Dory MacDuell

COHERENT™

Coherent Device Driver Kit

Mark Williams Company

COHERENT
Device Driver Kit

Release 1.2

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.

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.

Revision 4 Printing 5 4 3 2 1

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

Telephone:	(708) 291-6700	
FAX:	(708) 291-6750	
E-mail:	uunet!mwc!support support@mwc.com	(Technical Support)
	uunet!mwc!sales sales@mwc.com	(General Information)
BIX:	join mwc	
CompuServ:	76256,427	

Printed in the U.S.A.

Table of Contents

Section 1:
Introduction

This manual documents the COHERENT operating system's device driver kit. It describes the contents of the kit, introduces the COHERENT kernel, gives advice on how to go about writing a device driver, shows detailed examples of device drivers, and documents all of the kernel's accessible functions in Lexicon format.

Before you continue, please read the following carefully:

> *The COHERENT Device Driver 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.

The Kit

The COHERENT Device Driver Kit consists of the following:

- A set of relocatable object files from which the COHERENT kernel can be built.
- Configuration and documentation files for existing device drivers.
- Source files for selected device drivers.
- Header files that define functions, macros, and structures used by device drivers.

The following describes all directories found in the driver kit.

/conf/kbd
This directory contains the keyboard mapping table source files for various keyboards. Note that these can only be used with the **nkb** keyboard device driver.

/usr/sys

This is the root directory for the driver-configuration part of the driver kit. This includes commands to link a new COHERENT kernel and to create loadable drivers.

/usr/sys/confdrv

This directory contains shell scripts used by the **config** script (located in **/usr/sys**) that handle driver-specific parts of the configuration process. These include creating the device nodes to access the driver, setting up the parameters needed to link the driver into the kernel, etc. It holds the following files:

```
/usr/sys/confdrv/aha154x
/usr/sys/confdrv/al0
/usr/sys/confdrv/al1
/usr/sys/confdrv/at
/usr/sys/confdrv/ati
/usr/sys/confdrv/fl
/usr/sys/confdrv/gr
/usr/sys/confdrv/hs
/usr/sys/confdrv/kb
/usr/sys/confdrv/lp
/usr/sys/confdrv/mm
/usr/sys/confdrv/ms
/usr/sys/confdrv/msg
/usr/sys/confdrv/nkb
/usr/sys/confdrv/rm
/usr/sys/confdrv/rs0
/usr/sys/confdrv/rs1
/usr/sys/confdrv/sem
/usr/sys/confdrv/shm
/usr/sys/confdrv/ss
/usr/sys/confdrv/st
/usr/sys/confdrv/tn
```

/usr/sys/doc

This directory contains support files for the **config** script (located in **/usr/sys**). Each file corresponds to a driver, and holds a one-line description of the device the driver supports. It holds the following files:

```
/usr/sys/doc/aha165x
/usr/sys/doc/al
/usr/sys/doc/at
/usr/sys/doc/ati
/usr/sys/doc/fl
/usr/sys/doc/gr
/usr/sys/doc/hs
/usr/sys/doc/kb
/usr/sys/doc/lp
/usr/sys/doc/mm
/usr/sys/doc/ms
/usr/sys/doc/msg
/usr/sys/doc/nkb
/usr/sys/doc/rm
/usr/sys/doc/rs
/usr/sys/doc/sem
```

```
/usr/sys/doc/shm
/usr/sys/doc/ss
/usr/sys/doc/st
/usr/sys/doc/swap
/usr/sys/doc/tn
```

/usr/sys/ldrv
This is where the loadable drivers are stored after you run the script **ldconfig** (which resides in **/usr/sys**) to create a loadable driver.

/usr/sys/lib
This directory contains all the support objects used to build a loadable driver or a kernel. Each driver has an archive of the same name (i.e., **rm.a**) containing all the objects required for that type of driver. It holds the following files:

```
/usr/sys/lib/al.a
/usr/sys/lib/aha154x.a
/usr/sys/lib/at.a
/usr/sys/lib/ati.a
/usr/sys/lib/fl.a
/usr/sys/lib/gr.a
/usr/sys/lib/hs.a
/usr/sys/lib/kb.a
/usr/sys/lib/ldlib.a
/usr/sys/lib/ldmain.o
/usr/sys/lib/ldrts0.o
/usr/sys/lib/ldswap.o
/usr/sys/lib/lp.a
/usr/sys/lib/mm.a
/usr/sys/lib/ms.a
/usr/sys/lib/msg.a
/usr/sys/lib/nkb.a
/usr/sys/lib/rm.a
/usr/sys/lib/rs.a
/usr/sys/lib/sem.a
/usr/sys/lib/shm.a
/usr/sys/lib/ss.a
/usr/sys/lib/st.a
/usr/sys/lib/tn.a
/usr/sys/lib/tty.a
```

/usr/src/sys
Root of the subtree that contains the directories that hold driver sources, makefiles, etc.

/usr/src/sys/i8086/drv
Makefile and sources for all supplied drivers. It holds the following files:

```
/usr/src/sys/i8086/drv/Makefile
/usr/src/sys/i8086/drv/al.c
/usr/src/sys/i8086/drv/alx.c
/usr/src/sys/i8086/drv/at.c
/usr/src/sys/i8086/drv/atas.s
/usr/src/sys/i8086/drv/ati.s
/usr/src/sys/i8086/drv/fdisk.c
/usr/src/sys/i8086/drv/fl.c
```

```
/usr/src/sys/i8086/drv/gr.c
/usr/src/sys/i8086/drv/gras.s
/usr/src/sys/i8086/drv/hs.c
/usr/src/sys/i8086/drv/kb.c
/usr/src/sys/i8086/drv/lp.c
/usr/src/sys/i8086/drv/mm.c
/usr/src/sys/i8086/drv/mmas.s
/usr/src/sys/i8086/drv/ms.c
/usr/src/sys/i8086/drv/rm.c
/usr/src/sys/i8086/drv/rs.c
/usr/src/sys/i8086/drv/rsas.s
/usr/src/sys/i8086/drv/st.c
/usr/src/sys/i8086/drv/tn.c
/usr/src/sys/i8086/drv/tnas.s
```

/usr/kobj
Device driver objects.

/usr/src/sys/i8086/drv/tools
Support programs for driver development and testing. It holds the following files:

```
/usr/src/sys/i8086/drv/tools/fontgen.c
/usr/src/sys/i8086/drv/tools/prate.c
```

/usr/include/sys
Header files relating to hardware-dependent issues, system constants, structures, macros, etc. This directory also includes driver-specific information that a user program may need to include. For example, the mouse **ioctl** structure and parameters are defined in the header **/usr/include/sys/ms.h.** It holds the following files:

```
/usr/include/sys/al.h
/usr/include/sys/clist.h
/usr/include/sys/coherent.h
/usr/include/sys/devices.h
/usr/include/sys/dmac.h
/usr/include/sys/fun.h
/usr/include/sys/hdioctl.h
/usr/include/sys/i8086.h
/usr/include/sys/ins8250.h
/usr/include/sys/kb.h
/usr/include/sys/kbscan.h
/usr/include/sys/ktty.h
/usr/include/sys/mmu.h
/usr/include/sys/ms.h
/usr/include/sys/poll_clk.h
/usr/include/sys/ptrace.h
/usr/include/sys/sdioctl.h
/usr/include/sys/systab.h
/usr/include/sys/tnioctl.h
/usr/include/sys/tty.h
```

Installing the Device Driver Kit

Before attempting to install the COHERENT Device Driver Kit, be sure that you have thoroughly read sections one and two of this manual.

In order to perform the installation, you must first log in as **root** (the superuser).

To install the COHERENT Device Driver Kit from a high density 5.25 inch distribution in drive 0, enter the following command:

```
/etc/install Drv_120 /dev/fha0 1
```

Please note that the three characters after the underscore are **numeric** and represent the version number of the release you are about to install. If you are installing a version of the COHERENT Device Driver Kit more recent than version **1.2.0**, change the aforementioned three characters to match those of your release.

To install the COHERENT Device Driver Kit from a high density 3.5 inch distribution in drive 0, enter the following command:

```
/etc/install Drv_120 /dev/fva0 1
```

The installation program will prompt you to insert the write protected floppy disk into drive 0. After the installation completes, place your distribution disk in a safe place, away from heat or magnetic fields.

Driver Sources

Some of the device driver sources have restricted distribution rights, and, thusly, cannot be included with the COHERENT Device Driver Kit.

The following device driver sources are being shipped with this release of the driver kit:

al	Serial line (COM1 thru COM4)
at	AT hard disk
ati	ATI Graphics Solution adapter
fl	Floppy drive
gr	IBM Color card (640x200) graphics display
hs	Generic polled multi-port serial
kb	Keyboard
lp	Parallel line printer
mm	Memory mapped video
ms	Microsoft bus mouse
rm	Dual RAM disk
rs	Raw serial (COM1 and COM2)
st	Archive SC-499 streaming tape
tn	Tiac PC-234/6 ARCNET LAN driver

Section 2:
Compatibility Information

It is impossible for Mark Williams Company to directly test more than a small fraction of the many computers, controllers, BIOSes, disks, and other devices that purport to be compatible with the IBM AT. The COHERENT system has been installed on more than 20,000 computers throughout the world, and we have received reports from many of our customers who have successfully installed and run COHERENT on their systems (as well as from the few who could not do so).

This section names the machines, add-on cards and BIOSes that have been reported either to work or not to work with the COHERENT operating system.

Before you continue, please note the following caveats:

First, this is only a partial list of the hardware on which COHERENT runs. We receive confirmation of new machine configurations almost daily. If you believe that you have a machine, BIOS, or add-on board that is **not** compatible with COHERENT but is listed below, please call our technical support department.

Second, manufacturers make changes to their hardware as part of redesigns or product improvements. These can include logic, timing, firmware, or functionality changes. Although we do try to support tested products, Mark Williams Company cannot guarantee compatibility with products not under its control.

If you believe that your computer cannot run COHERENT, please contact the Mark Williams Company technical support department. If you do not find your machine in this section, that does not mean that it will not run COHERENT; chances are that it will. Whatever happens, please contact Mark Williams Company and let us know what happened, so we can make your experience available to future users of COHERENT

Compatible Systems

The following systems have been tested with COHERENT, and have been found to be compatible. Note that configurations vary, especially with respect to disk controllers, so not all possible configurations have been tested.

ABM AT
Acer 910, 1100, 1116
AGI 1800A, 3000D, 3000G
AGL 286-12
ALR PowerFlex, 386SX, 386/220
American Semiconductor 286 PC
AMI 386SX, 386

Arche 386/25
AST Premium 286, 386/33
AT&T 6386
Austin 386SX, 386/33
Bentley 286
Bitwise 33-386 Portable
Bondwell 286 Laptop
Cheetah International i486/25
Club AT, 1800
Commodore 286
Compaq 286, 386, 386 Portable
Compaq SLT 286, LTE/286
CompuAdd 286-10, 286-12
CompuAdd 216, 220, 320, 325
Compudyne 286, 386
Computer Directions 386SX
Comtex 386/20
Condor Adv 286 III
Dell System 210, 220, 300, 310, 325
DTK PEM-2000 386
Dyna 386/20
EDP 386SX
Emerson 8286ECV
EPS 386
Epson Equity II+, III+
Executive AT-286
Five Star 386/20
Gateway 2000 (RLL and ESDI)
Gateway 486, 33MHz (IDE)
GCH 386 AT
Giga-Byte 386-33
Hauppauge 386
HP Vectra RS/20 (ESDI), ES/12, QS/20
Hyundai LT3/286
IBM PC/AT (286)
Intel 301
Jameco 3550
JDR M386
Laser 286, 386, 486
Leading Edge 386, D3, 6000
Leading Technology 386SX
Logix 386-25
MAXAR 386
Micro-1 386
Micro-Designs 386, 25MHz
Micro Express 386
Micronics 386
Mitsubishi 286L, 386
MTEK MS-23, MS-28, MS-35, MS-37, MS-41
MultiTech 900
MYLEX MWS386, 25 MHz
NCR 386, PC-810
NEC 386/25, Powermate 386/20, 386SX

COHERENT Driver Kit

Northgate 286/20, 386/16, 486
Olivetti M280, H28, M380
Omega 386/20
Optima 386
Packard Bell Axcel 386SX, PB900
Packard Bell Pack-Mate, Legend V
Panasonic Notebook 270
PC Brand 386/20, 386/25
PC Designs ET 286
PC's Limited AT
PC Pros 486
PC Systems 386-20
PeaCock 286 AT
Pulse 386-SX
Samsung 5550, 5800
Schneider Euro AT
SEFCO 16 MHz 386SX
Sharp 5541
Siemens 750
Smart Micro 286, 386
Sperry IT 286
Standard Brands 386-25, 386/SX
Sunnytech 386-20
Sys Technologies 386
Tandon 386/20, 386/33
Tandy 3000HL, 3000HD, 3000NL, 4000DX, 4000SX
Televideo AT 8MHz
Telex 1280
Tera-Tek 386
Touche' 5550T
Tri-Star 386
Unibit DS212, DS216, DS316
Unisys 2850, 286 PW
UTI 386
Victor 386
Viglen Genie 1
Wang PC 240 AT, PC 350, PC 381
Wells American AT, 14 MHz
Wyse 2108, 2112, 2200, 3216
Zenith 248, SuperSport 286
Zenith TurboSport 386, 386/33
ZEOS 286, 386, 386SX, 386 Portable
ZEOS Notebook 286, 386SX

Compatible Add-On Products

The following add-on products have been tested with COHERENT, and have been found to be compatible. Note that board and firmware revisions may vary. Not all possible configurations have been tested.

Adaptec AHA-1540A, AHA-1542A SCSI Host Adapter
Adaptec AHA-1540B, AHA-1542B SCSI Host Adapter
Adaptec 2372B, 2372C RLL 1:1
Arnet Multi-8 8 port serial

Arnet COM4 QUAD RS-232, PLUS4 QUAD RS-232
ATI VGA Wonder
BTC 1505 Monochrome Graphic Printer Card
Chase Research DB4, DB8 serial card
Comtrol Hostess serial card
Connect Tech Inc. Dflex-8 serial
Data Technology DTC7287 RLL 1:1
Digiboard PC/x serial card
DPT Smart Connex SCSI Host Adapter (WD emulation)
DTK PTI-217 IDE HD/FD
DTK Graphicsmith
DTK PEI-301 32-bit memory expansion
Emulex DCP/MUX
Future Domain TMC-840/841/880/881 SCSI Host Adapter
Future Domain TMC-845/850/860/875/885 SCSI Host Adapter
Geesee Trading PC-COM 4 port serial
IBM monochrome printer card
Maxspeed intelligent serial card
Maxtor 7080AT IDE hard disk drive
National Computer Ltd NDC545 MFM
Perstore PS180-16FN RLL
Seagate ST01, ST02 SCSI Host Adapter
Seagate ST-157A
SEFCO serial adapter
SEFCO monochrome adapter
Ultrastore Ultra 12 ESDI
Western Digital WD1006V-MM2 1:1 MFM
Western Digital WD1006V-SR2 1:1 RLL
Western Digital WD1007 ESDI
Western Digital 930xx series IDE hard disks

Compatible BIOS ROMs

The following BIOS ROMs have been tested with COHERENT, and have been found to be compatible.

