT;
        int ret = 0;

        /* while any port is active
         *      call simple interrupt handler for active channel
         */
        while (status = ~inb(port)) {
                ret = 1;
                index = 0;
                if (status & 0xf0) {
                        status &= 0xf0;
                        index +=4;
                } else
                        status &= 0x0f;

                if (status & 0xcc) {
                        status &= 0xcc;
                        index +=2;
                } else
                        status &= 0x33;
```

```
                if (status & 0xaa)
                        index++;
                chan = gp->chan_list[index];
                asyintr(chan);
                if (safety-- == 0) {
                        printf("asy: p2 runaway - status %x\n", status);
                        break;
                }
        }
        return ret;
}
```

Interrupt Handler for GTEK-Type Port Groups

Function **p3()** is the interrupt handler for GTEK-type port groups.

```
static int
p3(g)
int g;
{
        asy_gp_t *gp = asy_gp + g;
        short port = gp->stat_port;
        unsigned char index, chan;

        /* Call simple interrupt handler for active channel. */
        index = inb(port) & 7;
        chan = gp->chan_list[index];
        return asyintr(chan);
}
```

Interrupt Handler for DigiBoard-Type Port Groups

Function **p4()** is the interrupt handler for DigiBoard-type port groups.

```
static int
p4(g)
int g;
{
        asy_gp_t *gp = asy_gp + g;
        short port = gp->stat_port;
        unsigned char index, chan;
        int ret = 0;
        int safety = LOOP_LIMIT;

        /* Status register has slot number for active port,
         * or 0xFF if no port is active.
         */
        for (;;) {
                index = inb(port);
                if (index == 0xFF)
                        break;

                if (safety-- == 0) {
                        printf("asy: p4 runaway - status %x\n", index);
                        break;
                }

                ret = 1;
                chan = gp->chan_list[index&0xF];
                asyintr(chan);
        }
        return ret;
}
```

The CON Structure

Finally, the **CON** structure holds pointers to the driver's functions that are invoked by the kernel.

```
CON asycon = {
        DFCHR|DFPOL,                    /* Flags */
        ASY_MAJOR,                      /* Major index */
        asyopen,                        /* Open */
        asyclose,                       /* Close */
        NULL,                           /* Block */
        asyread,                        /* Read */
        asywrite,                       /* Write */
        asyioctl,                       /* Ioctl */
        NULL,                           /* Powerfail */
        asytimer,                       /* Timeout */
        asyload,                        /* Load */
        asyunload,                      /* Unload */
        asypoll                         /* Poll */
};
```

Where To Go From Here

The Lexicon describes the functions invoked within this driver. The previous section gives an example of a driver for a block device.

Introduction to the Lexicon

The following section describes the functions and macros that are currently used in COHERENT device drivers. These include internal kernel routines, DDI/DKI routines, and STREAMS routines.

The root article is this Lexicon the one entitled **device driver**. If you are unfamiliar with how a Lexicon works, you should read this article first. It introduces the Overview articles, and discusses the families of articles in the Lexicon.

adjmsg() — DDI/DKI Kernel Routine

Clip a message
#include<sys/stream.h>
int adjmsg(*msgptr***,** *length***)**
mblk_t **msgptr***;*
int *length***;**

adjmsg() trims *length* bytes from from the message to which *msgptr* points. If *length* is greater than zero, **adjmsg()** trims the beginning of the message; if it is less than zero, **adjmsg()** trims the end.

If all goes well, **adjmsg()** returns one. It fails and returns zero if either of the following conditions occurs:

1. The absolute value of *length* exceeds the number of bytes to which *msgptr* points.

2. *length* spans more than one message block, but the message's blocks are not all of the same type.

See Also

DDI/DKI kernel routines, msgb

Notes

adjmsg() has base or interrupt level. It does not sleep. An application can hold driver-defined basic locks, read/write locks, and sleep locks across calls to this function.

If *length* exceeds the amount of data in a message block, **adjmsg()** sets the block's read and write pointers equal to each other to indicate that the block contains no data. It does not free the block.

allocb() — DDI/DKI Kernel Routine

Allocate a message block
#include <sys/types.h>
#include <sys/stream.h>
mblk_t *allocb(*size***,** *priority***)**
int *size***; uint_t** *priority***;**

allocb() allocates a STREAMS message block that is *size* bytes long.

priority gives the message's priority, as follows:

BPRI_LO	Low priority. Use this for routine allocation of data.
BPRI_MED	Medium priority. Use this to allocate blocks that are not critical, but are not data either.
BPRI_HI	High priority. Use this for allocations that must succeed; note, however, that the DDI/DKI does not guarantee success.

Note that some implementations ignore *priority*.

If all goes well, **allocb()** returns a pointer to the allocated message block; otherwise, it returns NULL.

See Also

DDI/DKI kernel routines, freeb(), msgb

Notes

allocb() has base or interrupt level. It does not sleep.

An application can hold driver-defined basic locks, read/write locks, and sleep locks across calls to this function.

altclk_in() — Internal Kernel Routine

Install polling function
int altclk_in(*hz, fn***)**
int *hz***, (***fn***)();**

altclk_in() increases the system clock rate from the value set by manifest constant **HZ** (at present, 100 Hertz) to *hz*. *fn* points to the function to be called whenever the clock interrupt occurs. *hz* must be an integral multiple of **HZ**; therefore, the rate of clock interrupts will be increased by a factor of *hz***/HZ**. *fn* is an **int**-valued function that must return zero every *hz***/HZ**'th time it is called, nonzero the rest of the time. The zero value returned from *fn* tells the COHERENT system's clock routine to do its usual processing.

altclk_in() returns zero if it completes normally; if argument *hz* is less than **HZ** or not an integral multiple of **HZ**, this function does nothing and returns -1.

See Also

altclk_out(), internal kernel routines

altclk_out() — Internal Kernel Routine

Uninstall polling function
int (*altclk_out)();

altclk_out() ends polling (previously installed with function **altclk_in()**). It restores the COHERENT clock rate to the value of the manifest constant **HZ** (at present, 100 Hertz) and unhooks the polling function. It returns the value of the previous pointer to the polling function.

See Also

alkclk_in(), internal kernel routines

ASSERT() — DDI/DKI Kernel Routine

Debug an expression
#include <sys/debug.h>
void ASSERT(*expression***)**
int *expression***;**

ASSERT() tests the Boolean *expression* for correctness. You can use this routine to verify expressions in programs that you have compiled with the symbol **DEBUG #define**'d (for example, with the option **-d** to the O compiler).

If *expression* evaluates to non-zero (that is, the expression is correct), the call to **ASSERT()** has no effect. If, however, *expression* evaluates to zero, **ASSERT()** panics the system. It prints a message on the console that identifies *expression*, its source file, and its line number.

See Also

cmn_err(), DDI/DKI kernel routines

Notes

ASSERT() has base or interrupt level. It does not sleep.

A program can hold driver-defined basic locks, read/write locks, and sleep locks across calls to **ASSERT()**.

backq() — DDI/DKI Kernel Routine

Get a pointer to the preceding queue
#include <sys/stream.h>
queue_t *backq(*queue***)**
queue_t **queue***;*

backq() returns the address of the queue that preceeds the queue to which *queue* points. If *queue* points to a read queue, **backq()** returns a pointer to the queue downstream from *queue*, unless *queue* is the end of the stream. If *queue*, is a write queue, **backq()** returns a pointer to the next queue upstream from *queue*, unless *queue* is the head of the stream. If something goes wrong, **backq()** returns NULL.

Level

Base or interrupt.

See Also

DDI/DKI kernel routines

Notes

backq() does not sleep.

The calling function cannot have the stream frozen when it calls this function.

The caller can hold driver-defined basic locks, read/write locks, and sleep locks across calls to this function.

bcanput() — DDI/DKI Kernel Routine

Test whether a priority band has room for a message
#include <sys/types.h>
#include <sys/stream.h>
int bcanput(*queue***,** *priority***)**
queue_t ******queue***; uchar_t** *priority***;**

bcanput() tests whether the priority band *priority* within *queue* has room for a message. If *priority* equals zero, **bcanput()** behaves identically to a call to **canput()**. *queue* must have a service procedure.

bcanput() returns one if the queue pointed to by *queue* contains room for a room with a priority of *priority*; it returns zero if *queue* does not.

See Also

bcanputnext(), canput(), canputnext(), DDI/DKI kernel routines, putbq(), putnext(), queue

Notes

bcanput() has base or interrupt level. It does not sleep.

queue argument cannot reference field **q_next**. Use **bcanputnext()** to perform this task.

Before you enqueue a message, you must first call **canput()** to test whether *queue* has room for a message of the given *priority*, even if **bcanput()** indicates that space is available. Do not send the message if no room is available.

Because of race conditions, **bcanput()** can state that *priority* has room for a message, but *priority* could be filled by another process before your process enqueues its message. This, however, is a benign problem.

Your process cannot have the stream frozen when it calls this function. Your process can hold driver-defined basic locks, read/write locks, and sleep locks across a call to this function.

bcanputnext() — DDI/DKI Kernel Routine

Test whether a priority band has room for a message
#include <sys/types.h>
#include <sys/stream.h>
int bcanputnext(*queue***,** *priority***)**
queue_t ******queue***; uchar_t** *priority***;**

bcanputnext() attempts to find a queue with a priority band of level *priority* that can hold a message.

bcanputnext() search the stream beginning at *queue***->q_next**. It seeks a queue that contains a service routine. If it finds one, it tests the queue to see whether it can hold a message in priority band *priority*. If the band is full, **bcanputnext()** marks the queue so that the caller's service routine is back-enabled automatically when the amount of data in the queue reaches its low-water mark.

bcanputnext() returns one if a message of *priority* can be put into the stream, or if it reached the end of the stream without finding a queue that has a service procedure. If the stream is full, it returns zero.

See Also

bcanput(), canput(), canputnext(), DDI/DKI kernel routines, putbq(), putnext(), queue

Notes

bcanputnext() has base or interrupt level. It does not sleep.

You must test whether *queue* has room for a message of the given *priority*, and not send the message if no room is available. Because of race conditions, **bcanput()** can state that *priority* has room for a message, but *priority* could be filled by another process before your process enqueues its message. This, however, is a benign problem.

Your process cannot have the stream frozen when it calls this function. Your process can hold driver-defined basic locks, read/write locks, and sleep locks across a call to this function.

bclaim() — Internal Kernel Routine

Claim a buffer
#include <sys/buf.h>
BUF *bclaim(*device***,** *block***)**
dev_t *device***; daddr_t** *block***;**

bclaim() locates or allocates a buffer associated with *block* on *device*. The buffer's contents are invalid if its field *b_flag* has bit **BFNTP** set.

bclaim() requires user context. Therefore, do not call it from within deferred or timed functions, or from within an interrupt handler.

See Also

internal kernel routines

bcopy() — DDI/DKI Kernel Routine

Copy data between locations within the kernel
#include <sys/types.h>
void bcopy(*source***,** *target***,** *count***)**
caddr_t *from***,** *to***; size_t** *count***;**

bcopy() copies *count* bytes from kernel address *source* to kernel address *target*. If the the chunk of memory pointed to by *source* overlaps *target*, the results are undefined (and probably unwelcome).

See Also

bzero(), copyin(), copyout(), DDI/DKI kernel routines, uiomove(), ureadc(), uwritec()

Notes

bcopy() has base or interrupt level. It does not sleep.

A function can hold driver-defined basic locks, read/write locks, and sleep locks across a call to this function.

To copy data between kernel and user space, use **copyin()**, **copyout()**, **uiomove()**, **ureadc()**, or **uwritec**().

This function is equivalent to the library routine **memcpy()**.

bdone() — Internal Kernel Routine

Block I/O completed
#include <sys/buf.h>
void bdone(*bp***)**
BUF * *bp***;**

A driver for a block device built around the internal kernel routines must call **bdone()** when it has completed I/O for the buffer pointed to by *bp*. If an I/O error occurred, the driver must set the **BFERR** bit in field *bp***->b_flag** before it calls **bdone()**.

See Also

internal kernel routines

bflush() — Internal Kernel Routine

Flush buffer cache
#include <sys/buf.h>
void bflush(*device***)**
dev_t *device***;**

bflush() synchronizes all blocks for *device* in the buffer cache, and invalidates all references. COHERENT 4.2 uses this routine when it unmounted a file system.

See Also

internal kernel routines

block — Entry-Point Routine

Invoke a driver block interface
#include <sys/con.h>
void *prefix***block(***bp***)**
buf_t **bp***;*

Under the internal COHERENT device-driver interface, the entry point **block** gives access to the driver's routine for executing a block interface with the device. The address of this routine is given in field **c_block** of the driver's **CON** structure.

Argument *bp* points to the **BUF** structure that describes how data are written to or read from this device. For details, see the entry for **BUF** in this manual.

See Also

BUF, CON, entry-point routines, strategy

bread() — Internal Kernel Routine

Read into buffer cache
#include <sys/buf.h>
BUF * **bread(***device***,** *bno***,** *flag***)**
dev_t *dev***; daddr_t** *bno***;**

bread() reads the block *bno* into the buffer cache. If *flag* is set, the read is synchronous (that is, **bread()** waits for I/O to complete), and returns a pointer to the buffer. Otherwise, the read is asynchronous (that it, it returns immediately), and **bread()** returns NULL. If the **BFERR** bit is set in the buffer's field **b_flag**, a read error occurred.

See Also

internal kernel routines

brelease() — Internal Kernel Routine

Release a buffer
#include <sys/buf.h>
void brelease(*bp***)**
BUF **bp***;*

brelease() unlocks and releases the buffer pointed to by *bp*.

A device driver built with the internal kernel routines must call **brelease()** when it no longer needs a buffer obtained via a **bread()**. If a driver mus read and modify a block, the recommended sequence is for it to call **bread()**, modify the block, set the **BFMOD** bit in the field **b_flag** field, then call **brelease()**.

See Also

internal kernel routines

bsync() — Internal Kernel Routine

Flush modified buffers
#include <sys/buf.h>
void bsync()

bsync() flushes modified buffers to all buffered devices. This synchronizes the entire buffer cache.

See Also

internal kernel routines

buf — Internal Data Structure

Buffer cache
#include <sys/buf.h>

A *buffer cache* is an area of memory that holds data being written to or read from a device. The kernel gives each block-special device its own buffer cache. The kernel, in turn, assigns to each buffer cache a copy of the structure **BUF**, which the kernel uses to manipulate that buffer cache. It is defined in header file **<sys/buf.h>**, and contains the following fields:

b_dev	This is a **dev_t** structure that describes the device being buffered. Use kernel macros **major()** and **minor()** to translate this structure into the device's major and minor numbers.
b_bno	This gives the number of the starting block.
b_req	This is the type of I/O requested, either **BREAD** or **BWRITE**.
b_count	This gives the number of bytes to read or write.
b_resid	This gives the number of bytes that remain to be transferred. Zero indicates that all data transferred correctly, i.e., that an error did not occur.
b_paddr	This gives the system global (DMA) address for the data.
b_vaddr	This field gives the virtual (non-DMA) address for the data.

See file **<sys/buf.h>** for full details on this structure.

See Also

internal data structures

bufcall() — DDI/DKI Kernel Routine

Call a function when a buffer becomes available
#include <sys/types.h>
#include <sys/stream.h>
toid_t bufcall(*size***,** *priority***,** *function***,** *argument***)**
uint_t *size***; int** *priority***;**
void (**function***)(); long** *argument***;**

bufcall() schedules *function* to be called with *argument* when a buffer of *size* bytes becomes available. You can use **bufcall()** to obtain a buffer at some time in the future, should a call to a buffer-allocation routine fail.

When *function* runs, all interrupts from STREAMS devices are blocked. *function* has no user context and cannot call any function that sleeps.

priority gives the priority of the request. You can use the following values:

BPRI_LO	Low (normal) priority.
BPRI_MED	Medium priority.
BPRI_HI	High priority.

bufcall() returns a non-zero value that identifies the scheduling request. You can pass this value to **unbufcall()** to cancel the request. If something goes wrong, **bufcall()** returns zero.

See Also

allocb(), DDI/DKI kernel routines, esballoc(), esbbcall(), itimeout(), unbufcall()

Notes

bufcall() has base or interrupt level. It does not sleep.

A function can hold driver-defined basic locks, read/write locks, and sleep locks across a call to this function.

bufcall() cannot guarantee that *function* will succeed. Although *function* will not be executed until a buffer of *size* bytes has become available, another driver may snatch the buffer between the time *function* is called and the time it tries to claim the memory for itself.

busyWait() — Internal Kernel Routine

Busy-wait the system, pending some event
int busyWait(*fn***,** *ticks***)**
int (**fn***)(),** *ticks***;**

busyWait() repeatedly calls the function to which *fn* points. It returns when *fn* returns a non-zero value, or after *ticks* clock ticks have elapsed, whichever happens first. If *fn* is NULL, **busyWait()** waits unconditionally.

busyWait() returns one if *fn* returned a nonzero value, or zero if it has timed out.

See Also

busyWait2(), drv_usecwait(), internal kernel routines

Notes

Each tick is one one-hundredth of a second. Busy-waiting the system for even one clock tick is a bad idea, except

while testing a driver or during system start-up.

busyWait2() — Internal Kernel Routine

Busy-wait the system, pending some event
int busyWait2(*fn***,** *ticks***)**
int (**fn***)(),** *ticks***;**

busyWait2() repeatedly calls the function to which *fn* points. It returns when *fn* returns a non-zero value, or after *ticks* timer ticks have elapsed, whichever happens first. If *fn* is NULL, **busyWait2()** waits unconditionally.

busyWait2() differs from the call **busyWait()** in that its granularity is finer: one count equals 1/(11932***HZ**) seconds, or about 0.84 microseconds.

busyWait2() returns one if *fn* returned a nonzero value, or zero if it has timed out.

See Also

drv_usecwait(), internal kernel routines

bwrite() — Internal Kernel Routine

Write buffer to disk
#include <sys/buf.h>
void bwrite(*bp***,** *flag***)**
BUF ****bp*; int** *flag***;**

bwrite writes out the buffer to which *bp* points. If *flag* is set, the write is synchronous and **bwrite()** does not return until I/O has completed; otherwise, the write is asynchronous and **bwrite()** returns immediately.

A device driver must lock the buffer gate before it calls **bwrite()**; if it does not, the buffer may be modified while it is being written.

See Also

internal kernel routines

bzero() — DDI/DKI Kernel Routine

Initialize a block of memory to zero
#include <sys/types.h>
void bzero(*address***,** *bytes***)**
caddr_t *address***; size_t** *number***;**

bzero() initializes to zero *number* bytes of memory, beginning at *address*. It returns nothing.

The block of memory described by *address* and *bytes* must lie within the kernel's address space and must reside in memory. If *address* lies within user space, the driver can corrupt the system in an unpredictable (and probably undesirable) way.

See Also

bcopy(), DDI/DKI kernel routines, kmem_zalloc()

Notes

bzero() has base or interrupt level. It does not sleep.

A function can hold driver-defined basic locks, read/write locks, and sleep locks across a call to this function.

This function is equivalent to the library routine **memset()**.

canput() — DDI/DKI Kernel Routine

Test whether a queue has room for a message
#include <sys/stream.h>
int canput(*queue***)**
queue_t ****queue*;**

canput() tests whether *queue* has room for a message. *queue* must have a service procedure.

canput() returns one if *queue* has room for a message; or zero if *queue* does not. Do *not* attempt to enqueue a message on *queue* if **canput()** does not return one.

See Also

bcanput(), bcanputnext(), canputnext(), DDI/DKI kernel routines, putbq(), putnext(), queue

Notes

canput() has base or interrupt level. It does not sleep.

A function can hold driver-defined basic locks, read/write locks, and sleep locks across a call to this function.

Note that **canput()** can indicate that *queue* has room for a message, but another message can fill *queue* before your process enqueues its message.

You cannot have the stream frozen when you call this function.

queue cannot reference field **q_next**. To examine the queue next to *queue*, call **canputnext()**.

canputnext() — DDI/DKI Kernel Routine

Test whether a queue has room for a message
#include <sys/stream.h>
int bcanputnext(*queue***)**
queue_t **queue***;*

canputnext() tests whether a queue has room for a message.

canputnext searches *queue* beginning at **canputnext()** until it finds a queue that has a service routine. If finds one, it tests whether that queue has room for a message. If the queue is full, **canputnext()** marks the queue so that the caller's service routine is back-enable automatically when the amount of data in the queue reaches its low-water mark.

canputnext() returns one if a message can be sent in the stream, or if it reaches the end of the stream without find a queue that contains a service routine. It returns zero if the queue with a service routine does not have room for a message.

See Also

bcanput(), bcanputnext(), canput(), DDI/DKI kernel routines, putbq(), putnext(), queue

Notes

canputnext() has base or interrupt level. It does not sleep.

A function can hold driver-defined basic locks, read/write locks, and sleep locks across a call to this function.

Note that **canputnext()** can indicate that *queue* has room for a message, but another message can fill *queue* before your process enqueues its message.

You cannot have the stream frozen when you call this function.

chpoll — Entry-Point Routine

Entry point for the polling routine
#include <sys/poll.h>
int *prefix***chpoll(***device***,** *events***,** *pointer***,** *events***,** *pollhead***)**
dev_t *device***; short** *events***; int** *pointer***; short** *events***;**
struct pollhead ***head***;**

chpoll is the entry point for polling a device. It is used only by character drivers that use the DDI/DKI interface; STREAMS drivers do not use it.

A **chpoll** routine takes the following arguments:

device The device being polled.

events A bitmask of the events to be polled, as follows:

POLLIN	Are data waiting to be read?
POLLOUT	May data be written without blocking?
POLLPRI	Are high-priority data waiting to be read?
POLLHUP	Has a device hung up?

POLLERR	Has a device error occurred?
POLLRDNORM	Are normal data waiting to be read?
POLLWRNORM	May normal data be written without blocking?
POLLRDBAND	Are out-of-band data waiting to be read?
POLLWRBAND	May out-of-band data be written without blocking?

pointer If this flag is set to true, the driver should return a pointer to its **pollhead** structure.

events The **chpoll** routine writes at this address a mask of the events that have occurred.

head The address of the **pollhead** structure to interrogate.

Note that the **pollhead** structure is totally opaque; a driver has no access to any of its fields.

The **chpoll** routine returns zero if all goes well. If something goes wrong, it returns an appropriate error value.

See Also

entry points, phalloc(), phfree(), pollhead, pollwakeup()

Notes

This entry point is used only drivers that use the DDI/DKI interface. It is optional.

close — Entry-Point Routine

Close a device

Internal-Kernel Interface:
#include <sys/cred.h>
#include <sys/ddi.h>
#include <sys/errno.h>
#include <sys/file.h>
#include <sys/open.h>
#include <sys/types.h>
int *prefix***close(***device***,** *mode***,** *flags***,** *credptr***,** *private***)**
dev_t *device***; int** *mode***,** *flags***; cred_t** **credptr***; **void** **private*

DDI/DKI:
#include <sys/cred.h>
#include <sys/ddi.h>
#include <sys/errno.h>
#include <sys/file.h>
#include <sys/open.h>
#include <sys/types.h>
int *prefix***close(***device***,** *flag***,** *type***,** *credptr***)**
dev_t *device***; int** *flag***,** *type***; cred_t** **credptr***;

STREAMS:
#include <sys/types.h>
#include <sys/stream.h>
#include <sys/file.h>
#include <sys/errno.h>
#include <sys/cred.h>
#include <sys/ddi.h>
int *prefix***close(***queue***,** *flag***,** *credptr***)**
queue_t *queue***; int** *flag***; cred_t** **credptr***;

A driver's **close** routine closes the connection between the user process and the device, and prepares the device to be opened again. Every driver must have this entry point. An application invokes it via the COHERENT system call **close()**. For details on this system call, see its entry in the COHERENT Lexicon.

The **close** routine should return zero if it succeeds in its tasks. If something goes wrong, it should return an appropriate error number. See the entry for **errno** in this manual for a list of error numbers. The driver determines how to react to an error.

The following describes the **close** routine for each flavor of driver-kernel interface.

Internal-Kernel Interface

Under the internal-kernel interface to a driver, field **c_close** in the driver's **CON** structure holds the address of this routine. It is customary to name the **close** routine with the word **close** prefixed by a unique identifier for your driver; but this is not required.

device is a **dev_t** that identifies the device to be closed.

mode and *flags* give, respectively, the mode into which *device* had been opened, and additional information about how it had been opened. See the article for the system call **open()** in the COHERENT Lexicon for a table of the legal values of these arguments.

credentials points to the credentials of the current user. If it wishes, your driver can read this structure to check the user's permissions before it closes *device*. Note that many drivers do not use this argument.

Finally, *private* points to a data element that is private to your driver. Note that many drivers do not use this argument.

DDI/DKI Interface

To invoke the **close** routine under the DDI/DKI interface, the kernel calls the function *prefix***close()**, where *prefix* is the unique prefix for this driver. The calling conventions are given in the second example, above.

device identifies the device to close.

flag gives the file-status flag. If the bits **FNDELAY** or **FNONBLOCK** are set, the driver should not sleep as it performs its close-related tasks.

type gives the type of the device. Your driver should use this field to determine how many times *device* was opened. At present, only one type is recognized:

OTYP_LYR
: A "layered" device. The kernel invokes the **close** routine for every corresponding call to the driver's **open** routine. With devices of this type, the driver must count each invocation of its **open** and **close** routines to determine when it should really close the device.

credptr points to the user's credential structure.

STREAMS Interface

The calling conventions for the **close** routine of a STREAMS driver are given in the third example at the beginning of this article.

queue points to the queue to be closed.

flag gives the file-status flag. For details, see the entry for **open** in this manual.

credptr points to the user's credential structure.

When a last reference to a stream is closed, the following steps are repeated in turn for every entity on the stream, from the stream head to the stream driver:

- If data are present on the write queue of the module or driver, the calling process waits up to 15 seconds for the data to drain normally. Once the queue is drained or the timeout expires, continue.
- The **close** routine of the module or driver is called.
- When the **close** routine returns, the queue and all the messages that were on it are deallocated automatically.

The 15-second timeout is to help prevent loss of data. In general, while the system can force data to be lost, it should try to avoid it. If an interactive process wants to hide these delays from the user, it can hand the final close-off to a child process.

See Also

CON, drv_priv(), entry-point routines, errno, open, qprocsoff(), queue, unbufcall(), untimeout()
COHERENT Lexicon: **close(), open()**

Notes

The **close** routine has user context and can sleep.

A STREAMS driver or module must call **qprocsoff()** to disable its **put** and **srv** routine before it returns from its **close**

routine.

clrivec() — Internal Kernel Routine

Clear interrupt vector
void clrivec(*level***)**
int *level***;**

clrivec() dissociates, or clears, the current handler for interrupt *level*.

See Also

internal kernel routines, setivec()

clrq() — Internal Kernel Routine

Clear character queue
#include <sys/clist.h>
void clrq(*cqp***)**
CQUEUE **cqp***;*

clrq() clears the character queue pointed to by *cqp*.

See Also

internal kernel routines

cltgetq() — Internal Kernel Routine

Get a char from a character queue
#include <sys/clist.h>
int cltgetq(*cqp***)**
CQUEUE **cqp***;*

cltgetq() returns the next character from character queue *cqp*. It returns -1 if the queue is empty.

See Also

internal kernel routines

Notes

Prior to release 4.2 of COHERENT, this function was named **getq()**. The name has been changed to avoid collision with a similarly named function in the STREAMS library.

cltputq() — Internal Kernel Routine

Put a character on a character queue
#include <sys/clist.h>
int cltputq(*cqp***,** *c***)**
CQUEUE **cqp***;* **char** *c***;**

cltputq() puts *c* onto the character queue referenced by *cqp*. It returns the character put, or -1 if something went wrong.

See Also

internal kernel routines

Notes

Prior to release 4.2, this function was named **putq()**. It has been renamed to avoid collision with a similarly named STREAMS function.

cmn_err() — DDI/DKI Kernel Routine

Handle an error
#include <sys/cmn_err.h>
void cmn_err(*level***,** *format***, ...)**
int *level***; char** **format***, ...;*

cmn_err() handles error conditions. It can display a message on the system console, or store the message within the kernel buffer **putbuf**. It can also panic the system.

level gives the severity of error, as follows:

CE_CONT Continue a previous message, or display a informative message that does not necessarily describe an error. This level tells **cmn_err()** to suppress the newline character that it normally adds to the end of the message it constructs; thus, this level permits you to build long messages. You can use messages of this type to help debug your driver, among other things; note, however, that using **cmn_err()** for this may change your system's timing.

CE_NOTE Display a message that begins with the string **NOTICE:**. Use messages of this type to report events that may not require action, but should interest the system administrator.

CE_WARN Display a messsage that begins with the string **WARNING:**. Use messages of this type to report events that require immediate action by the system administrator.

CE_PANIC Display a message that begins with the string **PANIC:**, and panic the system. Use this level only for errors so severe that the system must not continue to function.

format gives the message to be displayed. It can contain the conversion specifiers **%d**, **%o**, **%s**, **%u**, and **%x**. These specifiers work much the same as they under the COHERENT function **printf()**, except that **cmn_err()** does not recognize length specifications. *format* can be followed by zero or more arguments (indicated above by an ellipsis) that give the variables whose values are to be displayed. **cmn_err()** relates each argument to the corresponding format specifier within the string *format*, again just as **printf()** does.

By default, **cmn_err()** writes the message both onto the system console and into the kernel buffer **putbuf**. If the first character in *format* is an exclamation point '!', **cmn_err()** writes the message only into **putbuf**; whereas if the first character in *format* is a circumflex '^', it writes the message only onto the console. Kernel variable **putbufsz** sets the size of **putbuf**.

See Also

DDI/DKI kernel routines
COHERENT Lexicon: **db, printf(), tr (driver)**

Notes

cmn_err() does not sleep.

If *level* is anything other than **CE_PANIC**, the calling function can hold driver-defined basic locks, read/write locks, and sleep locks across a call to this function. If *level* is **CE_PANIC**, however, locks cannot be held — not that it matters, as the system is going down.

This function is equivalent to, and a replace for, the internal-kernel routines **printf()** and **devmsg()**.

con — Internal Data Structure

Structure of a device driver
The structure of a COHERENT device driver is set by the
CON
structure, which the header file
<sys/con.h>
defines as follows:

```
typedef struct con {
        int c_flag;
        int c_mind;
        void (*c_open);
        void (*c_close);
        void (*c_block);
        void (*c_read);
        void (* c_write);
        void (*c_ioctl);
        void (*c_power);
        void (*c_timer);
        void (*c_load);
        void (*c_uload);
        int  (*c_poll);
} CON;
```

Each of the fields in this header points to the equivalent of a DDI/DKI entry-point routine. The following subsections describe field in detail.

Flags

Field **c_flag** OR's the manners in which this device can be accessed, as followed:

DFBLK	Block-special device.
DFCHR	Character-special device.
DFTAP	Tape device.
DFPOL	Accessible via COHERENT system call **poll()**.

Major-Device Number

Field **c_mind** gives the device's major-device number. This number must be in the range of zero through 31. At present, the major-device number of each device driver is set in header file **devices.h**; in a future release of COHERENT, however, each device driver will be assigned its major-device number when the kernel is linked. Therefore, code should not depend upon the device having a particular "magic" major-device number.

Open Routine

Field **c_open** points to the routine within the device driver that is executed whenever the kernel opens the device. This function is always called with two arguments: the first is an **o_dev_t** that indicates the device being accessed, and the second is an integer that indicates the mode in which it is being opened. The mode can be **IPW** (write mode), **IPR** (read mode), or **IRW | IRP**. If an error occurs during execution of this function, it should call **set_user_error()** with an appropriate value.

Close Routine

Field **c_close** points to the routine that is executed whenever COHERENT closes the device. This function takes the same arguments as the open function.

Block Routine

Field **c_block** points to the routine within the device driver that is executed when the kernel reads a file in block mode. (As noted earlier, COHERENT — unlike most implementations of UNIX — permits your driver to open its device into either block- or character-special mode, should you wish.) This function is called with a pointer to a **buf** structure, which holds information about this driver's buffer cache. For more information on the **buf** structure, see its entry in this manual's Lexicon. The driver function that performs block transfers of data should first perform the I/O transfer, then set field **c_block->b_resid** to the appropriate number and call kernel function **bdone()** to clean up after itself.

Note that the function that performs block transfer must *never* sleep or access a process's **uproc** structure. This is because this function is asynchronous and therefore not pegged to a particular process.

Read Routine

Field **c_read** points to the driver's routine that is called when the kernel wishes to read data from that driver's device. It takes two arguments: an **o_dev_t** that indicates the device to read; and a pointer to that device's **io** structure, which is used by the read function. For more information on the **io** structure, see its entry in this manual's Lexicon.

Unlike a block transfer, the read function does not return until I/O is complete. Your driver can use the kernel functions **sleep()** and **wakeup()** to surrender the processor to another process while the read is being performed. It can also use the kernel function **ioputc()** to send characters to the user process and to update counter **io_ioc**.

Write Routine

Field **c_write** points to the function that the kernel executes when it wishes to write to this device. It behaves exactly the same as the function pointed to by field **c_read**, except that the direction of data transfer is reversed. Your driver can use kernel function **iogetc()** is used to fetch characters from the user process and to update counter **io_ioc**.

I/O Control Routine

Field **c_ioctl** points to the function that the kernel executes when it wishes to exert I/O control over a device. This function is called to perform non-standard manipulations of a device, e.g., format a disk, rewind a tape, or change the speed of a serial port.

The kernel always calls this function with three arguments. The first is an **o_dev_t** that identifies the device to be manipulated; the second is an integer that indicates the command to be executed; and the third points to an array

that can hold additional information, if any, that the command may need.

This function, by its nature, uses a considerable amount of device-specific information. The header files **<sys/tty.h>**, **<sys/mtioctl.h>**, and **<sys/lpioctl.h>** define codes for, respectively, teletypewriter devices (i.e., terminals), magnetic-tape devices, and line printers.

Power-Fail Routine

Field **c_power** points to the routine to be executed should power fail on the system. This field is not yet used by COHERENT.

Timeout Routine

Field **c_timer** points to the routine that the kernel executes when a device driver requests periodic scheduling. To request that the timeout routine for device *dev* be called once per second, set **drvl[major(***dev***)].d_time** to a nonzero value. The external variable **drvl[]** is declared in header file **<sys/con.h>**; macro **major()** is defined in header file **<sys/stat.h>**.

> The kernel's clock routines do not affect the value in field **d_time**. To stop invocations of the timeout routine, store zero in **drvl[major(***dev***)].d_time**.

dev is an **o_dev_t** that indicates which device is being timed out.

Load Routine

Field **c_load** points to the routine that would be executed when this device driver were loaded. It performs all tasks necessary to prepare the device and the driver to exchange information. Because COHERENT does not support loadable device drivers, the kernel executes this routine when COHERENT is booted.

Unload Routine

The field **c_uload** points to the driver's function that the kernel invokes when the driver is unloaded from memory. This routine is never invoked, because COHERENT does not support loadable device drivers.

Poll Routine

Field **c_poll** points to a function that can be accessed by commands or functions that poll the device. The driver's polling function is always called with three arguments. The first is an **o_dev_t** that indicates the device to be polled. The second is an integer whose bits flag which polling tasks are to be performed, as follows:

POLLIN	Input data is available
POLLPRI	Priority message is available
POLLOUT	Output can be sent
POLLERR	A fatal error has occurred
POLLHUP	A hangup condition exists
POLLNVAL	**fd** does not access an open stream

These are defined in the header file **<sys/poll.h>**. The third is an integer that gives the number of millseconds by which the response should be delayed. Note that the COHERENT clock timer runs at 100 Hz rather than the approximately 18 Hz clock used by MS-DOS.

The kernel functions **pollopen()** and **pollwake()**, respectively, initiate and terminate a polling event.

Example

The following shell script displays the values of the **CON** structure in a driver that uses the internal-kernel interface.