AMI 286, 386
AMI version 3.10, 3.10D
DTK 386
IBM AT (286)
OPTI-Modular
PHOENIX 386
PHOENIX 386SX

When running protected mode software, certain releases of the AMI 386 BIOS fail to reset the system correctly when rebooting via a **<ctrl-alt-del>** key sequence. If you have this BIOS, use the **<reset>** button to reset your system correctly.

COHERENT Driver Kit

Incompatible Hardware

The following hardware is known *not* to work with this release of COHERENT.

American Multi-Source model 1004 MFM/RLL
AT&T 6300, 6300+
Chicony 101B IDE adapter
Dataworld 386/33 (video incompatibility)
Fujitsu 2612ET IDE hard disk
IBM MicroChannel PS/1 and PS/2 computers.
Leading Edge D2
Microsoft InPort Mouse
OMTI 8620 disk controller
Orchid Privilege 386SX-16 motherboard
Suntac 286-chipset based motherboards
Western Digital 1004-27X, 1004-WX1, 1002 series
Western Digital XTGEN, XTGEN+, XTGEN-2, XTGEN-R
XT (i.e., all eight-bit) disk controllers
Zenith Z449 video card (older versions cause panics)

Section 3:
Writing a Device Driver

This section discusses how to write a device driver for the COHERENT system. It covers the following topics:

- How the COHERENT kernel works.
- How device drivers are structured, and how they work with the kernel.
- The steps needed to write a device driver, including defensive programming and testing of the new driver.

As noted above, this manual is not meant to teach a beginner how to write a device driver. If, however, you are experienced at writing device drivers, it should give you all the information you need to begin to work with the COHERENT system.

The COHERENT Kernel

The COHERENT kernel is the program that permanently resides in memory to control the moment-to-moment operation of the COHERENT system. It controls *processes* and *devices*.

Processes

A *process* is any program that is being run on the computer at a given time. Many operating systems (e.g., MS-DOS) can support only one process at a time: it loads a program into memory, the program runs it until it has completed, then returns control the operating system, which waits until the user asks it to run another program.

COHERENT, however, allows a user (or users) to request that it run many processes at the same time. If you type the command

```
ps -alxd
```

COHERENT will print all of the processes that it is now executing on your computer.

The kernel shares processor time among many processes simultaneously, which creates the illusion that COHERENT is running many programs simultaneously. To accomplish this, the kernel creates two queues of all processes that it has been asked to execute. One queue, the *ready* queue, describes all processes that are ready to be processed further by the microprocessor. The other queue, called the *suspended* queue, describes all processes that are waiting for something to happen; for example, a word-processing program that is waiting for the user to press a key will be placed on the suspended queue.

The kernel selects a process from the ready queue and executes it until it either has reached a stopping point or has exhausted the slice of time allotted to it. If a process has exhausted its slice of time, it is returned to the ready queue. If it is awaiting an event, it is moved to the suspended queue; a process on the suspended queue is said to be *sleeping*. The kernel saves the current state of the process, then jumps to another process on its queue and executes that process for a while.

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

Each process is described to the kernel by the **UPROC** structure, as follows:

```
typedef struct uproc {
       char u_error;                     /* Error number (must be first) */
       char u_flag;                      /* Flags (for accounting) */
       int u_uid;                        /* User id */
       int u_gid;                        /* Group id */
       int u_ruid;                       /* Real user id */
       int u_rgid;                       /* Real group id */
       unsigned u_umask;                 /* Mask for file creation */
       struct inode *u_cdir;             /* Current working directory */
       struct inode *u_rdir;             /* Current root directory */
       struct fd *u_filep[NUFILE];       /* Open files */
       struct sr u_segl[NUSEG];          /* User segment descriptions */
       int (*u_sfunc[NSIG])();           /* Signal functions */

       /* System working area. */
       struct seg *u_sege[NUSEG]; /* Exec segment descriptors */
       MPROTO u_sproto;           /* User prototype */
       MCON u_syscon;             /* System context save */
       MENV u_sigenv;             /* Signal return */
       MGEN u_sysgen;             /* General purpose area */
       int u_args[(MSASIZE*sizeof(char)+sizeof(int)-1)/sizeof(int)];
       struct io u_io;            /* User area I/O template */

       /* Set by ftoi. */
       ino_t u_cdirn;             /* Child inode number */
       struct inode *u_cdiri;     /* Child inode pointer */
       struct inode *u_pdiri;     /* Parent inode pointer */
       struct direct u_direct;    /* Directory name */

       /* Accounting fields. */
       char u_comm[10];           /* Command name */
       time_t u_btime;            /* Beginning time of process */
       int u_memuse;              /* Average memory usage */
       long u_block;              /* Count of disk blocks */
```

```
        /* Profiler fields. */
        vaddr_t u_ppc;              /* Profile pc from clock */
        vaddr_t u_pbase;            /* Profiler base */
        vaddr_t u_pbend;            /* Profiler base end */
        vaddr_t u_pofft;            /* Offset from base */
        vaddr_t u_pscale;           /* Scaling factor */

        /* Miscellaneous things. */
        int u_argc;                 /* Argument count (for ps) */
        unsigned u_argp;            /* Offset of argv[0] (for ps) */
        int u_signo;                /* Signal number (for debugger) */
} UPROC;
```

Devices

A *device* is a piece of hardware with which a process must communicate. These include physical memory, the hard disk, the floppy disk, the serial port, the console, etc. The kernel manages all transfers of data between a process and a device.

Devices come in two flavors: *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. The current edition of COHERENT defines a block as being one-half kilobyte (512 bytes). 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/const.h>**; this should be used to ensure that your driver does not have to be rewritten should future editions of the COHERENT system change the block size.

Note that the COHERENT system, unlike most other operating systems, can allow a device driver to be accessed in either block-special or character-special modes. This will be detailed below.

Communication with a device is set with an **IO** structure, which is defined in header file **<sys/io.h>** as follows:

```
typedef struct io {
        int io_seg;                 /* Space */
        unsigned io_ioc;            /* Count */
        fsize_t io_seek;            /* Seek posiion */
        char *io_base;              /* Virtual base */
        paddr_t io_phys;            /* Physical base */
        short io_flag;              /* Flags: 0, IONDLY */
} IO;
```

The fields in this structure will be described below.

Buffer Cache

A buffer cache is associated with all block-special devices. This is an area of memory that holds data being written to or read from the device. Each cache entry is accessed via its version of the **BUF** structure, which is defined in header file **<sys/buf.h>** as follows:

```
typedef struct buf {
        struct buf *b_actf;          /* First in queue */
        struct buf *b_actl;          /* Last in queue */
        GATE b_gate;                 /* Gate */
        unsigned b_flag;             /* Flags */
        dev_t b_dev;                 /* Device */
        daddr_t b_bno;               /* Block number */
        char b_req;                  /* I/O type */
        char b_err;                  /* Error */
        unsigned b_seqn;             /* Buffer sequence number */
        bold_t b_map;                /* Old map */
        vaddr_t b_count;             /* Size of I/O */
        vaddr_t b_resid;             /* Driver returns count here */
        faddr_t b_faddr;             /* Far Virtual address */
        paddr_t b_paddr;             /* Physical address */
} BUF;
```

The fields in this structure are described below.

Interrupts

Most peripheral devices gain the attention of the kernel by sending an *interrupt*, which is a signal that the device sends to the operating system to indicate that it needs attention.

Each device that uses interrupts has a unique pointer, or *interrupt vector*, assigned to it. A device's interrupt vector points to a routine, or *interrupt handler*, which is designed to service its device. The operating system stores a table of interrupt vectors at the beginning of main memory.

When a device completes an assigned task, it generates an interrupt to indicate that it is finished. When COHERENT receives the interrupt, it saves the state of the process currently being executed. It then jumps to the handler pointed to by the device's interrupt vector, and executes it. Executing the interrupt handler may require awakening some sleeping processes.

When the interrupt handler has finished its work, COHERENT resumes processing the interrupted process as if nothing had happened.

Devices, Drivers, and Device Files

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

The COHERENT file system communicates with a device via a special file called a *device file*, which is created with the command **mknod**.

Most devices 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:

```
Fields: 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 root       root        11 128 Wed May 16 18:19 at0x
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
brw-------  1 root       root        11 129 Fri Apr 27 16:56 at1x
crw-rw-rw-  1 bin        bin          5   0 Fri Apr 27 16:56 com1r
crw-rw-rw-  3 bin        bin          6 128 Sat Aug 18 12:57 com2
crw-rw-rw-  3 bin        bin          6 128 Sat Aug 18 12:57 com2l
crw-rw-rw-  1 bin        bin          6   0 Fri Apr 27 16:56 com2r
crwx------  1 fred       user         2   0 Sat Aug 18 13:58 console
crw-------  2 sys        sys         11   0 Fri Apr 27 16:56 dos
```

The listing consists of nine fields, as follows:

1	Permissions
2	Number of links to the file
3	Owner
4	Group
5	Major device number
6	Minor device number
7	Date last modified
8	Time last modified
9	Name of file

The first character in the permissions field indicates the type of device this is: **b** indicates a block-special device, and **c** indicates a character-special device.

The major device number, which is given in field 5, is a unique number that identifies a class of device to the kernel. The kernel can handle up to 32 devices at any given time, numbered zero through 31. See the table in the entry for "device drivers" in the Lexicon at the rear of this manual, for a table of all device drivers current recognized by the COHERENT system, and the major device number of each.

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.

Under the COHERENT system, a device driver can either be linked into the kernel itself, or it can be loaded or unloaded into memory like any other program. In most instances, devices that are commonly used (e.g., drivers for physical memory and the hard disk) are linked into kernel, while those that are not commonly used (e.g., drivers for semaphores, shared memory, or esoteric hardware) are written to be loadable. The details of creating each type of driver are discussed below.

Kernel Functions

The COHERENT kernel contains numerous functions that perform the basic work of driving a device. These are described in this manual's Lexicon, and will be referred to throughout the rest of this manual.

Structure of a Device Driver

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

```
typedef struct con {
        int     c_flag;             /* Flags */
        int     c_mind;             /* Major device number */
        int     (*c_open)();        /* Open */
        int     (*c_close)();       /* Close */
        int     (*c_block)();       /* Block */
        int     (*c_read)();        /* Read */
        int     (*c_write)();       /* Write */
        int     (*c_ioctl)();       /* Ioctl */
        int     (*c_power)();       /* Powerfail */
        int     (*c_timer)();       /* Timeout */
        int     (*c_load)();        /* Load */
        int     (*c_uload)();       /* Unload */
        int     (*c_poll)();        /* Poll */
} CON;
```

The following subsection describes each entry in detail.

Flags

This field 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

As described above, a driver's major device number is set when the command **mknod** is used to create a device driver's device file. This number must be in the range zero to 31, and should be a symbolic constant found in file **<sys/devices.h>**.

Open Routine

This 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.

The kernel function **dopen** can access this routine; for more information, see its entry in this manual's Lexicon.

Close Routine

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

The kernel function **dclose** can access this routine; for more information, see its entry in this manual's Lexicon.

Block Routine

This points to the routine within the device driver that is executed when the kernel reads a file in block mode. It is called with a pointer to a **BUF** structure. The fields in this structure hold the following information:

b_dev A **dev_t** structure that describes the device being buffered. Kernel macros **major()** and **minor()** can be used to translate this structure into the device's major and minor numbers.

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

b_bno Number of the starting block.

b_faddr Virtual (non-DMA) address for the data.

b_paddr Physical (DMA) address for the data.

b_count Number of bytes to read or write.

b_resid 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.

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

The kernel function **dblock** can access this routine; for more information, see its entry in this manual's Lexicon.

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: the first argument is a **dev_t** that indicates the device to read; the second points to the **IO** structure for that device. The read function uses the fields of the **IO** structure as follows:

io_seek Number of bytes from the beginning of the file/device where 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 dividable by 512 (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 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 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. **IO_NDLY** indicates that the request be 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 **sleep()** and **wakeup()** to surrender the processor to another process while the read is being performed. The kernel function **ioputc()** is used to send characters to the user process and to update counter **io_ioc**.

The kernel function **dread** can access this routine; for more information, see its entry in this manual's Lexicon.

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 **c_read**, except that the direction of data transfer is reversed. Kernel function **iogetc()** is used to fetch characters from the user process and to update counter **io_ioc**.

The kernel function **dwrite** can access this routine; for more information, see its entry in this manual's Lexicon.

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 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 command, 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.

The kernel function **dioctl** can access this routine; for more information, see its entry in this manual's Lexicon.

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. The kernel function **dpower** can access this routine; for more information, see its entry in this manual's Lexicon.

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 persecond, 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 header file **stat.h**. The value in field **d_time**is 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.

The kernel function **dtime** can access this routine; for more information, see its entry in this manual's Lexicon.

Load Routine

Field **c_load** 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. In the case of loadable drivers, it is executed whenever the command **drvld** is invoked to load the driver into memory.

Unload Routine

The field **c_uload** points to the driver's function that the kernel invokes when the driver is unloaded from memory. In the case of a driver that is linked into the kernel, this function is never called; in the case of a loadable driver, this function is called when the **kill** command is invoked to remove the driver from memory.

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 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.

The kernel functions **pollopen** and **pollwake**, respectively, initiate and terminate a polling event. The kernel function **dpoll** can access the driver's polling routine. For more information on these function, see their entries in this manual's Lexicon.

Writing a Device Driver

This section discusses how one goes about writing 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

As noted earlier in this manual, you should assume that you will damage the file systems on your COHERENT system *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. The entries for **cpio**, **dump** and **tar** in the COHERENT system's Lexicon will show you how to do this.

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. Procedures for doing this will be described below.

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 for all device drivers that are currently available for 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. 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 name indicates that it's for the IBM PC-AT, as distinguished from the hard-disk driver for the IBM PC-XT, which is called **xt**. 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**.

In order to avoid inadvertent name-space collisions, the names of functions, variables, and arrays within your device driver should be prefixed with the name of the device.

Errors

Each user process has a **uproc** structure, accessed through the kernel's global variable **u**. (**uproc** is defined in the header file **<sys/uproc.h>**. To report an error to the user's process, set the field **u.u_error** to an appropriate value.

For a list of legal error codes, see the entry for the header file **<errno.h>** in the COHERENT manuals' Lexicon.

Devising Functions

A device driver consists chiefly of the suite of functions pointed to by its **CON** structure. The example drivers in the following section show how to organize these functions into a whole.

The driver will constantly call the kernel functions **sleep()** and **wakeup()** to synchronize your device driver with events in the operating system. **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 **sleep()** and **wakeup()** to exploit them.

Note, however, that calling **sleep()** at the wrong time will trigger a "race condition", which under the wrong conditions could cause the device to hang. The entries for **sleep()** and **race condition** in this manual's Lexicon discuss the when you should use the sleep mechanism, and when you should not.

Adding the Driver to COHERENT

Once the driver is written and compiled, you must make it available to the kernel. As noted earlier, drivers can either be linked into the kernel, or loaded into memory.

Preparatory Work

Before you configure and test your driver, you must do some preparatory work.

Initially, you should perform all your development work in directory **/usr/src/sys/i8086/drv**, with your compiled/assembled objects being placed in **/usr/kobj**. The first step in installing your device driver is to archive its object modules. Each driver's object modules are kept in their own archive in directory **/usr/sys/lib**. Use the **cd** command to enter the directory where you have your driver's objects, then type the command

```
ar rcs /usr/sys/lib/drv.a *.o
```

where *drv* is the name of your driver.

Directory **/usr/src/sys/i8086/drv** has a **Makefile** that demonstrates how to use **make** to recompile and rearchive all the drivers that were included with the driver kits. You would be well advised to copy this **Makefile** and modify it to support your driver, as follows:

1. The macro **ARCHIVES** (found near the top of the **Makefile**) names the archives that this **Makefile** recreates. Add your driver's name to it.
2. The **Makefile**'s macro **DRVOBJ** names the object modules that must be compiled to create all of the archives. Add your driver's object modules to this macro. These should be files that end up in subdirectory **objects**.
3. The dependencies of each archive are given in the section of the **Makefile** that has a series of entries that begin with the macro **$(USRSYS)**. For example, the following gives the dependencies for the archive **at.a**, which holds the object modules for the COHERENT AT hard-disk driver:

   ```
   $(USRSYS)/lib/at.a: objects/at.o objects/atas.o objects/fdisk.o
   	rm -f $@
   	ar rc $@ objects/at.o objects/atas.o objects/fdisk.o
   ```

   Create a similar entry for your device driver.
4. The last section of the **Makefile** lists the dependencies for each of the components of each driver, as well as the compilation/assembly instructions needed to compile or assemble the module. Note that these dependencies also include header files. Create a similar entry for your driver's objects.

Once you have modified the **Makefile**, the next step is to create a configuration file for your driver. The file must be stored in directory **/usr/sys/confdrv**. The following gives a slightly simplified example of the configuration file for **lp**, the line-printer driver:

```
UNDEF="${UNDEF} -u lpcon_ lib/lp.a"
PATCH="${PATCH} drvl_+30=lpcon_"
:
if [ -d "${DEV-/dev}" ]
then
        umask 0555
        /etc/mknod -f ${DEV-/dev}/lpt1 c 3 0            || exit 1
        /etc/mknod -f ${DEV-/dev}/lpt2 c 3 1            || exit 1
        /etc/mknod -f ${DEV-/dev}/lpt3 c 3 2            || exit 1
fi
```

The line

```
        UNDEF="${UNDEF} -u lpcon_ lib/lp.a"
```

adds linker information specific to this driver. In this case, we undefine a symbol called **lpcon_**, which is the name of the **CON** structure for the line-printer device. This causes the linker to link in the **lp** driver to resolve the undefined reference to symbol **lpcon_**. The **lib/lp.a** specifies the archive containing the driver objects for the **lp** device.

The line

```
        PATCH="${PATCH} drvl_+30=lpcon_"
```

specifies the parameters that will be pased to the **patch** command after the kernel has been linked. In our example, **drvl_+30** specifies the offset into the driver list array for major number 3 (3 * 10). Each entry is ten bytes long, so the calculations are easy. The address of **lpcon_** is assigned to this table entry, thus linking the driver's **CON** structure to the system.

The line

```
        if [ -d "${DEV-/dev}" ]
```

tests whether the variable **DEV** has been set in the environment; if not, then it defaults to **/dev**. It then tests to see if this is a directory. This will be used when you build a version of COHERENT on a floppy disk.

The lines

```
        /etc/mknod -f ${DEV-/dev}/lpt1 c 3 0            || exit 1
        /etc/mknod -f ${DEV-/dev}/lpt2 c 3 1            || exit 1
        /etc/mknod -f ${DEV-/dev}/lpt3 c 3 2            || exit 1
```

create a device file for each of the physical devices to be handled by this driver **mknod** takes four arguments: the name of the device, the type of device, the device's major number, and its minor number. As you can see, the commands create devices **lpt1**, **lpt2**, and **lpt3**. Each device is a character-special device (as indicates by the **c** in the command), and has the major-device number of 3. Each device has its own minor device, from zero through two. See the COHERENT manual's Lexicon entry for **mknod** for more information on how this command works. You will need to build at least one device file for each physical device that your driver will handle.

The next step is to create a file in directory **/usr/sys/doc** that describes the device driver. For example, the following gives the contents of **/usr/sys/doc/lp**:

```
lp            - Parallel line printer (LPT1, LPT2, LPT3)
```

The command **/usr/sys/config** prints these files as part of its usage message.

With the preliminary work done, you can now configure and test your driver. The following two sub-sections describe how to do this for, respectively, loadable drivers and linked drivers.

Configuring a Loadable Driver

If you wish, you can configure your driver as a loadable device driver. Almost any driver can be loadable, with the exceptions of the root file system and the console. Loadable drivers are quite useful: they do not take up bytes in the kernel's code segment, and they can quietly reside on the disk until the user actually needs their services. The user, however, must use the command **drvld** load them.

The shell script **/usr/sys/ldconfig** will configure your driver into a loadable driver. This script is invoked by **/usr/src/sys/i8086/drv/Makefile** via the **make** command. To manually configure and load your driver, use the following commands:

```
cd /usr/sys
ldconfig drv
/etc/drvld -k /coherent /usr/sys/ldrv/drv
```

where *drv* is the name of your driver. **/coherent** is the name of the kernel to use for symbol-table information. **ldconfig** performs the necessary configuration on your driver by linking it with the loadable-driver run-time startup code and libraries. **drvld** loads your driver into memory and updates the kernel's internal table (among other necessary tasks).

The kernel sets aside a static amount of memory to service loadable drivers. This can cause a loadable driver to not be loadable on some systems, because different systems have different numbers of drivers linked into the kernel and already loaded. Thus, if the currently running kernel doesn't have enough free kernel data space, attempting to run **/etc/drvld** might fail. This is not a problem and should not cause any concern other than that you cannot run the driver.

To skirt this problem, you can use the debugger **db** to patch the kernel, then reboot your system. In this case, you must increase the size of the kernel's variable **NSLOT** (which sets the number of loadable drivers), then reboot. Because each loadable driver's slot occupies 64 bytes, you must decrease the kernel variable **ALLSIZE** by 64 times the amount you increase **NSLOT**. The following gives an example **db** session: the entries in Roman type give your commands, those in **bold** give **db**'s replies, and the text in *italics* comment on the proceedings. Note that all numeric values are given in hexadecimal:

```
db /coherent          Invoke db to patch the kernel
NSLOT?x               Find the size of NSLOT in hexadecimal
40
NSLOT=50              Increase NSLOT by 16 bytes entries
ALLSIZE?x             Find the size of ALLSIZE
2C00
ALLSIZE=2800          Shrink ALLSIZE by 64*16 bytes
<ctrl-D>              Quit
```

The entry for **kernel variables** in this manual's Lexicon describes all of the kernel's global variables.

Before you begin to modify the kernel with **db**, please read the following carefully:

> *Patching your copy of* **/coherent** *is dangerous! You should always make a copy (called, say,* **/testcoh***) and patch it rather than your working copy. When you reboot, be sure to type* **testcoh** *rather than* **coherent** *when you see the prompt* **AT**

> *BOOT. If your driver corrupts the kernel to the point where it does run, you can always reboot your original copy of* **/coherent**. *Note also that if file* **/autoboot** *exists, it will be booted automatically and you will not be prompted to enter the name of the kernel to boot.*

You can also use **db** to examine variables in your device driver, to see how it is working. Suppose, for example that you have written the driver **wg**, which supports the "widget" peripheral device. The command **db -f /tmp/wg /dev/kmem** will make the driver's symbol table available to **db**. To examine a driver variable, use **db**'s formatted-print command. (For more information on how to use **db**, see its entry in the COHERENT manual's Lexicon.)

This procedure may be useful in debugging a driver, but before you do this, please read the following carefully:

> *Running* **db** *on a driver is extremely dangerous.* **db** *not only allows you to look into the kernel's data space, but allows you to inadvertently change something, causing the system to crash or become sick. If you do not know exactly what you are doing, do not use* **db** *to debug a driver on a live system!*

If you wish to remove a loadable driver's symbol table after you have loaded it into memory, run the command

```
/etc/drvld -r drv
```

where *drv* is the name of the driver. Note that if you do not tell **drvld** to create a symbol table, you cannot use **db** to examine the contents of the driver's variables.

To unload a loadable device driver, use the command **ps -d** to find its process number, then use the command **kill -9** to kill the driver's process.

After you have thoroughly debugged and tested your loadable driver, move it to **/drv** (not **/dev**), which is where all the loadable drivers reside.

Linking a Driver Into the Kernel

If your device driver is going to be used frequently or is required for the system to boot, you may wish to link it into the kernel. The device-driver kit uses two shell scripts to make this process easy for you: **/usr/sys/config**, which creates the new kernel, and **/usr/sys/Build**, which oversees the processing of building the kernel. For the sake of ease, the following will describe how to modify **Build** to create your new kernel.

Before you begin, please copy the file **/coherent** to a safe place, so you can restore the old kernel should something go drastically wrong with the kernel you are rebuilding.

The following gives the contents of the first few lines of **Build**. Check the version supplied with the device driver kit for further details.

```
: default drivers to be linked into COHERENT
DRIVERS="fl lp mm rm"

: default root/pipe device
BOOTDEV="at0a"

: set the default keyboard driver
KB=nkb
```

To begin, the line

```
DRIVERS="fl lp mm rm"
```

sets the device drivers that are linked by default into the kernel. You should insert the name of your device driver into this list.

The next line

```
BOOTDEV="at0a"
```

sets the default boot device. It assumes that the default boot device is partition 0 (or **a**) on AT/IDE hard disk drive 0. If your system boots from another disk or another partition, change this variable to the appropriate setting.

The line

```
KB=nkb
```

selects which of the two keyboard drivers you wish to use by default.

The **Build** script invokes the **config** script to recreate the kernel via the command:

```
./config ibm-at $DRIVERS root=$BOOTDEV
```

This rebuilds the kernel in your current directory (**/usr/sys**) in the file **coherent** and then copies it to **/coh.***type*, where *type* is the driver name for the boot device (e.g., **at**, **ss**, etc.). Note that **config** does *not* touch the copy of **coherent** in the root directory!

If you change this command to read

```
./config ibm-at $DRIVERS stand=fha0 root=$BOOTDEV
```

config will create a bootable high-density 5.25-inch floppy disk in drive 0 that contains the basic COHERENT file system, a few basic commands, and the devices you need to access the device (from the **confdrv** entries for the devices you specified). The bootable floppy disk will contain two copies of **coherent**: the first is called "**coherent**", which has its **rootdev_** and **pipedev_** devices set to the value specified by the macro **BOOTDEV** in the script **Build**. The other copy of **coherent** is called "**stand**" — short for "stand-alone". This **coherent** has **rootdev_** and **pipedev_** set to the floppy-disk device. If you choose to do this, don't forget to insert a write-enabled, high-density floppy disk into floppy drive 0 before you run **Build**.

If, however, you modify this line to read:

```
./config ibm-at $DRIVERS stand=fva0 root=$BOOTDEV
```

config will build a bootable version of COHERENT on a high-density 3.5-inch floppy disk in drive 0.

Running COHERENT from the Floppy Disk Drive

As noted above, you can use **Build** to create a miniature version of COHERENT that uses your floppy disk drive as its root device. *This is the option to chose if you plan to test drivers. It will tend to limit the amount of damage that can be done by a driver that has gone wild or has stepped on the kernel's data segment!*

To run this mini-COHERENT, insert the floppy disk you just created into drive 0 (or A) on your machine; then reboot your system. When the prompt **AT BOOT**. appears, type the word **stand**. This will boot the copy of COHERENT that has the floppy disk as its rootdev/pipedev. Also note that if you are booting COHERENT from a hard disk, the secondary bootstrap routine will not prompt you for the name of the kernel to boot if file **/autoboot** exists.

Note that when you are debugging your device driver, you should *not* type **<ctrl><alt><del>** to reboot your machine. This signal is trapped by COHERENT and then processed by the BIOS. The BIOS of some clones of the IBM AT do not reset the hardware correctly; some, such as the AMI BIOS, even leave the processor in the wrong state. The correct way to reboot your machine is to press the **reset** button on the front panel. This is equivalent to turning the machine off and then on again, but does not stress the hardware.

Testing Your Device

This is specific to your device. We urge you, however, to test your device *thoroughly* before you release your driver for public use.

Where to Go from Here

The following section presents source code for two example device drivers: a simple hard-disk driver and a simple serial-port driver. The code is heavily annotated, and illustrates most of the issues that the present section presents only in the abstract.

The last section of this manual is a Lexicon for device-driver routines, commands, and header files. It has entries for all functions that are specific to the kernel (and so can be used in writing drivers), but are not otherwise of use to COHERENT users (and so are not included in the COHERENT system's manual). You should find this to be a good reference manual for all of the functions and most of the technical topics discussed in this manual.

Bibliography

The following references give useful information about the IBM AT, the Intel 80286 microprocessor, and related technical subjects:

Intel Corporation: *iAPX 286 Programmer's Reference Manual*. Santa Clara, Ca.: Intel Corporation, 1985 (part 210498).

Campbell, J.: *C Programmers Guide to Serial Communication*. Indianapolis: Howard Sams & Company, 19?? (ISBN 0-67222-584-0).

Vieillefond, C.: *Programming The 80286*. City, State: SYBEX Inc., 1987 (ISBN 0-89588-277-9).

Crawford, J.; Gelsinger, P.: *Programming The 80386*. City, State: SYBEX Inc., 1987 (ISBN 0-89588-381-3).

IBM Corporation: *Technical Reference, Personal Computer AT, ed. 1* Boca Raton, Fl.: International Business Machines Corporation, 1984.

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

The following publications are not specifically about the COHERENT operating system, but they do teach some basic concepts about device drivers that apply to COHERENT:

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

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

Section 4:
Example Device Drivers

The following appendices give examples of device drivers.

Sample Disk Driver

This simplistic driver is an operational example of a hard-disk driver under the COHERENT operating system. It has the following limitations:

- Works only on an IBM XT (eight-bit) disk controller
- I/O only supports 512 byte (one block) transfers
- Only supports one drive
- The only reported errors are DMA straddles
- No error recovery

The only error checking this driver performs is for DMA straddles and errors returned from the controller. It performs no error recovery, so if it receives an I/O error on a transfer it marks the transfer as bad. In the interest of simplicity, the driver understands only one physical disk drive.

In addition, the physical geometry for the drive is hard-wired into the driver as manifest constants. In a real driver, such as the COHERENT AT hard disk driver, these parameters are read from the system CMOS or from the controller; this avoids having to patch the kernel or recompile the driver in order to change drive types.