```
# drvldump - show drvl entry points in a kernel binary
# Usage: drvldump [-c] [kernel-name]

SHOW_CON_ADDRS=n

# a function - con_show name offset
con_show () {
      NAME=$1
      OFFSET=$2
      ADDR=‘/conf/patch -p $KER $DEVCON+$OFFSET | sed -e "s/^.*0x/0x/"‘
      if [ "$ADDR" != 0x00000000 ]; then
            echo "$ADDR $NAME $DEVCON"
      fi
}
```

LEXICON

```
for ARG; do
case $ARG in
-c)
      SHOW_CON_ADDRS=y
      shift
      ;;
--help|-h)
      echo "Usage: drvldump [-c] [kernel-name]"
      exit 1
      ;;
esac
done

KER=${1-/coherent}

if [ ! -f $KER ]; then
      echo "Can't open $KER"
      exit 1
fi

if [ "$SHOW_CON_ADDRS" = y ] ; then
      echo "Starting addresses CON structs in drvl table:\n"
      for D in `from 0 to 31`; do
            DO=`expr $D \* 8`
            CON_ADDR=`/conf/patch -p $KER drvl+$DO | sed -e "s/^.*0x/0x/"`

            if [ "$CON_ADDR" != 0x00000000 ]; then
                  echo "Major number $D: $CON_ADDR"
            fi
      done
      echo
fi

echo "Device driver entry points found in CON structs.\n"

for DEVCON in `nm -n $KER | grep "con\\$" | sed -e "s/^.* //"`; do
      con_show open 8
      con_show close 12
      con_show block 16
      con_show read 20
      con_show write 24
      con_show ioctl 28
      con_show powerfail 32
      con_show timeout 36
      con_show load 40
      con_show unload 44
      con_show poll 48
done
```

See Also

entry-point routines, internal data structures, internal kernel routines, set_user_error()

copyb() — DDI/DKI Kernel Routine

Duplicate a message block

#include <sys/stream.h>
mblk_t *copyb(*bufptr***)**
mblk_t ****bufptr*;**

copyb() allocates a message block and copies into it the contents of the block to which *bufptr* points. **copyb()** ensures that the newly allocated block is at least as large as that to which *bufptr* points.

If all goes well, **copyb()** returns a pointer to the newly allocated message block; otherwise, it returns NULL.

See Also

allocb(), copymsg(), DDI/DKI kernel routines, msgb

Notes

copyb() has base or interrupt level. It does not sleep.

A function can hold driver-defined basic locks, read/write locks, and sleep locks across a call to this function.

copyin() — DDI/DKI Kernel Routine

Copy data into a driver buffer from a user buffer
#include <sys/types.h>
int copyin(*user***,** *driver***,** *bytes***)**
caddr_t *user***,** *driver***; size_t** *bytes***;**

copyin() copies *bytes* of data from address *user*, which lies within the user's data space, to address *driver*, which lies within the kernel's data space. *driver* must point to at least *bytes* of allocated memory. If the memory to which *driver* points does not lie entirely within the kernel's space, the system may panic.

If all goes well, **copyin()** returns zero; otherwise, it returns -1.

See Also

copyout(), DDI/DKI kernel routines, uiomove(), ureadc(), uwritec()

Notes

copyin() has base level. It can sleep.

A driver cannot hold a driver-defined basic lock or a read/write lock across a call to this function; it can, however, hold a driver-defined sleep lock. When it holds a sleep lock, a driver must be careful to avoid creating a deadlock.

As data are being transferred, resolution of a page fault may result in another I/O to the same device.

This function is equivalent to the internal-kernel routine **ukcopy()**.

copymsg() — DDI/DKI Kernel Routine

Duplicate a message
#include <sys/stream.h>
mblk_t *copymsg(*msgptr***)**
mblk_t ****msgptr*;**

copymsg() duplicates the message to which *msgptr* points. It allocates enough message blocks to hold the message and calls **copyb()** to copy the message.

If all goes well, **copymsg()** returns a pointer to the duplicate message. Otherwise, it returns NULL.

See Also

allocb(), copyb(), DDI/DKI kernel routines, msgb

Notes

copymsg() has base or interrupt level. It does not sleep.

A driver can hold driver-defined basic locks, read/write locks, and sleep locks across a call to this function.

copyout() — DDI/DKI Kernel Routine

Copy data into a user buffer from a driver buffer
#include <sys/types.h>
int copyout(*driver***,** *user***,** *bytes***)**
caddr_t *driver***,** *user***; size_t** *bytes***;**

copyout() copies *bytes* of data from address *driver*, which lies within the kernel's data space, to address *user*, which lies within theuser's data space. *user* must point to at least *bytes* of allocated memory. If the memory to which *driver* points does not lie entirely within the kernel's space, the system may panic.

If all goes well, **copyout()** returns zero; otherwise, it returns -1.

See Also

copyin(), DDI/DKI kernel routines, uiomove(), ureadc(), uwritec()

Notes

copyout() has base level. It can sleep.

A driver cannot hold a driver-defined basic lock or a read/write lock across a call to this function; it can, however, hold a driver-defined sleep lock. When it holds a sleep lock, a driver must be careful to avoid creating a deadlock.

This function is equivalent to the internal-kernel routine **kucopy()**.

copyreq — STREAMS Data Structure

Structure for a request for a STREAMS transparent ioctl copy
#include <sys/stream.h>

The structure **copyreq** holds information used to process transparent **ioctl**s. A driver creates this structure by overlaying a STREAMS message of type **M_IOCTL** or **M_IOCDATA** and converting it into an **M_COPYIN** or **M_COPYOUT** message; thus, the driver lays **copyreq** upon the structures **ioctl** or **copyresp**. The stream head guarantees that the message is large enough to contain all of the structures.

The following fields within **copyreq** are available to drivers:

int cq_cmd This gives the **ioctl** command, as copied from field **ioc_cmd** in structure **iocblk**.

cred_t *cq_cr This points to the user's credentials. It is copied from field **ioc_cr** within structure **iocblk**.

uint_t cq_id This gives the **ioctl**'s identifier, as copied from the field **ioc_id** within structure **iocblk**.

caddr_t cq_addr If the message is of type **M_COPYIN**, **cq_addr** contains the address within user space from which the data are copied; if the message is **M_COPYOUT**, it contains the address in user space to which the data are copied.

uint_t cq_size The number of bytes to copy, regardless of the direction of copying.

int cq_flag This field is reserved for future use. The driver should initialize it to zero.

mblk_t *cq_private This field is reserved for the driver, which can use it to hold the information it needs to process the **ioctl**. The contents of this field are copied into field **cp_private** of the resulting **M_IOCDATA** message.

See Also

datab, DDI/DKI data structures, iocblk, msgb

Notes

When a message of type **M_COPYIN** or **M_COPYOUT** is freed, STREAMS does *not* free any message to which **cq_private** refers; the STREAMS module or driver must free these messages.

copyresp — STREAMS Data Structure

Structure for responding to STREAMS transparent ioctl copy
#include <sys/stream.h>

Structure **copyresp** contains the information needed to continue processing transparent **ioctl**s. No driver creates this structure: it is contained within any **M_IOCDATA** messages that the stream head generates.

The following fields within **copyresp** are available to drivers:

int cp_cmd The **ioctl** command, copied from field **cq_cmd** of the structure **copyreq**.

cred_t *cp_cr The user's credentials. It is copied from field **cq_cr** of structure **copyreq**.

uint_t cp_id The **ioctl** identifier, which uniquely identifies this **ioctl** within the stream. It is copied from field **cq_id** of the structure **copyreq**.

caddr_t cp_rval The value returned by the last copy request. Zero indicates that the request succeeded; a non-zero value indicates that the copy failed, with the nature of the failure indicated by the value. When this field indicates failure, the driver or module should abort processing the **ioctl** and free the message.

mblk_t *cp_private The contents of this field are copied from field **cq_private** of the structure **copyreq**. The driver defines what goes into this field.

See Also

datab, DDI/DKI data structures, mesgb, copyreq, iocblk

Notes

If a driver reuses an **M_IOCDATA** message, it must clear all unused fields.

When a STREAMS function frees an **M_IOCDATA** message, it does *not* free the memory to which **cp_private** points. Your driver must free this memory.

datab — STREAMS Data Structure

Structure for a STREAMS data block
#include <sys/types.h>
#include <sys/stream.h>

The data-block structure **datab** holds the data of a STREAMS message. The message-block structure **msgb** includes a field that points to it. The kernel allocates a **datab** when it creates structures of type **mblk_t**.

The following fields within **datab** are available to a driver:

uchar_t *db_base The beginning of the data buffer. Do not alter this field.

uchar_t *db_lim This field points to one byte past the end of the data buffer. Do not alter this field.

uchar_t db_ref The number of message blocks that share the data buffer. Only one message block can point to any given data block at any given time; therefore, if the value of this field is greater than one, do not change the contents of the data buffer. Do not alter this field.

uchar_t db_type The type of message within the data buffer. A driver can change this field — but only if field **db_ref** equals one, as described above.

See Also

DDI/DKI data structures, free_rtn, messages, msgb

Notes

The structure **datab** is defined as type **dblk_t**.

datamsg() — DDI/DKI Kernel Routine

Test whether a message type is a data type
#include <sys/types.h>
#include <sys/stream.h>
#include <sys/ddi.h>
int datamsg(*type***)**
uchar_t *type***;**

datamsg() tests whether *type* is a data type, i.e., any of the types **M_DATA**, **M_DELAY**, **M_PROTO**, or **M_PCPROTO**. **datamsg()** returns one if *type* is a data message, and zero if it is not. Use this function to examine field **db_type** within a message's **datab** structure, to see whether this is a data message. To access the type of the message to which *msgptr* points, use the construction *msgptr***->b_datap->db_type**.

See Also

allocb(), datab, DDI/DKI kernel routines, messages, msgb

Notes

datamsg() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

ddi_base_data() — DDI/DKI Kernel Routine

Get base data on per-process basis
#include <kernel/ddi_base.h>
dbdata_t *ddi_base_data();

ddi_base_data() returns a pointer to a table of information that the DDI/DKI must associate with a process but does not need to access outside the context of that process. The value it returns should be considered constant.

See Also

ddi_cpu_data, ddi_global_data, ddi_proc_data, DDK/DKI kernel routines

Notes

ddi_base_data() has base level only. It does not sleep.

Please note that although **ddi_base_data()** is used with COHERENT's implementation of the DDI/DKI interface, it is not part of the published description of the DDI/DKI. Code that uses it will not be portable to other operating systems.

ddi_cpu_data() — DDI/DKI Kernel Routine

Get global data on per-processor basis
#include <kernel/ddi_cpu.h>
dcdata_t *ddi_cpu_data();

ddi_cpu_data() returns a pointer to a table of information that can be considered DDI/DKI static data for a given processor. The value returned should be considered constant.

ddi_cpu_data() returns the base address of the DDI/DKI's data-table entry for the current CPU.

See Also

ddi_base_data, ddi_global_data, ddi_proc_data, DDI/DKI kernel routines

Notes

ddi_cpu_data() has base or interrupt level. It does not sleep.

Please note that although **ddi_cpu_data()** is used with COHERENT's implementation of the DDI/DKI interface, it is not part of the published description of the DDI/DKI. Code that uses it will not be portable to other operating systems.

ddi_global_data() — DDI/DKI Kernel Routine

Get global data
#include <kernel/ddi_glob.h>
dgdata_t *ddi_global_data();

ddi_global_data() returns a base pointer to a table of information that represents the global state of the DDI/DKI subsystem, with the possible exception of the STREAMS global state. The value returned should be considered constant.

See Also

ddi_base_data, ddi_cpu_data, ddi_proc_data, DDI/DKI kernel routines

Notes

ddi_global_data() has base or interrupt level. It does not sleep.

Please note that although **ddi_global_data()** is used with COHERENT's implementation of the DDI/DKI interface, it is not part of the published description of the DDI/DKI. Code that uses it will not be portable to other operating systems.

ddi_proc_data() — DDI/DKI Kernel Routine

Get global data on a per-process basis
#include <kernel/ddi_proc.h>
dpdata_t *ddi_proc_data();

ddi_proc_data() returns a base pointer to a table of information that the DDI/DKI needs to associate with a process, but may need to access outside the process context. The value it returns should be considered constant.

See Also

ddi_base_data(), ddi_cpu_data(), ddi_global_data(), DDI/DKI kernel routines

Notes

ddi_proc_data() has base or interrupt level. It does not sleep.

Please note that although **ddi_proc_data()** is used with COHERENT's implementation of the DDI/DKI interface, it is not part of the published description of the DDI/DKI. Code that uses it will not be portable to other operating systems.

DDI/DKI data structures — Overview

The COHERENT implementation of DDI/DKI and STREAMS uses the following data structures:

copyreq . . . . . Structure for a request for a STREAMS transparent ioctl copy
copyresp . . . . . Structure for responding to STREAMS transparent ioctl copy
datab . . . . . Structure for a STREAMS data-block structure
free_rtn . . . . . The free-message routine
iocblk . . . . . ioctl structure
iovec . . . . . Data-storage structure for scatter/gather I/O
linkblk . . . . . Multiplexor link structure
lkinfo . . . . . Information about a lock
module_info . . . . . Information about a STREAMS driver or module
msgb . . . . . Message block structure
pollhead . . . . . Structure for a STREAMS poll head
qinit . . . . . Initialization for queues
queue . . . . . Queue structure
streamtab . . . . . Driver/module declaration
stroptions . . . . . Structure for stream-head options
uio . . . . . Organize scatter/gather I/O requests

Note that each structure contains many more fields than those that are documented in this manual. However, no driver or module can manipulate any field in any structure other than those documented in this manual. Doing so risks corrupting kernel memory.

Note, too, that each structure has its own "legal" origin, as follows:

copyreq . . . . . Overlay an **M_IOCTL** or **M_IOCDATA** STREAMS message
copyresp . . . . . Not created by the driver
datab . . . . . Allocated when **mblk_t**s are allocated by kernel
free_rtn . . . . . Allocated by driver
iocblk . . . . . Not created by the driver
iovec . . . . . May be driver-allocated at present
linkblk . . . . . Not created by the driver
lkinfo . . . . . Allocated by driver
module_info . . . . . Statically allocated in driver source
msgb . . . . . Allocated only by **allocb()** or **esballoc()**
qinit . . . . . Statically allocated in driver source
queue . . . . . Not created by the driver
streamtab . . . . . Statically allocated in driver source
stroptions . . . . . Built by driver in STREAMS message memory
uio . . . . . May be driver-allocated at present

The kernel cannot police a driver to ensure that it does not create its own versions of these structures on the sly; however, doing so may corrupt kernel memory.

See Also

device driver, STREAMS

DDI/DKI kernel routines — Overview

The COHERENT implementation of the DDI/DKI includes the following routines: Note that the routines marked with an asterisk '*' are used with COHERENT's implementation of the DDI/DKI interface, but are not part of the published description of the DDI/DKI. Code that uses them will not be portable to other operating systems.

adjmsg() . . . . . Clip a message
allocb() . . . . . Allocate a message block
ASSERT() . . . . . Debug an expression
backq() . . . . . Get a pointer to the preceding queue
bcanput() . . . . . Test for flow control in a priority band
bcanputnext() . . . . . Test for flow control in a priority band
bcopy() . . . . . Copy data between locations within the kernel
bufcall() . . . . . Call a function when a buffer becomes available
bzero() . . . . . Initialize a block of memory to zero
canput() . . . . . Test for room in a queue

LEXICON

canputnext() . . . . . Test for room in a queue
cmn_err() . . . . . Display an error message or panic the system
copyb() . . . . . Copy a message block
copyin() . . . . . Copy data from a user's buffer to a driver's buffer
copymsg() . . . . . Copy a message
copyout() . . . . . Copy data from a driver's buffer to a user's buffer
datamsg() . . . . . Test whether a message is a data message
ddi_base_data()* . . . . Get DDI/DKI base data on per-process basis
ddi_cpu_data()* . . . . . Get DDI/DKI global data on per-processor basis
ddi_global_data()* . . . Get DDI/DKI global data
ddi_proc_data()* . . . . Get DDI/DKI global data on a per-process basis
drv_getparm() . . . . . Retrieve information about the kernel state
drv_hztousec() . . . . . Convert clock ticks to microseconds
drv_priv() . . . . . Determine whether credentials are privileged
drv_setparm() . . . . . Set kernel state information
drv_usectohz() . . . . . Convert microseconds to clock ticks
dupb() . . . . . Duplicate a message block
dupmsg() . . . . . Duplicate a message
enableok() . . . . . Enable a queue to be serviced
esballoc() . . . . . Allocate a message block with an externally supplied buffer
esbbcall() . . . . . Call a function when an externally-supplied buffer can be allocated
etoimajor() . . . . . Convert external to internal major device number
flushband() . . . . . Flush messages in a priority band
flushq() . . . . . Flush messages on a queue
freeb() . . . . . Free a message block
freemsg() . . . . . Free a message
freerbuf() . . . . . Free a buffer header used for raw I/O
freezestr() . . . . . Freeze a stream
getemajor() . . . . . Get external major-device number
geteminor() . . . . . Get external minor-device number
getmajor() . . . . . Get internal major-device number
getminor() . . . . . Get internal minor-device number
getq() . . . . . Get the next message from a queue
getrbuf() . . . . . Allocate a buffer header for raw I/O
inb() . . . . . Read a byte from an eight-bit I/O port
inl() . . . . . Read a 32-bit long word from a 32-bit I/O port
insq() . . . . . Insert a message into a queue
inw() . . . . . Read a 16-bit short word from a 16-bit I/O port
itimeout() . . . . . Execute a function after a given length of time
itoemajor() . . . . . Convert internal to external major number
kmem_alloc() . . . . . Allocate space from kernel free memory
kmem_free() . . . . . Free previously allocated kernel memory
kmem_zalloc() . . . . . Allocate space from kernel free memory
linkb() . . . . . Concatenate two message blocks
LOCK() . . . . . Acquire a basic lock
LOCK_ALLOC() . . . . . Allocate a basic lock
LOCK_DEALLOC() . . . Deallocate a basic lock
makedevice() . . . . . Make device number from major and minor numbers
msgdsize() . . . . . Return number of bytes of data in a message
msgpullup() . . . . . Concatenate bytes in a message
noenable() . . . . . Prevent a queue from being scheduled
OTHERQ() . . . . . Get pointer to queue's partner queue
outb() . . . . . Write a byte to an eight-bit I/O port
outl() . . . . . Write a long integer to an 32-bit I/O port
outw() . . . . . Write a word to an 16-bit I/O port
pcmsg() . . . . . Test whether a message is a priority-control message
phalloc() . . . . . Allocate and initialize a pollhead structure
phfree() . . . . . Free a pollhead structure
physiock() . . . . . Request and validate raw I/O
pollwakeup() . . . . . Inform polling process that an event has occurred
proc_ref() . . . . . Obtain a reference to a process for signalling
proc_signal() . . . . . Send a signal to a process

pullupmsg() . . . . . Concatenate bytes in a message
put(). . . . . Call a put procedure
putbq() . . . . . Place a message at the head of a queue
putctl() . . . . . Send a control message to a queue
putctl1() . . . . . Send a control message and a parameter to a queue
putnext(). . . . . Send a message to the next queue
putnextctl() . . . . . Send a control message to a queue
putnextctl1() . . . . . Send a control message and a parameter to a queue
putq() . . . . . Put a message onto a queue
qenable() . . . . . Schedule a queue's service routine for running
qprocsoff() . . . . . Disable put and service routines
qprocson() . . . . . Enable put and service routines
qreply(). . . . . Send a message in the opposite direction on a stream
qsize() . . . . . Find the number of messages on a queue
RD() . . . . . Get a pointer to a read queue
repinsb() . . . . . Read bytes from an I/O port to a buffer
repinsd() . . . . . Read 32-bit words from an I/O port to a buffer
repinsw(). . . . . Read 16-bit words from an I/O port to a buffer
repoutsb() . . . . . Read bytes from a buffer to an I/O port
repoutsd() . . . . . Read 32-bit words from a buffer to an I/O port
repoutsw() . . . . . Read 16-bit words from a buffer to an I/O port
rmvb() . . . . . Remove a message block from a message
rmvq() . . . . . Remove a message from a queue
RW_ALLOC(). . . . . Allocate and initialize a read/write lock
RW_DEALLOC(). . . . . Deallocate an instance of a read/write lock
RW_RDLOCK() . . . . . Acquire a read/write lock in read mode
RW_TRYRDLOCK(). . . Try to acquire a read/write lock in read mode
RW_TRYWRLOCK() . . Try to acquire a read/write lock in write mode
RW_UNLOCK() . . . . . Release a read/write lock
RW_WRLOCK() . . . . . Acquire a read/write lock in write mode
SAMESTR() . . . . . Test if next queue is same type
set_user_error(). . . . . Set an error code in the user space
SLEEP_ALLOC(). . . . . Allocate and initialize a sleep lock
SLEEP_DEALLOC(). . . Deallocate a sleep lock
SLEEP_LOCK() . . . . . Acquire a sleep lock
SLEEP_LOCK_SIG() . . Acquire a sleep lock
SLEEP_LOCKAVAIL() . Query whether a sleep lock is available
SLEEP_LOCKOWNED() Query whether a sleep lock is held by the caller
SLEEP_TRYLOCK(). . . Try to acquire a sleep lock
SLEEP_UNLOCK() . . . Release a sleep lock
splbase() . . . . . Block no interrupts
spldisk() . . . . . Block disk-device interrupts
splhi(). . . . . Block STREAMS interrupts
splstr() . . . . . Block STREAMS interrupts
spltimeout(). . . . . Block timeout interrupts
splx() . . . . . Reset an interrupt-priority level
strlog() . . . . . Submit messages to the log driver
strqget() . . . . . Get information about a queue
strqset() . . . . . Change information about a queue or band of a queue
sv_alloc(). . . . . Allocate and initialize a synchronization variable
SV_BROADCAST() . . . Wake up all processes sleeping on a synchronization variable
SV_DEALLOC() . . . . . Deallocate an instance of a synchronization variable
SV_SIGNAL(). . . . . Wake up one process sleeping on a synchronization variable
SV_WAIT() . . . . . Sleep on a synchronization variable
SV_WAIT_SIG() . . . . . Sleep on a synchronization variable
testb() . . . . . Check for an available buffer
TRYLOCK() . . . . . Try to acquire a basic lock
uiomove() . . . . . Copy data using **uio** structure
unbufcall(). . . . . Cancel a pending request to **bufcall()**
unfreezestr() . . . . . Unfreeze a stream
unlinkb(). . . . . Remove a message block from the head of a message
UNLOCK() . . . . . Release a basic lock

untimeout . . . . . Cancel execution of a previously scheduled function
ureadc() . . . . . Copy a character to space described by a **uio** structure
uwritec() . . . . . Copy a character from space described by a **uio** structure
WR() . . . . . Get a pointer to the write queue

See Also

device driver, internal kernel routines, STREAMS

defend() — Internal Kernel Routine

Execute deferred functions
void defend()

defend() tells the kernel to execute all functions that are on its deferred list. This function is never invoked by an interrupt handler.

See Also

internal kernel routines

defer() — Internal Kernel Routine

Defer function execution
void defer(*func***,** *arg***)**
void (**func***)(); char ****arg***;**

defer() defers execution of the function to which *func* points. *arg* is an argument passed to *func*. Execution of *func* remains deferred until the next context switch, transition from kernel to user mode, or invocation of the function **defend()**.

Deferred functions never call **sleep()** or access the **u** area, because the kernel can switch **u** areas as part of context switching. Up to 127 functions can be deferred at any one time. Exceeding this limit may lose all deferred functions.

defer() normally is used to minimize interrupt latency by deferring operations from interrupt level (where lower priority interrupts are disabled) to background level (where all interrupts normally are enabled). It is also used to help eliminate critical race conditions between task- and interrupt-related operations because deferred functions execute synchronously with each other, with timed functions, and with system calls.

See Also

internal kernel routines

device driver — Introduction

A *device driver* is a program that controls the action of one of the physical devices attached to your computer system. The following table lists the device drivers included with the COHERENT system. The first field gives the device's major device number; the second gives its name; and the third describes it. When a major device number has no driver associated with it, that device is available for a driver yet to be written. Note that the

0:	**null**	The "bit bucket"
0:	**mem**	Interface to memory and null device
0:	**kmem**	Device to manage kernel memory
0:	**kmemhi**	
0:	**clock**	System clock
0:	**cmos**	System CMOS
0:	**ps**	Processes currently being executed
0:	**idle**	System idle time
1:	**ct**	Controlling terminal device (**/dev/tty**)
2:	**console**	Video module for console (**/dev/console**)
2:	**nkb**	The "new" keyboard driver — loadable keyboard tables (**/dev/console**)
2.	**kb**	The "old" keyboard driver (**/dev/console**)
2.	**mm**	The video driver
3:	**lp**	Parallel line printer
4:	**fd**	Floppy-disk drive
4:	**fdc**	765 diskette and floppy-tape controller
4:	**ft**	Floppy-tape drive

5:	**asy**	Serial driver
6:		
7:		
8:	**rm**	Dual RAM disk
9:	**pty**	Pseudoterminals
10:		
11:	**at**	AT hard disk
12:		
13:	**hai**	Host adapter-independent SCSI driver
13:	**aha**	Older driver for Adaptec SCSI hard disks
13:	**ss**	Older driver for Seagate SCSI hard disks
14:		
15:		
16:		
17:		
18:		
19:		
20:		
21:		
22:		
23:		
24:		
25:		
26:		
27:		
28:		
29:		
30:		
31:		

Please note that the devices with major number 0 are not portable, and non-DDI/DKI. Also note that in future releases of COHERENT, the **hai** driver will be divided into several optional SCSI host-bus adapters (HBAs) and target devices.

It is not unusual for one major number to admit several driver service modules. Instances of this include the following major numbers:

0 This number is for a number of system-dependent drivers.

2 This number supports the console, both its keyboard modules and its video modules.

4 This describes varieties of floppy-disk and floppy-tape controllers and drives.

13 This describes a number of SCSI host modules, HBA modules, and target modules.

Major and Minor Numbers

COHERENT uses a system of *major* and *minor* device numbers to manage devices and drivers. In theory, COHERENT assigns a unique major number to each type of device, and a unique minor number to each instance of that type. In practice, however, a major number describes a device driver (rather than a device *per se*). The individual devices serviced by that driver are identified by a minor number. Sometimes, certain parts of the minor number specify configuration. For example, bits 0 through 6 of the minor number for COHERENT RAM disks indicate the size of the allocated device.

In future releases of COHERENT, major numbers will not be static, as they are in the above table. Rather, they will be assigned by the **config** script when you install COHERENT onto your system. This scheme will allow more flexible arrangements of drivers, and will also allow COHERENT to support more than 32 drivers at once. If you write code to work with device drivers, you should *not* make any assumptions about a given driver's major number.

Optional Kernel Components

The kernel also contains the following optional components:

em87	Emulate hardware floating-point routines
msg	Perform System V-style message operations
sem	Perform System V-style semaphore operations
shm	Perform System V-style shared-memory operations
streams	Perform STREAMS operations

These components resemble device drivers, in that that they perform discrete work and can be linked into or excluded from the kernel, as shown below. However, they do not perform I/O with a device, and so are not true drivers. For details on these modules, see their entries in the Lexicon.

Serial Ports

COHERENT manages serial ports with one driver, **asy**. It has major number 5, but it supports all four COM ports, and a variety of generic multi-port cards. The configuration of ports that **asy** supports is set when you install COHERENT; however, you can reconfigure **asy** should you wish to add more hardware to your system. See its Lexicon article for details.

Configuring Drivers and the Kernel

Beginning with release 4.2, COHERENT lets you tune kernel and driver variables, enable or disable drivers, and easily build a new bootable kernel that incorporates your changes.

The command **idenable** lets you enable or disable a driver within the kernel. The command **idtune** lets you set a user-modifiable variable within the kernel. Finally, the command **idmkcoh** generates a new kernel that incorporates all changes you have made with the other three commands. Changes are entered with **idenable** and **idtune** do not take effect until you invoke **idmkcoh** to generate a new kernel, and boot the new kernel. Scripts **/etc/conf/*/mkdev** simpify the choices of **idenable** and **idtune** during installation and reconfiguration: they invoke **idtune** and **idenable** in response to your choice of configuration options.

Adding a New Device Driver

The commands described above make it easy for you to add a new device driver to your COHERENT kernel.

The following walks you through the processing of adding a new driver. We will add the driver **foo**, which enables the popular "widget" device.

1. To begin, log in as the superuser **root**.

2. The next step is to create a directory to hold the driver's sources and object. Every driver must have its own directory under directory **/etc/conf**; and the sources must be held in directory **src** in that driver's directory. In this case, create directory **/etc/conf/foo**; then create directory **/et/conf/foo/src**.

3. Copy the sources for the driver into its source directory; in this case, copy them into **/etc/conf/foo/src**.

4. Create a **Makefile** in your driver's source directory, e.g., **/etc/conf/foo/src/makefile**. The easiest way to see what is required is to review several of the driver **Makefile**s shipped in the COHERENT driver kit. You can perform a test compilation of your driver by running **make** with the driver's **src** directory as the current directory. This should create one object file that has the suffix **.o**. Copy this file in the driver's home directory, and name it **Driver.o**. In this case, the object for the driver should be in file **/etc/conf/foo/Driver.o**. In some rare cases, a driver compile into more than one object. You should store all of these objects into one archive; name the archive **Driver.a** and store it in the driver's home directory. The COHERENT commands that build the new kernel know how to handle archives correctly. The main idea is that files **Space.c** (if one exists) and **Driver.o** or **Driver.a** be placed in the driver directory, i.e., the parent of the **src** directory.

5. Add an entry to file **/etc/conf/sdevice** for this driver. **sdevice**, as described above, names the drivers to be included in the kernel. The entries for practically every entry are identical; you need to note only that the second column marks whether to include the driver in the kernel. In this case, the entry for the driver **foo** should read as follows:

```
foo   Y     0     0     0     0     0x0   0x0   0x00x0
```

   For details on what each column means, read the comments in file **/etc/conf/sdevice**.

6. Add an entry to file **/etc/conf/mdevice** for the new driver. This file is a little more complex than **sdevice**; in particular, it distinguishes between STREAMS-style drivers and "old-style" COHERENT drivers. In most cases, you can simply copy an entry for an existing driver of the same type, and modify it slightly. In this case, the entry for **foo** should read as follows:

```
# full      func  misc  code  block char  minor minordma cpu
# name      flags flags prefix      major major minmax chan id
foo   -     CGo   foo   15    15    0     255   -1-1
```

In almost every case, the full name and the code prefix are identical. The code prefix also names the directory that holds the driver's object. Function flags are always always a hyphen, and miscellaneous flags almost always CGo. The block-major and character-major numbers again are almost always identical. The major number is usually assigned by the creator of the device driver. In future releases of the kernel, these will be assigned dynamically by the kernel itself; poorly written drivers that depend upon the driver having a magic major-device number will no longer work. Finally, the last four columns for non-STREAMS drivers are almost always 0, 255, -1, and -1, respectively. See the comments in file **/etc/conf/mdevice**.

7. If the driver has tunable variables, these should be set in the file **Space.c**, which should be stored in the driver's home directory. As it happens, **foo** does not need a **Space.c** file. For examples of such files, look in the various sub-directories of **/etc/conf**.

8. Type the command **/etc/conf/bin/idmkcoh** to build a new kernel. If necessary, move the new kernel into the root directory; you cannot boot it until it is in the root directory.

9. Save the old kernel and link the newly build kernel to **/autoboot**. You want save the old kernel, just in case the new one doesn't work. For directions on how to boot a kernel other than **/autoboot**, see the Lexicon entry for **booting**.

10. Back up your files! With a new driver in your kernel, it's best to play it safe.

11. Reboot your system to invoke the new kernel. If all goes well, you will now be enjoying the services of the new device driver.

If you wish to boot your test kernel from a floppy disk instead of from your development file system, execute script **/etc/conf/bin/Floppy** after step 8, above.

For scripts on how to add or remove individual drivers from your kernel, see the article of the driver in question.

Types of Device-Driver Interface

Beginning with release 4.2, COHERENT uses two types of device-driver interface:

Internal Kernel Interface
: This type of driver uses the routines internal to the COHERENT kernel. Examples of this interface include the **at** driver for the AT hard disk, and the **hai** driver for SCSI devices. See their sources in, respectively, directories **/etc/conf/at/src** and **/etc/conf/hai/src**.

DDI/DKI Interface
: The device-driver interface/driver-kernel interface (DDI/DKI) is a programmers' interface for UNIX System V release 4.

When you begin to write a driver for COHERENT, you should pick carefully between these two strategies:

- The internal-kernel interface is proven and works; however, note that this is a world apart from UNIX, and a driver written in this interface is not readily portable to any other operating system.
- The DDI/DKI interface does ensure portability with UNIX System V release 4; however, the COHERENT implementation lacks some features present in true UNIX. These features mainly center around features that are lacking from the COHERENT kernel itself, which means that the COHERENT must await a rewritten memory manager and file system before it will have a fully compliant DDI/DKI.

Sets of routines from the DDI/DKI can be combined with those from the internal-kernel interface. In some cases, the DDI/DKI offers the better method of performing a given task; in others, the internal-kernel interface offers the better (or, more likely, the only) method to perform a task. If you are importing a driver from UNIX System V release 4, then you should use the DDI/DKI routines primarily. Likewise, you should use them primarily if you are writing a driver that you wish to export to UNIX. Note, too, that as COHERENT evolves toward the standard of System V release 4, the DDI/DKI interface will grow in importance.

The sources included with release 4.2.05 of the device-driver kit are in the internal-kernel format rather than DDI/DKI. It was simply not practical to recast these drivers in the DDI/DKI mold at the present time; however, we are supplying information regarding DDI/DKI interfaces to inform developers of the future direction of COHERENT. In the development of new drivers, DDI/DKI facilities should be used wherever possible for greatest compatibility, e.g., with future releases of COHERENT.

LEXICON

To summarize, all else being equal, the DDI/DKI is preferred over the internal-kernel interface. The Lexicon entries themselves will alert you of the alternate ways of performing a given task, to help you decide which to use.

The best way to judge which interface you should use is to read the sources included with the COHERENT Device Driver Kit:

echo (**/etc/conf/echo/src**)
This driver gives a small example of a STREAMS driver.

at (**/etc/conf/at/src**)
This driver manipulates the AT hard disk. It gives the best demonstration of writing a block driver, with regard to compatibility with UNIX System V release 4.

hai devices (**/etc/conf/hai/src**)
This driver manipulates SCSI devices. It demonstrates how to use first-party DMA.

ss (**/etc/conf/ss/src**)
This driver manipulates the Seagate SCSI disk. It demonstrates how to use memory-mapped I/O.

fd (**/etc/conf/fd/src**)
fdc These drivers manipulate the floppy disks. It demonstrates how to perform DMA via the Intel controllers.

asy (**/etc/conf/asy/src**)
This driver manipulates serial ports COM1 through COM4 and multiport asynchronous serial boards using 8250- through 16550-type UARTs. It demonstrates how to write a non-STREAMS driver for a character device.

Beyond this, you must use your best judgement as you gain experience in working with COHERENT.

Coding Requirements

The following summarizes the coding requirements for device drivers that use the internal-kernel or DDI/DKI interfaces.

To begin, the coding requirements for the internal-kernel interface:

1. Put 'C' in the miscellaneous flags in the file **/etc/conf/mdevice**.
2. Do not define symbol **_DDI_DKI** in the driver's source file.
3. Place driver's entry points in a **CON** structure. For information on this structure, see the entries for **entry points** and **CON** in this Lexicon.
4. There is distinction between internal and external major- and minor-device numbers. A device number (**dev_t**) is a 16-bit object. Use internal-kernel routine **minor()**, *q.v.*, to obtain the minor-device number.
5. Either include **<sys/coherent.h>**, or explicitly define symbol **_KERNEL** to be one, before any other #include directives in the driver source.

The coding requirements for the DDI/DKI interface are as follows:

1. Do not put a 'C' into the miscellaneous-flags field in file **/etc/conf/mdevice** (*q.v.*).
2. Define symbol **_DDI_DKI** in the driver's source file, before any **#include** directives.
3. Put an entry into the function-flags field in **/etc/conf/mdevice** for each of the driver's entry points; do not put them into a **CON** structure.
4. A device number (**dev_t**) is a 32-bit object. There is some discussion in the literature of internal *vs.* external numbering for device numbers and for the major and minor parts of the device number as well. As of COHERENT 4.2.05, only external numbers are of interest to the writer of device drivers. Thus, when a **dev_t** is passed to a driver's entry point, it is an external device number. When major numbers are entered into file **/etc/conf/mdevice**, they are external major numbers. Unit numbers and device features are decoded from the external minor number, which is obtained from the external device number by calling the DDI/DKI routine **geteminor()**.
5. Define symbol **_KERNEL** to be one in the driver source, before any **#include** directives.

Using This Lexicon

This manual is organized into the Mark Williams Lexicon format. The following overview articles introduces the categories of articles within this manual:

DDI/DKI data structures
This article introduces the articles that describe the types of structures from which a stream is constructed.

DDI/DKI kernel routines
This article introduces the articles that discuss the DDI/DKI routines that are built into the kernel.

entry-point routines
This article introduces the articles that discuss the entry points into a driver.

internal data structures
This article introduces the data structures internal to the COHERENT kernel.

internal kernel routines
This article introduces the routines built into the COHERENT kernel that can be used in a device driver.

STREAMS
This article introduces STREAMS.

technical information
This article introduces articles that give technical information, such as types of messages or of signals.

See Also

COHERENT Lexicon: **asy, at, boot, console, ct, floppy disk, ft, hard disk, kernel, Lexicon, lp, mboot, mem, null, psy, sgtty, STREAMS, tape, termio**

device numbers — Technical Information

Device numbering is evolving under COHERENT toward compability with UNIX System V release 4. For this reason, the internal-kernel and DDI/DKI interfaces differ in their treatment of device numbers.

Under both the internal-kernel and DDI/DKI interfaces, a device number, or **dev_t**, combines major and minor numbers.

The major device number is arbitrarily assigned. A driver's logic should not rely on that driver being assigned any specific major-device numbers. A driver's major-device number is set by its entry in the file **mdevice**.

The driver's programmer assigns minor-device numbers. If a device driver controls several distinct units of the same device simultaneously (e.g., multiple floppy-disk drives or multiple partitions on a hard disk), the minor-device number often indicates which unit is being accessed. If a device driver allows different options for accessing a device, such as hardware flow control for a serial device or rewind-on-close for a tape device, it is common for some part of the minor-device number to indicate the user's choice of options.

The literature discusses internal versus external device numbers. As of release 4.2.05, this does not apply to COHERENT. The internal-kernel interface does not distinguish between these entities; and it has not yet been implemented for COHERENT's version of the DDI/DKI interface.

See Also

makedevice(), getemajor(), geteminor(), getmajor(), getminor(), technical information

devmsg() — Internal Kernel Routine

Print a message from a device driver
void devmsg(*dev***,** *fmt***, ...)**
dev_t *dev***; char** ******fmt***;**

devmsg() prints a message from a device driver on the system console. *fmt* and optional additional arguments are in the same form as used by the function **printf()**, except that **devmsg()** appends a newline onto *fmt*. Output from **devmsg()** is synchronous and at high priority, so its use is limited to brief error messages.

See Also

internal kernel routines, printf()

Notes

This function does much the same work as the DDI/DKI routine **cmn_err()**.

dmago() — Internal Kernel Routine

Enable DMA transfers
void dmago(*chan***)**
int *chan***;**

dmago() enables transfers on DMA channel *chan*. A call to **dmago()** must be preceded by a call to **dmaon()**, which sets the DMA parameters.

See Also

internal kernel routines

dmain() — Internal Kernel Routine

Copy from system global memory to kernel data
dmain(*nbytes***,** *src***,** *dest***)**
long *nbytes***; paddr_t** *src***; vaddr_t** *dest***;**

dmain() copies *nbytes* from system global address *src* to kernel address *dest*.

See Also

internal kernel routines

dmaoff() — Internal Kernel Routine

Disable DMA transfers
int dmaoff(*chan***)**
int *chan***;**

dmaoff() disables transfers on the DMA channel *chan*. It returns the residual count (i.e., the number of bytes not transferred). A call to **dmaoff()** must be preceded by calls to **dmaon()** and **dmago()**.

See Also

internal kernel routines

dmaon() — Internal Kernel Routine

Prepare for DMA transfer
#include <sys/types.h>
int dmaon(*chan***,** *paddr***,** *count***,** *wflag***)**
int *chan***,** *wflag***; paddr_t** *paddr***; unsigned** *count***;**

dmaon() programs DMA channel *chan* to transfer *count* bytes to or from physical-memory address *paddr*. If *wflag* is zero, the data are read from the device and written to memory.

If all goes well, **dmaon()** returns one. It returns zero if a "straddle condition" arises — that is, if an operation would cross the boundary of a 64-kilobyte "hunk" of physical memory — because the DMA controller cannot handle this situation.

See Also

internal kernel routines

dmaout() — Internal Kernel Routine

Copy from kernel data to system global memory
dmaout(*nbytes***,** *dest***,** *src***)**
long *nbytes***; vaddr_t** *dest***,** *src***;**

dmaout() copies *nbytes* from kernel address *src* to system global address *dest*.

See Also

internal kernel routines

dmareq() — Internal Kernel Routine

Request block I/O, avoiding DMA straddles
#include <sys/buf.h>
void dmareq(*bp***, *iop*, *dev*, *req*)**
BUF **bp*; **IO** **iop*; **dev_t** *dev*; **int** *req*;

dmareq(), like **ioreq()**, queues an I/O request through the block routine of a device driver. *bp* points to the **BUF** structure for the I/O. *iop* points to an **IO** structure. *dev* is the device to access. Finally, *req* requests the type of I/O: it must be either **BREAD** or **BWRITE**.

dmareq() converts I/O requests that straddle DMA boundaries into two or three non-straddling requests. It converts block DMA straddles into two non-straddling I/O requests; it converts other DMA straddles into three non-straddling I/O requests, where the DMA-straddling block is handled through the buffer cache. Note that the driver's block routine must be able to function with the smaller I/O requests.

See Also

ioreq(), internal kernel routines

drv_getparm() — DDI/DKI Kernel Routine

Retrieve information about the kernel state
#include <sys/types.h>
#include <sys/ddi.h>
int drv_getparm(*parameter***, *address*)**
ulong_t *parameter*; **ulong_t** **address*;

drv_getparm() retrieves the value of *parameter*, and writes it into *address*. *parameter* can be one of the following:

LBOLT The number of clock ticks since the kernel was last booted. The difference between successive values of this parameter can be used to calculate the number of ticks that elapsed between calls.

Under COHERENT, each tick is one one-hundredth of a second; however, the length of a clock tick varies among implementations, and if you wish your driver to be portable to other operating systems, do not hard-code this value. You can use functions **drv_hztousec()** and **drv_usectohz()** to convert between clock ticks and microseconds.

UPROCP The address of the current process's process (**UPROC**) structure. The value written at address *value* is of type **proc_t** * The only valid use of this value is as an argument to function **vtop()**. Because this value is associated with the current process, the caller must have process context (that is, must be at base level) when it attempts to retrieve this value. Use this value only within the context of the process within which it was retrieved.

UCRED The address of the structure that describes the current user's credentials for the current process. The value written at address *value* is of type **cred_t** *. The only valid use of this value is an an argument to function **drv_priv()**. Because this value is associated with the current process, the caller must have process context (that is, must be at base level) when it attempts to retrieve this value. Use this value only within the context of the process within which it was retrieved.

TIME Read the current time, in seconds. This is the same value returned by the system call **time()**, that is, in the number of seconds that have elapsed since January 1, 1970, 00:00:00 UTC. This definition presupposes that the administrator has set the system's date and time correctly.

drv_getparm() returns zero if all went well; otherwise, it returns -1. This usually indicates that *parameter* held an invalid parameter.

See Also

DDI/DKI kernel routines, drv_hztousec(), drv_priv(), drv_usectohz(), vtop()
COHERENT Lexicon: **time()**

Notes

drv_getparm() has base level *parameter* is set to **UPROCP** or **UCRED**, or base or interrupt level when *parameter* is set to **LBOLT** or **TIME**. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

drv_getparm() does not check whether the driver has the appropriate context when the driver calls it. Use this function only when it is appropriate.

drv_hztousec() — DDI/DKI Kernel Routine

Convert clock ticks into microseconds
#include <sys/types.h>
#include <sys/ddi.h>
clock_t drv_hztousec(*ticks***)**
clock_t *ticks***;**

drv_hztousec() returns the number of microseconds equivalent to *ticks*, which is in units of clock ticks. To convert between clock ticks and microseconds, use **drv_usectohz()**.

Several functions either take arguments in ticks, or return time in ticks. The length of a tick varies among operating systems; therefore, you should not hard-code any assumption about the length of a tick into your driver.

See Also

DDI/DKI kernel routines, delay(), drv_getparm(), drv_usectohz(), dtimeout(), itimeout()

Notes

drv_hztousec has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

If a **clock_t** is too small to hold the number of microseconds equivalent to *ticks*, **drv_hztousec()** returns then the maximum value of **clock_t**. Calling **drv_getparm()** with a parameter of **LBOLT** often returns a value whose equivalent in microseconds is too large to fit into a **clock_t**. If you wish to use **drv_getparm()** and **drv_hztousec()** to time an operation, you should subtract the values returned by successive calls to **drv_getparm()** and convert the difference, instead of converting the values and then performing the subtraction.

drv_priv() — DDI/DKI Kernel Routine

Check if a user has privileged credentials
#include <sys/types.h>
#include <sys/ddi.h>
int drv_priv(*credentials***)**
cred_t **credentials***;*

drv_priv() checks whether *credentials* identifies a process that is owned by a privileged user. Use this function only when file permissions and special minor-device numbers cannot guard the driver sufficiently.

The kernel passes a pointer to a credential structure to various entry-points into the driver (i.e., **open**, **close**, **read**, and **ioctl**). You can also obtain it by calling **drv_getparm()** from base-level driver code.

If *credentials* shows that the process is owned by a privileged user, **drv_priv()** returns zero; otherwise it returns **EPERM**.

See Also

DDI/DKI kernel routines, drv_getparm()

Notes

drv_priv() has base or interrupt levels. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

drv_setparm() — DDI/DKI Kernel Routine

Set an internal kernel variable
#include <sys/types.h>
#include <sys/ddi.h>
int drv_setparm(*action***,** *value***)**
ulong_t *action***,** *value***;**

drv_setparm() uses *action* to modify an internal-kernel variable by *value*. If all goes well, it returns zero; otherwise (e.g., because the user lacks permission to modify *variable*) it returns -1.

action can be one of the following:

SYSCANC Add *value* to the count of characters read from a terminal device. Exclude special characters, such as **break** or **backspace**.

SYSMINT Add *value* to the count of modem interrupts received.

SYSOUTC Add *value* to the count of characters written to a terminal device.

SYSRAWC Add *value* to the count of characters read from a terminal device. Do not exclude special characters.

SYSRINT Add *value* to the count of interrupts generated by data to be received from a terminal device.

SYSXINT Add *value* to the count of interrupts generated by data to be transmitted to a terminal device.

drv_setparm() returns zero if all goes well; otherwise, it returns -1.

See Also

DDI/DKI kernel routines, drv_getparm()

Notes

drv_setparm() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, a read/write lock, or a sleep lock across calls to this function.

drv_usectohz() — DDI/DKI Kernel Routine

Convert microseconds to clock ticks
#include <sys/types.h>
#include <sys/ddi.h>
clock_t drv_usectohz(*microseconds***)**
clock_t *microseconds***;**

drv_usectohz() converts *microseconds* to clock ticks. It returns the smallest number of clock ticks equal to or greater than *microseconds*; in other words, it rounds up, not down. If the number of ticks is too large to fit into a **clock_t**, it returns the maximum value that will fit into a **clock_t**.

See Also

DDI/DKI kernel routines, delay(), drv_getparm(), drv_hztousec(), dtimeout(), itimeout()

Notes

drv_usectohz() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

Some functions take time values expressed in clock ticks; others return time values expressed in clock ticks. Each operating system has its own notion of what constitutes a clock tick; therefore, a driver should not hard-code any assumption about the length of a tick. Rather, use **drv_usectohz()** and its complementary function **drv_hztousec()** to convert between microseconds and clock ticks.

dupb() — DDI/DKI Kernel Routine

Duplicate a message block
#include <sys/stream.h>
mblk_t *dupb(*bufferptr***)**
mblk_t ***bufferptr;**

dupb() creates a new **msgb** structure for the message block to which *bufferptr* points. Unlike the related function **copyb()**, **dupb()** does not copy data block (or blocks) to which the message block points; rather, it just creates a new structure to point to the data block.

If all goes well, **dupb()** returns a pointer to the newly created message block; otherwise, it returns NULL.

See Also

copyb(), datab, DDI/DKI kernel routines, dupmsg(), msgb

Notes

dupb() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

dupmsg() — DDI/DKI Kernel Routine

Duplicate a message

```
#include <sys/stream.h>
mblk_t *dupmsg(message)
mblk_t *message;
```

dupmsg() duplicates the message to which *message* points. It duplicates all of the message blocks in the message and links them together.

If all goes well, **dupmsg()** returns a pointer to the newly created duplicate message; otherwise, it returns NULL.

See Also

copyb(), copymsg(), datab, DDI/DKI kernel routines, dupb(), msgb

Notes

dupmsg() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

enableok() — DDI/DKI Kernel Routine

Enable a queue to be serviced

```
#include <sys/stream.h>
#include <sys/ddi.h>
void enableok(queue)
queue_t *queue;
```

enableok() cancels a previous call to **noenable()**. It permits the service routine of the queue to which *queue* points to be rescheduled.

See Also

DDI/DKI kernel routines, noenable(), queue, qenable()

Notes

enableok() has base or interrupt level. It does not sleep.

The caller cannot have the stream frozen when it calls this function.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

entry-point routines — Overview

Routines for managing requests to the driver

A driver contains entry-point routines via which the kernel or an application can ask the driver to do something. The following gives the legal entry points. Each is described in its own Lexicon entry. Those marked with an '*' apply only to the DDI/DKI interface; those marked with a '†' apply only to the internal COHERENT interface.

block† . . . . . Block interface to the device
chpoll* . . . . . Polling entry point
close . . . . . Close a device
halt* . . . . . Shut down a device upon system shut-down
init*. . . . . Initialize a device
intr*. . . . . Process an interrupt
ioctl. . . . . Control a character device
load† . . . . . Routine to execute upon loading the driver into memory
mmap* . . . . . Check virtual mapping for a memory-mapped device
open . . . . . Open a device
poll† . . . . . Poll the device
power† . . . . . Routine to execute if power fails
print* . . . . . Print a message on the system's console

put* . . . . . Receive messages from the preceding queue
read . . . . . Read data from a device
size* . . . . . Return the size of a logical block device
srv* . . . . . Service messages
start* . . . . . Initialize a device at system start-up
strategy* . . . . . Perform block I/O
time† . . . . . Routine to execute when timeouts occur
uload† . . . . . Routine to execute when driver is unloaded from memory
write . . . . . Write data to a device

Under the DDI/DKI, driver routines are accessed by the kernel by using the driver's unique prefix. For example, to access a driver's **read** routine for the driver whose prefix is **foo**, call the function **fooread()**.

Under the internal COHERENT kernel routines, entry points are accessed through the driver's copy of the structure **CON**. For example, the address of the **read** routine is kept in field **c_read** of the driver's **CON** structure. The kernel contains functions to invoke these routines. For example, to invoke the **read** routine for a device, call function **dread()** with that device's unique identifier.

See Also

con, DDI/DKI kernel routines, internal kernel routines, STREAMS

errors — Overview

List of error messages

The following gives the error codes that drivers can return from their entry-point routines or include within STREAMS messages:

EACCES Permission error: A processes tried to open a file that it has no permission to open.
EADDRINUSE The requested address is being used.
EADDRNOTAVAIL The requested address cannot be assigned.
EAFNOSUPPORT The requested family of addresses is not available.
EAGAIN An attempt to allocate a temporary resource (e.g., memory) failed.
EALREADY The requested operation is already underway.
EBUSY The device is busy.
ECONNABORTED A received-connect request has aborted.
ECONNREFUSED The requested host denied permission to connect.
ECONNRESET The connection was reset by the peer entity.
EDESTADDRREQ The requested operation required a destination address, but none was given.
EFAULT Bad address.
EHOSTDOWN Requested host is down.
EHOSTUNREACH No route to requested host.
EINPROGRESS The requested operation is in progress.
EINTR The operation was interupted.
EINVAL Invalid argument.
EIO An I/O error occurred.
EISCONN The requested endpoint was already connected.
EMSGSIZE The message is too long.
ENETDOWN The requested network is down.
ENETRESET The network dropped the connection because it is being reset.
ENETUNREACH The requested network is unreachable.
ENOBUFS No buffer space is available.
ENODEV The requested device is not available.
ENOMEM Not enough memory.
ENOPROTOOPT The requested protocol option is not available at the indicated level.
ENOSPC The device is out of free space.
ENOTCONN The requested operation requires that the endpoint be connected, but it is not.
ENXIO No such device or address.
EOPNOTSUPP The requested operation is not supported.
EPERM Permission denied.
EPROTO Protocol error.
ETIMEOUT The connection has timed out.

See Also

entry-point routines, geterror(), STREAMS

esballoc() — DDI/DKI Kernel Routine

Allocate a message block using a driver-supplied buffer
#include <sys/stream.h>
mblk_t *esballoc(*buffer***,** *size***,** *priority***,** *freefun***)**
uchar_t ****buffer*; int** *size***,** *priority***; frtn_t ****freefun***;**

esballoc() allocates a STREAMS message and the data-block header, and attaches to them a data buffer that the driver supplies.

buffer points to the base of the data buffer that the driver supplying, and which is *size* bytes long.

priority gives the priority of the allocation request. It must be one of **BPRI_LO**, **BPRI_MED**, or **BPRI_HI**, for, respectively, low-, medium-, or high-priority messages.

freefun points to the data structure that describes the routine to free the driver-allocated message buffer. When the kernel calls **freeb()** upon the last reference to this message, it invokes the routine *freefun***->free_func**. For details, see the Lexicon entry for **free_rtn**.

If all goes well, **esballoc()** returns a pointer to the newly allocated message block; if not, it returns NULL.

See Also

allocb(), DDI/DKI kernel routines, freeb(), free_rtn, msgb

Notes

esballoc() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

esbbcall() — DDI/DKI Kernel Routine

Call a function upon allocation of a buffer
#include <sys/types.h>
#include <sys/stream.h>
toid_t esbbcall(priority, *function***,** *arg***)**
int *priority***; int (***function***)(); long** *arg***;**

esbbcall() calls **esballoc()** to allocate a message-block header and a data-block header for a data buffer that the driver itself supplies. If **esballoc()** cannot immediately allocate the requested headers, **esbbcall()** schedules *function* to be run (with the argument *arg*) when memory becomes available. *function* has no user context and must not call any function that sleeps.

priority gives the priority of the allocation request. Legal values are **BPRI_LO**, **BPRI_MED**, and **BPRI_HI**, for, respectively, a low-, medium-, or high-priority request.

If all goes well, **esbbcall()** returns a non-zero value that identifies the scheduled request; you can pass this identifier to the function **unbufcall()** to cancel the request, should need arise. If, however, something goes wrong, **esbbcall()** returns zero.

See Also

allocb(), bufcall(), DDI/DKI kernel routines, esballoc(), itimeout(), unbufcall()

Notes

esbbcall() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock acoss a call to this function.

Even when *function* is called, **esballoc()** can fail if another module or driver grabbed the available memory before *function* could call **allocb()**.

etoimajor() — DDI/DKI Kernel Routine

Convert external major-device number to internal
#include <sys/types.h>
#include <sys/ddi.h>
major_t etoimajor(*external***)**
major_t *external***;**

etoimajor() converts the external major-device number *external* to an internal major-device number. If all goes well, **etoimajor()** returns the internal major number. If *external* is not valid, it returns **NODEV**.

See Also

DDI/DKI kernel routines, getemajor(), geteminor(), getmajor(), getminor(), itoemajor(), makedevice().

Notes

etoimajor() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

fdisk() — Internal Kernel Routine

Hard-disk partitioning
int fdisk(*dev***,** *fp***)**
dev_t *dev***; struct fdisk_s** *fp***[4];**

fdisk() attempts to read partitioning information from block 0 of the hard-disk device *dev*. If successful, **fdisk()** saves attributes for the four partitions in array *fp*, and returns one. If a read error occurs or it finds an invalid signature for the partition table, it returns zero.

See Also

internal kernel routines

flushband() — DDI/DKI Kernel Routine

Flush messages in a given priority band
#include <sys/types.h>
#include <sys/stream.h>
void flushband(*queue***,** *priority***,** *flag***)**
queue_t **queue***;* **uchar_t** *priority***; int** *flag***;**

flushband() flushes all messages in priority band *priority* of the message queue to which *queue* points. If *priority* is zero, **flushband()** flushes only messages with normal or high priority. Otherwise, it flushes messages from *priority* according to the value of *flag*, as follows:

FLUSHDATA Flush only data messages, i.e., messages of type **M_DATA**, **M_DELAY**, **M_PROTO**, or **M_PCPROTO**.

FLUSHALL Flush all messages.

See Also

DDI/DKI kernel routines, flushq(), put(), queue

Notes

flushband() has base or interrupt level. It does not sleep.

The calling process cannot have the stream frozen when calling this function.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

flushq() — DDI/DKI Kernel Routine

Flush the messages on a queue
#include <sys/stream.h>
void flushq(*queue***,** *flag***)**
queue_t **queue***;* **int** *flag***;**

flushq() frees messages within *queue*. *flag* indicates the messages to flush, as follows:

FLUSHDATA Flush only data messages, i.e, those with type **M_DATA**, **M_DELAY**, **M_PROTO**, or **M_PCPROTO**.

FLUSHALL Flush all messages.

If the number of messages within *queue* falls below its low-water mark, thus allowing another process to write a message onto it, **flushq()** enables the nearest service procedure upstream or downstream (as approriate).

See Also

DDI/DKI kernel routines, flushband(), freemsg(), put(), putq(), queue

Notes

flushq() has base or interrupt level. It does not sleep.

The calling process cannot have the stream frozen when it calls this function.

A driver can hold a driver-defined basic lock, read/write lock, or sleep locks across a call to this function.

free_rtn — STREAMS Data Structure

Structure for STREAMS message-free routine
#include <sys/stream.h>

The structure **free_rtn** holds information on how to invoke the driver's function for freeing a message buffer. When a driver calls routine **esballoc()** to allocate a message, **esballoc()** creates a copy of **free_rtn** and links it to the message. Thus, when routine **freeb()** is called to free the message and the message's reference count drops to zero, **freeb()** reads **free_rtn** and uses the information therein to invoke the driver's routine for freeing the data buffer.

The following fields within **free_rtn** are available to a driver:

void (*free_func)() This points to the driver's function that frees the data buffer. When this function runs, interrupts from all STREAMS devices are blocked. *free_func* has no user context and so cannot call any routine that sleeps, or access any dynamically allocated data structures that may no longer exist when it runs.

char *free_arg This points to an argument to pass to **free_func**. This function can take only one argument, a pointer to a string, it can use this argument creatively.

See Also

DDI/DKI data structures, freeb()

freeb() — DDI/DKI Kernel Routine

Free a message block
#include <sys/stream.h>
void freeb(*buffer***)**
mblk_t **buffer***;*

freeb() frees the message block to which *buffer* points. If the block's reference count, as held in field **db_ref** of structure **datab**, is greater than one, **freeb()** decrements it and returns. If **db_ref** equals one, **freeb()** deallocates the message block and the corresponding data block.

If the data buffer had been allocated by **esballoc()**, **freeb()** frees it by invoking the free routine indicated in its copy of the structure **free_rtn**. When the data buffer is freed, **freeb()** releases all STREAMS resources associated with the buffer, and returns.

See Also

allocb(), datab, DDI/DKI kernel routines, dupb(), esballoc(), free_rtn, msgb

Notes

freeb() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

freemsg() — DDI/DKI Kernel Routine

Free a message
#include <sys/stream.h>
void freemsg(*message***)**
mblk_t **message***;*

freemsg() frees *message*, including all of its message blocks and data buffers.

See Also

DDI/DKI kernel routines, freeb(), datab, msgb

Notes

freemsg() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

freerbuf() — DDI/DKI Kernel Routine

Free a buffer header used for raw I/O
#include <sys/buf.h>
#include <sys/ddi.h>
void freerbuf(*buf_header***)**
buf_t **buf_header***;*

freerbuf() frees a buffer header that had been allocated by a call to **getrbuf()**. It returns nothing. *buf_header* points to the buffer header to be freed.

This function normally is called through the I/O-done handler. For details on what the I/O-done handler is and how you establish it, see the Lexicon entry for **getrbuf()**.

See Also

buf, DDI/DKI kernel routines, getrbuf()

Notes

freerbuf() has base or interrupt level, and does not sleep.

A function can hold a basic lock, read/write lock, or sleep lock across a call to this function.

freezestr() — DDI/DKI Kernel Routine

Freeze a stream
#include <sys/types.h>
#include <sys/stream.h>
pl_t freezestr(*queue***)**
queue_t **queue***;*

freezestr() freezes the stream to which *queue* belongs.

When a stream is frozen, no process can invoke that stream's **open**, **close**, **put**, or service routines. No messages can be added to or removed from any queue, except by the process that called **freezestr()**. Freezing a stream does not stop the functions that are running within it: each continues until it attempts to do something that changes the state of the stream, at which point it must wait for the stream to be thawed.

A driver or module must freeze a stream while it manipulates its queues. This restriction applies to every function that searches a queue, as well as to the functions **insq()**, **rmvq()**, **strqset()**, and **strqget()**.

freezestr() returns the interrupt priority that the stream had had when it was frozen. You can use this value in a subsequent call to **unfreezestr()**, which thaws a stream, to restore the stream's interrupt priority to its pre-frozen level.

See Also

DDI/DKI kernel routines, unfreezestr()

Notes

freezestr() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

Calling **freezestr()** to freeze a stream that you have already frozen causes a deadlock.

Use **freezestr()** sparingly. It is rarely necessary to freeze a stream: most modules do not need to manipulate a queue directly, and freezing a stream slows performance significantly.

getDmaMem() — Internal Kernel Routine

Request virtual address of physical memory
char *getDmaMem(*numBytes***,** *align***)**
unsigned int *numBytes***,** *align***;**

getDmaMem() allocates physically aligned, physically and virtually contiguous blocks of RAM. It is used mainly for devices that use Intel DMA hardware without scatter/gather.

numBytes is the amount of memory requested, in bytes.

align gives the physical boundary to which memory must be aligned. For example, to request four-kilobyte alignment, set *align* to equal 4096. *align* must be a power of two.

If it can meet the request, **getDmaMem()** returns the virtual address of the start of the region allocated. If it cannot grant the request, **getDmaMem()** returns zero.

See Also

getPhysMem(), internal kernel routines

Notes

PHYS_MEM must be patched to at least *numBytes* for the call to **getDmaMem()** to work.

Once allocated, memory is not returned to the **physMem** pool.

getemajor() — DDI/DKI Kernel Routine

Get an external major-device number
#include <sys/types.h>
#include <sys/ddi.h>
major_t getemajor(*device***)**
dev_t *device***;**

getemajor() returns the external major number for *device*.

An external major-device number is the number visible to the user, e.g., through the command **ls -l**. An internal major-device number is visible only to the kernel. Because the range of major-device numbers is large and sparsely populated, the kernel maps external numbers to internal to save space. Every entry point to a driver uses an external, not internal, major-device number.

See Also

DDI/DKI kernel routines, device numbers, etoimajor(), geteminor(), getmajor(), getminor(), makedevice()

Notes

getemajor() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

geteminor() — DDI/DKI Kernel Routine

Get the external minor-device number
#include <sys/types.h>
#include <sys/ddi.h>
minor_t geteminor(*device***)**
dev_t *device***;**

geteminor() returns the external minor number for *device*.

An external minor-device number is visible to the user, e.g., through the command **ls -l**. An internal minor-device number is visible only to the kernel. Every entry point to a driver uses an external, not internal, minor-device number.

See Also

DDI/DKI kernel routines, device numbers, etoimajor(), getemajor(), getmajor(), getminor(), makedevice()

Notes

geteminor() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

getmajor() — DDI/DKI Kernel Routine

Get the internal major-device number
#include <sys/types.h>
#include <sys/ddi.h>
major_t getmajor(*dev***)**
dev_t *device***;**

getmajor() returns the internal major number for *device*. For a description of external and internal major numbers, see the entry for **getemajor()**.

See Also

DDI/DKI kernel routines, device numbers, etoimajor(), getemajor(), geteminor(), getminor(), makedevice()

Notes

getmajor() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

Note that **getmajor()** performs no validity checking: if *device* is bogus, it returns an bogus major-device number.

getminor() — DDI/DKI Kernel Routine

Get internal minor-device number
#include <sys/types.h>
#include <sys/ddi.h>
minor_t getminor(*device***)**
dev_t *device***;**

getminor() returns the internal minor-device number for *device*. For a description of external and internal minor-device numbers, see the entry for **geteminor()**.

See Also

DDI/DKI kernel routines, device numbers, etoimajor(), getemajor(), geteminor(), getmajor(), makedevice()

Notes

getminor() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

getminor() returns no validity checking: if *device* is bogus, it returns a bogus minor-device number.

getPhysMem() — Internal Kernel Routine

Request reserved physical memory
char *getPhysMem(*numBytes***)**
int *numBytes***;**

When the kernel starts up, it reserves a block of physically contiguous memory (of size **PHYS_MEM)** for one or more device drivers to use. Any device driver can request some of this memory; to do so, it calls **getPhysMem()** from within its load routine. *numBytes* gives the number number of bytes it needs.

If it can meet the request, **getPhysMem()** returns the virtual address of the start of the region allocated. This region has contiguous virtual addresses within kernel data space, as well as contiguous physical addresses. If it cannot grant the request, **getPhysMem()** returns 0. Use routine **vtop()** to determine the physical address of the region.

See Also

getDmaMem(), internal kernel routines, vtop()

getq() — DDI/DKI Kernel Routine

Get the next message from a queue
#include <sys/stream.h>
mblk_t *getq(*queue***)**
queue_t ****queue*;**

getq() returns the next message available from the top of *queue*. If no message is queued, it returns NULL. **getq()** handles flow control, and if necessary restarts I/O that was blocked.

See Also

bcanput(), canput(), DDI/DKI kernel routines, putbq(), putq(), qenable(), rmvq()

Notes

getq() has base or interrupt level. It does not sleep.

The calling process cannot have the stream frozen when it calls this function.

A driver can hold a driver-defined basic lock, read/write lock, or sleep locks across a call to this function.

getrbuf() — DDI/DKI Kernel Routine

Allocate a buffer header for raw I/O
#include <sys/buf.h>
#include <sys/ddi.h>
#include <sys/kmem.h>
buf_t *getrbuf(*flag***)**
long *flag*;

getrbuf() allocates a buffer header to be used for performing raw I/O. The driver can then initialize this header to the values that control I/O, then pass its address to the routines that perform I/O.

flag indicates whether the function is willing to sleep while it awaits free space. Setting *flag* to **KM_SLEEP** tells the kernel that if not enough memory is available to allocate a buffer header, the driver is willing to sleep until enough memory becomes available. Setting it to **KM_NOSLEEP** tells the kernel that the driver will not sleep.

getrbuf() returns NULL if something goes wrong; for example, insufficient memory is available to allocate a buffer header and *flag* is set to **KM_NOSLEEP**. If all goes well, however, it returns the address of the header, which is an object of type **buf_t**.

After the kernel has allocated the buffer header and returned its address, the driver must initialize the fields of the buffer header as follows:

b_bcount
The number of bytes to be transferred.

b_blkno
The number of the block to be accessed.

b_bufsiz
The size of the buffer that is associated with this header.

b_dev The non-extended device number. Note that this applies only to COHERENT.

b_edev The number of the device being manipulated.

b_flags The direction of data transfer: **B_READ** if the transfer moves from the kernel to the user's buffer; or **B_WRITE** if data moves from the user's buffer to the kernel. The setting must match that in field **b_req**. See below for details.

b_iodone
The address of the function to call when the raw I/O has finished.

b_paddr
The system global address of the data area. Note that this applies only to COHERENT.

b_req Set this to either **BREAD** or **BWRITE**. The setting must match that in field **b_flags**. See below for details.

b_resid The number of bytes to transfer. This field's value must match that of field **b_bcount**.

b_un.b_addr
The virtual address of the buffer that the user supplies. Note that whatever program invokes **getrbuf()** must also obtain the data area. It must do this before it calls **physiock()** or any other function to which it can pass a buffer.

The following gives the proper way to set or unset **b_flags** and **b_req** for reading or writing:

Reading