Again, please note that this code is meant as an example only. Attempting to use it with the COHERENT system will result in innumerable problems.

Comments that describe the code are interspersed throughout; the comments are printed in Roman type and should not be regarded as part of the code.

The Example

The first seven lines list the machine, system, and driver-specific header files that will be needed for the hard-disk driver.

```
#include <sys/coherent.h>
#include <sys/devices.h>
#include <sys/buf.h>
#include <sys/con.h>
#include <sys/stat.h>
#include <sys/fdisk.h>
#include <sys/uproc.h>
#include <errno.h>
```

The following lines give manifest constants. They define the drive geometry (number of heads, number of cylinders, and number of sectors-per-track); the interrupt vector; controller-port addresses; and bit-mapped definitions such as controller busy and bus direction.

```
#define NXT (1)                         /* # of drives */
#define NXTP (4)                        /* partitions per drive */
#define HEADS (4)                       /* heads per drive */
#define TRK_BLKS (17)                   /* blocks per track */
#define CYL_BLKS (HEADS * TRK_BLKS)     /* blocks per cylinder */
#define CYLINDERS (306)
#define XT_IVEC (5)                     /* hardware interrupt vector # */

#define XT_IO_BASE (0x320)
#define XT_DATA_REG (XT_IO_BASE+0)      /* controller data port address */
#define DISKERR (0x02)                  /* set if error occurred */
#define DRIVE_1 (0x20)                  /* set if err on drive 1 */

#define XT_RESET_REG (XT_IO_BASE+1)     /* controller reset on write */
#define XT_STAT_REG (XT_IO_BASE+1)      /* controller status register */
#define IREQ_STAT (0x20)                /* interrupt request */
#define BUSY_STAT (0x08)                /* controller busy */
#define BUS_STAT (0x04)
#define IO_STAT (0x02)
#define REQ_STAT (0x01)                 /* controller waiting */

#define XT_CONFIG_REG (XT_IO_BASE+2)    /* disk configuration (read) */
#define XT_ATTN_REG (XT_IO_BASE+2)      /* controller select register */
#define XT_ATTN_VAL (3)

#define XT_MASK_REG (XT_IO_BASE+3)      /* controller DMA/int mask reg */
#define XT_MASK_VAL (3)                 /* controller DMA/int mask value */
#define XT_CHAN (3)                     /* controller DMA channel */
```

The following lines define the functions to be used in the driver's configuration table.

```
int     hdopen();
int     hdblock();
int     hdread();
int     hdwrite();
int     hdload();
int     hdunload();
int     hdintr();
int     nulldev();
int     nonedev();
```

The following code defines the structure **hdcon**, which is the configuration table for the driver. The type **CON** comes from header file **<sys/con.h>** and associates the internal driver functions with an external entry point from the kernel.

The first field holds flags that determine the type of the driver, namely whether it is character-special, block-special, or both. In addition, various other attributes are tagged as well. Note that unlike drivers for most other operating systems, a COHERENT device driver can be both character-special and block-special, as in the case of this disk driver.

The second table entry is the driver's major number. This is the index into the driver list array (**drvl**) that the kernel maintains. This number must be in the range of 0-31 inclusive and must not "collide" with the major number of any other driver that *must run in the kernel at the same time*. Giving two device drivers the same major number will generate much unpleasantness. Header file **<sys/devices.h>** lists the major device number of each driver that is currently shipped under COHERENT.

The following fields point to the internal or system routines that are called when a user process attempts to open the device with the major number that corresponds to that found in the second field of this structure. In this case, any device in directory **/dev** that has a major number of **AT_MAJOR** will have all of its calls to **open()**, **close()**, **read()**, **write()**, etc., funnelled to the internal routines indicated here. These work as follows:

open This entry point is called when a user or the system opens the device.

close This entry point is called when a user- or system-level close is performed.

block This entry point provides the block-special interface to the driver. This is called only for devices that display the letter **b** when listed with the command **ls -l**.

read This entry point performs character-special or "raw" reads. It is only used for devices that display the letter **c** when listed with the command **ls -l**.

write This entry point performs character-special or "raw" writes. It is only used for devices that display the letter **c** when listed with the command **ls -l**.

ioctl This entry point provides a mechanism to perform device-specific controlling or requests. For example, on the AT hard-disk driver, it allows a user program to read the hard-disk partitioning information from the driver. In the sample serial program (which follows this example), the **ioctl** entry point could be used to change operation of a serial line, e.g., drop DTR or change word length from seven bits to eight bits.

powerfail This entry point is reserved for future use. When implemented, it will allow device-specific handling of a power fail condition, e.g., abort current hard-disk operation.

timeout This entry point is called periodically by the system. It helps to time or control external events, such as turning off the floppy-drive's motor after four seconds of inactivity.

load This entry point is called either when the system first boots (for drivers linked into the kernel) or when the command **/etc/drvld** loads them (for loadable drivers). This routine should perform all device-specific initialization and set up the internal driver state to run.

unload This entry point corresponds to the *load* entry point. It is called when a loadable driver is requested to unload (**exit**). This entry point is never called for a driver linked into the kernel.

```
CON hdcon = {
        DFBLK|DFCHR,            /* Flags */
        AT_MAJOR,               /* Major index */
        hdopen,                 /* Open */
        nulldev,                /* Close */
        hdblock,                /* Block */
        hdread,                 /* Read */
        hdwrite,                /* Write */
        nonedev,                /* ioctl */
        nulldev,                /* Power fail */
        nulldev,                /* Timeout */
        hdload,                 /* Load */
        hdunload                /* Unload */
};

/*
 * Commands to the controller
 */
#define READ (8)
#define WRITE (10)
```

These lines define the structure **hd**, which is an internal structure used to control operations. **hd** is the head of the list of requests queued for the driver. In addition, it also contains a flag that is set if the driver is busy working on a request.

```
struct {
        BUF     *d_actf;        /* First buffer in queue */
        BUF     *d_actl;        /* Last buffer in queue */
        int     d_busy;
} hd;

BUF hdbuf;          /* buffer used for raw I/O */
```

This line defines the partition table structure used for the hard disk. You can find the actual declaration in header file **<sys/fdisk.h>**.

```
struct fdisk_s hdinfo[NXTP];
```

Function **hdload()** defines the "load" function. Its first line outputs a zero byte to a control port on the disk controller. Its second line associates the internal routine **hdintr** with interrupt number **XT_IVEC** as defined earlier; after a call to **setivec()**, any interrupt processing must be handled by the function **hdintr()**.

```
hdload()
{
        outb( XT_MASK_REG, 0 );
        setivec( XT_IVEC, hdintr );
}
```

Function **hdunload** defines the "unload" function. The call to **clrivec()** resets the interrupt handler associated with interrupt **XT_IVEC** (defined earlier) to the default state (which is to ignore it). Note that your driver must call **clrivec()** before unloading a driver. If it does not, the next interrupt that occurs after the driver exits will will jump to where the interrupt handler used to be, and the system will crash.

In general, the "unload" routine must reset the device to prevent spurious interrupts, as well as reset all the interrupt vectors that were attached via calls to **setivec()**.

Although not demonstrated in the following code, the "unload" routine must also free any memory allocated via calls to any of the kernel-level allocation routines (e.g., **kalloc**), or that memory will be lost until the system is rebooted.

```
hdunload()
{
        outb( XT_MASK_REG, 0 );
        clrivec( XT_IVEC );
}
```

Function **hdopen()** defines the "open" routine that is called when the device is opened. The first argument is a **dev_t**, or device type, that contains the major and minor numbers of the device being opened. The second argument is an integer that gives the "mode," or type of operation desired. The mode flags are defined in header file **<sys/inode.h>**.

```
hdopen(dev, mode)
dev_t dev;
{
```

The following code verifies that the minor number is in range (i.e., makes sense) and that the device being requested actually exists on the machine (i.e., see if hard disk and controller really exist). Drivers for devices that are inherently single user (e.g., the line-printer port) must disallow opens to an already open port. In the case of this hard disk driver, the code noted here checks to see if the device being requested is the "special" device associated with the partition table.

```
        if ( minor(dev) == SDEV )
                return;
```

The following code checks for a valid partition number (i.e., only four partitions per device).

```
        if ( minor(dev) >= NXTP ) {
                u.u_error = ENXIO;          /* bad partition # /*
                return;
        }
```

The following code checks if a valid partition table exists in memory for this disk drive. If not, the call to **fdisk()** should load one into memory. If the load fails or if the requested partition does not exist, **hdopen()** returns an error by setting field **u.u_error** to a value defined in header file **<errno.h>**. In this example, **hdopen()** sets **u.u_error** to **ENXIO**, which indicates a non-existent I/O device.

```
        if (hdinfo[ minor(dev) ].p_size == 0)
                fdisk( makedev(major(dev),SDEV), hdinfo );

        if ( hdinfo[ minor(dev) ].p_size == 0)
                u.u_error = ENXIO;
}
```

Function **hdread()** defines the "read" routine that is called when a user does a read and the device is a "raw" device, as defined above. This simple function merely queues a normal read request through kernel function **dmareq()**, which is a special version of the kernel function **ioreq**. **dmareq()** works through the block I/O system and circumvents DMA straddles. Note that "raw" I/O differs from normal, or "cooked" I/O in that it uses the driver's internal buffer (here called **hdbuf**) to perform the I/O.

Argument **iop** points to the **IO** structure that contains all of the information needed to perform the I/O operation. The **IO** structure is defined in header file **<sys/io.h>**. It includes count, physical address of the I/O buffer, etc.

Argument **dev** is a **dev_t** that specifies the device on which the I/O is being requested.

The last argument to **dmareq()** is either **BREAD** or **BWRITE**. It determines the direction of data transfer.

```
hdread( dev, iop )
dev_t dev;
register IO *iop;
{
        dmareq( &hdbuf, iop, dev, BREAD );
}
```

Function **hdwrite()** defines the "write" routine called when a user does a write and the device is a "raw" device, as defined above. It operates exactly the same as **hdread()**, except that the direction of data transfer is changed from **BREAD** to **BWRITE**.

```
hdwrite( dev, iop )
dev_t dev;
register IO *iop;
{
        dmareq( &hdbuf, iop, dev, BWRITE );
}
```

Function **hdblock()** defines the driver's block I/O interface. It is called with one argument, which points to a **BUF** structure (defined in header file **<sys/buf.h>**).

Local variable **s** is used to store the old interrupt mask returned from the call to kernel function **sphi()**. Variable **lim** is used as a disk address for various computations.

```
hdblock(bp)
register BUF *bp;
{
        register int s;
        daddr_t lim;
```

The following code checks that the user requested exactly one block's worth of I/O. If he did not, it sets an error flag in the **BUF** structure to indicate that some sort of error occurred. The call to **bdone()** tells the block I/O subsystem that we are done with this block.

```
        if ( bp->b_count != BSIZE ) {
                bp->b_flag |= BFERR;
                bdone( bp );
                return;
        }
```

The following block of code checks if the device associated with the current buffer requested is a "special" device, such as the special disk device used to access the partition table on the drive. If it is, the code sets the block limit to the maximum number of blocks on the device (i.e., allow access to any block on the device); if not, it limits the request to any block within the requested partition by using the field **p_size** (partition size) of the partition structure for the given partition.

```
        /* entire device? */
        if ( minor(bp->b_dev) & SDEV )
                lim = CYLINDERS * CYL_BLKS;
        else
                /* single partition */
                lim = hdinfo[minor(bp->b_dev)].p_size;
```

This block block of code verifies that the requested block is within range.

```
        if ( bp->b_bno >= lim ) {
                bp->b_flag |= BFERR;
                bdone( bp );
                return;
        }
```

In the following code, the first line sets the residual count to be one block (i.e., the amount of I/O still to be done). The second line sets the link field in the buffer to NULL; this indicates that no subsequent work is needed after this operation is completed.

```
        bp->b_resid = bp->b_count;
        bp->b_actf  = NULL;
```

The code from this point to the end of the function form a critical section that is prone to "race conditions". Calls to kernel routines **sphi()** and **spl()**, bracket the code; these guarantee that the intervening code is executed as an indivisible operation, with no interrupts changing control flow. This is done to prevent a disk interrupt from accidently calling the hard-disk interrupt handler at a bad time. Usually, **sphi()** and **spl()** are called when manipulating pointers, linked lists, or other critical control structures in the driver. This protects the linked list from damage due to instructions being executed out of sequence.

The five lines following the call to **sphi()** check to see if the driver is busy processing work for a prior request. If not, the link field in the structure **hd** is pointed to the current buffer request. If so, the code links the current request onto the tail of the list that we had prior to **hdblock()** being called.

```
        s = sphi();
        if (hd.d_actf == NULL)
                hd.d_actf = bp;
        else
                hd.d_actl->b_actf = bp;
        hd.d_actl = bp;
```

The following **while** loop checks if the driver was already processing a prior request and if work is to be done. If not, the driver calls **hdgo()** to initiate the I/O to the controller.

```
        while ( !hd.d_busy && (hd.d_actf != NULL) )
                hdgo();
```

Finally, the call to **spl()** restores the processor interrupt mask to what it was prior to the initial call to **sphi()**. Thus, if the interups we enabled prior to the call to **sphi()** were disabled, they are now enabled again. Note that because the call to **hdgo()** is inside the **sphi()/spl()** pair, this function will also run with interrupts disabled.

```
        spl(s);
}
```

The following function **hdgo()** talks to the controller, i.e., "bangs on the hardware". Variable **bp** points to a buffer. The integer variables are self-explanatory. **cmdbuf** is a six-byte array in which the function contructs the command packet that it gives to the controller to initiate the I/O operation. Note that as this example driver supports only one drive, it does not support overlapping seeks or any of the other performance enhancements found in sophisticated disk drivers.

```
hdgo()
{
        register BUF *bp;
        register int i, blk, head, cyl, sector;
        register int loopcnt;
        char cmdbuf[6];
```

The following subroutine checks for work to do.

```
        if ( (bp = hd.d_actf) == NULL )
                return;
```

This subroutine sets up the DMA request for this I/O. The manifest constant **XT_CHAN** (defined above) gives the DMA channel to be used. Needless to say, the DMA channels must be chosen so there is no conflict between devices trying to perform DMA operations.

The second argument gives the physical address from/to which I/O will be performed.

The third argument gives the number of bytes to transfer.

The fourth argument indicates whether the I/O is a write operation, thus controlling the direction of the DMA transfer.

If **dmaon()** returns an error, it is due to a DMA straddle. This condition occurs when the buffers for an I/O request span a 64-kilobyte physical-address boundary. Due to the poor design of the DMA in the IBM PC family of computers, the DMA chip can only address 16 bits (64 kilobytes). To DMA from any location in memory, the hardware designers added a latch that controls the high-order address bits. In the case of the PC/XT/AT, the latch has four bits, giving a total of 20 bits (one megabyte) of addressability. Thus, I/O operations cannot cross 64-kilobyte physical address boundaries.

```
if (dmaon(XT_CHAN, bp->b_paddr, bp->b_count, \
          bp->b_req==BWRITE) == 0) {
      printf("hd: DMA straddle\n");
      goto error;
}

blk = bp->b_bno;
```

The first two lines of the following code increment variable **blk** which converts the logical block number to a physical block number. The following lines then convert the physical block number to the corresponding head/cylinder/sector numbers.

```
if ((bp->b_dev & SDEV) == 0)
      blk += hdinfo[ minor(bp->b_dev) ].p_base;
head = blk % CYL_BLKS / TRK_BLKS;                /* 0-3 */
cyl = blk / CYL_BLKS;                            /* 0-305 */
sector = blk % CYL_BLKS % TRK_BLKS ;             /* 0-16 */
```

These lines load the command packet that will be transfered to the controller.

```
cmdbuf[0] = (bp->b_req == BREAD) ? READ : WRITE;
cmdbuf[1] = ((minor(bp->b_dev) / NXTP) << 5) + head;
cmdbuf[2] = ((cyl >> 8) << 6) + sector;
cmdbuf[3] = cyl;
cmdbuf[4] = 1;           /* bp->b_count / BSIZE */
cmdbuf[5] = 5;           /* default 70 microsec per step */
```

These lines set up the controller for the I/O request.

```
/* attract controller's attention */
outb(XT_ATTN_REG, XT_ATTN_VAL);
/* set DMA/interrupt mask */
outb(XT_MASK_REG, XT_MASK_VAL);
```

These lines wait for the controller to enter a "request state" where it is ready to accept a command packet.

```
        loopcnt = 0;
        while ((inb(XT_STAT_REG)&0xf) != \
                        (BUSY_STAT|BUS_STAT|REQ_STAT))
                if ( --loopcnt == 0 )
                        goto error;
```

This block of code outputs the command packet to the controller. The code busy-waits until the command is executed. Given that the controller takes virtually no time to process each byte in the command packet, busy-waiting the bytes is not significant in terms of time.

```
        for ( i=0; i < 6; i++ ) {
                loopcnt = 0;
                while ( (inb(XT_STAT_REG) & REQ_STAT) != REQ_STAT )
                        if ( --loopcnt == 0 )
                                goto error;
                outb( XT_DATA_REG, cmdbuf[i] );
        }
```

This line enables the DMA controller for this channel. The DMA proceeds at its own rate, paced by the data going to or coming from the controller.

```
        dmago( XT_CHAN );
```

These lines check the controller to see that it has exited the "request state".

```
        if ( inb(XT_STAT_REG) & REQ_STAT )
                goto error;
```

This line sets an internal flag that indicates that we are now busy doing an I/O operation. This flag keeps this function from tripping over its own feet.

```
        hd.d_busy = 1;
        return;
```

The code that follows the label **error** shuts down the controller and DMA. The function **goto**'s this point if an error occurs, as well as flagging the current I/O as bad so the caller will know that the I/O failed for some reason. It calls **hddone()** to finish up processing for this block.

```
error:
        outb( XT_MASK_REG, 0 );
        dmaoff( XT_CHAN );
        bp->b_flag |= BFERR;
        hddone( bp );
}
```

Function **hdintr()** is the hard-disk interrupt handler. It is called when the system receives an interrupt from the disk controller, as set by the call to **setivec()** (see above). No further interrupts can nest while this interrupt is being processed, so the function need not call **sphi()** to disable interrupts.

```
hdintr()
{
        register BUF *bp;
```

This code checks to see if any work is in progress. If not, the interrupt handler ignores the interrupt and returns.

```
        if ( (bp = hd.d_actf) == NULL )
                return;
```

The first **if** statement in this block of code calls the kernel routine **inb()** to check whether the controller is in the correct state for further processing. The second **if** statement calls **inb** to check for an I/O error. If one has occurred, the code sets field **bp->b_flag** to constant **BFERR** to flag that the current block has had an error. If an I/O error has not occurred, we know that I/O has completed; thus, the code signifies this fact by setting the residual count to zero.

```
        if ( inb(XT_STAT_REG) & IREQ_STAT ) {
                if ( inb(XT_DATA_REG) & DISKERR )
                        bp->b_flag |= BFERR;
                else
                        bp->b_resid = 0;
```

Here, the first two lines shut down the controller and turn off the DMA for this channel. The third line calls **hddone()**, described below, to finish processing the current block.

```
                outb( XT_MASK_REG, 0 );
                dmaoff( XT_CHAN );
                hddone( bp );
```

The following lines check for more work to do. If so, it calls **hdgo()** to initiate requests to the controller for the next waiting request. At this point, the driver returns from the interrupt handler to the system interrupt handler that called it. The system part of the interrupt handler will context-switch back to where it was prior to the interrupt being serviced.

```
                while ( (hd.d_busy == 0) && (hd.d_actf != NULL) )
                        hdgo();
        }
}
```

Finally, function **hddone()** performs tail-end processing for a block. The first line of the function walks down the linked list to the next request to be processed, if any. The second line tells the block I/O subsystem that the driver is done with the current block. The third line sets the internal flag to indicate that the driver is no longer busy executing an I/O.

```
hddone( bp )
register BUF *bp;
{
        hd.d_actf = bp->b_actf;
        bdone( bp );
        hd.d_busy = 0;
}
```

Sample Serial Device Driver

The following code gives an example of a simple driver for a serial port. It has the following features:

- **Supports PC COM1 and COM2 serial ports**
- **Supports V7-compatible ioctl(), as defined in header file <sgtty.h>**

Again, please note that this code is meant as an example only. The code is interspersed with notes, which appear in Roman type. The notes mainly describe points where this driver differs from the one described in the previous example.

The Example

```
#include <sys/coherent.h>
#include <sys/ins8250.h>
#include <sys/clist.h>
#include <sys/stat.h>
#include <sys/uproc.h>
#include <sys/proc.h>
#include <sys/tty.h>
#include <sys/con.h>
#include <sys/devices.h>
#include <errno.h>

/*
 * Manifest constants.
 */
#define COM1VEC                 4           /* interrupt vector for COM1 */
#define COM2VEC                 3           /* interrupt vector for COM2 */
#define COM1PORT                0x3F8       /* i/o port address for COM1 */
#define COM2PORT                0x2F8       /* i/o port address for COM2 */
```

The following line defines the port address associated with a given **COM** port. In this case, we use the "device-dependent parameter" field in the **TTY** structure to store the port address that corresponds to the port. This field is a **char *** by definition, but can contain anything the programmer wishes; for our purposes, we must cast to **int** to ensure that we get the size/type correct for our uses.

```
#define      PORT    ((int)(tp->t_ddp))
```

```
/*
 * Functions.
 */
int    slload();
int    slunload();
int    slopen();
int    slclose();
int    slread();
int    slwrite();
int    slioctl();
int    slpoll();
int    nulldev();
int    nonedev();
```

The first two lines here declare the two interrupt handlers that the driver will use: one per interrupt line/port.

```
int    sl0intr();
int    sl1intr();
int    slparam();
int    slstart();
```

The following line specifies that the driver's routine **slcycle()** will be called when the kernel invokes our "timeout" handler. If enabled, this entry is called once per second and used either to time events or to handle some specific processing at regular intervals.

```
int    slcycle();

/*
 * Configuration table.
 */
CON slcon ={
       DFCHR,                          /* Flags */
       AL0_MAJOR,                      /* Major index */
       slopen,                         /* Open */
       slclose,                        /* Close */
       nulldev,                        /* Block */
       slread,                         /* Read */
       slwrite,                        /* Write */
       slioctl,                        /* Ioctl */
       nulldev,                        /* Powerfail */
       slcycle,                        /* Timeout */
       slload,                         /* Load */
       nulldev,                        /* Unload */
```

slpoll() is our "poll" routine, which lets the driver support UNIX System V-style device polling.

```
       slpoll                          /* Poll */
};
```

The array **sltty[]** holds the **TTY** structures for our two teletypewriter devices. See header file **<sys/ktty.h>** for details on the **TTY** structure. The first two structure members are aggregate types, so they need braces to initialize them. Member 3 is field **t_ddp**, which the driver uses to hold the hardware port address for the given port. The fourth member initializes the field **t_start**; it points to a function to be called when we desire to start output to a port. The common tty driver code calls it as needed. Member 5 initializes the field **t_param**; this points to the function to call when it is necessary to change port parameters, e.g., bit rate, word length, or parity. The common tty driver also calls it as needed. Members 6 and 7 initialize fields **t_dispeed** and **t_dospeed**, and correspond, respectively, to the default input and output speeds.

```
TTY sltty[2] = {
	{ {0}, {0}, COM1PORT, slstart, slparam, B9600, B9600 },
	{ {0}, {0}, COM2PORT, slstart, slparam, B9600, B9600 }
};
```

The array **timeconst[]** forms the divisor table that the driver uses to set the speed on a port. This table is indexed by the bit rates defined in the tty headers. The driver takes these values and outputs them to the divisor registers on the UARTs. The UART then divides its internal clock by this value to derive the bit-rate clock used for transmit and receive operations.

```
static
int timeconst[] = {
	0,			/* 0 */
	2304,			/* 50 */
	1536,			/* 75 */
	1047,			/* 110 */
	857,			/* 134.5 */
	768,			/* 150 */
	576,			/* 200 */
	384,			/* 300 */
	192,			/* 600 */
	96,			/* 1200 */
	64,			/* 1800 */
	58,			/* 2000 */
	48,			/* 2400 */
	32,			/* 3600 */
	24,			/* 4800 */
	16,			/* 7200 */
	12,			/* 9600 */
	6,			/* 19200 */
	6,			/* EXTA */
	6			/* EXTB */
};
```

Function **slload()** forms the "load" routine. Because it manipulates the hardware, the code brackets the internal operations with calls to the kernal routines **sphi()** and **spl()**, to protect internal structures from being updated incorrectly.

```
slload()
{
        register TTY *tp;
        register int s;
        static int init;

        s = sphi();
```

This **if** statement checks to if the driver has already gone through this routine; it bails out if this is the case.

```
        if ( !init ) {
```

In the following code, the first line initializes a pointer to a **TTY** structure so that it points to the parameters specific to this port. The following line, the call to **slparam()** sets up the port to the default values we specified.

```
                tp = &sltty[0];
                slparam( tp );
```

The **if** statement calls the kernel routine **inb()** to check whether the desired **COM** port exists. If the port exists, then the following lines set up the interrupt handler.

```
                if ( inb( PORT+IER ) == 0 ) {
                        setivec( COM1VEC, sl0intr );
                        init++;
                }

                /*
                 * Initialize COM2 and interrupt vector.
                 */
                tp = &sltty[1];
                slparam( tp );
                if ( inb( PORT+IER ) == 0 ) {
                        setivec( COM2VEC, sl1intr );
                        init++;
                }
```

The **if** statement checks if any ports were found. If so, the following line enables the periodic one-second timer by setting a flag in the driver list array for this driver.

```
                if ( init )
                        drvl[AL0_MAJOR].d_time = 1;
        }
        spl( s );
}
```

```
/*
 * Unload Routine.
 */
slunload()
{
	/*
	 * Reset COM1 and interrupt vector.
	 */
	clrivec( COM1VEC );                  /* release interrupt vector */
	outb( COM1PORT+IER, 0 );             /* disable port interrupts */
	outb( COM1PORT+MCR, MC_OUT2 );       /* hangup port */

	/*
	 * Reset COM2 and interrupt vector.
	 */
	clrivec( COM2VEC );                  /* release interrupt vector */
	outb( COM2PORT+IER, 0 );             /* disable port interrupts */
	outb( COM2PORT+MCR, MC_OUT2 );       /* hangup port */

	/*
	 * Cancel periodic polling.
	 */
	drvl[AL0_MAJOR].d_time = 0;
}

/*
 * Open Routine.
 */
slopen( dev, mode )
dev_t dev;
{
	register TTY *tp = &sltty[ dev & 1 ];
	register int s;

	/*
	 * Validate minor device.
	 */
	if ( minor(dev) > 1 ) {
		u.u_error = ENODEV;
		return;
	}

	/*
	 * Initialize hardware.
	 */
	slload();
```

```
        /*
         * Verify hardware exists.
         */
        if ( inb(PORT+IER) & -(IE_RxI|IE_TxI|IE_LSI) ) {
                u.u_error = ENXIO;
                return;
        }
```

In the function **slopen()**, this line calls the kernel routine **ttsetgrp()** to associate a process group with this port. This means that all processes related to the one that opened the port will have the port as the controlling terminal, and that they will be considered as a group for certain terminal-related functions.

```
        ttsetgrp( tp, dev );

        /*
         * Initialize if not already open.
         */
        if ( ++tp->t_open == 1 ) {

                tp->t_flags &= -T_MODC;
                tp->t_flags |=  T_CARR;
```

These lines call the common tty driver code to handle functions related to opening a terminal port. This call must be bracketed by calls to the kernel routines **sphi()** and **spl()** to avoid a race condition with the **slclose()** routine.

```
                s = sphi();
                ttopen( tp );
                spl( s );
```

These lines first set the input and output speeds to the default values from the port's **TTY** structure. Then, they call off **slparam()** to manipulate the hardware.

```
                tp->t_sgttyb.sg_ispeed = tp->t_dispeed;
                tp->t_sgttyb.sg_ospeed = tp->t_dospeed;
                slparam( tp );
        }
}
```

Function **slclose()** checks if this call is the last one to close a port. If this is not the case, then the function returns. This allows us to execute multiple opens and closes on a port, yet ensure that only the last one has to "turn out the lights". Once again, this function calls the kernel function **ttclose()** (the common tty-driver close routine) to clean up house; and does so at high priority to avoid race conditions with the open routine.

```
slclose( dev )
dev_t dev;
{
	register TTY *tp = &sltty[ dev & 1 ];
	register int s;

	/*
	 * Reset if last close.
	 */
	if (--tp->t_open == 0) {
		/*
		 * call common tty driver code
		 */
		s = sphi();
		ttclose( tp );
		spl( s );
	}
}
```

Function **slread()** is this driver's portion of the the "read" routine. For the sake of simplity (this is an example, after all), it just calls the kernel function **ttread()** and lets it do our work. Because **ttread()** handles the character queues for the ports, it will actually process the I/O request, blocking if necessary to wait for further input from the port.

```
slread( dev, iop )
dev_t dev;
register IO *iop;
{
	ttread( &sltty[ dev & 1 ], iop, 0 );
}
```

Function **slwrite()** is structured the same as **slread()**: it simply calls the kernel function **ttwrite()**, which performs writes for the common tty driver. It queues the characters and calls the routine specified in field **t_start** of the **TTY** structure for this device to perform the actual output.

```
slwrite( dev, iop )
dev_t dev;
register IO *iop;
{
	ttwrite( &sltty[ dev & 1 ], iop, 0 );
}
```