```
bp->b_flags |= B_READ;
bp->b_req = BREAD;
```

Writing

```
bp->b_flags &= ~B_READ;
bp->b_req = BWRITE;
```

By default, the buffer header has **B_READ** set to off. The driver is not allowed to modify flags pell-mell, or the results may crash the system.

See Also

buf, DDI/DKI kernel routines, freerbuf()

Notes

If *flag* is set to **KM_SLEEP**, **getrbuf()** has base level and can sleep; if it is set to **KM_NOSLEEP**, it has base or interrupt level and does not sleep.

If *flag* is set to **KM_SLEEP**, a function can hold driver-defined basic locks and read/write locks across a call to this function; otherwise, it cannot. A function can hold a sleep lock across a call to this function regardless of the value of *flag*.

getubd() — Internal Kernel Routine

Get a byte from user data space
char getubd(*u***)**
char **u**;

getubd() reads a byte from offset *u* in the current process's user data space. If an address fault occurs, **getubd()** calls **set_user_error()** with the value **EFAULT**.

See Also

internal kernel routines

getusd() — Internal Kernel Routine

Get a short from user data
short getusd(*usp***)**
short * *usp*;

getusd() gets a short (16-bit) integer from the user data space address pointed to by *usp*.

See Also

internal kernel routines

getuwd() — Internal Kernel Routine

Get a word from user data space
int getuwd(*u***)**
char **u**;

getuwd() reads a word from offset *u* in the current process's user data space. If an address fault occurs, **getuwd()** calls **set_user_error()** with the value **EFAULT**.

See Also

internal kernel routines

getuwi() — Internal Kernel Routine

Get a word from user code space
int getuwi(*u***)**
char **u**;

getuwi() reads a word from offset *u* in the current process's user code space. If an address fault occurs, it calls **set_user_error()** with the value **EFAULT**.

See Also

internal kernel routines

halt — Entry-Point Routine

Shut down a device upon system shut-down
void *prefix***halt();**

The kernel invokes a driver's **halt** routine when the system is shut down. The driver should not assume that interrupts are enabled. It should ensure that its device has no more interrupts pending, and it should inform its device to generate no more interrupts.

After the **halt** routine is called, no more calls can be made to the driver's entry points.

See Also

entry-point routines

Notes

This entry-point is used only by the DDI/DKI interface. It is optional.

This routine should never sleep.

inb() — DDI/DKI Kernel Routine

Read a byte from an eight-bit I/O port
#include <sys/types.h>
uchar_t inb(*port***)**
int *port***;**

inb() reads an unsigned character from *port*, which is a valid eight-bit I/O port, and returns it.

Function **repinsb()** resembles **inb()**, except that it reads a string of characters from a port.

See Also

DDI/DKI kernel routines, inl(), inw(), outb(), outl(), outw(), repinsb(), repinsd(), repinsw(), repoutsb(), repoutsd(), repoutsw()

Notes

inb() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

init — Entry-Point Routine

Initialize a device
void *prefix***init();**

init is the entry point to the routine with which the driver initializes its device. The kernel invokes this routine when it is booted. The related entry point, **start**, can be invoked after the system has been booted.

The **init** routine is executed before interrupts are enabled. It must not sleep, nor may it call any function that sleeps or requires user context.

An **init** routine can call the following DDI/DKI kernel functions:

ASSERT()	**getmajor()**	**makedevice()**
bcopy()	**getminor()**	**outb()**
bzero()	**inb()**	**outl()**
cmn_err()	**inl()**	**outw()**
drv_getparm()	**inw()**	**phalloc()**
drv_hztoused()	**itoemajor()**	**phfree()**
drv_usectohz()	**kmem_alloc()**	**repinsw()**
etoimajor()	**kmem_free()**	**repoutsb()**
getemajor()	**kmem_zalloc()**	**repoutsw()**
geteminor()	**LOCK_ALLOC()**	**SLEEP_ALLOC()**

See Also

entry-point routines, start

Notes

This entry-point is used only by the DDI/DKI interface. It is optional.

inl() — DDI/DKI Kernel Routine

Read a 32-bit value from an I/O port
ulong_t inb(*port***)**
int *port***;**

inl() reads an unsigned long integer from *port*, which is a valid 32-bit I/O port, and returns it.

See Also

DDI/DKi kernel routines, inb(), inw(), outb(), outl(), outw(), repinsb(), repinsd(), repinsw(), repoutsb(), repoutsd(), repoutsw()

Notes

inl() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

insq() — DDI/DKI Kernel Routine

Insert a message into a queue
#include <sys/stream.h>
int insq(*queue***,** *oldmsg***,** *newmsg***)**
queue_t ******queue***; mblk_t** ******oldmsg***,** ******newmsg***;**

insq() inserts *newmsg* into *queue*, at the point immediately preceding *oldmsg*. If *oldmsg* is NULL, **insq()** inserts *newmsg* at the end of *queue*.

If all goes well, **insq()** returns one; otherwise, it returns zero.

insq() updates all flow-control parameters. It schedules the service procedure to be run, unless it had been disabled by a call to **noenable()**.

STREAMS requires that messages be ordered by their priority. If a driver attempts to insert a message out of order, **insq()** will not enqueue it.

See Also

DDI/DKi kernel routines, freezestr(), getq(), putbq(), putq(), rmvq(), unfreezestr()

Notes

insq() has base or interrupt level. It does not sleep.

The caller must have the stream frozen when calling this function.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

internal data structures — Overview

The following articles define data structures that are internal to the O kernel:

buf . . . . . Buffer cache
con . . . . . Structure of a device driver
io . . . . . Manage communication with a device
uproc . . . . . Define a process

See Also

device driver, internal kernel routines

internal kernel routines — Overview

The following routines are internal to the COHERENT kernel.

actvsig() . . . . . Activate signal handler
altclk_in() . . . . . Install polling function
altclk_out() . . . . . Uninstall polling function
bclaim() . . . . . Claim a buffer
bdone() . . . . . Block I/O completed
bflush() . . . . . Flush buffer cache
bread() . . . . . Read into buffer cache
brelease() . . . . . Release a buffer
bsync() . . . . . Flush modified buffers
busyWait() . . . . . Busy-wait the system, pending some event
busyWait2() . . . . . Busy-wait the system, pending some event
bwrite() . . . . . Write buffer to disk
clrivec() . . . . . Clear interrupt vector
clrq() . . . . . Clear character queue
cltgetq() . . . . . Get a char from a character queue
cltputq() . . . . . Put a character on a character queue
dblock() . . . . . Call the device block interface
dclose() . . . . . Invoke the driver's close routine
defend() . . . . . Execute deferred functions
defer() . . . . . Defer function execution
devmsg() . . . . . Print a message from a device driver
dioctl() . . . . . Call a device-driver's I/O control point
dmago() . . . . . Enable DMA transfers
dmain() . . . . . Copy from system global memory to kernel data
dmaoff() . . . . . Disable DMA transfers
dmaon() . . . . . Prepare for DMA transfer
dmaout() . . . . . Copy from kernel data to system global memory
dmareq() . . . . . Request block I/O, avoiding DMA straddles
dopen() . . . . . Invoke the driver's open routine
dpoll() . . . . . Invoke the driver's poll routine
dpower() . . . . . Invoke the driver's power-fail routine
dread() . . . . . Invoke the driver's read routine
dtime() . . . . . Invoke the driver's timeout routine
dwrite() . . . . . Invoke the driver's device write routine
fdisk() . . . . . Hard-disk partitioning
getDmaMem() . . . . . Request virtual address of physical memory
getPhysMem() . . . . . Request reserved physical memory
getubd() . . . . . Get a byte from user data space
getusd() . . . . . Get a 16-bit short integer from user data space
getuwd() . . . . . Get a 32-bit word from user data space
getuwi() . . . . . Get a word from user code space
iogetc() . . . . . Get a character from I/O segment
iomapand() . . . . . Set bits in the I/O privilege bitmap
iomapOr() . . . . . Clear bits in the I/O privilege bitmap
ioputc() . . . . . Put a character into I/O segment
ioread() . . . . . Read from I/O segment
ioreq() . . . . . Re-queue I/O request through block routine
iowrite() . . . . . Write to I/O segment

kalloc() . . . . . Allocate kernel memory
kfree() . . . . . Free kernel memory
kiopriv() . . . . . Write a bit into the I/O privilege bitmap
kucopy() . . . . . Kernel-to-user data copy
lock() . . . . . Lock a gate
locked(). . . . . See if a gate is locked
major() . . . . . Extract major-device number
map_pv() . . . . . Map physical to virtual addresses
MAPIO() . . . . . Return global address
mapPhysUser() . . . . . Overlay user data with memory-mapped hardware
minor() . . . . . Extract minor-device number
nondsig(). . . . . Non-default signal pending
P2P() . . . . . Convert system global to physical address
panic() . . . . . Fatal system error
pollopen() . . . . . Initiate driver polled event
pollwake() . . . . . Terminate driver polled event
printf() . . . . . Formatted print
putubd() . . . . . Store a byte into user data space
putusd() . . . . . Store a short into user data data
putuwd() . . . . . Store a 32-bit word into user data space
putuwi() . . . . . Put a word into user code space
pxcopy() . . . . . Copy from physical or system global memory to kernel data
read_t0() . . . . . Read the system clock t0
salloc() . . . . . Allocate a memory segment
sendsig() . . . . . Send a signal
setivec() . . . . . Set an interrupt vector
sigdump() . . . . . Generate core dump
sphi() . . . . . Disable interrupts
spl() . . . . . Adjust interrupt mask
splo() . . . . . Enable interrupts
super() . . . . . Verify super-user
timeout(). . . . . Defer function execution
ttclose() . . . . . Close tty
ttflush() . . . . . Flush a tty
tthup() . . . . . tty hangup
ttin() . . . . . Pass character to tty input queue
ttinp(). . . . . See if tty input queue has room for more input
ttioctl(). . . . . Perform tty I/O control
ttopen() . . . . . Open a tty
ttout() . . . . . Get next character from tty output queue
ttoutp(). . . . . See if tty input queue has data available
ttread(). . . . . Read from tty
ttread0() . . . . . Read from tty
ttsetgrp(). . . . . Set tty process group
ttsignal() . . . . . Send tty signal
ttstart(). . . . . Start tty output
ttwrite() . . . . . Write to tty
ttwrite0() . . . . . Write to tty
ukcopy() . . . . . User to kernel data copy
unlock() . . . . . Unlock a gate
unmap_pv() . . . . . Dissociate virtual addresses from physical addresses
vtop() . . . . . Translate virtual address to physical address
wakeup() . . . . . Wakeup processes sleeping on an event
x_sleep() . . . . . Wait for event or signal
xpcopy() . . . . . Copy from kernel data to physical or system global memory

See Also

DDI/DKI kernel routines, device driver, internal data structures, STREAMS

intr — Entry-Point Routine

Process an interrupt
void *prefix***intr(***vector***)**
int *vector***;**

The **intr** routine is the interrupt handler for both block and character drivers. This entry point applies *only* to drivers that use the DDI/DKI interface to the kernel.

vector gives the number that COHERENT uses to associate a driver's interrupt handler with an interrupting device. This number is set in the file **sdevice**.

The **intr** routine performs all tasks specific to the driver and its device. You should know the exact chip set that produces the interrupt for your device, the bit patterns of the device's control and status register, and how data are moved into and out of your computer.

If the driver called **biowait()** or **SV_WAIT()** to await the completion of an operation, the **intr** routine must call **biodone()** or **SV_SIGNAL()** to tell the process to resume.

See Also

biodone(), entry-point routines, spl(), SV_SIGNAL()

Notes

This entry-point is used only by the DDI/DKI interface. It is required only in drivers for hardware that generate interrupts. It is not used with software drivers.

The **intr** routine must never do the following:

- Call a function that sleeps.
- Lower the level of interrupt priority below that at which the interrupt routine was entered.
- Call any routine that requires user context.
- Call **uiomove()**, **ureadc()**, or **uwritec()** when field **uio_segflg** of structure **uio** equals **UIO_USERSPACE**, which indicates that data are being transferred between the user and kernel spaces.

inw() — DDI/DKI Kernel Routine

Read a 16-bit word from an I/O port
ushort_t inw(*port***)**
int *port***;**

inw() reads an unsigned short integer from *port*, which is a valid 16-bit I/O port, and returns it.

See Also

DDI/DKI kernel routines, inb(), inl(), outb(), outl(), outw(), repinsb(), repinsd(), repinsw(), repoutsb(), repoutsd(), repoutsw()

Notes

inw() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

io — Internal Data Structure

Manage communication with a device
#include <sys/io.h>

The kernel uses structure **io** to manage communication with a device. It is defined in header file **<sys/io.h>**.

A driver's **read** function can use the following fields within **io**, as follows:

io_seek Point at which reading begins. It is in the form of the number of bytes from the beginning of the file or device. This is, of course, is meaningless for devices for devices like serial ports. In the case of disk drives, this number must indicate the block to be read, i.e., the number must be evenly divisible by constant **BSIZE**, which gives the size of a COHERENT block. If this is not true, an error has occurred.

io_ioc The number of bytes to read or write. When the read is completed, this should be set to the number of bytes that remain to be read or written. If it is not reset to zero, then an error has occurred.

io_base The offset of data to be transferred into the user's memory space. This is converted to a physical or virtual memory address before performing the read.

io_flag Flags. **IONDLY** indicates that the request should not delay for data if the requested results are not immediately available; it is equivalent to the DDI/DKI value **O_NDELAY**. **IONONBLOCK** indicates that the driver is non-block; it is equivalent ot the DDI/DKI value **O_NONBLOCK**.

See Also

internal data structures

iocblk — STREAMS Data Structure

STREAMS ioctl structure
#include <sys/stream.h>

The structure **iocblk** defines an **ioctl** request. Messages of types **M_IOCTL**, **M_IOCACK**, and **M_IOCNAK** use it. The driver does not create this structure.

A module or driver usually converts a message of type **M_IOCTL** into one of type **M_ICACK** or **M_IOCNAK** by changing its type and updating the relevant fields within **iocblk**. When a driver processes a transparent **ioctl** (defined below), it usually overlays **iocblk** with a copy of the structure **copyreq**. The stream head guarantees that the message is large enough to hold either structure.

The following fields within **iocblk** are available to a driver:

int ioc_cmd This holds the **ioctl** command that the user issued.

cred_t *ioc_cr This points to the user's credentials.

uint_t ioc_id This uniquely identifies the **ioctl** request within the stream.

uint_t ioc_count This gives the number of bytes of data within the **M_IOCTL** message. Data are passed in message blocks of type **M_DATA** that are linked to the **M_IOCTL** message. In a message of type **M_IOCACK**, this field gives the number of bytes to copy into the user's buffer.

If this field is set to the special value **TRANSPARENT**, the **ioctl** request is *transparent*. This means that the the user did not use the **I_STR** format of STREAMS **ioctl**s. The module or driver must use **M_COPYIN** messages to obtain user data, and use **M_COPYOUT** messages to change user data. The message block **M_DATA** linked to the **M_IOCTL** message block contains the value of the *arg* parameter to the COHERENT system call **ioctl()**.

int ioc_error This field holds the error, if any, for a message of type **M_IOCACK** or **M_IOCNAK**.

int ioc_rval This field is set to the return value, if any, for a message of type **M_IOCACK**. The kernel returns this value to the system call **ioctl()** that generated the request.

See Also

copyreq, copyresp, data structures, datab, msgb

ioctl — Entry-Point Routine

Control a character device

Internal-Kernel Interface:
#include <sys/types.h>
#include <sys/cred.h>
#include <sys/file.h>
#include <sys/errno.h>
void *prefix***ioctl(***device*, *command*, *arg*, *mode*, *credptr*, *retptr*, *private***)**
dev_t *device*; **int** *command*, *mode*, **retptr*; **_VOID** **arg*; **cred_t** **credptr*, **void** **private*;

DDI/DKI or STREAMS:
#include <sys/types.h>
#include <sys/cred.h>
#include <sys/file.h>
#include <sys/errno.h>
int *prefix***ioctl(***device*, *command*, *arg*, *mode*, *credptr*, *retptr***)**
dev_t *device*; **int** *command*, *mode*, **retptr*; **_VOID** **arg*; **cred_t** **credptr*;

The **ioctl** (i.e., "I/O control") routine gives a non-STREAMS character driver an alternate entry point that it can use for almost any operation other than a transfer of data, e.g., to implement terminal setting, format disk devices, implement a trace driver for debugging, and flush queues. An application can invoke the **ioctl** routine through the COHERENT system call **ioctl()**.

Internal-Kernel Interface

Under the internal-kernel inteface to a driver, the address of the **ioctl** routine is kept in field **c_ioctl** of the driver's **CON** structure. It is customary to name the **ioctl** routine with the word **ioctl** prefixed by a unique identifier for your driver; but this is not required.

An **ioctl** routine takes the following arguments:

device This is a **dev_t** that identifies the device to be manipulated.

command
This gives the number of the operation to perform. These numbers are specific to the driver.

arg This points to the parameters passed between the user and the driver. The nature of the arguments depends upon the driver and on the *command* being executed.

mode This gives the modes to set when the device was opened. See the description of the entry point **open** for a description of the legal values for this argument.

credptr This points to the user's credential structure.

retptr This gives the address at which the **ioctl** routine must write its return value.

private This points to a data item that is unique to your driver. Note that most drivers do not use this argument.

The **ioctl** returns nothing. The kernel determines what the system call **ioctl()** (which invokes this entry-point routine) returns to the user application: the kernel returns -1 (and sets **errno** to an appropriate value) if the **ioctl** entry-point routine called **set_user_error()** to return an error. If the **ioctl** routine exits normally, **ioctl()** returns the value that the **ioctl** routine writes at address *retptr*; if this is not set, **ioctl()** returns zero.

DDI/DKI or STREAMS

The rest of this article describes how to invoke this function under the DDI/DKI interface, using the calling conventions given at the beginning of this article. The kernel invokes it by calling function *prefix***ioctl()**, where *prefix* is the unique prefix for this driver. The function takes the following arguments:

device The number of the device to manipulate.

command The number of the operation to perform. These numbers are specific to the driver.

arg A pointer to the parameters passed between the user and the driver. The nature of the arguments depends upon the driver and on the *command* being executed. If *arg* points to the user space, the driver can use functions **copyin()** and **copyout()** to transfer data between kernel and user space.

mode The modes set when the device was opened. See the description of the entry point **open** for a description of the legal values for this argument.

credptr A pointer to the user's credential structure.

retptr The address at which the **ioctl** routine must write its return value.

The **ioctl** routine returns an **int** to the kernel. The value that the kernel's system call **ioctl()** returns user-level program is determined by how the **ioctl** routine exits. If the **ioctl** routine called **set_user_error()** to report an error, **ioctl()** returns -1.

However, if the **ioctl** routine exits normally, it should return zero to the kernel; the system call **ioctl()**, will return to the user-level program the value that the **ioctl** routine wrote at the address *retptr*. If the **ioctl** routine fails, it should return -1 to the kernel; **ioctl()** will return that value to the user-level program.

See Also

CON, copyin(), copyout(), drv_priv(), entry-point routines, errno, open
COHERENT Lexicon: **ioctl()**

Notes

This entry point is optional.

The **ioctl** routine has user context and can sleep.

STREAMS drivers do not have **ioctl** routines. The stream head converts I/O control commands to **M_IOCTL** messages, which are handled by the driver's **put** or **srv** routines.

iogetc() — Internal Kernel Routine

Get a character from I/O segment
#include <sys/io.h>
int iogetc(*iop***)**
IO **iop*;*

iogetc() reads a character from the I/O segment referenced by *iop*. If an address fault occurs, it calls **set_user_error()** with value **EFAULT** and returns -1; otherwise, it decrements *iop***->ioc** by one and returns the value of the character read. If *iop***->io_ioc** (the I/O count) is zero, **iogetc()** returns -1.

See Also

internal kernel routines

iomapAnd() — Internal Kernel Routine

int iomapAnd(*val***,** *offset***)**
int *val***,** *offset***;**

iomapAnd() "bitwise AND's" the 32-bit mask *val* at the word offset *offset* within the I/O privilege map. This permits a usr's code to to enable a given option on a given port or ports.

If *offset* is zero, **iomapAnd()** covers ports zero through 31; if *offset* is one, it covers ports 32 through 63; and so on. The current valid range for *offset* is zero through 63, corresponding to ports in the range of zero through 0x7FF.

iomapAnd() returns the updated map word.

See Also

internal kernel routines

iomapOr() — Internal Kernel Routine

Clear bits in the I/O privilege bitmap
int iomapOr(*val***,** *offset***)**
int *val***,** *offset***;**

iomapOr() "bitwise OR's" the 32-bit mask *val* at the word offset *offset* within the I/O privilege map. This permits a usr's code to to disable a given option on a given port or ports.

If *offset* is zero, **iomapOr()** covers ports zero through 31; if *offset* is one, it covers ports 32 through 63; and so on. The current valid range for *offset* is zero through 63, corresponding to ports in the range of zero through 0x7FF.

iomapOr() returns the updated map word.

See Also

internal kernel routines

ioputc() — Internal Kernel Routine

Put a character into I/O segment
#include <sys/io.h>
int ioputc(*c***,** *iop***)**
char *c***; IO** **iop*;*

ioputc() writes character *c* into the I/O segment referenced by *iop*. If an address fault occurs, **ioputc()** calls **set_user_error()** with value **EFAULT** and returns -1; otherwise, it decrements *iop***->io_ioc** by one and returns the value of the character written. If *iop***->io_ioc** (the I/O count) is zero, it returns -1.

See Also

internal kernel routines

ioread() — Internal Kernel Routine

Read from I/O segment
#include <sys/io.h>
void ioread(*iop***, *v*, *n*)**
IO ****iop*; char ****v***; unsigned** *n***;**

ioread() copies *n* bytes from the I/O segment referenced by *iop* to address *v* in the kernel's data segment. If an address fault occurs, it calls **set_user_error()** with value **EFAULT**.

See Also

internal kernel routines

ioreq() — Internal Kernel Routine

Re-queue I/O request through block routine
#include <sys/io.h>
void ioreq(*bp***, *iop*, *dev*, *req*, *f*)**
BUF ****bp*; IO ****iop***; dev_t** *dev***;**

ioreq() queues a request through the *block* routine of the driver. If a request is already pending on the IO structure referenced by *iop*, queuing will not occur until the previous request is completed. *req* should be **BREAD** or **BWRITE**. *f* should be **BFIOC|BFRAW** under normal circumstances. **ioreq()** normally is called from the read/write routines of a block device that does not support DMA.

See Also

dmareq(), internal kernel routines

iovec — DDI/DKI Data Structure

DDI/DKI data-storage structure for scatter/gather I/O
#include <sys/types.h>
#include <sys/uio.h>

The structure **iovec** describes a data storage area that is used in a scatter/gather I/O transfer. The structure **uio** controls such a transfer; it contains a pointer to an array of **iovec** structures, each of which describes one hunk of memory to be used in the transfer.

The kernel or a driver can create an **iovec**. The rules by which an **iovec** is manipulated depend upon its origin — and therefore, upon which entity "owns" it. A driver can set the fields within **iovec** structure only for the **uio** structures that it has created. It can, however, read all **iovec** structures.

iovec contains two fields that are available to drivers:

caddr_t iov_base;
The base address of the "hunk" of memory.

int iov_len;
The size, in bytes, of the "hunk" of memory.

See Also

data structures, physiock(), uio, uiomove(), ureadc(), uwritec()

Notes

No function exists for allocating **iovec** structures when a driver needs to create them. Therefore, a driver can either use **kmem_zalloc()** to allocate them, or allocate them statically.

iowrite() — Internal Kernel Routine

Write to I/O segment
#include <sys/io.h>
void iowrite(*iop***, *v*, *n*)**
IO ****iop*; char ****v***; unsigned** *n***;**

iowrite() writes *n* bytes from address *v* in the kernel's data segment to the I/O segment referenced by *iop*. If an address fault occurs, **iowrite()** calls **set_user_error()** with **EFAULT**.

See Also

internal kernel routines

itimeout() — DDI/DKI Kernel Routine

Execute a function after a given length of time
#include <sys/types.h>
toid_t itimeout(*function***,** *argument***,** *ticks***,** *priority***)**
void (* *function***)(); void** ******argument***;**
long *ticks***; pl_t** *priority***;**

The DDI/DKI function **itimeout()** schedules a function to be executed after a given amount of time has passed. If something goes wrong and it could not schedule the function, **itimeout()** returns zero. Otherwise, it returns an identifier other than zero. You can pass this value to the function **untimeout()** to cancel this request, should you wish.

function points to the function to execute. It must neither sleep nor reference the process's context. *ticks* gives the number of clock ticks to wait before executing the function. *argument* points to an argument to pass to *function*. Finally, *priority* gives the function's interrupt priority. The value supplied must be at least **pltimeout**, i.e., **plbase** is not valid.

The kernel may not execute *function* exactly after *ticks* clock ticks have passed; however, it will wait at least *ticks*-1 ticks before it executes *function*.

See Also

DDI/DKI kernel routines, LOCK_ALLOC(), untimeout()

Notes

itimeout() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

Before you de-initialize or deallocate a data structure, first cancel every function that **itimeout()** has scheduled to access that structure.

When the time comes to execute a scheduled function, the kernel will run it only if the processor is at base level. If the processor is not at base level, execution of the function is deferred.

itoemajor() — DDI/DKI Kernel Routine

Convert internal to external major number
#include <sys/types.h>
#include <sys/ddi.h>
major_t itoemajor(*imajor***,** *prevemajor***)**
major_t *imajor***,** *prevemajor***;**

itoemajor() returns the external major number that corresponds to the internal major number *imajor*. See **getemajor()** for an explanation of external and internal major numbers.

prevemajor gives the most recently obtained external major number. When you call **itoemajor()** for the first time, set this to **NODEV**. Because an internal major number can be associated more than external major number, this mechanism lets you call **itoemajor()** repeated to find all of the external major numbers. When it has returned all external major numbers associated with *imajor*, it returns **NODEV**.

See Also

DDI/DKI kernel routines, etoimajor(), getemajor(), geteminor(), getmajor(), getminor(), makedevice()

Notes

itoemajor() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

kalloc() — Internal Kernel Routine

Allocate kernel memory
#include <sys/coherent.h>
char * **kalloc(***n***)**
int *n***;**

kalloc() is a macro that allocates *n* bytes in the kernel's data segment. The amount of space available to **kalloc()** is limited by the kernel variable **ALLSIZE**. **kalloc()** returns a pointer to the allocated buffer, or NULL if space is insufficient.

The storage space returned contains garbage. Most drivers use **memset()** to clear the storage, if needed. Space allocated with **kalloc()** must be deallocated with **kfree()**.

See Also

kfree(), internal kernel routines

kernel variables — Technical Information

Variables set within COHERENT kernel

The following lists most of the variables that you can "hot patch" within the kernel.

For a complete list of tunable variables, see the file **/etc/conf/mtune**. Note, however, that this file does not name the variables themselves; rather, it uses the variable's enumeration constant initialized; e.g., **mtune** identifies variable **ALLSIZE** as **ALLSIZE_SPEC**.

The clock rate is defined as the manifest constant **HZ** (hertz), which is set in header file **kernel/const.h**. Normally, this value is set to 100, which translates into 100 ticks per second, or approximately 10 milliseconds per tick.

By using command **/conf/patch** to reset one or more of these variables, you can change the behavior of the kernel. Note that it is possible to reset these variables in such a way that the kernel is unusable, memory is destroyed, or other undesirable consequences occur. *If you do not know* exactly *what you are doing, you are well advised to leave these variables alone!*

ALLSIZE — Size of kernel memory allocation pool

```
int ALLSIZE;
```

ALLSIZE gives the number of bytes in the kernel's memory allocation pool. This pool is manipulated by the functions **kalloc** and **kfree**. **ALLSIZE** is "auto sized" unless patched to a non-zero value.

ISTSIZE — Initial stack size

```
int ISTSIZE = 4096;
```

ISTSIZE specifies the size of the user stack, in bytes. This affects all processes. It can be increased if required. Reducing the size of the user's stack may cause programs to crash due to stack overflow. The kernel stack associated with a process will not change.

KBBOOT — Toggle MS-DOS-style booting

```
int KBBOOT = 1;
```

KBBOOT flags whether your system can be rebooted MS-DOS fashion, i.e., by typing **<ctrl><alt><del>**. When set to a non-zero value, it enables MS-DOS rebooting; this is the default. You can use **patch** to reset this variable to zero, as follows:

```
/conf/patch /coherent KBBOOT=0
```

Thereafter, typing **<ctrl><alt><del>** displays the value of function key 0 rather than rebooting. Function key 0 defaults to the phrase "reboot", as a reminder that this key normally reboots your system. However, this never actually prints since the system normally reboots. You can set the value of function key 0 to anything you want, either via the command **fnkey** or directly in the keyboard tables located in directory **/conf/kbd**.

NBUF — Number of blocks in buffer cache

NBUF specifies the number of blocks in the buffer cache. It is auto-sized unless you patch it to a non-zero value.

NCLIST — Number of clists

```
int NCLIST = 64;
```

NCLIST specifies the number of clists in kernel memory. clists are used by the canonical tty routines to store input/output data.

NINODE — Number of in-memory i-nodes

```
int NINODE = 128;
```

NINODE specifies the maximum number of i-nodes that can be open simultaneously.

NMSC — Number of characters per message

```
int NMSC = 640;
```

NMSC gives the maximum number of characters per message.

NMSG — Number of message buffers

```
int NMSG = 10;
```

NMSG gives the number of message buffers allocated.

NMSQB — Maximum characters per message queue

```
int NMSQB = 2048;
```

NMSQB gives the default maximum number of bytes of messages on any one message queue.

NMSQID — Maximum number of message queues

```
int NMSQID = 9;
```

NMSQID specifies the maximum number of message queues in the system.

NPOLL — Number of simultaneous pending polls

```
int NPOLL = 0;
```

NPOLL specifies the maximum number of polls that can be pending simultaneously. If it is zero, dynamic allocation will occur, in groups of 32 pending polls. You increase variable **ALLSIZE** by eight bytes per pending poll.

PHYS_MEM — Amount of memory reserved for drivers
This variable is an **int**. Its value is the number of bytes needed in the block of physically contiguous memory reserved for special-purpose device drivers.

RLOCKS — Number of available locks
This variable is an **int**. By default, it is set to 100.

SHMMAX — Maximum size of a shared-memory segment
This variable gives the maximum size of a shared-memory segment. By default, it is set to 0x10000.

SHMMNI — Maximum number of shared-memory segments
This gives the maximum number of shared-memory segments that can exist at any one time. By default, it is set to 100.

VIDSLOW — Slow (no snow) video updates

```
int VIDSLOW = 0;
```

Set **VIDSLOW** to non-zero to enable video memory updates only during vertical retrace. This reduces snow on the display with some older video controller cards.

condev — Console device

```
dev_t condev = makedev(2,0);
```

condev specifies the console device that the kernel's **printf** or **putchar** routines write to. This normally is the memory-mapped video driver, but it can be mapped to any terminal driver that recognizes data written from the kernel's data segment. The drivers for the **console** and **serial** devices are currently supported as the kernel's console devices.

cprocp — Pointer to current process

```
PROC *cprocp;
```

cprocp points to the **proc** structure that is associated with the user process that is currently executing.

drvl — Device driver list

```
#include <sys/con.h>
#include <kernel/param.h>
DRV drvl[drvn];
```

drvl is an array that references device drivers. Field **d_conp** points to a table of driver access routines, or is NULL. Field **d_time** is non-zero if the driver timed routine is to be invoked once per second.

drvn — Number of device drivers

```
int drvn;
```

drvn gives the maximum number of device drivers available to the kernel.

lbolt — Clock ticks since system startup (lightning bolt)

```
time_t lbolt;
```

lbolt is the number of clock ticks since system startup. A clock tick normally occurs **HZ** times per second.

pipedev — File system used for pipes

```
dev_t pipedev;
```

pipedev gives the file system to be used for pipes. It is normally the same as **rootdev** (the root device).

ronflag — Root file system is read-only

```
int ronflag;
```

If **ronflag** is set to non-zero, the root file system has read-only access.

rootdev — File system used for root device

```
dev_t rootdev;
```

rootdev specifies the root file system's device.

See Also

technical information

Notes

You can modify most kernel variables by modifying file **/etc/conf/stune**. If a variable is not named in that file, you can modify it with the debugger **db** using the command line:

`db -a` *kernel*`.sym` *kernel*

where *kernel* and *kernel.sym* name, respectively, the kernel to be patched and its symbol-table file. Once you have invoked the debugger, type the name of the variable; **db** will display its address and current value. If the value is other than what you want, type the command

varname=*value*

which patches variable *varname* to *value*. Then type the command **:q** to exit from **db**. This approach works, but has the disadvantage that you must repeat it each time you link a new kernel.

kfree() — Internal Kernel Routine

Free kernel memory
#include <sys/coherent.h>
void kfree(*k***)**
char **k***;*

kfree() is a macro that frees a dynamic buffer that had been obtained from **kalloc()**.

See Also

internal kernel routines

kiopriv() — Internal Kernel Routine

Write a bit into the I/O privilege bitmap
int kiopriv(*port***,** *bit***)**
unsigned int *port***,** *bit***;**

kiopriv() writes the bit value *bit* into the I/O privilege bitmap for the address of *port*. A *bit* value of zero enables user I/O for the port address; whereas a *bit* value of one disables user I/O for that port address.

kiopriv() returns zero if the port address was invalid (i.e., out of range for the bitmap); otherwise, it returns one.

See Also

internal kernel routines

kmem_alloc() — DDI/DKI Kernel Routine

Allocate space from kernel free memory
#include <sys/types.h>
#include <sys/kmem.h>
void *kmem_alloc(*bytes***,** *flag***)**
size_t *bytes***; int** *flag***;**

kmem_alloc() allocates *bytes* of kernel memory.

flag indicates whether the driver can sleep while waiting for memory. **KM_SLEEP** indicates that the driver will sleep, if necessary, until the requested amount of memory is available; therefore, **kmem_alloc()** waits until *bytes* of memory is available. **KM_NOSLEEP** indicates that it will not sleep; therefore, **kmem_alloc()** returns NULL if the requested amount of memory is not available immediately.

kmem_alloc() returns a pointer to the newly allocated memory. If *flag* is set to **KM_NOSLEEP** and *bytes* of memory is not available, it returns NULL. It always returns NULL if you set *bytes* to zero.

See Also

DDI/DKI kernel routines, kmem_free(), kmem_zalloc()

Notes

kmem_alloc() has base level only if *flag* equals **KM_SLEEP**; it has base or interrupt level if *flag* equals **KM_NOSLEEP**. It can sleep if *flag* is set to **KM_SLEEP**.

A driver can hold a driver-defined basic lock or read/write lock across a call to this function if *flag* is **KM_NOSLEEP**, but may not hold it if *flag* is **KM_SLEEP**. It can hold a driver-defined sleep lock regardless of the value of *flag*.

kmem_free() — DDI/DKI Kernel Routine

Free previously allocated kernel memory
#include <sys/types.h>
#include <sys/kmem.h>
void kmem_free(*address***,** *bytes***)**
void ****address*; size_t** *bytes***;**

kmem_free() returns to the free pool *bytes* of memory at *address*. This memory must have been allocated by a call to **kmem_alloc()** or **kmem_zalloc()**.

See Also

DDI/DKI kernel routines, kmem_alloc(), kmem_zalloc()

Notes

kmem_free() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

kmem_zalloc() — DDI/DKI Kernel Routine

Allocate space from kernel free memory
#include <sys/types.h>
#include <sys/kmem.h>
void *kmem_zalloc(*bytes***,** *flag***)**
size_t *bytes***; int** *flag***;**

kmem_zalloc() allocates *bytes* of kernel memory and initializes them to zero.

flag indicates whether the driver can sleep while waiting for memory. **KM_SLEEP** indicates that the driver will sleep, if necessary, until the requested amount of memory is available; therefore, **kmem_zalloc()** waits until *bytes* of memory is available. **KM_NOSLEEP** indicates that it will not sleep; therefore, **kmem_zalloc()** returns NULL if the requested amount of memory is not available immediately.

kmem_zalloc() returns a pointer to the newly allocated memory. If *flag* is set to **KM_NOSLEEP** and *bytes* of memory is not available, it returns NULL. It always returns NULL if you set *bytes* to zero.

See Also

DDI/DKI kernel routines, kmem_alloc(), kmem_free()

Notes

kmem_zalloc() has base level only if *flag* equals **KM_SLEEP**; it has base or interrupt level if *flag* equals **KM_NOSLEEP**. It can sleep if *flag* is set to **KM_SLEEP**.

A driver can hold a driver-defined basic lock or read/write lock across a call to this function if *flag* is **KM_NOSLEEP**, but may not hold it if *flag* is **KM_SLEEP**. It can hold a driver-defined sleep lock regardless of the value of *flag*.

kucopy() — Internal Kernel Routine

Kernel-to-user data copy
unsigned
kucopy(*k***,** *u***,** *n***)**
char ****k*;**
char ****u*;**
unsigned *n***;**

kucopy() copies *n* bytes from offset *k* in the kernel's data segment to offset *u* in user's data segment. It returns the number of bytes copied. If an address fault occurs, **kucopy()** calls **set_user_error()** with the value **EFAULT** and returns zero.

See Also

internal kernel routines, ukcopy()

Notes

This function is equivalent to the DDI/DKI routine **copyout()**.

linkb() — DDI/DKI Kernel Routine

Concatenate two message blocks
#include <sys/stream.h>
void linkb(*first***,** *second***)**
mblk_t ****first*, ****second***;**

linkb() concatenates the message *second* onto message *first*. It sets field **b_cont** within *first* to point to *second*.

See Also

DDI/DKI kernel routines, msgb, unlinkb()

Notes

linkb() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

linkblk — STREAMS Data Structure

Structure for a STREAMS multiplexor link
#include <sys/stream.h>

The structure **linkblk** contains the information that a driver needs to establish or break a multiplexor link. It is part of the **M_DATA** portion of **M_IOCTL** messages generated the **ioctl()** calls **I_LINK**, **I_UNLINK**, **I_PLINK**, and **I_PUNLINK**. The driver does not create this structure.

A driver or module can use the following fields within **linkblk**:

queue_t *l_qtop The address of the multiplexing driver's write queue. Set this field to NULL if the link persists across a close of the driver.

queue_t *l_qbot The address of the lower stream's upper write queue. These queues' **qinit** structures are those that the driver's **streamtab** structure specifies for lower processing.

int l_index The multiplexing link in the system.

See Also

datab, DDI/DKI data structures, iocblk, msgb, qinit, streamtab

lkinfo — DDI/DKI Data Structure

DDI/DKI structure for a lock
#include <sys/ksynch.h>

Structure **lkinfo** describes a lock. Basic and read/write locks can share a **lkinfo**; however, a basic lock may not share a **lkinfo** with a sleep lock, or vice versa. A driver allocates this structure on its own.

The following fields within **lkinfo** are available to a driver:

char *lk_name The address of the lock's name. The driver must initialize this field. The name should begin with the driver's prefix.

int lk_flags Flags. As of this writing, the COHERENT implementation of STREAMS recognizes no flags. The driver must initialize this field to zero. **LOCK_ALLOC()** or **RW_ALLOC()**.

long lk_pad[2] This field is reserved for future use. The driver must initialize both array members to zero.

See Also

DDI/DKI data structures, LOCK_ALLOC(), RW_ALLOC(), SLEEP_ALLOC()

Notes

The structure **lkinfo** is defined as type **lkinfo_t**.

load — Entry-Point Routine

Routine to execute upon loading the driver into memory

Under the internal COHERENT device-driver interface, the entry point **load** gives access to the routine to execute when the driver is first loaded into memory. Its address is kept in field **c_load** of the driver's **CON** structure. The kernel executes this routine when it is booted.

See Also

CON, entry-point routines

lock() — Internal Kernel Routine

Lock a gate
#include <sys/types.h>
void lock(*g***)**
GATE *g***;**

lock() waits for the gate *g* to unlock, then locks it. When the gate of a system resource is locked, no other processes can use the resource. Gates must be in the kernel's data segment, not on the stack. Because it may call **sleep()**, **lock()** is *never* called from within an interrupt handler, block routine, deferred function, or timed function.

See Also

internal kernel routines, locked()

LOCK() — DDI/DKI Kernel Routine

Acquire a basic lock
#include <sys/types.h>
#include <sys/ksynch.h>
pl_t LOCK(*lock***,** *level***)**
lock_t **lock***; **pl_t** *level***;**

LOCK() sets the interrupt priority to *level* and acquires the lock to which *lock* points. To be portable, a driver must specify a *level* high enough to block any interrupt handler that may attempt to acquire this lock. If the lock is not available, the caller must wait until it becomes so.

LOCK() returns the previous interrupt-priority level.

See Also

DDI/DKI kernel routines, LOCK_ALLOC(), LOCK_DEALLOC(), TRYLOCK(), UNLOCK()

Notes

LOCK() has base or interrupt level. Basic locks are not recursive. A driver can hold a driver-defined sleep across a call to this function. It can also hold a driver-defined basic lock or read/write lock if it observes the priority restrictions described above.

A driver must honor the order of the lock hierarchy when it calls **LOCK()**. If it does not, it can create a deadlock. When a driver calls **LOCK()** from interrupt level, it cannot set *level* to a priority below that at which the interrupt handler is running.

LOCK_ALLOC() — DDI/DKI Kernel Routine

Allocate a basic lock
#include <sys/types.h>
#include <sys/kmem.h>
#include <sys/ksynch.h>
lock_t ***LOCK_ALLOC(***hierarchy***, minimum,** *lock_info***,** *flag***)**
uchar_t *hierarchy***; pl_t** *mininum***; lkinfo_t** ****lock_info*****; int** *flag***;**

Function **LOCK_ALLOC()** allocates a basic lock. It initializes the lock to the unlocked state.

hierarchy gives the order in which the newly created lock is to be acquired relative to other locks. It must no less than one and no greater than 32, and must be selected so that locks normally are acquired in increasing order. If your driver acquires locks at more than one level of interrupt priority and you wish it to be portable, you should organize the hierarchy among those locks such that the hierarchy increases with level of priority.

minimum gives the minimum level of priority to be used by any function that attempts to acquire this lock. Any subsequent calls to **LOCK()** to acquire the lock that **LOCK_ALLOC()** creates must pass in a priority at least as great as *minimum*.

LOCK_ALLOC() recognizes the following values for *minimum*:

plbase	Block no interrupts.
pltimeout	Block the functions scheduled by functions **itimeout()** and **dtimeout()**.
pldisk	Block disk-device interrupts.
plstr	Block STREAMS interrupts.
plhi	Block all interrupts.

The above assumes the following order of priority levels:

```
plbase < pltimeout <= pldisk, plstr <= plhi
```

The order of **pldisk** and **plstr** relative to each other is not defined.

lock_info points to a **lkinfo** structure. Fields **lk_flags** and **lk_pad** must be initialized to zero. Field **lk_name** points to the string that names the lock. Names are used only for collecting statistics. A name should begin with the driver's magic prefix; and it should be unique to the lock or group of locks for which the driver wishes to collect statistics. A given *lock_info* can be shared among multiple basic locks and read/write locks, but not between a basic lock and a sleep lock.

flag specifies whether you can sleep for the lock to be allocated, should sufficient memory not be available immediately. **KM_SLEEP** indicates that the caller will sleep if necessary; **KM_NOSLEEP** indicates that it will not.

If all goes well, **LOCK_ALLOC()** returns a pointer to the newly allocated lock. If *flag* is set to **KM_NOSLEEP** and sufficient memory is not available, it returns NULL.

See Also

DDI/DKI kernel routines, lkinfo, LOCK(), LOCK_DEALLOC(), TRYLOCK(), UNLOCK()

Notes

If *flag* is set to **KM_SLEEP**, **LOCK_ALLOC()** has base level and can sleep. If *flag* is set to **KM_NOSLEEP**, it has base or interrupt level and cannot sleep.

If *flag* is set to **KM_NOSLEEP**, a driver can hold a driver-defined basic lock or read/write lock across a call to this function. If *flag* is set to **KM_SLEEP**, it cannot. A driver can hold a driver-defined sleep lock across a call to this function regardless of the value of *flag*.

LOCK_DEALLOC() — DDI/DKI Kernel Routine

Deallocate a basic lock
#include <sys/ksynch.h>
void LOCK_DEALLOC(*lock***)**
lock_t **lock**;

LOCK_DEALLOC() deallocates the lock to which *lock* points.

See Also

DDI/DKI kernel routines, LOCK(), LOCK_ALLOC(), TRYLOCK(), UNLOCK()

Notes

LOCK_DEALLOC() has base or interrupt level. It does not sleep.

Do not attempt to deallocate a lock that is locked or is awaited. Doing so triggers undefined behavior.

A driver can hold a driver-defined basic lock (other than the one being deallocated), read/write lock, or sleep lock across a call to this function.

locked() — Internal Kernel Routine

See if a gate is locked
#include <sys/proc.h>
#include <sys/types.h>
int locked(*gate***)**
GATE *gate*;

the macro **locked()** determines if *gate* is locked.

See Also

internal kernel routines, lock()

major() — Internal Kernel Routine

Extract major-device number
#include <sys/stat.h>
#include <sys/types.h>
int major(*dev***)**
dev_t *dev*;

The macro **major()** returns a device's major number.

See Also

internal kernel routine

makedevice() — DDI/DKI Kernel Routine

Make a device number
#include <sys/types.h>
#include <sys/ddi.h>
dev_t makedevice(*major***, minor)**
major_t *major*; **minor_t** *minor*;

makedevice() creates a device number from the external *major* and *minor* device numbers. It returns the device number, which contains both *major* and *minor*.

See Also

DDI/DKI kernel routines, getemajor(), geteminor(), getmajor(), getminor()

Notes

makedevice() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

makedevice() does not validate *major* or *minor*. *Caveat utilitor!*

map_pv() — Internal Kernel Routine

Map physical to virtual addresses
vaddr_t map_pv(*paddr***,** *len***)**
paddr_t *paddr*;
fsize_t *len*;

map_pv() initializes a virtual address to access physical memory at location *paddr* of size *len* bytes. It provides read and write (but not execute) access. When no longer required, a virtual address should be released by invoking **unmap_pv()**.

See Also

internel kernel routines

MAPIO() — Internal Kernel Routine

Return global address
#include <sys/mmu.h>
int MAPIO(*table_address***,** *offset***)**
SEG ******table_address*; **int** *offset*;

The macro **MAPIO()** passes an absolute page-table address for a segment and an offset into the segment, and returns the system global address of the desired location. It is used when a region of memory is available in a mapped segment, but could be unmapped later when it is needed. A driver uses the corresponding system global address range to refer to the memory whether or not the segment that contains it is mapped into virtual space.

See Also

internal kernel routines, salloc()

mapPhysUser() — Internal Kernel Routine

Overlay user data with memory-mapped hardware
int mapPhysUser(*virtAddr***,** *physAddr***,** *numBytes***)**
int *virtAddr*, *physAddr*, *numBytes*;

mapPhysUser() mapped the virtual address *virtAddr* into the user data-page table for the current process at address *physAddr*, for the byte count of *numBytes*.

This function must observe the following restrictions:

1. Addresses *virtAddr* and *physAddr* must be aligned to four-kilobyte boundaries, i.e., must be multiples of 4,096. This is a consequence of the 386/486 paging hardware.
2. Addresses from *virtAddr* to *virtAddr* plus *numBytes* minus one must be in the user data segment (*not* within the user stack).

3. Addresses from *physAddr* to *physAddr* plus *numBytes* minus one must be either all outside of installed RAM or all inside the range of physical addresses within the **PHYS_MEM** pool.

If any of these conditions are not met, **mapPhysUser()** does nothing. Otherwise, all subsequent access to the specified range of user addresses goes to the physical range requested.

mapPhysUser() returns zero if any of the above requirements is not met; otherwise, it returns one.

See Also

internal kernel routines

Notes

numbytes must be an exact multiple of four kilobytes (4,096 bytes). If it is not, COHERENT rounds it up to the next multiple of four kilobytes.

mdevice — System Administration

Describe drivers that can be linked into kernel
/etc/conf/mdevice

File **mdevice** describes each device driver that can be linked into the COHERENT kernel. The command **idmkcoh** uses the information in this file when it builds and configures a new kernel.

mdevice contains one line for each driver or kernel component that can be linked into the kernel. Each line, in turn, consists of ten fields. The following describes the ten fields in order, from left to right:

1. *Name*
This field gives the name of the driver or component. Each name must uniquely identify the driver or component within the kernel. This field cannot be longer than eight characters.

2. *Function Flags*
This field holds a flag for each function (that is, entry point) within the driver or component. This field is used only by drivers or components that use the STREAMS or DDI/DKI interfaces; drivers that use the internal-kernel interface should place a hyphen '-' in this field. The legal flags are as follows:

o	Open
c	Close
r	Read
w	Write
i	Ioctl
s	Startup
x	Exit
I	Init
h	Halt
p	Poll — that is, **chpoll()**

3. *Miscellaneous Flags*
These flags give information about the device. They are set by most varieties of driver; the only exception is a STREAMS driver, for which only the flag **S** matters. The legal flags are as follows:

c	Character device
b	Block device
f	Driver conforms to the DDI/DKI
o	Driver has only one entry in **/etc/conf/sdevice**
r	Driver is required in all configurations of the kernel
S	STREAMS module; or STREAMS device when used with the 'c' flag
H	Device driver controls hardware
C	Driver uses interal-kernel (**CON**) interface

Note that the 'C' flag is unique to COHERENT, and cannot be used under other operating systems.

4. *Code Prefix*
This gives the "magic prefix" by which the kernel identifies the entry-point routines for this driver. In most instances, this is identical with the driver's name.

5. *Block Major-Device Number*
This gives the major-device number of this driver when it is accessed in block mode. In most instances, this and the following field are identical.

6. *Character Major-Device Number*
This gives the major-device number of this driver when it is accessed in character (raw) mode. In most instances, this and the preceding field are identical.

7. *Minor Device Numbers, Minimum*
This gives the smallest number that can be held by a minor-device number under this driver. Most drivers set this field to the lowest legal value, which is zero.

8. *Minor Device Numbers, Maximum*
This gives the largest number that can be held by a minor-device number under this driver. Most drivers set this field to the highest legal value, which is 255.

9. *DMA Channel*
This gives the DMA channel by which the device is accessed. Under COHERENT, this is always set to -1.

10. *CPU ID*
This gives the CPU that controls this driver, should the driver be running in a multiprocessor environment and be dedicated to a particular processor. Under COHERENT, this is always set to -1.

For an example of modifying this file, see the entry for **device drivers**.

Example

The following gives some example entries from **mdevice**:

```
1     2     3     4     5     6     7     8     9     10
###
# Example of an kernel components:  floating-point emulator and STREAMS
###
em87  -     -     em87  0     0     0     0     -1    -1
streams     I     -     streams     0     0     0     0-1 -1

###
# Example of a STREAMS driver:  note flags 'c' and 'S' both set in field 3
###
echo  -     cSf   echo  0     33    0     255   -1    -1

###
# Example DDI/DKI character driver:  Note that field 2 is initialized.
###
trace ocriI cfo   tr    0     34    0     255   -1    -1

###
# Example IK driver:  Note flag 'C' in field 3
###
at    -     CGHo  at    11    11    0     255   -1    -1
```

See Also

Administering COHERENT, device drivers, idmkcoh, mtune, sdevice, stune

messages — Technical Information

Types of STREAMS messages
#include <stream.h>

The following lists the types of STREAMS messages that drivers can use. **M_DATA** is a data message. An asterisk '*' indicates that the message is a control message with normal priority. A dagger '†' indicates that the message is a control message with high priority.

M_BREAK* Generate a line break.
M_COPYIN† Copy data from the user to a STREAMS message during transparent **ioctl**.
M_COPYOUT† Copy data from a STREAMS message to the user during transparent **ioctl**.
M_CTL* Control message used between neighboring modules and drivers.
M_DATA Data message.

M_DELAY*	Generate a real-time delay.
M_ERROR†	The stream has incurred a fatal error.
M_FLUSH†	Flush queue.
M_HANGUP†	The device has been disconnected.
M_IOCACK†	An **ioctl** request has succeeded
M_IOCNAK†	An **ioctl** request has failed.
M_IOCDATA†	The status and data of a previous **M_COPYIN** or **M_COPYOUT** request during a transparent **ioctl** request.
M_IOCTL*	A user has made an **ioctl** request.
M_PCCTL†	Message passed between neighboring drivers.
M_PCPROTO*	Protocol control message.
M_PCSETOPTS*	Set stream-head options.
M_PCSIG*	Send a signal to a process.
M_PROTO†	Protocol message.
M_READ†	Indicate the occurrence of a **read** routine when the stream head's read queue has no data.
M_SETOPTS†	Set stream-head options.
M_SIG†	Send a signal to a process.
M_START†	Output can be restarted.
M_STARTI†	Input can be restarted.
M_STOP†	Stop output immediately.
M_STOPI†	Stop input immediately.

See Also

technical information

minor() — Internal Kernel Routine

Extract minor-device number

#include <sys/stat.h>
int minor(*dev***)**
dev_t *dev***;**

The macro **minor()** returns a device's minor number.

See Also

internal kernel routines

mmap — Entry-Point Routine

Check virtual mapping for a memory-mapped device

#include <sys/types.h>
#include <sys/mman.h>
#include <sys/vm.h>
int *prefix***mmap(***device***,** *offset***,** *protection***)**
dev_t *device***; off_t** *offset***; int** *protection***;**

mmap is the entry point to the driver's internal routine for memory-mapped devices. The COHERENT system call **mmap()**, when applied to a character-special file, maps this device's memory into user space, for direct manipulation by the user's application.

device gives the device whose memory is being mapped. *offset* gives the offset within device memory at which mapping begins. *protection* gives the protection flags. The following flags are Valid:

PROT_READ	Page can be read
PROT_WRITE	Page can be written on
PROT_EXEC	Page can be executed
PROT_USER	Page is accessible from user level
PROT_ALL	All of the above

The **mmap** routine must check whether *offset* is within the range that the device supports. If the offset does not exist, then it should return **NOPAGE**.

If *offset* exists, the **mmap** routine returns its physical-page identifier.

See Also

entry-point routines

Notes

This entry-point is used only by the DDI/DKI interface. It is optional.

The driver's **mmap** routine has user context and can sleep.

As of this writing, the COHERENT kernel does not contain the system call **mmap()**.

module_info — STREAMS Data Structure

Information about a STREAMS driver or module
#include <sys/conf.h>
#include <sys/stream.h>
#include <sys/types.h>

Structure **module_info** holds the identification data and limits that with which a queue is initialized. The kernel creates one **module_info** for each driver or module, and initializes it to the values that the driver or module requires. A driver or module can have separate **module_info** structures for its read queues and write queues, or it can use the same **module_info** for both.

A driver can view the following fields within **module_info**:

ushort_t mi_idnum — A number that uniquely identifies the driver or module.

char *mi_idname — The address of the name of the driver or module. A name cannot have more than **FMNAMESZ** characters, not including the terminating NUL. At present, this constant is set to eight.

long mi_minpsz — The minimum size of a message packet.

long mi_maxpsz — The maximum size of a message packet.

ulong_t mi_hiwat — The default high-water mark for the queue. If the queue's messages together hold more than this number of bytes of data, the queue is declared to be full and is flow-controlled.

ulong_t mi_lowat — The default low-water mark for the queue. If the queue's messages together hold fewer than this number of bytes of data, the queue is declared not to be full and is not flow-controlled.

module_info is read-only; however, its fields **mi_minpsz**, **mi_maxpsz**, **mi_hiwat**, and **mi_lowat** can be copied into a **queue** structure, where they can be modified.

See Also

DDI/DKI data structures, queue

msgb — STREAMS Data Structure

Structure of a STREAMS message block
#include <sys/types.h>
#include <sys/stream.h>

A STREAMS message consists of one or message blocks. A message block, in turn, is referenced by a pointer to the structure **msgb**. The functions **allocb()** and **esballoc()** automatically create a **msgb** when they allocate a message; note that this structure must be created *only* by those functions.

The following fields within **msgb** can be read by drivers and modules:

struct msgb *b_next
struct msgb *b_prev — The addresses of, respectively, the next and previous blocks of the message queue. These fields bind a queue's messages into a link list, and they bind a message onto its queue.

struct msgb *b_cont — The address of the next message block within the message. This field is initialized when a message consists of more than one message block.

uchar_t *b_rptr

uchar_t *b_wptr The addresses of, respectively, the first unread byte within the data buffer associated with this message, and the next byte to be written into the data buffer. These fields together define the message's region within the associated data buffer.

struct datab *b_datap The address of this message's data block. *Never* change this field!

uchar_t b_band The message's priority band. In normal- and high-priority messages, **b_band** is set to zero. Use the value of this field to position the message within its queue: the higher its priority band (that is, the lower the value of this field), the closer to the head of the queue it should be.

ushort_t b_flag A bitmask of flags that control how the stream head processes the message. At present, the only valid flag is **MSGMARK**, which indicates that the last byte in the message is "marked".

When a message is on a queue, all fields within its **msgb** are read-only.

See Also

allocb(), datab, DDI/DKI data structures, esballoc(), free_rtn, freeb(), getq(), messages, rmvq()

Notes

The structure **msgb** is defined as type **mblk_t**.

msgdsize() — DDI/DKI Kernel Routine

Get the number of bytes of data that a message holds
#include <sys/stream.h>
int msgdsize(*message***)**
mblk_t **message***;*

msgdsize() counts and returns the bytes of data within *message*. It counts only the data within message blocks of type **M_DATA**.

See Also

DDI/DKI kernel routines, msgb

Notes

msgdsize has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

msgpullup() — DDI/DKI Kernel Routine

Copy message data into a new message
#include <sys/stream.h>
mblk_t ***msgpullup(***message***,** *bytes***)**
mblk_t ***message***; int** *bytes***;**

msgpullup() copies into a new message the first *bytes* of data from *message*. If *bytes* equals -1, it copies all data from *message*. **msgpullup()** does not affect *message* in any way.

If *message* does not contain *bytes* of the same message type, **msgpullup()** fails and returns NULL. Otherwise, it returns the address of the new message.

See Also

allocb(), DDI/DKI kernel routines, msgb

Notes

msgpullup() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

mtune — System Administration

Define tunable kernel variables
/etc/conf/mtune

File **mtune** defines all of the tunable variables within the kernel. These variables let you configure some aspects of your kernel, without having to modify the kernel's drivers or recompile the kernel.

Command **idmkcoh** reads this file when it builds a new kernel, and uses its contents to help patch the newly build kernel. A **mkdev** script (kept in a subdirectory of **/etc/conf**) also sets appropriate variables within this file, based on your answers to its questions.

Each line within **mtune** defines one tunable parameter. A line consists of four fields, as follows:

1. *Name*
This field names the parameter. It cannot exceed 20 characters.

2. *Minimum Value*
The legal minimum value of this parameter.

3. *Default Value*
The default value for this parameter. This value can be overridden by an entry in file **/etc/conf/stune**.

4. *Maximum Value*
The legal maximum value of this parameter.

Note that under UNIX System V, fields 2 and 3 are reversed. A line that begins with the pound sign '#' is a comment, and is ignored by **idmkcoh** when it builds a new kernel.

For details on the parameters that this file sets, read the comments within this file.

See Also

Administering COHERENT, device drivers, mdevice, sdevice, stune

Notes

mtune contains comments that describe the kernel variables that you can tune. If you wish to tune the kernel, you should read this file and modify it appropriately. The variables are documented in this file rather than in the COHERENT manual to ensure that you have exactly accurate information about the variables that reside in the version of the kernel on your system.

noenable() — DDI/DKI Kernel Routine

Stop scheduling of a queue service routine
#include <sys/stream.h>
void noenable(*queue***)**
queue_t ******queue***;**

noenable() stops functions **insq()**, **putbq()**, and **putq()** from scheduling the service routine of *queue* when they enqueue a message that does not have high priority. **noenable()** does not stop these functions from enabling *queue*'s service routine when they enqueue a high-priority message, nor does it stop the service routine from being enabled when a function calls **qenable()** explicitly.

See Also

DDI/DKI kernel routines, enableok(), insq(), putbq(), putq(), qenable(), queue

Notes

noenable() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

The caller cannot have frozen the stream when it calls **noenable()**.

To undo the action of **noenable()**, call function **enableok()**.

nondsig() — Internal Kernel Routine

Non-default signal pending
int nondsig()

nondsig() returns the signal number if the current process has a non-ignored signal. If there are no non-ignored signals, **nondsig()** returns zero.

See Also

internal kernel routines

nonedev() — Internal Kernel Routine

Illegal device request
void nonedev()

nonedev() calls function **set_user_error()** with value. **ENXIO**. This function is placed in the configuration table to provide a routine that sets this error status. It does not return anything useful.

See Also

internal kernel routines

nulldev() — Internal Kernel Routine

Ignored device request
int nulldev()

The function **nulldev()** does nothing. It is placed in the configuration table to supply something to call when a function is required to do nothing. **nulldev()** returns nothing useful.

See Also

internal kernel routines

open — Entry-Point Routine

Open a device

Internal Kernel Interface:
#include <sys/cred.h>
#include <sys/errno.h>
#include <sys/file.h>
#include <sys/open.h>
#include <sys/types.h>
int *prefix***open(***device*, *mode*, *flags*, *credentials*, *inodep***)**
dev_t *device*; **int** *mode*, *flags*; **cred_t** **credentials*; **struct inode** **inodep*;

DDI/DKI:
#include <sys/cred.h>
#include <sys/errno.h>
#include <sys/file.h>
#include <sys/open.h>
#include <sys/types.h>
int *prefix***open(***device*, *oflag*, *otype*, *credentials***)**
dev_t **device*; **int** *oflag*, *otype*; **cred_t** **credentials*;

STREAMS:
#include <sys/cred.h>
#include <sys/errno.h>
#include <sys/file.h>
#include <sys/stream.h>
#include <sys/types.h>
int *prefix***open(***queue*, *device*, *oflag*, *sflag*, *credentials***)**
queue_t **queue*; **dev_t** **device*; **int** *oflag*, *sflag*; **cred_t** **credentials*;

A driver's **open** routine prepares a device to be manipulated. Every driver must have this entry point. An application invokes it via the COHERENT system call **open()**. For details on this system call, see its entry in the COHERENT Lexicon.

LEXICON

The **open** routine returns zero on success, or an appropriate error number. See the entry for **errno** in this manual for a list of error numbers. The driver determines how to react to an error.

The following describes the **open** routine for each flavor of driver-kernel interface.

Internal-Kernel Interface

Under the internal-kernel interface to a driver, field **c_open** in the driver's **CON** structure holds the address of this routine. It is customary to name the **open** routine with the word **open** prefixed by a unique identifier for your driver; but this is not required.

device is a **dev_t** that identifies the device to be opened.

mode and *flags* give, respectively, the mode into which *device* should be opened, and additional information about how it should be opened. See the article for the system call **open()** in the COHERENT Lexicon for a table of the legal values of these arguments.

credentials points to the credentials of the current user. If it wishes, your driver can read this structure to check the user's permissions before it opens *device*. Note that many drivers do not use this argument.

Finally, *inodep* points to a structure that holds information about the i-node that is current being manipulated. Note that many drivers do not use this argument.

DDI/DKI Interface

To invoke the **open** routine under the DDI/DKI interface, the kernel calls the function *prefix***open()**, where *prefix* is the unique prefix for this driver. The function takes the following arguments:

device The address of the **dev_t** structure that identifies this device.

oflag Flag that indicate how *device* is to be opened. The **open** routine must recognize the following values:

FEXCL Open the device in a driver-dependent manner.

FNDELAY
Open the device and return immediately, even if a problem occurs.

FNONBLOCK
Same as **FNDELAY**.

FREAD Open *device* for reading.

FWRITE
Open *device* for writing.

otype The type of interface into which *device* is to be opened. The **open** routine can recognize the following values:

OTYP_BLK
Open into a block interface.

OTYP_CHR
Open into a character (raw) interface.

OTYP_LYR
Open into a layered interface.

credentials
The address of the user's credentials.

The **open** routine has user context and can sleep.

STREAMS Interface

To invoke the **open** routine under the STREAMS interface, the kernel calls the function *prefix***open()**, where *prefix* is the unique prefix for this driver. The function takes the following arguments:

queue The address of the queue into the driver's read side.

device The address of the **dev_t** structure that identifies this device. For modules, *device* identifies the device of the driver that is at the end of the stream.

oflag Flag that indicate how *device* is to be opened. The **open** routine must recognize the same values as those given above for the DDI/DKI interface.

sflag STREAMS flag. The **open** routine must recognize the following values:

CLONEOPEN
A "clone" open. If the driver supports cloning, it must alter the value of *device* to that of the "clone" device. Cloning is discussed below.

MODOPEN
Open a module, not a driver.

0 Open *device* directly, without cloning.

credentials
The address of the user's credentials.

Before it returns, the **open** routine of a STREAMS driver or module must call **qprocson()** to enable its **put** and **srv** routines — but only after it has allocated and intialized all resources upon which the **put** and **service** routines depend.

Only one instance of the **open** routine can be running at any given time for a given queue. The **open** routine has user context and can sleep.

Cloning

Cloning is the process by which a driver selects an unused device for a user. This spares the user the bother of polling many devices as he looks for one that is not being used. When an applications opens the clone driver, that driver calls the **open** routine of the real driver with *sflag* set to **CLONEOPEN**. This spares a driver from having to reserve special minor numbers as entry points for clones.

There are two common methods of cloning. The first does not use the flag **CLONEOPEN**; the second does.

1. The driver reserves special minor numbers for clones. When a user opens one of these, the driver searches for an unused device and, if it finds one, sets *device* to identify the unused device it has discovered.
2. When it sees the flag **CLONEOPEN**, the driver can invoke a special "cloning" driver to find and open a clone device. Here, *device*'s major number identifies the cloning driver, and its minor number the driver to be cloned.

Multiple clone opens can run concurrently. Support of cloning is optional, so a driver should react rationally if does not support cloning but is asked to clone a driver.

See Also

close, CON, drv_priv(), entry-point routines, errno, qprocson(), queue
COHERENT Lexicon: **open()**

OTHERQ() — DDI/DKI Kernel Routine

Get the other queue
#include <sys/ddi.k>
#include <sys/stream.h>
queue_t * OTHERQ(*queue***)**
queue_t ******queue***;***

OTHERQ() returns the address of *queue*'s partner queue. If *queue* points to the write queue, **OTHERQ()** returns a pointer to the read queue, and vice versa.

See Also

DDI/DKI kernel routines, RD(), WR()

Notes

OTHERQ() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

outb() — DDI/DKI Kernel Routine

Output a byte to an I/O port
#include <sys/types.h>
void outb(*port***,** *c***)**
int *port***; uchar_t** *c***;**

outb() writes byte *c* to *port*.

See Also

DDI/DKI kernel routines, inb(), inl(), inw(), outl(), outw()

Notes

outb() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

This function is meaningful only on implementations that support I/O-mapped I/O.

outl() — DDI/DKI Kernel Routine

Write a long integer to an I/O port
#include <sys/types.h>
void outl(*port***,** *datum***)**
int *port***; ulong_t** *datum***;**

outl() writes the unsigned long (32-bit) integer *datum* to *port*.

See Also

DDI/DKI kernel routines, inb(), inl(), inw(), outb(), outw()

Notes

outl() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

This function is meaningful only on implementations that support I/O-mapped I/O.

outw() — DDI/DKI Kernel Routine

Output a short integer (word) to an I/O port
#include <sys/types.h>
void outw(*port***,** *datum***)**
int *port***; ushort_t** *datum***;**

outw() writes the short (16-bit) integer *datum* to *port*.

See Also

DDI/DKI kernel routines, inb(), inl(), inw(), outb(), outl()

Notes

outw() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

This function is meaningful only on implementations that support I/O-mapped I/O.

P2P() — Internal Kernel Routine

Convert system global to physical address
P2P(*gl_addr***)**
vaddr_t *gl_addr***;**

Macro **P2P** converts a system global address to a physical address. For example, the code sequence