Function **slioctl()** creates a simple ioctl function. Because the driver does not support any ioctl's other than the basic ones provided by the common tty driver, this function just calls the tty driver to do the work. **slioctl()** does this at high priority to avoid race conditions with interrupts.

```
slioctl( dev, com, vec )
dev_t dev;
int com;
struct sgttyb *vec;
{
	register int s;

	s = sphi();
	ttioctl( &sltty[ dev & 1 ], com, vec );
	spl( s );
}

/*
 * Polling Routine.
 * [System V.3 Compatible]
 */
slpoll( dev, ev, msec )
dev_t dev;
int ev;
int msec;
{
	return ttpoll( &sltty[dev&1], ev, msec );
}
```

Function **slcycle()** is the timeout-processing function mentioned earlier; as noted there, this function runs at one-second intervals. **slcycle()** checks both **COM1** and **COM2** to see if any of the modem-control leads have changed state since the function last ran (i.e., in the previous second). If this is so, it calls the appropriate interrupt handler to service the modem-control changes.

```
slcycle()
{
	register TTY *tp;
	register int s;

	s  = sphi();

	tp = &sltty[0];
	if ( (inb(PORT+IER) & -(IE_RxI|IE_TxI|IE_LSI)) == 0 )
		sl0intr();

	tp = &sltty[1];
	if ( (inb(PORT+IER) & -(IE_RxI|IE_TxI|IE_LSI)) == 0 )
		sl1intr();

	spl( s );
}
```

Function **sl0intr()** is the interrupt handler for COM1. The main body of code is within a **for** loop; this allows the driver to process multiple conditions that may exist simultaneously.

```
sl0intr()
{
        register TTY *tp = &sltty[0];
        register int b;

        /*
         * Service serial port interrupt requests, highest
         * to lowest priority.
         * Pass off to common tty driver code as needed.
         */
        for (;;) {
                b = inb( PORT+IIR );
                switch ( b ) {
```

Case **LS_INTR** is for line-status interrupts. Here, if the driver detects a framing error (break condition), it calls the kernel function **ttsignal()** to send an interrupt signal to all processes within the process group.

```
                case LS_INTR:
                        if ( inb( PORT+LSR ) & LS_BREAK )
                                ttsignal( tp, SIGINT );
                        break;
```

Case **Rx_INTR** is a receive-interrupt condition. If this occurs, the driver calls the kernel function **inb()** to read the character from the UART. If the port is currently open, the driver calls the kernel function **ttin()** to pass the character to the tty driver's input routine; **ttin()**, in turn, queues it in the queue associated with this port.

```
                case Rx_INTR:
                        b = inb( PORT+DREG );
                        if ( tp->t_open )
                                ttin( tp, b );
                        break;
```

Case **Tx_INTR** indicates that a transmit interrupt occurred due to the transmit buffer on the UART becoming empty. Here, the driver calls the kernel function **ttstart()** to let the common tty driver know that we can send another character.

```
                case Tx_INTR:
                        ttstart( tp );
                        break;
```

Finally, case **MS_INTR** indicates that a modem-status interrupt occurred. Here, the driver simply calls the kernel function **inb()** to read the modem-status register. This acknowledges that the error occurred, but does nothing about it; this is, after all, a simple driver.

```
                case MS_INTR:
                        inb( PORT+MSR );
                        break;

                default:
                        return;
                }
        }
}
```

Function **sl1intr()** is the interrupt handler for port **COM2**. It behaves the same as **sl0intr()**.

```
sl1intr()
{
        register TTY *tp = &sltty[1];
        register int b;

        /*
         * Service serial port interrupt requests,
         * highest to lowest priority.
         * Pass off to common tty driver code as needed.
         */
        for (;;) {
                switch ( inb( PORT+IIR ) ) {
                case LS_INTR:
                        if ( inb( PORT+LSR ) & LS_BREAK )
                                ttsignal( tp, SIGINT );
                        break;

                case Rx_INTR:
                        b = inb( PORT+DREG );
                        if ( tp->t_open )
                                ttin( tp, b );
                        break;

                case Tx_INTR:
                        ttstart( tp );
                        break;

                case MS_INTR:
                        inb( PORT+MSR );
                        break;
```

```
		default:
			return;
		}
	}
}
```

Function **slstart()** is the "start" routine that the tty driver calls when someone (or something) needs to write a character to a port. The body of this function is bracketed by calls to the kernel functions **sphi()** and **spl()**, to protect it against untoward interruption.

The driver first calls the kernel function **inb()** and checks what it returns to see if the port is already busy sending data. If it is not, the funtion then calls the kernel function **ttout()** to check if characters must be output on this port. Note that **ttout()** returns an eight-bit unsigned character in the low-order eight bits, so there is no chance of any valid output character evaluating to less than zero (i.e., nothing to send). If characters are to be sent, then the function calls the kernel function **outb()** to send the character it obtained from **ttout()**.

```
slstart( tp )
register TTY * tp;
{
	register int b;
	int s;

	s = sphi();
	if ( inb( PORT+LSR ) & LS_TxRDY )
		if ( (b = ttout(tp)) >= 0 )
			outb( PORT+DREG, b );
	spl( s );
}
```

Function **slparam()** is the machine-dependent code that sets parameters on the specified device. These include modem control leads, character size, and parity.

```
slparam( tp )
register TTY * tp;
{
	register int b;
	int s;

	s = sphi();

	/*
	 * Assert required modem control lines (DTR, RTS).
	 */
	b = MC_OUT2;
	if ( tp->t_sgttyb.sg_ospeed != B0 )
		b |=  MC_DTR | MC_RTS;
	outb( PORT+MCR, b );
```

```
        /*
         * Program baud rate.
         */
        if (b = timeconst[ tp->t_sgttyb.sg_ospeed ]) {
                outb( PORT+LCR, LC_DLAB );
```

These two lines output to the UART, respectively, the low and high bytes of the divisor.

```
                outb( PORT+DLL, b );
                outb( PORT+DLH, b >> 8 );
        }

        /*
         * Program character size, parity.
         */
        switch ( tp->t_sgttyb.sg_flags & (EVENP|ODDP) ) {
        case ODDP:
                b = LC_CS7|LC_PARENB;
                break;

        case EVENP:
                b = LC_CS7|LC_PARENB|LC_PAREVEN;
                break;
```

Finally, this case tests to "ignore parity", since simultaneously setting **EVENP** and **ODDP** allows for either parity.

```
        case EVENP|ODDP:
        default:
                b = LC_CS8;
                break;
        }
        outb( PORT+LCR, b );

        /*
         * Enable desired serial interrupts.
         * Unreliable operation if both receive and modem
         * interrupts enabled.
         */
        b = 0;

        if ( tp->t_sgttyb.sg_ispeed != B0 )
                b |= IE_TxI | IE_LSI;
        if ( tp->t_open != 0 )
                b |= IE_RxI;
        outb( PORT+IER, b );
```

```
		spl( s );
	}
```

Section 5:
The Lexicon

The following section describes each function and macro available for use with device drivers, in Lexicon format.

The following overview articles introduce clusters of related articles:

accessible kernel routines
block-device routines
driver-access routines
header files
interrupt-handler routines
I/O routines
kernel variables
memory-manipulation routines
segment-manipulation routines
signal-handler routines
terminal-device routines

Each overview article introduces and lists its set of related articles.

accessible kernel routines — Overview

The COHERENT kernel contains a number of routines that can be accessed by device drivers. They are as follows:

defend	Execute deferred functions
defer	Defer function execution
dmago	Enable DMA transfers
dmaoff	Disable DMA transfers
dmaon	Prepare for DMA transfer
dmareq	Request block I/O, avoiding DMA straddles
inb	Read a byte from an I/O port
lock	Lock a gate
locked	See if a gate is locked
outb	Output a byte to an I/O port
panic	Fatal system error
pollopen	Initiate driver polled event
pollwake	Terminate driver polled event
printf	Formatted print
sleep	Wait for event or signal
super	Verify super-user
timeout	Defer function execution
unlock	Unlock a gate
wakeup	Wakeup processes sleeping on an event

See Also

device drivers

actvsig() — Signal-Handler Routine

Activate signal handler
actvsig()

The routine **actvsig** activates a signal handler. For example:

```
          if (SELF->p_ssig && nondsig())
               actvsig();
```

If the current process has received a signal (**p_ssig** being non-zero) that is not ignored (not default signal handling), calling **actvsig** will activate it. "Activate" means that the process is moved from the kernel's "suspended" list to its "ready" list, where it will await further execution by the kernel. If the current process is terminated, **actvsig** will not return.

See Also

signal-handler routines

aha154x — Device Driver

Adaptec AHA-154x device driver

The device driver **aha154x** lets you use SCSI interface devices attached to an Adaptec AHA-154x series host adapter. This driver has major number 13. It can be accessed either as a block-special device or as a character-special device. The minor number specifies the device and partition number for disk-type devices, letting you use up to eight SCSI-IDs, with up to four logical unit numbers (LUNs) per SCSI-ID and up to four partitions per LUN.

The first **open** call on a SCSI disk device allocates memory for the partition table and reads it into memory.

Controller Configuration

Prior to installing the Adaptec host adapter in your system, you must configure the I/O base address, interrupt vector and DMA channel as follows:

I/O base address:	0x330
DMA channel:	5
Interrupt vector:	IRQ11

In addition, if you are using any synchronous SCSI peripherals, disable the synchronous transfer option on the Adaptec host adapter.

After verifying that your controller works with COHERENT, you may select an alternate I/O base address or an alternate interrupt vector. Device driver variables **SDBASE_** and **SDIRQ_** correspond to the I/O base address and interrupt vector, respectively. See Lexicon article **hs** for an example of how to configure a device driver.

When processing BIOS I/O requests prior to booting COHERENT, the Adaptec host adapter uses "translation mode" drive parameters: number of heads, cylinders, and sectors per track. Most current versions of the AHA-154x use values of 64 heads and 32 sectors per track, and calculate the number of cylinders based upon drive capacity. Note that these numbers are called translation-mode parameters because they have nothing to do with the physical drive geometry. Some early versions of the AHA-154x, and some versions distributed by Tandy, use 16 heads and 32 sectors per track. Device driver variable **SD_HDS_** is initialized to 64 as shipped; it should be patched to a value of 16 for adapters whose BIOS code uses 16-head translation mode. The translation-mode parameters used by the BIOS code present on your host adapter can be obtained using the **dpb** utility found on the boot diskette of versions 3.2.0 and later of COHERENT. Note that the BIOS code is executed by COHERENT only during initial bootstrap. After that, drive parameters are of no consequence since SCSI I/O requests are based upon logical block number, rather than on cylinder/head/sector addressing.

The installation procedure for COHERENT versions 3.2.0 and later patches all necessary variables for the accompanying version of the **aha154x** driver by executing the command:

```
/etc/mkdev scsi
```

Minor Device Numbers

The minor device number is decoded as follows:

Bit number:	7	6	5	4	3	2	1	0
Meaning:	S	I	I	I	L	L	P	P

where *S* indicates the "special" bit, *III* indicates a three-bit field containing the SCSI-ID in the range of zero through seven, *LL* indicates a two-bit field containing a LUN in the range of zero through three, and *PP* indicates a two-bit field that contains either a partition number for disk-type devices or a set of special modes for devices other than disks.

The "special" bit and the partition number interact as follows:

Description	*S Bit*	*PP*	*Device*	*Type*
partition a	0	00	/dev/sd?a	disk
partition b	0	01	/dev/sd?b	disk
partition c	0	10	/dev/sd?c	disk
partition d	0	11	/dev/sd?d	disk
partition table	1	00	/dev/sd?x	disk
no rewind	1	01	/dev/sd?n	tape
RESERVED	1	10	---	----
rewind on close	1	11	/dev/sd?	tape

Loading the Driver

The **aha154x** loadable device driver must be loaded on a system that does not have a SCSI hard disk as the root device. To do so, use the command **/etc/drvld**, as follows:

```
/etc/drvld -r /drv/aha154x
```

Files

/dev/sd* — block-special devices
/dev/rsd* — character-special devices

See Also

device drivers, drvld, scsi

Notes

This release of the **aha154x** device driver only supports disk-type devices. A future version of the driver will add support for tape-type and other devices.

altclk_in() — Accessible 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*. Function *fn* will be called every time 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 0 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 0 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.

Example

The following gives a partial example of how to use **altclk_in** in a device driver.

```
#include <sys/const.h>      /* #define's HZ */
...

static int scale_factor;
static int poll_fn();
...
```

```
          /* install high-speed polling of I/O device */
          poll_rate = ...;
          scale_factor = poll_rate/HZ;
          altclk_out();
          altclk_in(poll_rate, poll_fn);
     ...
     /* polling function */
     int poll_fn()
     {
          static int count;

          ...do device polling...

          count++;
          if (count >= scale_factor)
               count = 0;
          return count;
     }
```

See Also

accessible kernel routines, altclk_out

Notes

To use this function, link module **clocked.o** into the kernel. Avoid naming the polling function **altclk:** there is already a kernel symbol with this name.

altclk_out() — Accessible Kernel Function

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.

Calling **altclk_out** when polling is not already in effect does not affect the system; the function simply does nothing and returns NULL. To change polling rate, call **altclk_out**, then **altclk_in.**

See Also

accessible kernel routines, alkclk_in

Notes

To use this function, link module **clocked.o** into the kernel. Avoid naming the polling function **altclk:** there is already a kernel symbol with this name.

at — Device Driver

Drivers for hard-disk partitions

/dev/at* are the COHERENT system's AT devices for the hard-disk's partitions. Each device is assigned major-device number 11, and may be accessed as a block- or character-special device.

The **at** hard-disk driver handles two drives with up to four partitions each. Minor devices 0 through 3 identify the partitions on drive 0. Minor devices 4 through 7 identify the partitions on drive 1. Minor device 128 allows access to all of drive 0. Minor device 129 allows access to all of drive 1. To modify the offsets and sizes of the partitions, use the command **fdisk** on the special device for each drive (minor devices 128 and 129).

To access a disk partition through COHERENT, directory **/dev** must contain a device file that has the appropriate type, major and minor device numbers, and permissions. To create a special file for this device, invoke the command **mknod** as follows:

```
/etc/mknod /dev/at0a b 11   0 ; : drive 0, partition 0
/etc/mknod /dev/at0b b 11   1 ; : drive 0, partition 1
/etc/mknod /dev/at0c b 11   2 ; : drive 0, partition 2
/etc/mknod /dev/at0d b 11   3 ; : drive 0, partition 3
/etc/mknod /dev/at0x b 11 128 ; : drive 0, partition table
```

Drive Characteristics

When processing BIOS I/O requests prior to booting COHERENT, many IDE drives use "translation-mode" drive parameters: number of heads, cylinders, and sectors per track. These numbers are called translation-mode parameters because they do not reflect true physical drive geometry. The translation-mode parameters used by the BIOS code present on your host adapter can be obtained using the **dpb** utility found on the boot diskette of versions 3.2.0 and later of COHERENT. It is often necessary to patch the **at** driver with BIOS values of translation-mode parameters in order to boot COHERENT on IDE hard drives. In COHERENT versions 3.1.0 and later, drive parameters are stored in table **atparm_** in the driver. For the first hard drive, number of cylinders is a two-byte value at **atparm_+0**, number of heads is a single byte at **atparm_+2**, and number of sectors per track is a single byte at **atparm_+14**. For the second hard drive, number of cylinders is a two-byte value at **atparm_+16**, number of heads is a single byte at **atparm_+18**, and number of sectors per track is a single byte at **atparm_+30**. For example, if **testcoh** is a kernel linked with the **at** driver and you want to patch it for a second hard drive with 829 cylinders, 10 heads, and 26 sectors per track, you can do

```
    /conf/patch testcoh atparm_+16=829 atparm_+18=10:c atparm_+30=26:c
```

To read the characteristics of a hard disk once the **at** driver is running, use the call to **ioctl** of the following form:

```
    #include <sys/hdioctl.h>
    hdparm_t hdparms;
        . . .
    ioctl(fd, HDGETA, (char *)&hdparms);
```

where *fd* is a file descriptor for the hard disk device and *hdparms* receives the disk characteristics.

Non-Standard and Unsupported Types of Drives

Prior releases of the the COHERENT **at** hard-disk driver would not support disk drives whose geometry was not supported by the BIOS disk parameter tables. COHERENT adds support for these drives during installation by "patching" the disk parameters into the bootstrap and the **/coherent** image on the hard disk.

Files

/dev/at* — Block-special files
/dev/rat* — Character-special files

See Also

device drivers, fdisk

ati — Device Driver

ATI Graphics Solution Driver

ati is a special version of the normal console driver that lets you use the ATI Graphic Solution adapter's ability to change the size of the screen. Normally, this driver is major device 2 and minor device 0, and is accessed as a character-special device (default, **/dev/console**).

The following special escape sequences apply to the ATI Graphics Solution adaptor: 132 columns are supported with both the monochrome and color modes of the adaptor.

<ctrl-N>
Place the console into 40-column mode.

<ctrl-O>
Place the console into 80-column mode.

<ctrl-W>
Place the console into 132-column mode.

All other capabilities that apply to the normal console driver also apply to the ATI driver.

See Also

device drivers

Files

/dev/console — Character-special file

Notes

Color is supported by this interface.

bclaim() — Block-Device 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 contents are invalid if its field *b_flag* has the BFNTP bit set.

bclaim should not be called from deferred or timed functions, or by interrupt handlers.

See Also

block-device routines

bdone() — Block-Device Routine

Block I/O completed

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

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

See Also

block-device routines

bflush() — Block-Device 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. The kernel typically uses this routine when it unmounts file systems.

See Also

block-device routines

block-device routines — Overview

The following routines can be used by device drivers to access block-special devices:

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
bwrite	Write buffer to disk

See Also

device drivers

bread() — Block-Device 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** will wait for I/O to complete), and **bread** will return 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

block-device routines

brelease() — Block-Device 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 should always call **brelease** when it no longer needs a buffer obtained via a **bread**. If a driver needs to 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

block-device routines

bsync() — Block-Device Routine

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

bsync flushes modified buffers to all buffered devices, thus synchronizing the entire buffer cache.

See Also

block-device routines

Build — Command

Build a new version of the kernel
/usr/sys/Build *option_list*

Build is a shell script that automates the building of a new version of the COHERENT kernel. It invokes **make** to recreate each device driver to be linked into the kernel, as set by an internal variable, then calls the command **config** to recreate the kernel.

option_list is a list of device drivers which need to be linked into the kernel.

This script is meant to be used only by experienced writers of device drivers. Directions for modifying it to recreate the kernel are given in section 2 of the manual to the COHERENT Device Driver kit.

Examples

For example, an invocation of:

```
Build at nkb
```

would build a COHERENT kernel using the **at** device driver for the AT/IDE interface hard disk, using device driver **nkb** which is the user configurable keyboard device driver.

An alternate configuration could be:

```
Build ss kb
```

which would build a COHERENT kernel using the **ss** device driver for the Seagate and Future Domain SCSI interface hard disk, using device driver **kb** which is the traditional COHERENT keyboard device driver.

See Also

config, device drivers

bwrite() — Block-Device Routine

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

bwrite writes out the buffer pointed to by *bp*. If *flag* is set, the write is synchronous, and **bwrite** will not return until the I/O has completed; otherwise, it is asynchronous and **bwrite** will return immediately.

A device driver must first lock the buffer gate before it calls **bwrite**; otherwise, the buffer may be modified while it is being written.

See Also

block-device routines

clist.h — Header File

Character-list structures
#include <sys/clist.h>

The header file **clist.h** holds definitions useful to functions that manipulate character lists. It defines the character-list structure **CLIST** and the character-queue structure **CQUEUE**.

See Also

device drivers, header files

clrivec() — Interrupt-Handler Routine

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

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

See Also

interrupt-handler routines, setivec

Notes

You should call **clrivec** only from the **load()** or **unload()** routines of a driver.

clrq() — Terminal-Device Routine

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

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

See Also

terminal-device routines

coherent.h — Header File

Miscellaneous useful definitions
#include <sys/coherent.h>

The header file **coherent.h** holds miscellaneous definitions that are useful to writers of device drivers. Among other things, it defines the structure **TIME**, and declares most of the accessible kernel variables.

See Also

device drivers, header files

com — Device Driver

Device drivers for asynchronous serial lines

The COHERENT system has drivers for four asynchronous serial lines, **com1** through **com4.**

A serial line can be opened into any of four different "flavors", as follows:

com?l	Interrupt driven, local mode (no modem control)
com?r	Interrupt driven, remote mode (modem control)
com?pl	Polled, local mode (no modem control)
com?pr	Polled, remote mode (modem control)

"Local mode" means that the line will have a terminal plugged into it, to directly access the computer. "Modem control" means that the line will have a modem plugged into it. Modem control is enabled on a serial line by resetting the modem control bit (bit 7) in the minor number for the device. This allows the system to generate a hangup signal when the modem indicates loss of carrier by dropping DCD (Data Carrier Detect). A modem line should always have its DSR, DCD and CTS pins connected. If left hanging, spurious transitions can cause severe system thrashing. To disable modem control on a given serial line, use the minor device which has the modem control bit set (bit 7). An **open** to a modem-control line will block until a carrier is detected (DCD goes true).

"Interrupt mode" means that the port can generate an interrupt to attract the attention of the COHERENT system; "polled mode" means that the port cannot generate an interrupt, but must be checked (or "polled") constantly by the COHERENT system to see if activity has occurred on it.

The COHERENT system uses two device drivers to manage serial lines: one driver manages COM1 and COM3, and the other manages COM2 and COM4. Due to limitations in the design of the ports, you can enable interrupts on either COM1 or COM3 (or on COM2 or COM4), but not both. If you wish to use both ports simultaneously, one must be run in polled mode. For example, if you wish to open all four serial lines, you can open two of the lines in interrupt mode: you can open either COM1 or COM3 in interrupt mode, and you can open either COM2 or COM4 in interrupt mode. The other two lines must be opened in polled mode.

Opening a device in polled mode consumes many CPU cycles, based upon the speed of the highest baud rate requested. For example, on a 20 MHz 80386-based machine, polling at 9600-baud was found to consume about 15% of the CPU time. As only one device can use the interrupt line at any given time, the best approach is to make the high-speed line of the pair interrupt driven and open the low-speed or less-frequently used line in polled mode. However, if you enable a polled line for logins, the port is open and will be polled as long as the port remains open (enabled). Thus, even if a port is not in use, the fact that it has a **getty** on it consumes CPU cycles. As a rule of thumb, try and open a port in interrupt mode. If you cannot, use the polled version. Also note that use of any of the four serial ports in polled mode prevents other polled serial device drivers, such as the **hs** generic multi-port polled serial driver, from being used at the same time.

If you intend to use a modem on your serial port, you must insure that the DCD signal from the modem actually *follows* the state of carrier detect. Some modems allow the user to "strap" or set the DCD signal so that it is always asserted (true). This incorrect setup will cause COHERENT to think that the modem is "connected" to a remote modem, even when there is no such connection.

In addition, if you wish to allow remote logins to your COHERENT system via your modem, you must insure that the modem does **not** echo any commands or status information. Failure to do so will result in severe system thrashing due to the **getty** or **login** processes endlessly "talking" to your modem.

Changing Default Port Speeds

Serial lines **com1** through **com4** default to 9600 baud when opened. This default speed can be permanently changed on a "per port" basis by changing the value of driver variables **C1BAUD_**, **C2BAUD_**, **C3BAUD_** or **C4BAUD_**. The list of acceptible values can be found in header file **<sgtty.h>** and range from 1, corresponding to 50 baud, up to 17, which corresponds to 19,200 baud. For a table of legal baud rates, see the Lexicon entry for **sgtty.h**.

To change the default value for a port, you must use the **/conf/patch** command. For example, to change the default speed for port **com2** to 2400 baud, enter the following command while running as the superuser:

```
/conf/patch /coherent C2BAUD_=12
```

The change will not take effect until the next time that you boot your system.

See Also

com1, com2, com3, com4, device drivers

Diagnostics

An attempt to open a non-existent device will generate error messages. This can occur if hardware is absent or not turned on.

Notes

The **com*** series of devices are not compatible with the **ioctl()** parameters defined in header file **<termio.h>**. Be sure to include header file **<sgtty.h>** if you wish to perform terminal specific **ioctl()** calls.

In the current version of these drivers, the following sequence of steps results in a panic:

```
enable com4pl
enable com3pl
disable com4pl
kill kill <all driver process id>
```

The key is that the driver containing the polling routine cannot be unloaded if the other driver is still polling.

Note, too, that if any **com** device driver is used in polling mode, the **hs** driver cannot be used, and vice versa.

com1 — Device Driver

Device driver for asynchronous serial line COM1

/dev/com1 is the COHERENT system's standard interface to asynchronous serial line COM1. The interface is assigned major device 5, and is accessed as a character-special device. The I/O address for the corresponding 8250 SIO is 0x3F8 (COM1). **com1** generates interrupt IRQ4.

Four versions of device **com1** are in directory **/dev**, as follows:

Device Name	*Major*	*Minor*	*I/O Type*	*Modem Control?*
/dev/com1l	5	128	Interrupts	No
/dev/com1r	5	0	Interrupts	Yes
/dev/com1pl	5	192	Polled	No
/dev/com1pr	5	64	Polled	Yes

For details on how these versions differ, see the entry for **com**.

Files

/dev/com1l — Interrupt-driven, non-modem (local) line
/dev/com1r — Interrupt-driven, modem (non-local) line
/dev/com1pl — Polled, non-modem (local) line
/dev/com1pr — Polled, modem (non-local) line

See Also

com, com3, stty

com2 — Device Driver

Device driver for asynchronous serial line COM2

/dev/com2 is the COHERENT system's standard interface to asynchronous serial line COM2. The interface is assigned major device 6, and is accessed as a character-special device. The I/O address for the corresponding 8250 SIO is 0x2F8 (COM2). **com2** generates interrupt IRQ3.

Four versions of device **com2** are in directory **/dev**, as follows:

				Modem
Device Name	*Major*	*Minor*	*I/O Type*	*Control?*
/dev/com2l	6	128	Interrupts	No
/dev/com2r	6	0	Interrupts	Yes
/dev/com2pl	6	192	Polled	No
/dev/com2pr	6	64	Polled	Yes

For details on how these differ, see the entry for **com**.

Files

/dev/com2l — Interrupt-driven, non-modem (local) line
/dev/com2r — Interrupt-driven, modem (non-local) line
/dev/com2pl — Polled, non-modem (local) line
/dev/com2pr — Polled, modem (non-local) line

See Also

com, com4, stty

com3 — Device Driver

Device driver for asynchronous serial line COM3

/dev/com3 is the COHERENT system's standard interface to asynchronous serial line COM3. The interface is assigned major device 5, and is accessed as a character-special device. The I/O address for the corresponding 8250 SIO is 0x3E8 (COM3). **com3** generates interrupt IRQ4.

Four versions of device **com3** are in directory **/dev**, as follows:

				Modem
Device Name	*Major*	*Minor*	*I/O Type*	*Control?*
/dev/com3l	5	129	Interrupts	No
/dev/com3r	5	1	Interrupts	Yes
/dev/com3pl	5	193	Polled	No
/dev/com3pr	5	65	Polled	Yes

For details on how these differ, see the entry for **com**.

Files

/dev/com3l — Interrupt-driven, non-modem (local) line
/dev/com3r — Interrupt-driven, modem (non-local) line
/dev/com3pl — Polled, non-modem (local) line
/dev/com3pr — Polled, modem (non-local) line

See Also

com, com1, stty

com4 — Device Driver

Device driver for asynchronous serial line COM4

/dev/com4 is the COHERENT system's standard interface to asynchronous serial line COM4. The interface is assigned major device 6, and is accessed as a character-special device. The I/O address for the corresponding 8250 SIO is 0x2E8 (COM4). **com4** generates interrupt IRQ3.

Four versions of device **com4** are in directory **/dev**, as follows:

Device Name	*Major*	*Minor*	*I/O Type*	*Modem Control?*
/dev/com4l	6	129	Interrupts	No
/dev/com4r	6	1	Interrupts	Yes
/dev/com4pl	6	193	Polled	No
/dev/com4pr	6	65	Polled	Yes

For details on how these differ, see the entry for **com**.

Files

/dev/com4l — Interrupt-driven, non-modem (local) line
/dev/com4r — Interrupt-driven, modem (non-local) line
/dev/com4pl — Polled, non-modem (local) line
/dev/com4pr — Polled, modem (non-local) line

See Also

com, com2, stty

con.h — Header File

Configure device drivers
#include <sys/con.h>

The header file **con.h** gives the configuration for each device driver included with the COHERENT system. Each driver is defined using the structure **CON**, which is declared in **<sys/con.h>**.

See Also

header files, sload()

config — Command

Build a new COHERENT kernel
/usr/sys/config
/usr/sys/config [stand={fha0,fva0}] [standard] [root=*DEV***] [swap=***DEV***]** [*DRV* **...]**

The command **config** builds a new COHERENT kernel.

Invoking this command with the argument **help** prints a usage message on the screen. Otherwise, the command describes the type of kernel to build.

The argument **standard** tells **config** to build the "standard" COHERENT AT kernel. The standard kernel uses **/dev/at0a** as its root device.

The argument **stand** allows you to reset the standard configuration of the kernel. **stand=fha0** builds a kernel that runs off of a 5.25-inch, high-density floppy disk in drive 0 (otherwise known as drive A). **stand=fva0** builds a kernel that runs off of a 3.5-inch, high-density floppy disk in drive 0. Each floppy-disk edition of COHERENT includes a large-enough file system and enough system commands to allow you to do real work.

The **root** option lets you reset the root device and pipe device to *DEV*. The **swap** option lets you set the swap device to *DEV*. Obviously, the swap device and the root device must be different devices. Note that unlike other systems, COHERENT does not require the use of a swapper in order to run. Some releases of COHERENT do not include support for swapping.

Each *DRV* argument names a device driver to include with the kernel. Each driver must exist in the form of an archive of relocatable object modules in directory **/usr/sys/lib**.

The shell script **/usr/sys/Build** invokes this command and otherwise manages the complexity of recreating a COHERENT kernel. You are well advised to modify this script to build your kernel rather than attempt to run **config** from the command line. For directions on how to do so, see section 3 of the manual for the COHERENT device driver kit.