```
#include <sys/mmu.h>
phys_addr = P2P(sys_gl_addr);
```

converts system global address *sys_gl_addr* and stores it into variable *phys_addr*.

See Also

internal kernel routines

panic() — Internal Kernel Routine

Fatal system error
void panic(*format***,** *arg***, ...)**
char **format***;*

panic() prints an error message and halts the system. Normally, it is called only when a catastrophic event occurs.

format gives formatting information for the error message, accompanied by zero or more *arg* arguments. Syntax for *format* is the same as for the kernel function **printf()**.

See Also

internal kernel routines, printf()

pcmsg() — DDI/DKI Kernel Routine

Test if a message type indicates high priority
#include <sys/ddi.h>
#include <sys/stream.h>
#include <sys/types.h>
int pcmsg(*type***)**
uchar_t *type***;**

pcmsg() tests the message *type* to see if it indicates high priority. It returns one if does, or zero if it does not. Field **datab.db_type** holds a message's type.

See Also

allocb(), datab, DDI/DKI kernel routines, msgb, messages

Notes

pcmsg() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

phalloc() — DDI/DKI Kernel Routine

Create a pollhead structure
#include <sys/kmem.h>
#include <sys/poll.h>
struct pollhead ***phalloc(***flag***)**
int *flag***;**

phalloc() allocates and initializes structure of type **pollhead**. A character driver that uses the DDI/DKI interface can use **pollhead** to support polling.

The calling routine should use **flag** to indicate whether it will sleep. Setting *flag* to **KM_SLEEP** indicates that if not enough memory is available to allocate a **pollhead** structure, the caller will sleep until memory becomes available. Setting *flag* to **KM_NOSLEEP** indicates that the caller will not sleep; if not enough memory is available to allocate a **pollhead** structure, **phalloc()** immediately returns NULL.

See Also

DDI/DKI kernel routines, phfree(), pollwakeup()

Notes

If *flag* equals **KM_SLEEP**, **phalloc()** has base level only and can sleep; if *flag* equals **KM_NOSLEEP**, it has base or interrupt level and does not sleep.

A driver can hold a driver-defined basic lock or read/write lock across a call to this function if *flag* equals **KM_NOSLEEP**; it cannot hold such locks if *flag* equals **KM_SLEEP**.

Driver-defined sleep locks may be held across calls to this function regardless of the value of *flag*.

phfree() — DDI/DKI Kernel Routine

Free a pollhead structure
#include <sys/poll.h>
void phfree(*head***)**
struct pollhead **head***;*

phfree() frees the **pollhead** structure *head*, which must have been allocated by the function **phalloc()**.

See Also

DDI/DKI kernel routines, phalloc(), pollwakeup()

Notes

phfree() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

physiock() — DDI/DKI Kernel Routine

Request and validate raw I/O
#include <sys/buf.h>
#include <sys/types.h>
#include <sys/types.h>
int physiock(*strategy***,** *buf_ptr***,** *device***,** *rwflag***,** *blocks***,** *uio_ptr***)**
int (*strategy***);** **buf_t** **buf_ptr***;* **dev_t** *device***;**
int *rwflag***;** **daddr_t** *blocks***;** **IO** **uio_ptr***;*

physiock() uses a buffer header to perform unbuffered I/O. Thus, it provides block drivers with a character (or "raw") interface to a device.

When it executes a request for raw I/O, **physiock()** performs the following tasks:

1. Check whether the request runs to or past the end of the device. If a read request runs past the end of the device or a write request runs to or past the end of the device, the request is invalid and **physiock()** rejects it. See the description of return values, below, for details on how it handles this situation.
2. Set up a buffer header to describe the I/O task. For details, see the Lexicon entry for the function **getrbuf()**.
3. Lock pages so they cannot be swapped out of memory. (NB, this currently does not apply to COHERENT, but will when demand paging is added to its kernel.)
3. Call the driver's **strategy** routine.
4. Sleep until the transfer is complete. It awakens when the I/O-done handler calls **biodone()** to awaken it. (For details on the I/O-done handler, see the Lexicon entry for **getrbuf()**.)
5. Update various structures where necessary, tidy up memory, and return.

physiock() takes the following parameters:

strategy
The address of the driver's strategy routine.

buf_ptr The address of the instance of type **buf** that describes the I/O request. If this is initialized to NULL, **physiock()** allocates a buffer and a buffer header from the buffer pool, then frees them after the I/O request has been executed.

device The number of the external device with which I/O is to be performed.

rwflag The direction of I/O. If it is set to **B_READ**, the data moves from the kernel into the user's buffer; if to **B_WRITE**, then the data moves in the opposite direction.

blocks The number of blocks that a logical device can support. One block equals **NBPSCTR** bytes, as defined by header file
.BR <sys/param.h> . Note that for some devices (e.g., tape devices), this should be set to an arbitrarily large value.

uio_ptr The address of the instance of type **IO** that defines the user space which the I/O procedure is to use. Note that under UNIX's implementation of the DDI/DKI, this argument has type **uio_t**.

Return Values

physiock() returns zero if the I/O executed without trouble or if it read data at the end of a device. It returns a value other than zero if any of the following conditions occurred:

- Only a partial transfer of data occurred. **physiock()** updates the **uio** instance to which *uio_ptr* points to reflect this partial transfer, and returns a non-zero value.
- The I/O request attempts to read beyond the end of this device, or write at or beyond the end of a device. **physiock()** returns **ENXIO**.
- The user memory to which *uio_ptr* points is not valid. **physiock()** returns **EFAULT**.
- **physiock()** could not lock pages for DMA. It returns **EAGAIN**.

See Also

DDI/DKI kernel routines, freerbuf(), getrbuf(), strategy

Notes

physiock() has base level. It can sleep.

A function cannot hold a basic lock or read/write lock across a call to this function. It can, however, hold a sleep lock.

poll — Entry-Point Routine

Poll the device
int *prefix***poll(***device***,** *events***,** *msec***,** *private***)**
device_t *device***; int** *events***,** *msec***; void** **private***;*

Under the internal COHERENT device-driver interface, the entry point **poll** gives access to the driver's routine for polling the device. The address of this routine is given in field **c_poll** of the driver's **CON** structure.

By convention, the **poll** routine is named with the word **poll** prefixed with your driver's unique prefix. This, however, is not required.

device identifies the device to be polled.

events gives the number of events to be polled.

msec gives the number of seconds to wait before the call times out.

Finally, *private* points to a data item that is private to this driver. Note that most drivers do not use this argument.

See Also

CON, entry-point routines

pollhead — STREAMS Data Structure

Structure for a STREAMS poll head
#include <sys/poll.h>

The structure **pollhead** is used in System V polling. This structure is meant to be totally opaque; no access to its internal structure is permitted.

A driver must provide one **pollhead** structure for each minor device that it supports. It must call **phalloc()** to allocate the structure; it must call **phfree()** to free the structure once it is no longer needed. A DDI/DKI driver can use a **pollhead** structure only if **phalloc()** has allocated and initialized it.

The driver entry point **chpoll** gives access to the driver's polling routine.

See Also

chpoll, DDI/DKI data structures, phalloc(), phfree()

pollopen() — Internal Kernel Routine

Initiate driver polled event
void pollopen(*eventp***)**
event_t **eventp*;**

pollopen() creates a polled event on the event structure to which *eventp* points. The event structure must reside in static kernel data space.

See Also

internal kernel routines

pollwake() — Internal Kernel Routine

Terminate driver polled event
#include <sys/types.h>
void pollwake(*eventp***)**
event_t **eventp*;**

pollwake() generates a polled event report on the event structure pointed to by *eventp*. The event structure must reside in static kernel data space. If the field

*eventp***->e_eprocp**

is NULL, no events are still pending and the driver has no need to call **pollwake()**.

See Also

internal kernel routines

pollwakeup() — DDI/DKI Kernel Routine

Inform polling process that an event has occurred
#include <sys/poll.h>
void pollwakeup(*head***,** *event***)**
struct pollhead **head*; **short** *event*;**

pollwakeup() notifies all processes that are polling for *event*. It should be used only by drivers that use the DDI/DKI interface. A driver should call this function for each occurrence *event*. A process can register for notification by invoking the COHERENT system call **poll()**.

head points to the **pollhead** structure for for the minor device. It must have been allocated with the function **phalloc()**.

See Also

DDI/DKI kernel routines, phalloc(), phfree()
COHERENT Lexicon: **poll()**

Notes

pollwakeup() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

power — Entry-Point Routine

Routine to execute if power fails
int *prefix***power(***device***)**
dev_t *device*;**

Under the internal COHERENT device-driver interface, the entry point **power** points to the routine to execute if power fails. The address of this routine is given in field **c_power** of the driver's **CON** structure.

device identifies the device being manipulated.

See Also

CON, entry-point routines

print — Entry-Point Routine

Print a message on the system's console
#include <sys/types.h>
#include <sys/errno.h>
int *prefix* **(***device***,** *message***)**
dev_t *device***; char** ******message***;**

The **print** routine prints *message* on the system's console, plus any text the driver itself cares to add. The kernel invokes this routine when something has gone wrong with the block *device*.

The driver should call the function **cmn_err()** to display its own message.

See Also

cmn_err(), entry-point routines, errno
COHERENT Lexicon: **errno.h, printf()**

Notes

This entry point is used only by the DDI/DKI interface. It is optional.

The driver should not interpret the string text passed to it.

The **print** routine must not call any routine that sleeps.

printf() — Internal Kernel Routine

Formatted print
void printf(*format***,** *arg***, ...)**
char ******format***;**

printf() is a kernel routine that offers a simplified version of the function found in the standard C library. This version recognizes the formatting conversions **%**, **c**, **d**, **o**, **p**, **r**, **s**, **u**, **x**, **D**, **O**, **U**, and **X**. It also recognizes the length modifier **l**. It does not recognize left justification, field widths, or zero padding. For details on each conversion specification, see the Lexicon entry for **printf()** in the COHERENT Lexicon.

See Also

internal kernel routines
COHERENT Lexicon: **printf()**

Notes

Unlike the library version of this function, the kernel version of **printf()** is synchronous; that is, it does not wait until the next context switch before it prints your message.

This function does much the same work as the DDI/DKI routine **cmn_err()**.

proc_ref() — DDI/DKI Kernel Routine

Identify a process
#include <sys/types.h>
void ******proc_ref* **();**

proc_ref() returns a pointer to the process in whose context the driver is running. A non-STREAMS character driver can pass the value returned by **proc_ref()** to **proc_signal()** to signal that process, or to **proc_unref()** to un-reference this value. There is no other use for this value.

See Also

DDI/DKI kernel routines, proc_signal(), proc_unref()

Notes

proc_ref() has base level only. It requires user context. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

proc_signal() — DDI/DKI Kernel Routine

Send a signal to a process
#include <sys/types.h>
#include <sys/signal.h>
int proc_signal(*process***,** *signal***)**
void **process***; **int** *signal***;**

proc_signal() sends *signal* to *process*, which must have been obtained by a call to **proc_ref()**. *signal* can be one of the following:

SIGHUP	The device has "hung up".
SIGINT	The driver has received an interrupt.
SIGQUIT	The driver has received the quit character.
SIGWINCH	The size of a window has changed.
SIGURG	Urgent data are waiting.
SIGPOLL	A pollable event has occurred.

If *process* still exists, **proc_signal()** sends sends it *signal* and returns zero. However, if it has exited, **proc_signal()** does nothing and returns -1.

See Also

DDI/DKI kernel routines, proc_ref(), proc_unref(), signals

Notes

proc_signal() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

A STREAMS driver or module should not use **proc_signal()**. Rather, it should send a message of type **M_SIG** or **M_PCSIG** to the stream head. A driver must not use **proc_signal()** to send **SIGTSTP**.

proc_signal() interrupts any process that is blocked in **SV_WAIT_SIG()** or **SLEEP_LOCK_SIG()**. In most cases, this causes these functions to return prematurely.

pullupmsg() — DDI/DKI Kernel Routine

Concatenate bytes in a message
#include <sys/stream.h>
int pullupmsg(*message***,** *bytes***)**
mblk_t **message***; **int** *bytes***;**

pullupmsg() concatenates and aligns the first *bytes* of *message.* If *bytes* equals -1, it concatenates all data. If it cannot find *bytes* of the same message type, it fails and returns zero; otherwise, it returns the number of characters concatenated.

See Also

allocb(), DDI/DKI kernel routines, msgpullup()

Notes

pullupmsg() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a calls to this function.

This function is provided for compatibility with obsolete versions of the DDI/DKI. You should instead use the function **msgpullup()**.

put — Entry-Point Routine

Receive a message from a queue
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/types.h>

Read Side:
int *prefix***rput(***queue***,** *message***)**
queue_t *queue***; mblk_t** *message***;**

Write Side:
int *prefix***wput(***queue***,** *message***)**
queue_t *queue***; mblk_t** *message***;**

The **put** routine passes *message* onto *queue*.

No driver or module can call a **put** routine directly; rather, it should use the function **put()** to invoke it.

A **put** routine is designated *read* or *write*, depending upon the direction of message flow. A module or driver must have a write **put** routine. A module must have a read **put** routine, but a driver is not required to have it because its interrupt handler can do that work instead.

See Also

datab, entry-point routines, flushband(), flushq(), msgb, putctl(), putctl1(), putnext(), putq(), qreply(), queue, srv

Notes

This entry point is used only by the DDI/DKI interface. Under this interface, it is required.

A **put** routine does not have user context, and therefore cannot call any function that sleeps.

No locks should be held when passing messages to other queues in the stream.

Multiple copies of the same **put** routine for a given queue, as well as the service routine for the queue, can be running concurrently. Drivers and modules must synchronize access to their private data structures accordingly.

DDI/DKI drivers cannot call **put** procedures directly. They must now call the appropriate STREAMS utility function, e.g., **put()**, **putnext()**, **putctl()**, **putnextctl()**, or **qreply()**.

put() — DDI/DKI Kernel Routine

Call a put procedure
#include <sys/stream.h>
void put(*queue***,** *message***);**
queue_t **queue***; **mblk_t** **message***;*

put() calls the driver's **put** procedure for *queue*. *message* points to the message being passed.

See Also

DDI/DKI kernel routines, putctl(), putctl1(), putnext(), putnextctl(), putnextctl1(), qreply(), query

Notes

put() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

The caller cannot have the stream frozen when it calls this function.

putbq() — DDI/DKI Kernel Routine

Place a message at the head of a queue
#include <sys/stream.h>
int putbq(*queue***,** *message***)**
queue_t **queue***, **mblk_t** **message***;*

putbq() places *message* at the head of *queue*. If *queue* contains a message whose priority exceeds that of *message*, **putbq()** places *message* at the head of the appropriate priority band. A driver usually calls **putbq()** when **bcanputnext()** or **canputnext()** indicates that *message* cannot be passed to the next stream component.

putbq() updates all flow-control parameters. It schedules *queue*'s service routine if it had not been disabled by a previous call to **noenable()**.

If all goes well, **putbq()** returns one. Otherwise, it returns zero.

See Also

bcanputnext(), canputnext(), DDI/DKI kernel routines, getq(), insq(), msgb, putq(), queue, rmvq()

Notes

putbq() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

The caller cannot have the stream frozen when calling this function.

putbq() can fail if insufficient memory is available to allocate the accounting data structures used with messages whose priority bands are greater than zero.

A high-priority message should never be placed onto a queue from within a service routine.

putctl() — DDI/DKI Kernel Routine

Put a control message onto a queue
#include <sys/stream.h>
int putctl(*queue***,** *type***)**
queue_t **queue***;* **int** *type***;**

putctl() allocates a message block and calls the driver's **put** to put it onto *queue*. *type* is the type of the message to be created; it must not be a data type, i.e., **M_DATA**, **M_PROTO**, or **M_PCPROTO**.

putctl() returns one if it could allocate and put the the message block. It fails and returns zero if *type* is a data type, or if it cannot allocate a message block.

See Also

DDI/DKI kernel routines, put(), putctl1(), putnextctl(), putnextctl1(), queue

Notes

putctl() has base or interrupt type. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

The caller cannot have the stream frozen when it calls this function.

queue cannot reference field **q_next**. Rather, use the function **putnextctl()**

putctl1() — DDI/DKI Kernel Routine

Enqueue a control message and one-byte parameter
#include <sys/stream.h>
int putctl1(*queue***,** *type***,** *parameter***)**
queue_t **queue***;* **int** *type***, parameter;**

putctl1(), like **putctl()**, allocates a message block and calls the driver's **put** routine to put it onto *queue*. *type* is the type of message to create; it must not be a data type, i.e., **M_DATA**, **M_PROTO**, or **M_PCPROTO**.

parameter gives a one-byte parameter. What this parameter represents depends upon the type of message being created.

putctl1() returns one if it could allocate and put the the message block. It fails and returns zero if *type* is a data type, or if it cannot allocate a message block.

See Also

DDI/DKI kernel routines, put(), putctl(), putnextctl(), putnextctl1(), queue

Notes

putctl1() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

The caller cannot have the stream frozen when it calls this function.

queue must not reference the field **q_next**. To pass a message to the next queue in a stream, use the function **putnextctl1()**.

putnext() — DDI/DKI Kernel Routine

Send a message to the next queue
#include <sys/stream.h>
int putnext(*queue***,** *message***)**
queue_t ******queue***; mblk_t** ******message***;**

putnext() invokes the driver's **put** routine to put *message* onto *queue*'s next queue (i.e., to *queue***->q_next**). It does not return a meaningful value.

See Also

DDI/DKI kernel routines, msgb, putnextctl(), putnextctl1(), queue

Notes

putnext() has base or interrupt level. It does not sleep.

The caller cannot have the stream frozen when it calls this function.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

putnextctl() — DDI/DKI Kernel Routine

Send a control message to a queue
#include <sys/stream.h>
int putnextctl(*queue***,** *type***)**
queue_t ******queue***; int** *type***;**

putnextctl() allocates a message block and calls the driver's **put** routine to put it onto the queue next to *queue* (i.e., to *queue***->q_next**). *type* is the type of the message to be created; it must not be a data type, i.e., **M_DATA**, **M_PROTO**, or **M_PCPROTO**.

putnextctl() returns one if it could allocate and put the the message block. It fails and returns zero if *type* is a data type, or if it cannot allocate a message block.

See Also

DDI/DKI kernel routines, put, putctl(), putctl1(), putnextctl1(), queue

Notes

putnextctl() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

The caller cannot have the stream frozen when it calls this function.

putnextctl1() — DDI/DKI Kernel Routine

Send a control message and a parameter to a queue
#include <sys/stream.h>
int putnextctl1(*queue***,** *type***,** *parameter***)**
queue_t ******queue***; int** *type***, parameter;**

putnextctl1(), like **putctl()**, allocates a message block and calls the driver's **put** routine to put it onto the queue next to *queue* (i.e., to *queue***->q_next**). *type* is the type of the message to be created; it must not be a data type, i.e., **M_DATA**, **M_PROTO**, or **M_PCPROTO**.

parameter gives a one-byte parameter. What this parameter represents depends upon the type of message being created.

putnextctl1() returns one if it could allocate and put the the message block. It fails and returns zero if *type* is a data type, or if it cannot allocate a message block.

See Also

DDI/DKI kernel routines, put(), putctl(), putnextctl(), putnextctl1(), queue

Notes

putnextctl1() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

The caller cannot have the stream frozen when it calls this function.

putq() — DDI/DKI Kernel Routine

Put a message onto a queue
#include <sys/stream.h>
int putq(*queue***,** *message***)**
queue_t ******queue***; mblk_t** ******message***;**

putq() puts *message* onto *queue* after its **put** routine has finished processing it.

putq() places *message* onto *queue* after all other messages with the same priority. It updates all flow-control parameters, and schedules *queue*'s service routine if it had not been disabled by a previous call to **noenable()**.

If all goes well **putq()** returns one; otherwise, it returns zero.

See Also

DDI/DKI kernel routines, getq(), insq(), msgb, putbq(), queue, rmvq()

Notes

putq() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

The caller cannot have the stream frozen when it calls this function.

putq() fails if not enough memory is available to allocate the accounting data structures used with messages with priority greater than zero.

putubd() — Internal Kernel Routine

Store a byte into user data space
putubd(*u***,** *b***)**
char ******u***,** *b***;**

putubd() stores byte *b* at address *u* in the user's data segment. If an address fault occurs, it calls **set_user_error()** with value **EFAULT**.

See Also

internal kernel routines

putusd() — Internal Kernel Routine

Store a short to user data
putusd(*addr***,** *s***)**
short ******addr***,** *s***;**

putusd() writes short (16-bit) integer *s* into the user data space addressed by *addr*.

See Also

internal kernel routines

putuwd() — Internal Kernel Routine

Store a word into user data space
putuwd(*u***,** *w***)**
char ******u***; int** *w***;**

putuwd() stores word *w* at address *u* of the user's data segment. If an address fault occurs, it calls **set_user_error()** with value **EFAULT**.

See Also

internal kernel routines

putuwi() — Internal Kernel Routine

Put a word into user code space
putuwi(*u***, *w*)**
char **u*; **int** *w*;*

putuwi() puts word *w* into address *u* of the user's code segment. If an address fault occurs, it calls **set_user_error()** with value **EFAULT**.

See Also

internal kernel routines

pxcopy() — Internal Kernel Routine

Copy from physical or system global memory to kernel data
#include <sys/seg.h>
pxcopy(*src*, *dest*, *num_bytes*, *flag***)**
paddr_t *src*; **vadr_t** *dest*; **unsigned int** *num_bytes*; **int** *flag*;

Kernel function **pxcopy()** copies data from physical or system-global memory into kernel memory. You can invoke it either of two forms.

The first form copies *num_bytes* from physical address *src* into kernel data virtual address *dest*. This form is selected by setting argument *flag* to manifest constant **SEL_386_KD**.

The second form copies *num_bytes* from system global address *src* to kernel data virtual address *dest*. This form is selected by setting argument *flag* to manifest constant **SEL_386_KD|SEL_VIRT**.

Note that num_bytes *must be less than or equal to four kilobytes (4,096 bytes).*

No alignment restrictions are placed on *src* or *dest*.

See Also

internal kernel routines

qenable() — DDI/DKI Kernel Routine

Enable a queue
#include <sys/stream.h>
void qenable(*queue***);**
queue_t **queue*;*

qenable() enables *queue*. It tells the STREAMS scheduler that *queue*'s service routine is ready to be called.

See Also

DDI/DKI kernel routines, enableok(), noenable(), queue, srv

Notes

qenable() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

The caller cannot have the stream frozen when it calls this function.

qenable() works regardless of whether *queue*'s service routine has been disabled by a call to **noenable()**.

qinit — STREAMS Data Structure

Structure to initialize a STREAMS queue
#include <sys/stream.h>

The structure **qinit** contains pointers to procedures that initialize or manipuate a queue. A driver or module declares one **qinit** structure for all of its read queues, and one for all of its write queues. The driver's **streamtab** structure holds the address of each **qinit** structure.

Once they are initialized, all fields within **qinit** are read-only. A driver or module can read the following fields:

int (*qi_putp)() The address of the driver's **put** routine.

int (*qi_srvp)() The address of the driver's **srv** (service) routine. This is initialized to NULL if the driver has no **srv** routine.

int (*qi_qopen)() The address of the driver's **open** routine. Only read queues need an **open** routine; the **qinit** structure for write queues initializes this field to NULL.

int (*qi_qclose)() The address of the driver's **close** routine. Only read queues need an **open** routine; the **qinit** structure for write queues initializes this field to NULL.

int (*qi_qadmin)() This field reserved for future use. Always initialize it to NULL.

struct module_info *qi_minfo
The address of the driver's **module_info** structure.

struct module_stat *qi_mstat
The address of structure **module_stat**, which is defined in header file **<sys/strstat.h>**. **module_stat** holds statistics; if the driver or module does not keep statistics, it initializes this field to NULL.

See Also

DDI/DKI data structures, module_info, queue, streamtab

qprocsoff() — DDI/DKI Kernel Routine

Turn off a driver or module
#include <sys/stream.h>
void qprocsoff(*readqueue***)**
queue_t **readqueue***;*

qprocsoff() "turns off" the driver or module that owns *readqueue*. It removes *readqueue*'s service routines from the list of service routines to be run; then it waits until all concurrent **put** or **service** routines are finished, disables the **put** routine, and returns.

When these routines are disabled in a module, messages flow around it as if it were not present in the stream. When they are disabled in a driver, of course, the queue halts.

To "turn on" the driver or module, call function **qprocson()**.

See Also

DDI/DKI kernel routines, qprocson()

Notes

qprocsoff() has base level only. It can sleep.

A driver cannot hold a driver-defined basic lock or read/write lock across a call to this function. However, it can hold a driver-defined sleep lock.

The caller cannot have the stream frozen when it calls this function.

The **close** routine of a driver must call **qprocsoff()** before it deallocates any resources upon which a driver's **put** or **service** routines depend.

qprocson() — DDI/DKI Kernel Routine

Turn on a driver or module
#include <sys/stream.h>
void qprocson(*readqueue***)**
queue_t **readqueue***;*

qprocson() "turns on" the driver or module that owns *readqueue*. It enables its **put** and **service** routines.

See Also

DDI/DKI kernel routines, qprocsoff()

Notes

qprocson() has base level only. It can sleep.

A driver cannot hold a driver-defined basic lock or read/write lock across a call to this function. However, it can hold a driver-defined sleep lock.

The caller cannot have the stream frozen when it calls this function.

The **open** routine within a driver or module must call **qprocson()** when it (the routine) is first invoked — but only after it has allocated and intialized all resources upon which the **put** and **service** routines depend.

qreply() — DDI/DKI Kernel Routine

Reply to a message
#include <sys/stream.h>
void qreply(*queue***,** *message***)**
queue_t **queue***;* **mblk_t** **message***;*

qreply() replies to a message. It does so by calling **OTHERQ()** to find *queue*'s partner, then calls its **put** routine to place *message* upon it.

See Also

DDI/DKI kernel routines, put, OTHERQ(), putnext()

Notes

qreply() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

The caller cannot have the stream frozen when it calls this function.

qsize() — DDI/DKI Kernel Routine

Count the messages on a queue
#include <sys/stream.h>
int qsize(*queue***)**
queue_t **queue***;*

qsize() counts the messages on *queue*, and returns the sum.

See Also

DDI/DKI kernel routines, msgb, queue

Notes

qsize() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep locks across a call to this function.

The caller cannot have the stream frozen when it calls this function.

queue — STREAMS Data Structure

Structure of a STREAMS queue
#include <sys/stream.h>
#include <sys/types.h>

The structure **queue** holds the information with which a STREAMS queue is managed. A STREAMS driver or module has two such structures: one for the stream's read queue and one for its write queue.

A driver or module can access the following fields within **queue**:

struct qinit *q_qinfo
: The address of the structure **qinfo**, which holds the addresses of the routines with which the driver or module processes this queue. This field must not be altered.

struct msgb *q_first
struct msgb *q_last
: The addresses of, respectively, the first and last messages in a queue. If the queue is empty, both are initialized to NULL. No driver or module should modify these fields.

struct queue *q_next
The address of the next queue in the stream, should there be one. No driver or module should modify this field.

void *q_ptr This field is reserved for use by the driver or module.

ulong_t q_count The number of bytes of all data messages in the queue's priority-band 0. Note that this band normally holds messages with normal or high priority.

ulong_t q_flag Flags that define the characteristics of the queue. A driver or module cannot set or clear a flag; it can, however, test for the presence of flags. At present, only one flag is recognized for this field: **QREADR** which indicates that this is a read queue.

long q_minpsz
long q_maxpsz Respectively, the minimum and maximum sizes for a packet. These fields are initialized from, respectively, fields **mi_minpsz** and **mi_maxpsz** within structure **module_info**. A driver or module may alter these fields. For more information on these fields, see the entry for **module_info** in this manual.

ulong_t q_hiwat
ulong_t q_lowat Respectively, the high- and low-water marks for this queue. These fields are initialized from, respectively, fields **mi_hiwat** and **mi_lowat** within structure **module_info**. A driver or module may alter these fields. For more information on these fields, see the entry for **module_info** in this manual.

See Also

bcanputnext(), canputnext(), DDI/DKI data structures, getq(), insq(), module_info, msgb, putq(), qinit, qsize(), rmvq() strqget(), strqset()

Notes

The structure **queue** is defined as type **queue_t**.

race condition — Technical Information

The term *race condition* refers to the condition that exists when the the outcome of a sequence of instructions cannot be guaranteed. This occurs when program has two sections of code that can run in any order and either share a variable or change the state of the machine: the code executed first wins the "race" and so controls execution of the program. Obviously, it is desirable to avoid this situation; you can do so if you can force a certain ordering of the code sections.

Race conditions most often happen in operating system related environments. If, as in the case of a device driver, your program has a main section of code that manipulates a few variables and it also has an interrupt handler that does the same, your program must lock out interrupts during certain critical times to guarantee that the variables will not be compromised.

Consider, for example, the following pseudo-code:

```
set interrupt priority to keep out the gremlins
while (work is not yet completed)
      sleep_routine( &some_variable_in_the_kernel_data_area )
restore interrupt mask
```

If an interrupt were to occur between the **while** statement and the call to the sleep routine, the driver would never wake up because the event it was waiting for (sleeping on) will have already occurred.

In most cases, drivers lock out interrupts when manipulating the internal linked lists associated with tasks to be performed or buffers in use. This keeps the interrupt handler from using stale data or, worse yet, a linked list that isn't correctly linked.

See Also

technical information

RD() — DDI/DKI Kernel Routine

Get a pointer to a read queue
#include <sys/stream.h>
queue_t *RD(*queue***)**
queue_t ****queue*;**

RD() returns the address of the read queue associated with *queue*. If *queue* is itself the read queue, then its address is returned.

See Also

DDI/DKI kernel routines, OTHERQ(), WR()

Notes

RD() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

read — Entry-Point Routine

Read data from a device
Internal-Kernel Interface:
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/uio.h>
#include <sys/cred.h>
int *prefix***read(***device***,** *uioptr***,** *credptr***,** *private***)**
dev_t *device***; IO ****ioptr***; cred_t ****credptr***; void ****private*
DDI/DKI or STREAMS:
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/uio.h>
#include <sys/cred.h>
int *prefix***read(***device***,** *uioptr***,** *credptr***)**
dev_t *device***; uio_t ****uioptr***; cred_t ****credptr***;**

A driver's **read** routine moves data from *device* into the user's data area. An application can invoke it via the COHERENT system call **read()**.

Internal-Kernel Interface

Under the internal-kernel interface to a driver, field **c_read** in the driver's **CON** structure holds the address of this routine. It is customary to name the **read** routine with the word **read** prefixed by a unique identifier for your driver; but this is not required.

device is a **dev_t** that identifies the device to be read.

ioptr points to the **IO** structure that manages communication with *device*.

Finally, *private* points to a data element that is private to your driver. Note that many drivers do not use this argument.

DDI/DKI or STREAMS

The rest of this article describes how to invoke this function under the DDI/DKI or STREAMS interfaces. To invoke it, the kernel calls function *prefix***read()**, where *prefix* is the unique prefix for this driver.

uioptr holds the address of structure **uio**, whose contents set where the data can be written, and how many can be written. Function **uiomove()** provides a convenient way to use the **uio** structure to manage the copying of data.

credptr points to the user's credential structure. The driver can read that structure to see if the user can read privileged information, should the driver provide access to any.

The **read** routine returns zero for success, or an appropriate error number.

See Also

CON, drv_priv(), entry-point routines, errno, kernel routines, strategy, uio, uiomove(), ureadc(), write

LEXICON

COHERENT Lexicon: **read()**

Notes

This entry point is optional.

The **read** routine has user context and can sleep.

read_t0() — Internal Kernel Routine

Read the system clock t0
int read_t0()

read_t0() reads channel 0 (t0) of the programmable interval timer (**PIT**), which drives the system clock. A system clock tick is the time it takes timer t0 to decrease from 11,932 to zero. A driver can read the timer whether interrupts are masked or not, and receive a number between 11,932 and zero. Each unit, therefore, represents a little less than a microsecond. Overhead per call to **read_t0()** is about five to ten microseconds, depending upon speed of the CPU and clock speeds of the system upon which a program is being run.

See Also

internal kernel routines

repinsb() — DDI/DKI Kernel Routine

Read bytes from a port
#include <sys/types.h>
void repinsb(*port***,** *address***,** *count***)**
int *port***,** *count***; uchar_t** ******address***;**

repinsb() reads *count* bytes from the eight-bit *port* and writes them at *address*.

See Also

DDI/DKI kernel routines, inb(), repinsd(), repinsw(), repoutsb()

Notes

repinsb() has base or interrupt level. It does not sleep.

Driver-defined basic locks, read/write locks and sleep locks may be held across calls to this function.

This function may not be meaningful on all implementations because some implementations may not support I/O-mapped I/O.

repinsd() — DDI/DKI Kernel Routine

Read double (32-bit) words from a port
#include <sys/types.h>
void repinsd(*port***,** *address***,** *count***)**
int *port***,** *count***; ulong_t** ******address***;**

repinsd() reads *count* double (32-bit) words from *port* and writes them at *address*.

See Also

DDI/DKI kernel routines, inl(), repinsb(), repinsw(), repoutsd()

Notes

repinsd() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

This function may not be meaningful on all implementations because some implementations may not support I/O-mapped I/O.

repinsw() — DDI/DKI Kernel Routine

Read a word from a port
#include <sys/types.h>
void repinsw(*port***,** *address***,** *count***)**
int *port***,** *count***; ushort_t** ******address***;**

repinsw() reads *count* 16-bit words from *port* and writes them at *address*.

See Also

DDI/DKI kernel routines, inw(), repinsb(), repinsd(), repoutsw()

Notes

repinsw() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

This function is meaningful only on implementations that support I/O-mapped I/O.

repoutsb() — DDI/DKI Kernel Routine

Write bytes to a port
#include <sys/types.h>
void repoutsb(*port***,** *address***,** *count***)**
int *port***,** *count***; uchar_t** ******address***;**

repoutsb() writes *count* bytes from *address* to *port*.

See Also

DDI/DKI kernel routines, outb(), repinsb(), repoutsl(), repoutsw()

Notes

repoutsb() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep locks across a call to this function.

This function is meaningful only on implementations that support I/O-mapped I/O.

repoutsd() — DDI/DKI Kernel Routine

Write double (32-bit) words to a port
#include <sys/types.h>
void repoutsd(*port***,** *address***,** *count***)**
int *port***,** *count***; uchar_t** ******address***;**

repoutsd() writes *count* double (32-bit) words from *address* to *port*

See Also

DDI/DKI kernel routines, outl(), repinsd(), repoutsb(), repoutsw()

Notes

repoutsd() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

This function is meaningful only on implementations that support I/O-mapped I/O.

repoutsw() — DDI/DKI Kernel Routine

Write words to a port
#include <sys/types.h>
void repoutsw(*port***,** *address***,** *count***)**
int *port***,** *count***; uchar_t** ******address***;**

repoutsw() writes *count* 16-bit words from *address* to *port*.

See Also

DDI/DKI kernel routines, outw(), repinsw(), repoutsb(), repoutsd()

Notes

repoutsw() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

This function is meaningful only on implementations that support I/O-mapped I/O.

LEXICON

rmvb() — DDI/DKI Kernel Routine

Remove a block from a message
#include <sys/stream.h>
mblk_t *rmvb(*message***,** *block***)**
mblk_t ****message*, ****block***;**

rmvb() removes *block* from *message*. If all goes well, it returns the address of the altered message. It fails and returns NULL if *block* is the only block within *message*; and it fails and returns -1 if *block* is not associated with *message*.

The caller must free *block*: **rmvb()** only removes it from *message*.

See Also

DDI/DKI kernel routines, msgb

Notes

rmvb() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

rmvq() — DDI/DKI Kernel Routine

Remove a message from a queue
#include <sys/stream.h>
void rmvq(*queue***,** *message***)**
queue_t ****queue*; mblk_t ****message***;**

rmvq() removes *message* from *queue*.

See Also

DDI/DKI kernel routines, getq(), insq()

Notes

rmvq() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

The caller must have the stream frozen when calling this function.

If *message* does not point to a message within *queue*, the system panics: *Caveat utilitor*.

RW_ALLOC() — DDI/DKI Kernel Routine

Create a read/write lock
#include <sys/types.h>
#include <sys/kmem.h>
#include <sys/ksynch.h>
rwlock_t *RW_ALLOC(*hierarchy***,** *priority***,** *lock_info***,** *flag***)**
uchar_t *hierarchy***; pl_t** *priority***; lkinfo_t ****lock_info***; int** *flag***;**

RW_ALLOC() allocates a read/write lock, and initializes it into the unlocked state.

hierarchy is a number from one through 32 that gives the order in which the newly created lock was acquired. This gives the lock's position relative to other basic and read/write locks, and therefore gives its position within the hierarchy of locks. Your driver must set *hierarchy* such that it acquires locks in order of increasing hierarchy number. To be portable across all implementations of STREAMS, your driver must increase *hierarchy* with priority — that is, no lock's *hierarchy* should be less than that of a lock with a lower priority.

priority gives the minimum interrupt priority that a function must have to acquire this lock. The following gives the recognized values for *priority*, from least to most restrictive:

plbase	Block no interrupts.
pltimeout	Block functions scheduled by functions **itimeout()** and **dtimeout()**.
pldisk	Block disk-device interrupts.

plstr Block STREAMS interrupts.
plhi Block all interrupts.

priority the following order of priorities:

```
plbase < pltimeout <= pldisk, plstr <= plhi
```

STREAMS does not define how **pldisk** and **plstr** relate to each other.

priority must be high enough to block any interrupt handler that attempts to acquire this lock.

lock_info gives the address of the **lkinfo** structure that describes this lock. The caller must initialize this structure. For detail on initializing and using a lock, see the article **lkinfo** in this Lexicon.

flag indicates whether the caller can sleep as it awaits the lock. If *flag* equals **KM_SLEEP**, the caller indicates that if insufficient memory is available for the lock, it will sleep until enough memory becomes available. However, if *flag* equals **KM_NOSLEEP**, the caller will not sleep.

If all goes well, **RW_ALLOC()** returns the address of the newly allocated lock. If insufficient memory was available to allocate a lock and *flag* equals **KM_NOSLEEP**, then it returns NULL.

See Also

DDI/DKI kernel routines, lkinfo, RW_DEALLOC(), RW_RDLOCK(), RW_TRYRDLOCK(), RW_TRYWRLOCK(), RW_UNLOCK(), RW_WRLOCK()

Notes

If *flag* equals **KM_SLEEP**, **RW_ALLOC()** has base level only and can sleep. If it equals **KM_NOSLEEP**, it has base or interrupt level and does not sleep.

If *flag* equals **KM_NOSLEEP**, a driver can hold a driver-defined basic lock or read/write lock across a call to this function. It can hold a driver-defined sleep lock across a call to this function regardless of the value of *flag*.

RW_DEALLOC() — DDI/DKI Kernel Routine

Deallocate a read/write lock
#include <sys/ksynch.h>
void RW_DEALLOC(*lock***)**
rwlock_t **lock***;*

RW_DEALLOC() deallocates the read/write *lock*. Calling **RW_DEALLOC()** when *lock* is locked or is being awaited triggers behavior that is undefined — and probably unwelcome.

See Also

DDI/DKI kernel routines, lkinfo

Notes

RW_DEALLOC() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock (other than the one being deallocated), or a sleep lock across a call to this function.

RW_RDLOCK() — DDI/DKI Kernel Routine

Acquire a read/write lock in read mode
#include <sys/ksynch.h>
#include <sys/types.h>
pl_t RW_RDLOCK(*lock***,** *priority***)**
rwlock_t **lock***;* **pl_t** *priority***;**

RW_RDLOCK() sets the interrupt priority to *priority* and acquires *lock*. If it is not available in read mode, the caller must wait until it is. When it acquires the lock, it returns the previous level of interrupt priority.

See Also

DDI/DKI kernel routines

Notes

RW_RDLOCK() has base or interrupt level.

A driver can hold a driver-defined sleep lock across a call to this function. It can also hold a driver-defined basic lock or read/write locks; however, note that attempting to acquire a lock that is already held by the calling context can trigger a deadlock.

When this function is called from interrupt level, *priority* must not be below the level at which the interrupt handler is running.

RW_TRYRDLOCK() — DDI/DKI Kernel Routine

Try to acquire a read/write lock in read mode
#include <sys/ksynch.h>
#include <sys/types.h>
pl_t RW_TRYRDLOCK(*lock***,** *priority***)**
rwlock_t ******lock***;* **pl_t** *priority***;**

RW_TRYRDLOCK() sets the level of interrupt priority to *priority* and acquires *lock* in read mode.

If all goes well, **RW_TRYRDLOCK()** returns the previous level of interrupt priority. Unlike the related function **RW_RDLOCK()**, this function does not wait for *lock* to become available; rather, if *lock* is not available, it fails and returns the value **invpl**.

See Also

DDI/DKI kernel routines

Notes

RW_TRYRDLOCK() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

You can call **RW_TRYRDLOCK()** acquire a lock in an order other than that defined by the lock hierarchy.

When this function is called from interrupt level, *priority* must not be below the level at which the interrupt handler is running.

RW_TRYWRLOCK() — DDI/DKI Kernel Routine

Try to acquire a read/write lock in write mode
#include <sys/types.h>
#include <sys/ksynch.h>
pl_t RW_TRYWRLOCK(*lock***,** *priority***)**
rwlock_t ******lock***;* **pl_t** *priority***;**

RW_TRYWRLOCK() sets the level of interrupt priority to *priority*, and attempts to acquire *lock* in write mode.

If all goes well, **RW_TRYWRLOCK()** returns the previous level of interrupt priority. Unlike the related function **RW_WRLOCK()**, this function does not wait for a lock; rather, if *lock* is not available, it fails and returns **invpl**.

See Also

DDI/DKI kernel routines

Notes

RW_TRYWRLOCK() has base or interrupt priority. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

You can call **RW_TRYWRLOCK()** to acquire a lock in an order other than that defined by the lock hierarchy.

When this function is called from interrupt level, *priority* must not be below the level at which the interrupt handler is running.

RW_UNLOCK() — DDI/DKI Kernel Routine

Release a read/write lock
#include <sys/ksynch.h>
#include <sys/types.h>
void RW_UNLOCK(*lock***,** *priority***)**
rwlock_t ******lock***; pl_t** *priority***;**

RW_UNLOCK() releases the basic *lock*, and sets the level of interrupt priority to *priority*.

See Also

DDI/DKI kernel routines

Notes

RW_UNLOCK has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

RW_WRLOCK() — DDI/DKI Kernel Routine

Acquire a read/write lock in write mode
#include <sys/ksynch.h>
#include <sys/types.h>
pl_t RW_WRLOCK(*lock***,** *priority***)**
rwlock_t ******lock***; pl_t** *priority***;**

RW_WRLOCK() sets interrupt priority to *priority* and acquires *lock*. points. If the lock is not available, **RW_WRLOCK()** waits until it becomes available in write mode.

When it acquires *lock*, **RW_WRLOCK()** returns the previous level of interrupt priority.

See Also

DDI/DKI kernel routines

Notes

RW_WRLOCK() has base or interrupt level.

A driver can hold a driver-defined sleep lock across a call to this function. It can also hold a driver-defined basic lock or read/write locks; however, note that attempting to acquire a lock that is already held by the calling context can trigger a deadlock. To avoid deadlock, the caller should honor the hiererarchy of locks.

When this function is called from interrupt level, *priority* cannot be less than that of the interrupt handler.

salloc() — Internal Kernel Routine

Allocate a memory segment
#include <sys/seg.h>
SEG * salloc(*len***,** *flag***)**
fsize_t *len***; int** *flags***;**

salloc() allocates a segment of memory that is *len* bytes long. The segment reference count is set to one. If more than one reference is made to the segment (where each reference calls **sfree()** when done), the device driver increments the fields **s_urefc** and **s_refc** in the **SEG** structure.

flags can be bitwise OR'd to contain any combination of the following values:

SFDOWN The segment grows downward (e.g., stack segment for a process).

SFNCLR Do *not* clear memory in the allocated segment (usually to save time).

SFTEXT The segment may not be written to from user mode

If allocation succeeds, **salloc()** returns a pointer to a **SEG** structure that describes the requested segment. The **SEG** structure has been taken from the **kalloc()** pool; memory for the segment itself is from the **sysmem** pool. If allocation fails (i.e., not enough memory is available), **salloc()** returns NULL.

See Also

internal kernel routines

SAMESTR() — DDI/DKI Kernel Routine

Check type of next queue
#include <sys/stream.h>
int SAMESTR(*queue***)**
queue_t ******queue***;**

SAMESTR() checks whether the next queue in a stream is of the same type as *queue*. It returns one if the next queue is the same type as *queue*; it returns zero if it is not, or if there is no next queue.

See Also

DDI/DKI kernel routines, OTHERQ()

Notes

SAMESTR() has base or interrupt level. It does not sleep.

A driver cannot hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

The caller cannot have the stream frozen when calling this function.

sdevice — System Administration

Configure drivers included within kernel
/etc/conf/sdevice

File **sdevice** configures the drivers that can be included within the COHERENT kernel. Command **idmkcoh** reads this file when it builds a new COHERENT kernel, and uses the information within it to configure the suite of drivers it links into the kernel.

There is one line within the file for each type of hardware device; if a driver manipulates more than one type of device, then it has one entry for each type of device it manipulates. A driver's entry within file **/etc/conf/mdevice** indicates how many entries a driver can have with **sdevice**: if field 3 contains flag 'o', the device can have only one entry; whereas if field 3 does not contain this flag, it can have more than one entry (although it is not required to do so). An entry that begins with a pound sign '#' is a comment, and is ignored by **idmkcoh**.

Each entry within **sdevice** consists of ten fields, as follows:

1. *Name*
This gives the name of driver, and must match the name given in **mdevice**. It cannot exceed eight characters.

2. *Included in Kernel?*
This field indicates whether the driver is to be linked into the kernel: 'Y' indicates that it is, 'N' that it is not.

3. *Number of Units*
The number of the hardware units that this driver can manipulate. Under COHERENT, this is always set to zero.

4. *Interrupt Priority*
The device's interrupt priority. This must be a value between 0 and 8: zero indicates that this device is not interrupt driven, whereas a value from 1 to 8 gives the interrupt priority.

5. *Interrupt Type*
The type of interrupt for this device. The legal values are as follows:

0 This device is not interrupt driven.

1 The device is interrupt driven. If the driver controls more than one device, each requires a separate interrupt.

2 The device is interrupt driven. If the driver supports more than one device, all share the same interrupt.

3 The device requires an interrupt line. If the driver supports more than one device, all share the same interrupt. Multiple device drivers that the same interrupt priority can share this interrupt; however, this requires special hardware support.

6. *Interrupt Vector*
The interrupt vector used by the device. If field 5 is set to zero, this must be also.

7. *Low I/O Address*
The low I/O address through which the driver communicates with the device. Set this field to zero if it is not used.

8. *High I/O Address*
The high I/O address through which the driver communicates with the device. Set this field to zero if it is not used.

9. *Low Memory Address*
The low address of memory within the controller of the device being manipulated. Set this field to zero if it is not used.

10. *High Memory Address*
The high address of memory within the controller of the device being manipulated. Set this field to zero if it is not used.

Note that all COHERENT drivers current set fields 7 through 10 to zero.

For examples of settings for this, read the file itself. For an example of modifying this file to add a new driver, see the Lexicon entry for **device drivers**.

See Also

Administering COHERENT, device drivers, mdevice, mtune, stune

sendsig() — Internal Kernel Routine

Send a signal
#include <sys/proc.h>
#include <signal.h>
void sendsig(*sig***,** *pp***)**
int *sig***; PROC** ******pp***;**

sendsig() sends signal *sig* to process *pp*.

See Also

internal kernel routines
COHERENT Lexicon: **signal(), sigset()**

set_user_error() — DDI/DKI Kernel Function

Set an error code in the user space
#include <sys/errno.h>
void set_user_error(*error***)**
int *error***;**

Function **set_user_error()** writes code *error* into the user space, where it can be examined by the process that owns that space.

This function replaces setting field **u_error** in the **UPROC** structure. Note that this field no longer exists, and therefore can no longer be modified or examined directly.

See Also

DDI/DKI kernel routines

Notes

Please note that like sleeping and some other situation, your driver can set the user error status only when user control is valid. A driver can call **set_user_error()** only from within driver functions invoked through the system calls **open()**, **close()**, **read()**, **write()**, **ioctl()**, and **poll()**.

setivec() — Internal Kernel Routine

Set an interrupt vector
void setivec(*level***,** *function***)**
int *level***; int (***function***)();**

setivec() establishes the routine to which *function* points as the handler for interrupt vector *level*. If the interrupt routine is in use, does not set the vector.

See Also

clrivec(), internal kernel routines

sigdump() — Internal Kernel Routine

Generate core dump
void sigdump()

sigdump() writes a dump of the current process into file *core* in the current directory, and terminates the current process.

sigdump() writes its core file in the following way:

- All segments appearing in the core file have **SRFDUMP** flags set to one, and will be complete (never truncated).
- All missing segments have **SRFDUMP** flags set to zero.
- Any segment larger (in bytes) than **DUMP_LIM** will not appear in the core file.
- If **DUMP_TEXT** is patched to one, the text segment will appear in the core file (unless it is too big). Thus, it is possible for a core file to contain text but no data.

See Also

internal kernel routines

signals — Technical Information

List recognized signals

To send a signal to a process under the DDI/DKI, invoke the function **proc_signal()** with the identity of the process being signalled. Under STREAMS, you should invoke the functions **putctl1()** or **putnextctl1()** to send a message of type **M_SIG**, plus the signal as an argument. For details, see the Lexicon entries for these functions.

The following lists the signals that a driver can send to a process:

SIGHUP	The device has "hung up," or disconnected.
SIGINT	The interrupt character has been received.
SIGQUIT	The quit character has been received.
SIGTSTP	The user has requested that the process stop.
SIGURG	Urgent data have become available.
SIGWINCH	The size of a window has changed.
SIGPOLL	A pollable event has occurred.

See Also

proc_signal(), putctl1(), putnextctl1(), technical information
COHERENT Lexicon: **sigaction(), signal(), signal.h, sigset()**

size — Entry-Point Routine

Return the size of a block device
#include <sys/types.h>
#include <kernel/param.h>
int *prefix***size(***device***)**
dev_t *device***;**

The **size** routine returns the size, in blocks, of *device*. Should this routine fail (e.g., *device* cannot be read), it returns -1. The number of bytes in a block is set by the manifest constant **NBPSCTR** which is defined in the header file **<kernel/param.h>**.

See Also

entry-point routines

Notes

This routine is used only by the DDI/DKI interface. Under this interface, it is required for block drivers.

The **size** routine has user context and can sleep.

SLEEP_ALLOC() — DDI/DKI Kernel Routine

Create a sleep lock
#include <sys/kmem.h>
#include <sys/types.h>
#include <sys/ksynch.h>
sleep_t *SLEEP_ALLOC(*unused***,** *lock_info***,** *flag***)**
int *unused***,** *flag***; lkinfo_t ****lock_info***;**

SLEEP_ALLOC() allocates a sleep lock, and initializes it to the unlocked state.

unused is reserved for future development. Always initialize it to zero.

lock_info points to a a copy of the structure **lkinfo**. Initialize it as follows:

lk_pad Initialize this field to zero.

lk_name Initialize this field to the address of the string that names the lock.

lk_flags To turn off the gathering of statistics, initialize this field to **LK_NOSTATS**; otherwise, initialize it to zero.

Multiple sleep locks can share one **lkinfo** structure; however, a sleep lock and a read/write lock cannot.

flag specifies whether the caller is willing to sleep if the lock cannot be created immediately. If *flag* equals **KM_SLEEP**, the caller will sleep if the lock cannot be created; if it equals **KM_NOSLEEP**, it will not.

If all goes well, **RW_ALLOC()** returns the address of the newly created lock. If sufficient memory is not available for the lock and *flag* equals **KM_NOSLEEP**, it returns NULL.

See Also

DDI/DKI kernel routines, lkinfo

Notes

If *flag* equals **KM_SLEEP**, **SLEEP_ALLOC()** has base level only and may not sleep; if, however, *flag* equals **KM_NOSLEEP**, it has base or Interrupt level and may sleep.

A driver can hold a driver-defined basic lock or read/write lock across a call to this function only if *flag* equals **KM_NOSLEEP**. It can hold a driver-defined sleep lock regardless of the value of *flag*.

SLEEP_DEALLOC() — DDI/DKI Kernel Routine

Deallocate a sleep lock
#include <sys/ksynch.h>
void SLEEP_DEALLOC(*lock***)**
sleep_t ****lock*;**

SLEEP_DEALLOC() deallocates the sleep lock *lock*. If *lock* is being held or awaited, attempting to deallocate triggers behavior that is undefined — and probably unwelcome.

See Also

DDI/DKI kernel routines

Notes

SLEEP_DEALLOC() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock (other than the one being deallocated) across a call to this function.

SLEEP_LOCK() — DDI/DKI Kernel Routine

Acquire a sleep lock
#include <sys/ksynch.h>
void SLEEP_LOCK(*lock***, *priority*)**
sleep_t **lock*; **int** *priority*;**

SLEEP_LOCK() acquires the sleep lock *lock*. If *lock* is not available, it puts the calling function to sleep; when the lock becomes available, the the calling process awakens and returns If *lock* is already held by the calling context, a deadlock occurs.

The caller is not interrupted by signals while it sleeps within **SLEEP_LOCK()**. The related function **SLEEP_LOCK_SIG()** also acquires a sleep lock but can be interrupted by signals.

priority gives the priority that the calling process wishes to receive after it awakens. **SLEEP_LOCK()** recognizes the following values for this argument:

pridisk	Priority for a disk driver.
prinet	Priority for a network driver.
pritty	Priority for a terminal driver.
pritape	Priority for a tape driver.
prihi	High priority.
primed	Medium priority.
prilo	Low priority.

You can specify positive or negative offsets from these values; positive offsets request favorable priority. The maximum allowable offset is three.

See Also

DDI/DKI kernel routines

Notes

SLEEP_LOCK() has base level only. It can sleep.

A driver cannot hold a driver-defined basic lock or read/write locks across a call to this function. A driver can hold a driver-defined sleep lock, assuming that it does not attempt to acquire that lock with this function.

SLEEP_LOCK_SIG() — DDI/DKI Kernel Routine

Acquire a sleep lock
#include <sys/ksynch.h>
#include <sys/types.h>
bool_t SLEEP_LOCK_SIG(*lock***, *priority*)**
sleep_t **lock*; **int** *priority*;**

SLEEP_LOCK_SIG() acquires the sleep lock *lock*. If *lock* is not available, **SLEEP_LOCK_SIG()** puts the caller to sleep; when *lock* becomes available, it awakens the caller and returns a non-zero value. The calling function can then return with the lock in its possession. If *lock* is already held by the calling context, a deadlock occurs.

priority gives the priority that the calling process wishes to have when it awakens. For a list of legal values for this argument, see the entry for **SLEEP_LOCK()** in this manual.

Unlike the related function **SLEEP_LOCK()**, **SLEEP_LOCK_SIG()** and its caller can be interrupted by a signal. If **SLEEP_LOCK_SIG()** receives a signal (or if the caller receives a signal other than a job-control-stop signal), it immediately returns zero without waiting to acquire *lock*. If, however, the caller receives a job-control-stop signal, **SLEEP_LOCK_SIG()** stops but restarts the lock operation as soon as the stop signal is released. If all goes well, **SLEEP_LOCK_SIG()** returns a non-zero value.

See Also

DDI/DKI kernel routines, signals

Notes

SLEEP_LOCK_SIG() has base level only. It can sleep.

A driver cannot hold a driver-defined basic lock or read/write lock across a call to this function. It can, however, hold a driver-defined sleep lock, subject to the restriction described above.

SLEEP_LOCKAVAIL() — DDI/DKI Kernel Routine

Query whether a sleep lock is available
#include <sys/ksynch.h>
#include <sys/types.h>
bool_t SLEEP_LOCKAVAIL(*lock***)**
sleep_t ***lock*****;**

SLEEP_LOCKAVAIL() returns a non-zero value if the sleep lock **lock** is available. If it is not available, it returns zero. Note that the state of a lock can change rapidly: the value **SLEEP_LOCKAVAIL()** returns may no longer be valid by the time the caller sees it.

See Also

DDI/DKI kernel routines

Notes

SLEEP_LOCKAVAIL() has base or Interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

SLEEP_LOCKOWNED() — DDI/DKI Kernel Routine

See if the caller holds a given sleep lock
#include <sys/ksynch.h>
#include <sys/types.h>
bool_t SLEEP_LOCKOWNED(*lock***)**
sleep_t ***lock*****;**

SLEEP_LOCKOWNED() returns a non-zero value if the caller holds the sleep lock *lock*. It returns zero if the caller does not hold it. You should use **SLEEP_LOCKOWNED()** only within an **ASSERT()** expression.

See Also

DDI/DKI kernel routines

Notes

SLEEP_LOCKOWNED() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

SLEEP_TRYLOCK() — DDI/DKI Kernel Routine

Try to acquire a sleep lock
#include <sys/ksynch.h>
#include <sys/types.h>
bool_t SLEEP_TRYLOCK(*lockptr***)**
sleep_t ***lockptr*****;**

SLEEP_TRYLOCK() attempts to acquire the sleep lock **lock**. If it succeeds, it returns a non-zero value. However, unlike the related function **SLEEP_LOCK()**, it fails and returns zero if it cannot acquire *lock*.

See Also

DDI/DKI kernel routines

Notes

SLEEP_TRYLOCK() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

SLEEP_UNLOCK() — DDI/DKI Kernel Routine

Release a sleep lock
#include <sys/ksynch.h>
void SLEEP_UNLOCK(*lock***)**
sleep_t ***lock*****;**

SLEEP_UNLOCK() releases the sleep lock *lock*. If a process is awaiting the lock, it is awakened.

See Also

DDI/DKI kernel routines

Notes

SLEEP_UNLOCK() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

sphi() — Internal Kernel Routine

Disable interrupts
int sphi()

sphi() disables hardware interrupts. It returns a value that describes the previous hardware interrupt state. A driver could later pass that value to **spl()** to restore the previous hardware interrupt state.

See Also

internal kernel routines, spl()

spl() — Internal Kernel Routine

Adjust interrupt mask
int spl(*s***)**
int *s***;**

spl() restores the hardware-interrupt state to *s*, which was returned by functions **sphi()** or **splo()**.

See Also

internal kernel routines, sphi(), splo()

splbase() — DDI/DKI Kernel Routine

Block no interrupts
#include <sys/inline.h>
pl_t splbase();

The functions with the prefix **spl** set the level of interrupt priority. The level assigned depends upon the type of device in question.

Each **spl** function blocks interrupts at or below its level. The following gives the order of the levels set by the **spl** functions:

splbase() <= spltimeout() <= spldisk(),splstr() <= splhi()

STREAMS does not define how **spldisk()** and **splstr()** relate to each other.

splbase() sets the interrupt priority to its lowest level, i.e., it blocks no interrupts. It returns the previous priority level.

See Also

DDI/DKI kernel routines, spldisk(), splhi(), splstr(), spltimeout()

Notes

splbase() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic locks or read/write lock across a call to this function; however, the call to **splbase()** must not lower the interrupt priority below that associated with the lock. A driver can hold a driver-defined sleep lock across a call to this function.

spldisk() — DDI/DKI Kernel Routine

Block disk-device interrupts
#include <sys/inline.h>
pl_t spldisk();

spldisk() sets the interrupt priority to the level associated with disk devices. It returns the previous priority level. For information on how the **spl** family of functions relate to each other, see the entry for **splbase()**.

See Also

DDI/DKI kernel routines, splbase() splhi(), splstr(), spltimeout()

Notes

spldisk() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined sleep locks across a call to this function. It can also hold driver-defined basic lock or read/write lock; however, the call to **spldisk()** must not lower the level of interrupt priority below that associated with the lock.

Interrupt-level code must never lower the level of interrupt priority below that at which the interrupt handler was entered.

splhi() — DDI/DKI Kernel Routine

Block STREAMS interrupts
#include <sys/inline.h>
pl_t splhi();

splhi() sets the interrupt priority to the highest level, i.e., it blocks all interrupts. It returns the previous priority level. For information on how the **spl** family of functions relate to each other, see the entry for **splbase()**.

See Also

DDI/DKI kernel routines, splbase(), spldisk(), splstr(), spltimeout()

Notes

splhi() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined sleep locks across a call to this function. It can also hold driver-defined basic lock or read/write lock; however, the call to **splhi()** must not lower the level of interrupt priority below that associated with the lock.

Interrupt-level code must never lower the level of interrupt priority below that at which the interrupt handler was entered.

splo() — Internal Kernel Routine

Enable interrupts
int splo()

splo() enables hardware interrupts. It returns a value that describes the previous hardware-interrupt state.

See Also

internal kernel routines spl()

splstr() — DDI/DKI Kernel Routine

Block STREAMS interrupts
#include <sys/inline.h>
pl_t splstr();

splstr() sets the interrupt priority to the level associated with STREAMS interrupts. It returns the previous priority level. For information on how the **spl** family of functions relate to each other, see the entry for **splbase()**.

See Also

DDI/DKI kernel routines, splbase() spldisk(), splhi(), spltimeout()

Notes

splstr() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined sleep locks across a call to this function. It can also hold driver-defined basic lock or read/write lock; however, the call to **splstr()** must not lower the level of interrupt priority below that associated with the lock.

Interrupt-level code must never lower the level of interrupt priority below that at which the interrupt handler was entered.

spltimeout() — DDI/DKI Kernel Routine

Block STREAMS interrupts
#include <sys/inline.h>
pl_t spltimeout();

spltimeout() sets the interrupt priority to the level associated with timeout functions, i.e., all functions scheduled by the function **itimeout()**. It returns the previous priority level. For information on how the **spl** family of functions relate to each other, see the entry for **splbase()**.

See Also

DDI/DKI kernel routines, splbase() spldisk(), splhi(), splstr()

Notes

spltimeout() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined sleep locks across a call to this function. It can also hold driver-defined basic lock or read/write lock; however, the call to **spltimeout()** must not lower the level of interrupt priority below that associated with the lock.

Interrupt-level code must never lower the level of interrupt priority below that at which the interrupt handler was entered.

splx() — DDI/DKI Kernel Routine

Reset an interrupt-priority level
#include <sys/inline.h>
pl_t splx(*oldlevel***)**
pl_t *oldlevel***;**

splx() sets the level of interrupt priority to *oldlevel*, which must have been returned by a previous call to **splbase()**, **spldisk()**, **splhi()**, **splstr()**, or **spltimeout()**. It returns the previous level of interrupt priority.

See Also

DDI/DKI kernel routines, splbase(), spldisk(), splhi(), splstr(), spltimeout()

Notes

splx() does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

srv — Entry-Point Routine

Service queued messages
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/types.h>
int *prefix***rsrv(***queue***)**
queue_t *queue***;**

int *prefix***wsrv(***queue***)**
queue_t **queue***;*

The **srv**, or "service," routine services queued messages. *queue* points to the message queue to be serviced. The **rsrv** routine reads *queue*, and **wsrv** writes to it. Neither returns a meaningful value. A drivers or modules should call **qenable()** to invoke a service routine, rather than invoke it directly.

A **srv** routine allows a driver or module to process messages. When the STREAMS scheduler calls a **srv** routine, it processes all messages on its queue. Processing continues until the queue is empty or is flow-controlled, or an allocation error occurs.

Because some networking protocols require multiple bands of data flow, STREAMS messages can have up to 256 different priorities. A stream must at least distinguish between normal messages and high-priority messages. A queue orders its messages by priority: a high-priority message always is written at the head of the queue, after all

other high-priority messages already enqueued. Each priority band has its own flow-control limits; high-priority messages are not affected by flow control. If a band of messages is stopped by flow control, all bands with lower priority are also stopped.

See Also

bcanputnext(), bufcall(), canputnext(), datab, entry-point routines, getq(), msgb, pcmsg(), put, putbq(), putnext(), putq(), qenable(), qinit, queue, timeout(),

Notes

This entry point is used only by the DDI/DKI interface. It is optional for modules and drivers, but required for multiplexing drivers. If a service routine is not needed, initialize to NULL field **qi_srvp** within the module's **qinit** structure.

If the service routine finishes running because of any reason other than flow control or an empty queue, it must explicitly arrange to be rescheduled.

Service routines do not have user context, and so many not call any function that sleeps. Only one copy of a queue's service routine will run at a time.

start — Entry-Point Routine

Initialize a device at system start-up
void *prefix***start();**

The **start** routine intiailizes the driver's data structures or hardware. The kernel calls it at system boot time after system services are available and interrupts have been enabled.

See Also

entry-point routines, init

Notes

This entry point is used only by the DDI/DKI interface. It is optional.

The **start** routine may not call routines that sleep, or that require user context.

strategy — Entry-Point Routine

Perform block I/O
#include <sys/types.h>
#include <sys/buf.h>
int *prefix***strategy(***buffer***)**
buf **buffer***;*

The **strategy** routine sets up and initiates data transfer with a block device. *buffer* points to the **buf** structure to be used in the transfer of data. It does not return a meaningful value.

The kernel calls the **strategy** routine to read and write data on the block device. A driver's **read**, **write**, or **ioctl** routines may also call its **strategy** routine to support the character (raw) interface of a block device.

The **strategy** routine can be called either directly, or via a call to the kernel function **physiock()**.

The **strategy** routine first validate the I/O request; if the request passes validation test, it enqueues the request. If no transfer is underway, it starts the I/O; then returns. When the I/O is complete, the driver calls **biodone()** to free the buffer and to notify everyone who had called **biowait()** to wait for the I/O to finish.

If the amount of data to be transferred exceeds the amount that can be transferred, a driver that supports partial reads and writes can do the following: First, transfer as much data as possible. Second, call **bioerror()** to set the buffer error number to **EIO**. Third, set *buffer***->b_resid** equal to the number of bytes not transferred. The remaining data can then be handled rationally. If **strategy** succeeded in transferring all of the data requested, it should set *buffer***->b_resid** to zero.

See Also

biodone(), bioerror(), biowait(), block, bp_mapin(), buf, entry-point routines, errno, getnextpg(), physiock(), pptophys(), read, write

Notes

This entry point is used only by the DDI/DKI interface. Under this interface, it is required in every block driver.

The **strategy** entry point has the context needed to sleep, but it cannot assume it is called from the same context of the process that initiated the I/O request. Further, the process that initiated the I/O might not even exist when the **strategy** routine is called.

STREAMS — Overview

This article introduces STREAMS.

STREAMS is a programmer's interface for performing modular, character-level I/O at the kernel level, but without modifying the kernel itself. You can write a program that invokes STREAMS facilities within the kernel, but your code does not need to be linked into the kernel itself.

The "metaphor" of STREAMS is, as its name suggests, that of streams of information being passed between the user's process and the device being manipulated. At the top of the stream, closest to the user's process, stands the *stream head*. At the bottom, closest to the hardware, stands the *stream driver*. In between can stand an indefinite number of modules. Two streams run from the stream head, through the modules, to the stream driver: one stream goes from the head to the driver ("downstream"), while the other stream goes from the driver to the head ("upstream"). Each stream can consist of one or more queues; each queue has a different priority, and a message is read or written in with the priority dictated by its queue.

When a user's process wishes to perform I/O with a device, the kernel joins it to a stream head. The process passes its request to the head via normal system calls, plus the additional STREAMS calls **getmsg()** and **putmsg()**. The stream head translates the request into one or more *messages*, which it passes downstream to the first module that stands between it and the stream driver. A message can convey a request to the stream driver, convey a packet of data, or both. That module can perform some transformation upon the messages or the data they contain, then pass them on to the next module; and so on, until the message reaches the stream driver.

The stream driver translates the message into a task to perform with the hardware, and executes that task. The stream driver, in turn, generates messages. The message can consist of a terse reaction, such as an error message; it can convey a packet of data; or both. The stream driver passes it message or messages upstream to the penultimate module; the module performs the same transformation it did on the message going downstream, only in reverse, and passes the modified message upstream. This continues until the message reaches the stream head, which translates the message into a form that can be understood by the user process.

As you can see, a stream can consist of an indefinite number of modules, each of which can modify the messages that pass through it. The functions that establish a stream and manipulate it are built into the kernel; however, the STREAMS code that comprises the stream head, the stream driver, and the modules lies outside of the kernel. This modularity and independence of the kernel means that, among other things, STREAMS drivers are independent of the operating system — in theory, at any rate. As long as they adhere to the published descriptions of STREAMS, they should run on any operating system that has a conforming implementation of STREAMS. STREAMS, in effect, separates hardware and software aspects of device drivers.

A full description of STREAMS lies outside the scope of this manual. As the COHERENT implementations of STREAMS was performed with the DDI/DKI in mind, there is a significant degree of overlap between the structures and functions used by both. If a function is described as being part of the DDI/DKI, you should assume that it is available under STREAMS as well, unless the Lexicon entry explicitly says that it is not. To begin to explore STREAMS, read the articles **DDI/DKI data structures** and **DDI/DKI kernel routines**.

See Also

device driver, DDI/DKI kernel routines, internal kernel routines
COHERENT Lexicon: **getmsg(), putmsg()**

streamtab — DDI/DKI Data Structure

Initialize a STREAMS driver or module
#include <sys/stream.h>

Each STREAMS driver or module has one **streamtab** structure that is statically allocated within its sources. It must be named **prefixinfo**, where *prefix* is the driver's magic prefix.

streamtab contains the addresses of the **qinit** structures for the read and write queues of a driver or module. The **qinit** structure, in turn, contains the addresses for the routines with which a driver or module manages its queues.

The following fields in structure **streamtab** are available to the driver or module:

struct qinit *st_rdinit
struct qinit *st_wrinit
The addresses of, respectively, the read-side and write-side **qinit** structures.

struct qinit *st_muxrinit
struct qinit *st_muxwinit
The addresses of, respectively, the lower read-side and the lower write-side **qinit** structures for multiplexing drivers. Modules and non-multiplexing drivers should initialize these to NULL.

See Also

DDI/DKI data structures, qinit

strlog() — DDI/DKI Kernel Routine

Submit messages to the log driver
#include <sys/log.h>
#include <sys/stream.h>
#include <sys/strlog.h>
#include <sys/types.h>
int strlog(*module***,** *minor***,** *level***,** *flags***,** *format***, ...)**
short *module*, *minor*; **char** *level*, ******format*; **uchar_t** *flags*;

strlog() submits formatted messages to the **log** driver. It returns zero if the message is not seen by all the readers, or one if it is. The messages can be retrieved by the COHERENT system call **getmsg()**.

module identifies the module or driver that submitted the message. *minor* identifies the minor device in question.

flags is a bitmask of flags that indicate the purpose of the message. **strlog()** recognizes the following values for this argument:

SL_ERROR	Message is for the error logger.
SL_TRACE	Message is for tracing.
SL_CONSOLE	Message is for the console logger.
SL_NOTIFY	Mail a copy of the error message to the system administrator.
SL_FATAL	Error is fatal.
SL_WARN	Error is a warning.
SL_NOTE	Error is a notice.

format is a **printf()**-style formatting string. The formats **%s**, **%e**, **%g**, and **%G** are not allowed. For a detailed discussion of how to build a format string, see the entry for **printf()** in the COHERENT Lexicon.

All subsequent arguments give variables. The number of variables, and the type of each, is determined by the *format* string, as with the function **printf()**.

See Also

DDI/DKI kernel routines, print
COHERENT Lexicon: **getmsg(), printf()**

Notes

strlog() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

stroptions — DDI/DKI Data Structure

Stream-head options
#include <sys/stream.h>
#include <sys/stropts.h>

A driver can send a message of type **M_SETOPTS** or **M_PCSETOPTS** upstream to the stream head to set options within the stream head. These messages contain the structure **stroptions**, which encodes the options to be set within the stream head.

The following fields within **stroptions** are available to a driver or module:

short so_readopt;
This field sets the options in the stream head that affect how the stream handles the system call **read()**. It holds two sets of flags; you can set only one flag within each set.

The first set determines how **read()** handles data messages:

RNORM Normal mode. This is the default. The system call **read()** returns the number of bytes requested or the number of bytes available, whichever is less. If a message's data are only partially read, **read()** returns that message to the beginning of the stream head's read queue.

RMSGD Message-discard mode. **read()** returns the numer of bytes requested or the number of bytes within the first message on the stream, whichever is less. It discards a message if its data are read only partially.

RMSGN Message non-discard mode. **read()** returns the number of bytes requested and the number of bytes in the first message on the stream head's read queue, whichever is less. It returns a message to the beginning of the stream head's read queue if its data are only partially read.

The second set of flags determines how **read()** handles the protocol messages **M_PROTO** and **M_PCPROTO**:

RPROTNORM Normal mode. If a protocol message is at the beginning of the stream head's read queue, **read()** fails with the error code **EBADMSG**.

RPROTDIS Discard mode. **read()** discards the **M_PROTO** or **M_PCPROTO** portions of the message and returns any **M_DATA** portions that may be present. In this mode, **read()** also discards messages of type **M_PASSFP**.

RPROTDAT Data mode. **read()** delivers to the user the **M_PROTO** or **M_PCPROTO** portions of a message, just as if they were normal data.

ushort_t so_wroff;
The offset, in bytes, to be included in the first message block of each message of type **M_DATA** that system call **write()** creates.

long so_minpsz;
long so_maxpsz;
Respectively, the minimum and maximum sizes of packets for the stream head's read queue.

ulong_t so_hiwat;
ulong_t so_lowat;
Respectively, the "high-water" and "low-water" marks for the stream head's read queue.

uchar_t so_band;
The priority band of messages to which the fields **so_hiwat** and **so_lowat** should be applied.

ulong_t so_flags;
This is a bitmask that encodes the options to set. It can contain any combination of the following values:

SO_READOPT Set the read option to that given in **so_readopt**.
SO_WROFF Set the write offset to that given in **so_wroff**.
SO_MINPSZ Set the minimum packet size to that given in **so_minpsz**.
SO_MAXPSZ Set the maximum packet size to that given in **so_maxpsz.**
SO_HIWAT Set the high-water mark to that given in **so_hiwat**.
SO_LOWAT Set the low-water mark to that given in **so_lowat**.
SO_ALL Set all of the above options.
SO_MREADON Permit the stream head to generate messages of type **M_READ**.
SO_MREADOFF Forbid the stream head to generate messages of type **M_READ**.
SO_NDELON For no-delay reads and writes, use TTY semantics.
SO_NDELOFF For no-delay reads and writes, use STREAMS semantics.
SO_ISTTY The stream acts as a terminal.
SO_ISNTTY The stream does not act as a terminal.
SO_TOSTOP Stop processes that are writing to this stream in the background.
SO_TONSTOP Do not stop processes that are writing to this stream in the background.

SO_BAND Set the priority band affected by the high- and low-water marks to that given in field **so_band**.

See Also

datab, DDI/DKI data structures, msgb
COHERENT Lexicon: **read(), write()**

strqget() — DDI/DKI Kernel Routine

Get information about a priority band
#include <sys/stream.h>
int strqget(*queue***,** *datum***,** *priority***,** *value***)**
queue_t ******queue***; qfields_t** *datum***;**
uchar_t *priority***; long** ******value***;**

strqget() retrieves *datum* that describes priority band *priority* within *queue*. It writes *datum* at address *value*. If all goes well, **strqget()** returns zero; otherwise, it returns an appropriate, non-zero error number.

strqget() recognizes the following values for *datum*:

QHIWAT The high-water mark.
QLOWAT The low-water mark.
QMAXPSZ The maximum size of a packet.
QMINPSZ The minimum size of a packet.
QCOUNT The number of bytes of data in messages.
QFIRST The address of the first message.
QLAST The address of the last message.
QFLAG Its flags.

See Also

DDI/DKI kernel routines, queue, strqset()

Notes

strqget() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

The caller must have the stream frozen when calling this function.

strqset() — DDI/DKI Kernel Routine

Modify a priority band
#include <sys/types.h>
#include <sys/stream.h>
int strqset(*queue***,** *what***,** *priority***,** *datum***)**
queue_t ******queue***; qfields_t** *datum***;**
uchar_t *priority***; long** *value***;**

strqset() sets to *value* the parameter *datum* within priority band *priority* of *queue*. If all goes well, it returns zero; otherwise, it returns a non-zero error code.

datum identifies parameter of *priority* that you wish to modify, as follows:

QHIWAT Its high-water mark.
QLOWAT Its low-water mark.
QMAXPSZ Its maximum packet size.
QMINPSZ Its minimum packet size.

See Also

DDI/DKI kernel routines, queue, strqget()

Notes

strqset() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

The caller must have the stream frozen when calling this function.

LEXICON

stune — System Administration

Set values of tunable kernel variables
/etc/conf/stune

File **stune** names each tunable variable within the kernel, and gives the value to which it is actually set. Command **idmkcoh** reads this file when it builds a new kernel, and uses its contents to patch the kernel appropriately.

Each entry within this file has two fields. The first field names the variable; the name must match that given in **stune**. The second field gives the value of the variable; this value must fall between the minimum and maximum values given in **stune**.

If a line begins with a pound sign '#', it is a comment and **idmkcoh** ignores it. If a tunable variable is not named in this file, **idmkcoh** uses the default value given in **stune**.

See Also

Administering COHERENT, device drivers, mdevice, mtune, sdevice

super() — Internal Kernel Routine

Verify super-user
int super()

super() checks whether the user has super-user privileges. It return one if the user has these privileges (i.e., if **u.u_uid == 0**). Otherwise, it calls **set_user_error()** with value **EPERM** and returns zero.

See Also

internal kernel routines

SV_ALLOC() — DDI/DKI Kernel Routine

Create a synchronization variable
#include <sys/kmem.h>
#include <sys/ksynch.h>
sv_t *SV_ALLOC(*flag***)**
int *flag***;**

SV_ALLOC() allocates and initializes a synchronization variable. *flag* specifies whether the caller can sleep, should insufficient memory not be available to create the variable: **KM_SLEEP** indicates that the caller can sleep until enough memory becomes available; **KM_NOSLEEP** indicates that it cannot.

SV_ALLOC() returns the address of the newly created synchronization variable. If not enough memory is available to hold a synchronization variable and *flag* equals **KM_NOSLEEP**, it returns NULL.

See Also

DDI/DKI kernel routines

Notes

If *flag* equals **KM_NOSLEEP**, **SV_ALLOC()** has base or interrupt level and cannot sleep; if it is sett to **KM_SLEEP**, it has base level only and can sleep.

If *flag* equals **KM_NOSLEEP**, a driver can hold a driver-defined basic lock or read/write lock across a call to this function; if *flag* equals **KM_SLEEP**, it cannot. It can hold a driver-defined sleep lock regardless of the value of *flag*.

SV_BROADCAST() — DDI/DKI Kernel Routine

Awaken processes sleeping on a synchronization variable
#include <sys/ksynch.h>
void *SV_BROADCAST(*synch***,** *flags***)**
sv_t **synch*; int *flags***;**

SV_BROADCAST() awakens every process blocked on the synchronization variable *synch*. Because a synchronization variable is stateless, a call to **SV_BROADCAST()** affects only the processes now blocked on *synch*, not processes that may later block on it.

flags is reserved for future use. Initialize it to zero.

See Also

DDI/DKI kernel routines

Notes

SV_BROADCAST() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

SV_DEALLOC() — DDI/DKI Kernel Routine

Deallocate a synchronization variable
#include <sys/ksynch.h>
void SV_DEALLOC(*synch***)**
sv_t **synch***;*

SV_DEALLOC() deallocates the synchronization variable *synch*.

See Also

DDI/DKI kernel routines

Notes

SV_DEALLOC() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, and sleep lock across a call to this function.

SV_SIGNAL() — DDI/DKI Kernel Routine

Awaken one process sleeping on a synchronization variable
#include <sys/ksynch.h>
void SV_SIGNAL(*synch***,** *flags***)**
sv_t **synch***;* **int** *flags***;**

SV_SIGNAL() awakens one process of those blocked on the synchronization variable *synch*. Because synchronization variables are stateless, a call to **SV_SIGNAL()** affects only the processes now blocked on *synch*, not a process that blocks on it later.

flags is reserved for future use. Initialize it to zero.

See Also

DDI/DKI kernel routines

Notes

SV_SIGNAL() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

SV_WAIT() — DDI/DKI Kernel Routine

Sleep on a synchronization variable
#include <sys/ksynch.h>
#include <sys/types.h>
void SV_WAIT(*synch***,** *priority***,** *lock***)**
sv_t ***synch*****; int** *priority***; lock_t** **lock***;*

SV_WAIT() puts the calling process to sleep. The calling process sleeps on the synchronization variable *synch*, until it is awakened by a call to **SV_SIGNAL()** or **SV_BROADCAST()**.

lock points to the basic lock that the caller must hold. **SV_WAIT()** releases the lock and sets the interrupt priority to *priority* after it queues the process on the synchronization variable, but before it switches context to another process. When the caller returns from **SV_WAIT()**, the basic lock is not held and the interrupt-priority level is set to **plbase**. The caller will not be interrupted by signals while it sleeps within **SV_WAIT()**.

priority gives the relative priority that the caller wants after it wakes up. **SV_WAIT()** recognizes the following values:

pridisk Priority appropriate for disk driver.
prinet Priority appropriate for network driver.
pritty Priority appropriate for terminal driver.
pritape Priority appropriate for tape driver.
prihi High priority.
primed Medium priority.
prilo Low priority.

You can also specify positive or negative offsets from these values. Positive offsets request favorable priority; the maximum allowable offset is three. Offsets can help you to define the relative importance of the locks and resources that a driver holds. In general, the more highly sought a lock or resource is, or the more locks or kernel resources a driver holds, the higher *priority* should be.

See Also

DDI/DKI kernel routines

Notes

SV_WAIT() has base level only. It can sleep.

A driver cannot hold a driver-defined basic lock or read/write lock across a call to this function; it can, however, hold a driver-defined sleep lock.

SV_WAIT_SIG() — DDI/DKI Kernel Routine

Sleep on a synchronization variable
#include <sys/types.h>
#include <sys/ksynch.h>
bool_t SV_WAIT_SIG(*synch***,** *priority***,** *lock***)**
sv_t **synch***; **int** *priority*; **lock_t** **lock*;

SV_WAIT_SIG() puts the calling process to sleep on the synchronization variable *synch*. The calling process sleeps until it is awaked by a call to **SV_SIGNAL()** or **SV_BROADCAST()**. When the calling process awakens, **SV_WAIT_SIG()** returns a non-zero value to indicate success. Unlike **SV_WAIT()**, a process that sleeps under **SV_WAIT_SIG()** can be awakened by a signal, as described below.

lock points to a basic lock that a function must hold when it calls **SV_WAIT_SIG()**. It releases the lock and sets the level of interrupt priority to **plbase** after it queues the process on *synch*, but before it switches context to another process.

priority gives the relative priority that the caller wants after it wakes up. This can be one of the following values:

pridisk Priority appropriate for disk driver.
prinet Priority appropriate for network driver.
pritty Priority appropriate for terminal driver.
pritape Priority appropriate for tape driver.
prihi High priority.
primed Medium priority.
prilo Low priority.

You can also specify positive or negative offsets from these values. Positive offsets request favorable priority. The maximum allowable offset is three. Offsets can help you to define the relative importance of the locks and resources that a driver holds. In general, the more highly contended a lock or resource is, or the more locks or kernel resources a driver holds, the higher *priority* should be.

A process that sleeps under **SV_WAIT_SIG()** can be interrupted by a signal. If the calling function receives a job-control signal that stops it, as soon as the calling process returns from the signal **SV_WAIT_SIG()** returns a non-zero value, just as if calling processing had been awakened by a call to **SV_SIGNAL()** or **SV_BROADCAST()**. If the caller is interrupted by a signal other than a job-control signal or by a job-control signal that does not stop the calling process, **SV_WAIT_SIG()** immediately returns zero.

If **SV_WAIT_SIG()** itself is interrupted by a signal, it immediately returns a non-zero value, just as if the calling process had been awakened by a call to **SV_SIGNAL()** or **SV_BROADCAST()**.

See Also

DDI/DKI kernel routines

Notes

SV_WAIT_SIG() has base level only. It may sleep.

A driver can hold a driver-defined basic lock or read/write lock cannot be held across a call to this function. However, it can hold driver-defined sleep lock.

technical information — Overview

The following Lexicon articles in this section give technical information:

errors	List all recognized error messages.
kernel variables	Variables that you can set within the COHERENT kernel.
mdevice	Format of the kernel's master file.
mtune	Format of file that defines tunable parameters.
messages	List all legal types of messages.
race condition	Define what a race condition is.
sdevice	Name drivers linked into your system's kernel.
stune	Name parameters set in your system's kernel.
signals	List all recognized signals.
trace	How to interpret a kernel traceback.

See Also

STREAMS

testb() — DDI/DKI Kernel Routine

Check for an available buffer
#include <sys/stream.h>
int testb(*size***,** *priority***)**
int *size***,** *priority***;**

testb() checks whether a call to **allocb()** call is likely to succeed. *size* and *priority* give, respectively, the size and the priority of the proposed allocation. **testb()** returns one if the proposed call is likely to succeed, and zero if it is not.

See Also

allocb(), bufcall(), DDI/DKI kernel routines

Notes

testb() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

This function is provided purely as a porting convenience for developers. You should replace calls to this function with calls to functions that do the real work.

time — Entry-Point Routine

Routine to execute when a timeout occurs
*prefix***time(***device***)**
dev_t *device***;**

Under the internal COHERENT device-driver interface, the entry point **time** points to the routine to execute when a timeout occurs. The address of this routine is given in field **c_timer** of the driver's **CON** structure.

device identifies the device to be manipulated.

See Also

CON, entry-point routines

timeout() — Internal Kernel Routine

Defer function execution
#include <kernel/timeout.h>
void timeout(*tp***,** *n***,** *function***,** *a***)**
TIM ***tp***; **int** *n*, **(***function***)();**

timeout() sets *function* to be called with integer argument *a* after *n* clock ticks. *tp* points to a timing structure to insert into the timing queue. The timing structure is a static structure located in the kernel's data segment. Any previous activation of a timer on the same timing structure is cancelled.

Calling **timeout()** with *function* set to NULL cancels a timer. A timed function never sleeps or alters the contents of the **u** structure.

To request that the timeout routine for device *dev* be called once per second, a driver sets **drvl[major(***dev***)].d_time** to a nonzero value. **drvl** is declared in header file **con.h**; macro **major()** is defined in header file **stat.h**. The value in field **d_time** is not altered by the kernel clock routines. A driver stops invocations of the timeout routine by storing a zero in **drvl[major(***dev***)].d_time**.

See Also

internal kernel routines

trace — Technical Information

COHERENT kernel traceback procedure

The following describes how to interpret the COHERENT kernel's page-fault message:

- First, look at the value of register **cr2**. This is the address that was illegal. Find which register (**eax**, **esi**, **edi**, etc.) matches the address in **cr2** so you can look at the assembly later and figure out the instruction.
- Check the value of register **eip**. This is the instruction that caused the page fault.
- Check the kernel backtrace. The first number (*aaa***->***bbb*, where *aaa* is the first number) is probably the kernel page-fault routine itself. However, check it anyhow. The rest is the backtrace, which is useful in determining why it panicked (now that you should know where from **eip**).
- Find the ***.sym** file for the kernel you were running. Sort it with the following command:

  ```
  sort kernel.sym > kernel.ssym
  ```

  This puts the addresses into numeric order.
- Pull the file *kernel*.ssym into an editor and look for the address. You may well not find the exact address. For example, if the value of register **eip** is FFF00030, you may find

  ```
  FFF00010  func1
  FFF00020  func2
  FFF10020  func3
  ```

  So, you know it bailed within **func2**, because address 030 is between addresses 020 and 10020. Some functions, however, are declared **static**, so they do not show up in the symbol table. If you are unlucky enough to have failed near a static function, you will just have to go into **db** and find where the functions end and where you really are at.
- Now you can use **db** and go to that function and look at it. Because you know which register held the illegal value (from looking at the value of register **cr2**), you can match the assembly language to the C original and find out exactly where in the code the program failed.

See Also

technical information

TRYLOCK() — DDI/DKI Kernel Routine

Acquire a basic lock

#include <sys/ksynch.h>
#include <sys/types.h>
pl_t TRYLOCK(*lock***,** *priority***)**
lock_t ******lock***;** **pl_t** *priority***;**

TRYLOCK() acquires *lock*, with interrupt *priority*. If *lock* is available, **TRYLOCK()** returns its previous level of interrupt priority. However, if it is not, **TRYLOCK()** immediately returns **invpl** without waiting to acquire the lock.

For a driver to be portable, it should set *priority* to a value high enough to block any interrupt handler that attempts to acquire *lock*. When a function calls **TRYLOCK()** from interrupt level, it must not set *priority* to less than the level at which the interrupt handler runs. For a list of the valid values for *priority*, see the entry for

LOCK_ALLOC().

See Also

DDI/DKI kernel routines, LOCK(), LOCK_ALLOC(), LOCK_DEALLOC(), UNLOCK()

Notes

TRYLOCK() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

ttclose() — Internal Kernel Routine

Close tty
#include <sys/tty.h>
void ttclose(*tp***)**
TTY **tp*;*

ttclose() is called by a terminal device driver on the last close. It waits for pending output to be sent, then flushes input and resets the internal state information for the given tty.

See Also

internal kernel routines

ttflush() — Internal Kernel Routine

Flush a tty
#include <sys/tty.h>
void ttflush(*tp***)**
TTY * *tp*;

ttflush() clears the input and output queues, and resets most state flags.

See Also

internal kernel routines

tthup() — Internal Kernel Routine

tty hangup
#include <sys/tty.h>
void tthup(*tp***)**
TTY **tp*;*

tthup() flags loss of carrier, flushes the tty queues, then sends the hangup signal to every process in the tty process group.

See Also

internal kernel routines

ttin() — Internal Kernel Routine

Pass character to tty input queue
#include <sys/tty.h>
int ttin(*tp***, *c*)**
TTY **tp*; **char** *c*;

ttin() passes character *c* to the device-independant teletypewriter (tty) input routines. It is called with interrupts disabled.

See Also

internal kernel routines, ttinp()

ttinp() — Internal Kernel Routine

See if tty input queue has room for more input
#include <sys/tty.h>
int ttinp(*tp***)**
TTY **tp*;*

ttinp() returns a nonzero value if the tty input queue can accept input, and returns zero if the queue is full. A driver always calls **ttinp()** before it calls **ttin()** to see if it is safe to do so.

See Also

internal kernel routines, ttin(), ttoutp()

ttioctl() — Internal Kernel Routine

Perform tty I/O control
#include <sys/tty.h>
#include <sgtty.h>
void ttioctl(*tp***,** *com***,** *vec***)**
TTY **tp***tp***;** **int** *com***;** **struct sgttyb** **vec***vec***;**

ttioctl() handles common typewriter I/O control (ioctl) operations, as defined in header file **sgtty.h**. It may call **(****tp***->t_param)(***tp***)** to initialize the hardware. If an error occurs, it calls **set_user_error()** with an appropriate value.

See Also

internal kernel routines

ttopen() — Internal Kernel Routine

Open a tty
#include <sys/tty.h>
#include <sgtty.h>
void ttopen(*tp***) TTY** **tp***tp***;**

a teletypewriter (tty) device driver calls **ttopen()** on the first open. It sets up default parameters, and invokes **(****tp***->t_param)(***tp***)** to initialize the hardware.

See Also

internal kernel routines

ttout() — Internal Kernel Routine

Get next character from tty output queue
#include <sys/tty.h>
int ttout(*tp***)**
TTY **tp***tp***;**

ttout() returns the next character to be output. If the output queue is empty, it returns -1. It should be called with interrupts disabled.

See Also

internal kernel routines, ttoutp()

ttoutp() — Internal Kernel Routine

See if tty input queue has data available
#include <sys/tty.h>
int ttoutp(*tp***)**
TTY **tp***tp***;**

ttoutp() returns a nonzero value if the tty input queue has output data available, and returns zero if the queue is empty. A driver calls **ttoutp()** before it calls **ttout()** to see if it is safe to do so.

See Also

internal kernel routines, ttout(), ttinp()

ttread() — Internal Kernel Routine

Read from tty
#include <sys/io.h>
#include <sys/tty.h>
void ttread(*tp***,** *iop***)**
TTY **tp***tp***;** **IO** **iop***iop***;**

ttread() moves data from the input queue associated with *tp* to the I/O segment referenced by *iop*. If an error occurs, **ttread()** calls **set_user_error()** with an appropriate value.

ttyread() may block, depending on settings of the flags in the **termio** structure.

See Also

internal kernel routines

ttread0() — Internal Kernel Routine

Read from tty
#include <sys/io.h>
#include <sys/tty.h>
void ttread0(*tp***,** *iop***,** *func1***,** *arg1***,** *func2***,** *arg2***)**
TTY **tp***; **IO** **iop***; **int (******func1***)(),** *arg1***, (******func2***)(),** *arg2***;**

ttread0() moves data from the input queue associated with *tp* to the I/O area referenced by *iop*. If an error occurs, it calls **set_user_error()** with an appropriate value.

The following behavior allows a driver to prevent deadlocks between coupled drivers, such as master-slave pairs of pseudoterminals. If *func1* is not null, the function call **(****func1***)(***arg1***)** is performed whenever **ttread0()** is about to sleep. Likewise, if *func2* is not null, **(****func2***)(***arg2***)** is performed whenever **ttread0()** is about to sleep.

ttread(*tp***,** *iop***)** is equivalent to **ttread0(***tp***,** *iop***, 0, 0, 0, 0)**.

See Also

internal kernel routines

ttsetgrp() — Internal Kernel Routine

Set tty process group
#include <sys/tty.h>
#include <sys/types.h>
void ttsetgrp(*tp***,** *ctdev***)**
TTY **tp***; **dev_t** *ctdev***;**

ttsetgrp() sets the process group if the current process does not have one. It also sets up the controlling terminal for the process if there is none.

See Also

internal kernel routines

ttsignal() — Internal Kernel Routine

Send tty signal
#include <signal.h>
#include <sys/tty.h>
void ttsignal(*tp***,** *sig***)**
TTY **tp***; **int** *sig***;**

ttsignal() sends signal *sig* to every process in the tty process group associated with *tp*.

See Also

internal kernel routines

ttstart() — Internal Kernel Routine

Start tty output
#include <sys/tty.h>
void ttstart(*tp***)**
TTY **tp***;

ttstart() begins output on a teletypewriter (tty) device if output is not disabled. It calls the start function indicated by the structure **tty**.

See Also

internal kernel routines

ttwrite() — Internal Kernel Routine

Write to tty
#include <sys/io.h>
#include <sys/tty.h>
void ttwrite(*tp***,** *iop***)**
TTY **tp***;* **IO** **iop***;*

ttwrite() moves data to an output queue associated with *tp*, from the I/O segment referenced by *iop*. If an error occurs, it calls **set_user_error()** with an appropriate value.

ttwrite() blocks either if the queue is full and **IONDLY** is clear for the transfer, or if the line discipline has run out of **clist**s.

See Also

internal kernel routines

ttwrite0() — Internal Kernel Routine

Write to tty
#include <sys/io.h>
#include <sys/tty.h>
void ttwrite0(*tp***,** *iop***,** *func1***,** *arg1***,** *func2***,** *arg2***)**
TTY **tp***;* **IO** **iop***;* **int (***func1***)(),** *arg1***, (***func2***)(),** *arg2***;**

ttwrite0() moves data to an output queue associated with *tp*, from the I/O area referenced by *iop*. If an error occurs, it calls **set_user_error()** with an appropriate value.

The following behavior allows a driver to prevent deadlocks between coupled drivers, such as master-slave pairs of pseudoterminals. If *func1* or is not NULL, the function call **(***func1***)(***arg1***)** is performed whenever **ttwrite0()** is about to sleep. Likewise, if *func2* is not NULL, **(***func2***)(***arg2***)** is performed whenever **ttwrite0()** is about to sleep.

ttwrite(*tp***,** *iop***)** is equivalent to **ttwrite0(***tp***,** *iop***, 0, 0, 0, 0)**.

See Also

internal kernel routines

uio — DDI/DKI Data Structure

Structure to organize scatter/gather I/O requests
#include <sys/file.h>
#include <sys/types.h>
#include <sys/uio.h>

The structure **uio** describes an I/O request that is split across more than one data-storage area (also called *scatter/gather I/O*). It describes the request and contains the address of an array of **iovec** structures that, in turn, indicate where the data are to be read or written. An **iovec**, in turn, can point either into user space or kernel space.

A **uio** can be created either by the kernel or by the driver. The rules by which an **uio** is manipulated differ depending upon its origin (and therefore, upon the entity that "owns" it). These are described below.

The kernel passes the contents of **uio** to the driver through the driver's entry-point routines. The driver should never change them. Functions **uiomove()**, **ureadc()**, and **uwritec()** maintain **uio**. Function **physiock()** also helps maintain **uio**; a block driver can call it to perform unbuffered I/O.

A driver that creates its own **uio** for a data transfer must initialize it to zero before it initializes the fields that are accessible to it. Thereafter the driver must not change its **uio**: the DDI/DKI functions maintain it.

A driver can read the following fields within **uio**:

iovec_t *uio_iov — The address of the array of **iovec** structures that describe where the data are stored. If a driver creates a private **uio** for a data transfer, it must also create an array of **iovec** structures.

int uio_iovcnt — The number of **iovec** structures in the array to which **uio_iov** points.

off_t uio_offset The starting address on the device to/from which the data are to be transferred. This field applies to every device that is randomly accessed (e.g., a floppy-disk drive), but not to every device that is sequentially accessed (e.g., a tape drive).

short uio_segflg This flag gives the space within memory from/to whence the data are to be transferred. **UIO_SYSSPACE** indicates kernel space; **UIO_USERSPACE** indicates that data are split between kernel space and user space.

short uio_fmode Flags that give the access mode of the data transfer. The following gives the legal values for this field:

- **FNDELAY** If the requested data transfer cannot occur immediately, terminate the request without indicating that an error occurred.
- **FNONBLOCK** If the requested data transfer cannot occur immediately, terminate the request and return error **EAGAIN**.

int uio_resid The number of bytes not yet been transferred to/from from the data area. If the driver creates the **uio** structure for a data transfer, it must initialize this field to the number of bytes to be transferred.

See Also

data structures, iovec, physiock(), read, uiomove(), ureadc(), uwritec() write

Notes

The DDI/DKI does not have a special function with which a driver can create a **uio** or **iovec** structure. Therefore, it should use either **kmem_zalloc()** or allocate them statically.

uiomove() — DDI/DKI Kernel Routine

Use a uio structure to copy data
#include <sys/types.h>
#include <sys/uio.h>
int uiomove(*address***,** *bytes***,** *flag***,** *uioptr***)**
caddr_t *address***; long** *bytes***; uio_rw_t** *flag***; uio_t** **uioptr***;*

uiomove() copies *bytes* of data between *address* and the space defined by the **uio** structure to which *uioptr* points.

address always gives a location within kernel space. *uioptr* can describe an area in either kernel space or user space, depending upon the value of its field **uio_segflg**: **UIO_SYSSPACE** indicates kernel space, whereas **UIO_USERSPACE** indictes user space. The system will panic if *address* lies within user space, or if **uio_segflg** is not consistent with the space that **uio** defines.

flag gives the direction of the copy: **UIO_READ** moves data from *address* to *uioptr*, whereas **UIO_WRITE** does the opposite.

If **uiomove()** copies *bytes* of data, it updates the appropriate fields within the structures **uio** and **iovec** and returns zero. If it could not copy all of the requested data, it updates **uio** to record the number of bytes not transferred and returns an appropriate error code.

See Also

bcopy(), copyin(), copyout(), DDI/DKI kernel routines, iovec, uio, ureadc(), uwritec()

Notes

If **uio_segflg** equals **UIO_USERSPACE**, **uiomove()** has base level only and can sleep; if it equals **UIO_SYSSPACE**, the function has base or interrupt level and cannot sleep.

If **uio_segflg** equals **UIO_SYSSPACE**, a driver can hold a driver-defined basic lock or read/write lock across a call to this function; if it equals **UIO_USERSPACE** it cannot. In either case, a driver can hold a driver-defined sleep lock. When it holds a lock across a call to this function, a driver must be careful not to create a deadlock.

ukcopy() — Internal Kernel Routine

User to kernel data copy
unsigned int ukcopy(*u***,** *k***,** *n***)**
char **u*, **k*; **unsigned** *n*;

ukcopy() copies *n* bytes from offset *u* in the user's data segment to offset *k* in the kernel's data segment. It returns the number of bytes copied. If an address fault occurs, it calls **set_user_error()** with value **EFAULT**, and returns zero.

See Also

internal kernel routines, kucopy()

Notes

This function is equivalent to the DDI/DKI routine **copyin()**.

uload — Entry-Point Routine

Routine to execute upon unloading the driver from memory

Under the internal COHERENT device-driver interface, the entry point **uload** gives access to the routine to execute when the driver is unloaded from memory. Its address is kept in field **c_uload** of the driver's **CON** structure.

This routine is unused at this time because COHERENT does not support loadable drivers.

See Also

CON, entry-point routines

unbufcall() — DDI/DKI Kernel Routine

Cancel a request to bufcall()
#include <sys/stream.h>
void unbufcall(*request***)**
toid_t *request*;

unbufcall() cancels *request* to the function **bufcall()**. *request* must have been returned by a call to **bufcall()** or **esbbcall()**.

See Also

bufcall(), DDI/DKI kernel routines, esbbcall()

Notes

unbufcall() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

unfreezestr() — DDI/DKI Kernel Routine

Unfreeze a stream
#include <sys/stream.h>
#include <sys/types.h>
void unfreezestr(*queue***,** *priority***)**
queue_t **queue*; **pl_t** *priority*;

unfreezestr() unfreezes the stream that contains *queue*. It sets the newly un-frozen stream's level of interrupt priority to *priority*; this must have been returned by the call to **freezestr()** with which the caller froze the stream, unless the caller needs to set a different level of interrupt priority. For this field's legal values, see the entry for **LOCK_ALLOC()** in this manual.

See Also

DDI/DKI kernel routines, freezestr()

Notes

unfreezestr() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

The caller must have frozen the stream before it calls this function.

unlinkb() — DDI/DKI Kernel Routine

Remove a block from the head of a message
#include <sys/stream.h>
mblk_t *unlinkb(*message***)**
mblk_t ****essage*;**

unlinkb() removes the first block from *message*. It returns the address of the remaining stump of *message*; if the message contained only one block, it returns NULL.

See Also

DDI/DKI kernel routines, linkb()

Notes

unlinkb() has base or interrupt level It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

unlinkb() does not free the block that it removes. You must free it.

unlock() — Internal Kernel Routine

Unlock a gate
#include <sys/types.h>
void unlock(*g***)**
GATE *g***;**

unlock() unlocks gate *g*. When the gate of a system resource is locked, no other processes can use it. Unlocking a gate allows the kernel to reschedule processes that had previously been blocked.

See Also

internal kernel routines, lock()

UNLOCK() — DDI/DKI Kernel Routine

Release a basic lock
#include <sys/ksynch.h>
#include <sys/types.h>
void UNLOCK(*lock***,** *priority***)**
lock_t ****lock*; pl_t** *priority***;**

UNLOCK() releases *lock*, which must be a basic lock. *priority* gives the level of interrupt priority that the calling process wants once the lock is released. Normally, this is the value that had been returned by the call that set the lock, that is, the level of interrupt priority the calling process had had before it set the lock. However, the calling function can set a different level of interrupt priority should it need to. See entry for **LOCK_ALLOC()** in this manual for a list of all legal values for this argument.

See Also

DDI/DKI kernel routines, LOCK(), LOCK_ALLOC(), LOCK_DEALLOC(), TRYLOCK()

Notes

UNLOCK() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

unmap_pv() — Kernel Routine

Dissociate virtual addresses from physical addresses
void unmap_pv(*vaddr***)**
vaddr_t *vaddr***;**

unmap_vp() releases virtual address *vaddr*. *vaddr* must have been previously obtained via function **map_pv()**.

See Also

internal kernel routines

untimeout() — DDK/DKI Kernel Routine

Cancel execution of a previously scheduled function
#include <sys/types.h>
void untimeout(*function_id***)**
toid_t *function_id***;**

Function **untimeout()** cancels the request to execute a given function at a future time. It returns nothing.

function_id identifies the request to cancel; it had been returned to **itimeout()** when the function was first scheduled for execution.

If you call **untimeout()** as the function is running, **untimeout()** does not return until the function has run to completion.

See Also

DDI/DKI kernel routines, itimeout()

Notes

untimeout() has base or interrupt level. It does not sleep. Note that **untimeout()** can be executed only from an interrupt level less than or equal to the level that **itimeout()** specified when it scheduled the function to be executed.

A driver cannot hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function if that lock is being contended by the scheduled function.

Note that the scheduled function cannot call **untimeout()** to cancel itself.

uproc — Internal Data Structure

Structure that defines a process
#include <sys/uproc.h>

Structure **uproc** describes a process. The kernel allocates one such structure for each process that is running. Header file **<sys/uproc.h>** defines this structure.

Each process has its own 'u' (or "user") area. This area holds a copy of the **uproc** structure, which the process (or things acting upon the process) can modify.

See Also

internal data structures

Notes

Please note that this structure is being redesigned to help COHERENT conform more closely to published standards. You should not write code that depends upon any part of this structure remaining stable.

ureadc() — DDI/DKI Kernel Routine

Copy a character to space that uio describes
#include <sys/uio.h>
int ureadc(*c***,** *uioptr***)**
int *c***; uio_t** ***uioptr***;**

ureadc() copies the character *c* into the space described by the **uio** structure to which *uioptr* points.

uioptr describes an area in either user or kernel space: if its field **uio_segflg** equals **UIO_SYSSPACE**, then it points to kernel space; whereas if the field equals **UIO_USERSPACE**, it points to the user's address space.

If all goes well, **ureadc()** updates the appropriate fields within structures **uio** and **iovec**, and returns zero. If something goes wrong, **ureadc()** returns an appropriate error number. For details, see the Lexicon entries for these structures.

See Also

DDI/DKI kernel routines, iovec, uio, uiomove(), uwritec()

Notes

ureadc() uio_segflg equals **UIO_USERSPACE**, **uread()** has base level only and can sleep; however, if it equals **UIO_SYSSPACE ureadc()** has base or interrupt level and cannot sleep.

If **uio_segflg** equals **UIO_SYSSPACE**, a driver can hold a driver-defined basic lock or read/write lock across a call to this function; however, if **uio_segflg** equals **UIO_USERSPACE**, it cannot. A driver can hold a sleep lock across a call to this function regardless of the value of **uio_segflg**. When it holds a lock across a call to this function, a driver must be careful not to create a deadlock.

uwritec() — DDI/DKI Kernel Routine

Copy character from space described by uio structure
#include <sys/uio.h>
int uwritec(*c***,** *uioptr***)**
int *c***; uio_t** ******uioptr***;**

uwritec() copies the character *c* from the space described by the **uio** structure to which *uioptr* points. *uioptr* describes an area in either user or kernel space: if its field **uio_segflg** equals **UIO_SYSSPACE**, then it points to kernel space; whereas if the field equals **UIO_USERSPACE**, it points to the user's address space.

If it copies *c* successfully, **uwritec()** updates the appropriate members of structures **uio** and **iovec** to reflect this fact, and returns the copied character. If something goes wrong, **uwritec()** returns -1.

See Also

DDI/DKI kernel routines, iovec, uio, uiomove(), ureadc()

Notes

uwritec()

Level

If **uio_segflg** equals **UIO_USERSPACE**, **uwrite()** has base level only and can sleep; however, if it equals **UIO_SYSSPACE**, then **uwritec()** has base or interrupt level and can sleep.

If **uio_segflg** equals **UIO_SYSSPACE**, then a driver can hold a driver-defined basic lock or read/write lock over a call to this function; however, if **io_segflg** equals **UIO_USERSPACE**, then it cannot. A driver can hold a driver-defined sleep lock regardless of the value of **uio_segflg**. If a function holds a lock across a call to **uwritec()**, it must be careful not to create a deadlock.

vtop() — Internal Kernel Routine

Translate virtual address to physical address
#include <sys/coherent.h>
#include <sys/types.h>
paddr_t vtop(*vaddr***)**
vaddr_t *vaddr***;**

vtop() returns the current physical address associated with virtual address *vaddr*.

See Also

internal kernel routines

wakeup() — Internal Kernel Routine

Wakeup processes sleeping on an event
void wakeup(*e***)**
char ******e***;**

wakeup() "wakes up" all processes that went to sleep on event *e*, so they can run again.

See Also

internal kernel routines

WR() — DDI/DKI Kernel Routine

Get a pointer to the write queue
#include <sys/stream.h>
queue_t *WR(*queue***)**
queue_t ****queue*;**

WR() returns a pointer to the write queue of *queue*.

See Also

DDI/DKI kernel routines, OTHERQ(), queue, RD()

Notes

WR() has base or interrupt level. It does not sleep.

A driver can hold a driver-defined basic lock, read/write lock, or sleep lock across a call to this function.

write — Entry-Point Routine

Write data to a device

Internal-Kernel Interface:
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/uio.h>
#include <sys/cred.h>
int *prefix***write(***device***,** *uioptr***,** *credptr***,** *private***)**
dev_t *device***; IO ****ioptr***; cred_t ****credptr***; void ****private*
DDI/DKI or STREAMS:
#include <sys/cred.h>
#include <sys/errno.h>
#include <sys/types.h>
#include <sys/uio.h>
int *prefix***write(***device***,** *uioptr***,** *credentials***)**
dev_t *device***; uio_t ****uioptr***; cred_t ****credentials***;**

The **write** routine copies data from the user's data area to the device. A user's application can invoke it via the function call **write()**.

Internal-Kernel Interface

Under the internal-kernel inteface to a driver, the address of the **write** routine is kept in field **c_write** of the driver's **CON** structure. It is customary to name the **write** routine with the word **write** prefixed by a unique identifier for your driver; but this is not required.

device is a **dev_t** that identifies the device to be written.

ioptr points to the **IO** structure that manages communication with *device*.

Finally, *private* points to a data element that is private to your driver. Note that many drivers do not use this argument.

DDI/DKI or STREAMS

The rest of this article describes how to invoke this function under the DDI/DKI interface. To invoke it, call function *prefix***write()**, where *prefix* is the unique prefix for this driver. *device* identifies the device to which the data are to be written. *uioptr* points to the **uio** structure that holds the information about the data to be copied. *credentials* points to the user's credential structure, which the driver should examine to see if the user has permission to write to *device*. The **write** routine returns zero if it succeeded in copying the data, or an appropriate error number should something have gone wrong.

An application calls the **write** routine via the system call **write()**.

Function **uiomove()** lets you use the **uio** structure to copy data. Block drivers that provide a character interface can call **physiock()** to perform data transfer via the driver's **strategy** routine.

The **write** operation should appear to the user to run synchronously. At very least, it should not return until the caller's buffer is no longer needed. A driver that is scrupulous about returning errors should not return until it

has committed the data to *device*. Drivers that are less fastidious about errors can return once they have entrusted the data to a local staging buffer; the data can be committed to the device asynchronously, but should an error occur driver will not be able to notify the user that his request failed.

See Also

CON, drv_priv(), entry-point routines, errno, physiock(), read, strategy, uio, uiomove(), uwritec(),
COHERENT Lexicon: **write()**

Notes

This entry point is optional.

The **write** routine has user context and can sleep.

x_sleep() — Internal Kernel Routine

Wait for event or signal
#include <sys/sched.h>
int x_sleep(*event***,** *schedPri***,** *isleepPri***,** *reason***);**
char **event***, ***reason***; int** *schedPri***,** *sleepPri***;**

x_sleep() surrenders control of the processor while this process awaits some event or resource. In effect, the process "sleeps" until a particular event occurs.

event gives the address of a data item in the kernel's static-data space. To awaken the sleeping process, call the function **wakeup()** with the same *event*.

schedPri gives a value used to the hint the scheduler once the process is asleep. It is one of the following: **prilo**, **primed**, **prihi**, **pritape**, **pritty**, **pridisk**, or **prinet**.

sleepPri is a flag that indicates what should happen if a signal is sent to the process while it sleeps (or is about to sleep). Values are one of the following:

slpriNoSig
: Signals cannot interrupt sleep. Use **slpriNoSig** if you want the driver never to be awakened by the arrival of a signal. The risk is, that if you lose the wakeup, the driver hangs forever.

slpriSigLjmp
: Signals cause whatever system call was in progress to end immediately with an error value of **EINTR**. Use **slpriSigLjmp** if you can afford to throw away the entire system call and return to the user with an **EINTR**. This is *not* valid from within **open()** or **close()**, as it causes a fatal imbalance in internal reference counts.

slpriSigCatch
: Signals cause a return from the call to **x_sleep()**. Use **slpriSigCatch** if you want to detect a non-ignored, non-delayed signal and do something about it.

reason points to text that explains why the process is sleeping. This text appears in output of the command **ps**. This text can be no more than **U_SLEEP_LEN** bytes long. If text contains fewer than **U_SLEEP_LEN** bytes, it must be terminated by a NUL character.

x_sleep() must obey the following rules:

First, a driver can **x_sleep()** while it waits for some condition to be satisfied. However, **x_sleep()** may return prematurely; therefore, the driver must place the call to **x_sleep()** within a loop and check for the initial condition to still be valid. Normally, a sleep is performed in the following manner:

```
set interrupt priority to keep out the gremlins
while (work is not yet completed)
        x_sleep( &some_variable_in_the_kernel_data_area,... )
restore interrupt mask
```

The interrupt routine, in turn, calls **wakeup()** or defers wakeup for later background processing if time is not an issue. This causes the aforementioned code to return from the call to **x_sleep()**.

As you can see, there is an inherent race condition between the **while** and **x_sleep()**. If the work is serviced while the driver is **x_sleep()**ing, the **while** loop works correctly. However, should the last interrupt happen after the **while** but before the call to **x_sleep()**, the driver deadlocks — in effect, it awaits an event that will never occur.

x_sleep() returns for various reasons, but a driver cannot depend upon it to return for reasons other than a process calling **wakeup()** with the variable upon which the driver fell asleep. If the driver awaits an event based

upon an interrupt, a driver must bracket the call to **x_sleep()** with calls to the kernel routines **sphi()** and **spl()**.

x_sleep() returns **PROCESS_NORMAL_WAKE** if it has received a wakeup call. It returns **PROCESS_SIGNALLED** it has received a signal (other than **SIGSTOP** or **SIGCONT**). Both constants are defined in the header file **<kernel/_sleep.h>**.

See Also

internal kernel routines, sphi(), spl(), wakeup()

Notes

x_sleep() replaces the function **v_sleep()**.

Because a driver that is "asleep at the wheel" can cause a great deal of trouble, you must use **x_sleep()** only during situations when the kernel can awaken it again. Observe the following rules when you use **x_sleep()**:

- Never call **x_sleep()** from within a driver's block routine, either directly or indirectly.
- Never call **x_sleep()** from within an interrupt handler, either directly or indirectly. Doing so can cause a deadlock, as described above.
- Never call **x_sleep()** from the load routine of a driver; doing so will cause a panic.
- Your driver must always check for signals while sleeping. Failure to do so will create "zombies" — that is, user processes that cannot be terminated. For example, the following code fragment shows how a blocking driver's **open** routine can let the user break out of a sleep by pressing the interrupt character on the keyboard:

```
if (nondsig()) {  /* received a signal? */
      set_user_error(EINTR);    /* indicate that we were interrupted */
      return;                   /* return to user process */
}
```

- If **longjmp()** occurs, there is no return from **x_sleep()**.

xpcopy() — Internel Kernel Routine

Copy from kernel data to physical or system global memory
#include <sys/seg.h>
xpcopy(*src***,** *dest***,** *num_bytes***,** *flag***)**
vaddr_t *src***; paddr_t** *dest***; unsigned int** *num_bytes***; int** *flag***;**

Kernel function **xpcopy()** copies kernel data to an address that you specify. You can invoke it in either of two forms.

The first form copies *num_bytes* from kernel data virtual address *src* to physical address *dest*. This form is selected by setting argument *flag* to manifest constant **SEL_386_KD|SEG_VIRT**.

The second form copies *num_bytes* from kernel data virtual address *src* to system global address *dest*. This form is selected by setting argument *flag* to manifest constant **SEL_386_KD**.

Note well that num_bytes *must be less than or equal to four kilobytes (4,096 bytes).*

No alignment restrictions are placed on *src* or *dest*.

See Also

internal kernel routines

Index

to _

A

B

C

D

E