See Also

Build, device drivers, ldconfig

dblock() — Driver-Access Routine

Call device block entrypoint

#include <sys/buf.h>
void
dblock(*dev*, *bp***)**
dev_t *dev***;**
BUF **bp***;**

dblock calls the function pointed to by field **c_block** in the device driver's **CON** structure. *dev* indicates the device. *bp* points to the buffer's **BUF** structure.

See Also

driver-access routines

dclose() — Driver-Access Routine

Device close

#include <sys/types.h>
void
dclose(*dev***)**
dev_t *dev***;**

dclose calls the function pointed to by field **c_close** in the device driver's **CON** structure. This function closes the device. *dev* indicates the device to be closed.

dclose should never be called from an interrupt or a deferred routine.

See Also

driver-access routines

defend() — Accessible Kernel Routine

Execute deferred functions
void
defend()

defend tells the kernel to execute all functions that are on its deferred list. This function should **never** be invoked by an interrupt handler.

See Also

accessible kernel routines

defer() — Accessible Kernel Routine

Defer function execution
void
defer(*func*, *arg***)**
void (**func***)();**
char ****arg*;**

defer defers execution of function *func* with argument *arg*. Execution of *func* remains deferred until the next context switch, transition from kernel to user mode, or invocation of the function **defend**.

Deferred functions should 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 is normally used to minimize interrupt latency by deferring operations from interrupt level, where lower priority interrupts are disabled, to background level, where all interrupts are normally enabled. It is also useful in eliminating 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

accessible kernel routines

device drivers — Overview

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 this edition of 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.

0:	***mem**	Interface to memory
1:	**tty**	Primitive tty driver
2:	**nkb/kb/mm**	Keyboard and video
3:	**lp**	Parallel line printer
4:	**fl**	Floppy drive
5:	**al0**	Serial line 0 (COM1 and COM3)
5:	**rs0**	Raw serial 0 (COM1)
5:	**sl**	Primitive serial line sl0 (COM1), sl1 (COM2)
6:	**rs1**	Raw serial 1 (COM2)
6:	**al1**	Serial line 1 (COM2 and COM4)
7:	**hs**	Generic polled multi-port serial card
8:	**rm**	Dual RAM disk

9:		
10:	**ms**	Microsoft Mouse
11:	**at**	AT hard disk
11:	**hd**	Primitive sample XT disk driver
12:	**st**	Archive Streaming Tape
13:	**scsi**	SCSI device drivers: **aha154x, ss**
14:		
15:		
16:		
17:		
18:		
19:		
20:	**tn**	Tiac PC-234/6 ARCNET LAN driver
21:	**pe**	Intelligent multiport serial board
22:		
23:	**sem**	System V compatible semaphores
24:	**shm**	System V subset shared memory
25:	**msg**	System V compatible messaging
26:		
27:		
28:		
29:		
30:	**gr**	IBM Color card (640x200) graphics display
31:		

Also included are drivers for the following devices:

console	Console driver
ct	Controlling terminal driver
null	The "bit bucket"

Please note that these device drivers are distributed with the COHERENT system in binary form only. For proprietary reasons, source code for some drivers cannot be included with the COHERENT Device Driver Kit.

The commands **Build, config, ldconfig** are used to recreate device drivers; **Build** and **config** link the drivers into a new version of the kernel, whereas **ldconfig** creates a loadable device driver. See their respective entries in this manual for more information.

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*). Each device driver uses one or more unique major numbers, and the individual devices serviced by that driver are identified by a minor number. There are, however, a number of exceptions to this scheme:

1. 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.

2. In COHERENT, devices using different IRQ's may have different major numbers, even if the devices are of the same general type. For example, devices **com1*** and **com3*** have major number 5, while **com2*** and **com4*** have major number 6.

See Also

accessible kernel routines, block-device routines, driver-access routines, header files, interrupt-handler routines, I/O routines, kernel variables, memory-manipulation routines, race condition, segment-manipulation routines, swap, terminal-device routines

devices.h — Header File

Define major numbers for device drivers
#include <sys/devices.h>

The header file **<sys/devices.h>** defines the major number for each COHERENT device driver.

See Also

header files

devmsg() — Driver-Access 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 kernel function **printf**, except that a newline is appended to *fmt*. Output from **devmsg** is synchronous and at high priority, so its use should be limited to brief error messages.

See Also

driver-access routines, printf()

dioctl() — Driver-Access Routine

Call a device-driver's I/O control point
void
dioctl(*dev*, *com*, *vec***)**
dev_t *dev*;
int *com*;
union ioctl **vec*;

dioctl calls the ioctl entrypoint for a device driver. *dev* is the device number for the device; *com* is the command to be executed; and *vec* is its argument vector (i.e., address).

See Also

driver-access routines

dmac.h — Header File

DMA definitions
#include <sys/dmac.h>

The header file **dmac.h** holds manifest constants that are used by routines that perform direct-memory access (DMA).

See Also

device drivers, header files

dmago() — Accessible 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

accessible kernel routines

dmaoff() — Accessible 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

accessible kernel routines

dmaon() — Accessible Kernel Routine

Prepare for DMA transfer
#include <sys/types.h>
int
dmaon(*chan*, *paddr*, *count*, *wflag*)
int *chan*;
paddr_t *paddr*;
unsigned *count*;
int *wflag*;

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 the operation is successfully programmed, **dmaon** returns one. A DMA straddle arises when an operation would cross a 64-kilobyte physical memory boundary. As the DMA controller cannot handle a straddle condition, the operation is not programmed and **dmaon** returns zero.

See Also

accessible kernel routines

dmareq() — Accessible 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

accessible kernel routines, ioreq

dopen() — Driver-Access Routines

Device open
void
dopen(*dev*, *mode*, *flags*)
dev_t *dev*;

dopen calls the function pointed to by field **c_open** in the driver's **CON** structure. This function opens the device.

dev is the device being opened. *mode* gives the mode in which it is being opened; valid *modes* include **IPR**(read), **IPW**(write), or **IPR | IPW**. Valid *flags* are **DFBLK** or **DFCHR**. If the open fails, **u.u_error** is set.

See Also

driver-access routines

dpoll() — Driver-Access Routine

Device poll
int
dpoll(*dev*, *ev*, *msec*)
dev_t *dev*;
int *ev*;
int *msec*;

dpoll calls the function pointed to by field **c_poll** in the driver's **CON** structure. This function polls the device. *dev* is the device to be polled.

If the driver does not support polling, **dpoll** returns **POLLNVAL**.

See Also

driver-access routines

dpower() — Driver-Access Routine

Device power-fail
void
dpower(*dev*)
dev_t *dev*;

dpower calls the function pointed to by field **c_power** in the device's **CON** structure. This function can be executed should the power fail. *dev* indicates the device in question.

See Also

driver-access routines

dread() — Driver-Access Routine

Device read
#include <sys/types.h>
void
dread(*dev*, *iop***)**
dev_t *dev***;**
IO **iop***;**

dread calls the function pointed to by field **c_read** in the device driver's **CON** structure. This function reads from the device. *dev* indicates the device to be read. *iop* points to the **IO** structure.

See Also

driver-access routines

driver-access routines — Overview

The following kernel routines access the functions that are pointed to by the fields in a driver's configuration table:

dblock	Call device block entry point
dclose	Device close
dioctl	Call a device-driver's ioctl entry point
dopen	Device open
dpoll	Device poll
dpower	Device power-fail
dread	Device read
dtime	Device timeout
dwrite	Device write

The following routines are also used to access a device or retrieve information about it:

devmsg	Print a message from a device driver
fdisk	Hard-disk partitioning
major	Extract major device number
minor	Extract minor device number
nonedev	Illegal device request
nulldev	Ignored device request

See Also

device drivers

drvld — Command

Load a loadable driver into memory
/etc/drvld *options driver*

drvld loads a loadable driver into memory. *driver* names a loadable driver. Only the superuser **root** can run **drvld.**

A loadable driver is one that is not linked into the kernel when it was built. The current suite of loadable drivers include multi-port serial cards, various SCSI host adaptors, and a variety of add-on cards. The COHERENT drivers for shared memory, semaphores, and message passing are also implemented as loadable drivers, due to the efficient size of the COHERENT kernel.

drvld recognizes the following *options*:

-k *kernel*
By default, **drvld** assumes that file **/coherent** holds the symbol table for the in-core copy of COHERENT. The **-k** option tells **drvld** to load the driver using a version of COHERENT other than the default. You must use this option if you are running an alternate copy of COHERENT (e.g., a version based on the floppy disk drive).

-r Supress generation of a debugging symbol table.

-o *outfile*
By default, **drvld** writes the driver's debugging symbol table into a file that has the same name as the driver but is located in directory **/tmp**. The **-o** options tells **drvld** to output the symbol table to *outfile* rather than the default.

Files

/drv — directory containing loadable drivers

See Also

commands, device drivers, sload()

Notes

COHERENT supports user-written, loadable device drivers generated with the COHERENT device-driver kit. Loadable device drivers produced by **ldconfig** reside in **/usr/sys/ldrv**. By convention, loadable drivers that have been tested thoroughly and released for production reside in directory **/drv**, not in **/dev**.

dtime() — Driver-Access Routine

Device timeout
void
dtime(*dev***)**
dev_t *dev***;**

dtime calls the function pointed to by field **c_time** in the device driver's **CON** structure. This function is executed if a device driver has requested periodic timer service. *dev* indicates the device in question.

See Also

driver-access routines

dwrite() — Driver-Access Routine

Device write
void
dwrite(*dev, iop***)**
dev_t *dev***;**
IO **iop***;**

dwrite calls the function pointed to by field **c_write** in the device driver's **CON** structure. This function writes to a device. *dev* indicates the device in question; *iop* points to the **IO** structure.

See Also

driver-access routines

fclear() — Memory-Manipulation Routine

Clear far memory
#include <sys/types.h>
void
fclear(*fp*, *n*)
faddr_t *fp*;
unsigned *n*;

fclear clears *n* bytes of memory at far address *fp*.

See Also

memory-manipulation routines

fdisk() — Driver-Access Routine

Hard-disk partitioning
int
fdisk(*dev*, *fp*)
dev_t *dev*;
struct fdisk_s *fp*[4];

fdisk attempt to read partitioning information from block 0 of the hard disk *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

driver-access routines

ffbyte() — Memory-Manipulation Routine

Fetch a far byte
#include <sys/types.h>
int
ffbyte(*fp*)
faddr_t *fp*;

ffbyte reads a byte from far address *fp*. Note that if an address fault occurs, the system will panic.

See Also

memory-manipulation routines

ffword() — Memory-Manipulation Routine

Fetch a far word
#include <sys/types.h>
int
ffword(*fp*)
faddr_t *fp*;

ffword reads a word from far address *fp*. Note that if an address fault occurs, the system will panic.

See Also

memory-manipulation routines

fkcopy() — Memory-Manipulation Routine

Copy from far address to kernel
#include <sys/types.h>
unsigned
fkcopy(*fp***,** *k***,** *n***)**
faddr_t *fp***;**
char ***k*;**
unsigned *n***;**

fkcopy copies *n* bytes from far address *fp* to address *k* in the kernel data segment. It returns the number of bytes copied.

See Also

memory-manipulation routines

fun.h — Header File

Miscellaneous definitions
#include <sys/fun.h>

The header file **fun.h** holds miscellaneous definitions that may be useful to writers of device drivers.

See Also

device drivers, header files

getq() — Terminal-Device Routine

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

getq returns the next character from character queue *cqp*. It returns -1 if the queue is empty.

See Also

terminal-device routines

getubd() — Memory-Manipulation 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** sets **u.u_error** to **EFAULT**.

See Also

memory-manipulation routines

getuwd() — Memory-Manipulation 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** sets **u.u_error** to **EFAULT**.

See Also

memory-manipulation routines

getuwi() — Memory-Manipulation 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 sets **u.u_error** to **EFAULT**.

See Also

memory-manipulation routines

gr — Device Driver

Graphics Driver

/dev/gr is a low-level graphics interface that lets you use graphics on the IBM PC color card. It is assigned major device 30, and is accessed as a character-special device. The supported resolution is 640 pixels across (80 bytes) by 200 pixels high; thus, a bit-map of the entire screen takes 16,000 bytes.

Graphics memory can be manipulated by read and write calls to **/dev/gr**. The lseek() library call should be used to specify the byte at which the read or write is to start. To read the entire screen, use the following sample code:

```
#define NLINES 200
#define BYTESPERLINE 80
int  fd;
char image[NLINES][BYTESPERLINE];

fd = open( "/dev/gr", 2 );
lseek( fd, 0L, 0 );
read( fd, image, sizeof image );
```

The following code fragment reads, inverts all bits, then writes the bottom half of the screen:

```
int  fd, row, col;
char image[NLINES/2][BYTESPERLINE];

fd = open( "/dev/gr", 2 );
lseek( fd, (long)(NLINES/2) * (long)BYTESPERLINE, 0 );
read( fd, image, sizeof image );

for ( row=0; row < NLINES/2; row++ )
     for ( col=0; col < BYTESPERLINE; col++ )
          image[row][col] ^= 0xFF;
lseek( fd, (long)(NLINES/2) * (long)BYTESPERLINE, 0 );
write( fd, image, sizeof image );
```

Characters written to **/dev/console** are painted onto the graphics screen. The cursor is also painted onto the screen. Subsequent reads through **/dev/gr** includes the painted characters and cursor. Subsequent writes to **/dev/gr** can erase the painted characters or make the cursor invisible.

Files

/dev/gr — Character-special file

See Also

device drivers

Notes

This interface does not support color.

header files — Overview

The following header files are included in the COHERENT system's device-driver kit:

clist.h	Character-list structures
coherent.h	Miscellaneous useful definitions
con.h	Configure device drivers
devices.h	Device major numbers
dmac.h	DMA definitions
fun.h	Miscellaneous definitions
i8086.h	Machine-dependent information
ins8250.h	Definitions used with i8250 chip
ktty.h	Kernel portion of tty structure
mmu.h	Definitions for memory-management unit
ms.h	Header for Microsoft Mouse driver
ptrace.h	Process trace
systab.h	System-call table

See their respective entries in this manual for more information.

See Also

device drivers

hs — Device Driver

Device driver for polled serial ports

The COHERENT **hs** driver adds support for up to eight serial lines, **/dev/hs00** through **/dev/hs07**.

Serial lines controlled via the **hs** driver can be opened in one of two ways, as follows:

/dev/hs??
Polled, local mode (no modem control).

/dev/hs??r
Polled, remote mode (modem control).

Any port used with the **hs** device driver will be polled, i.e., interrupt operation is not used. Please refer to the Lexicon article **com** for explanations of "local" vs "remote" and "polled" vs "interrupt-driven".

To use the **hs** driver, first configure it to match your equipment (see below), then load the driver using the following command while running as the superuser **root**:

```
/etc/drvld -r /drv/hs
```

To unload the driver without rebooting COHERENT, first use the **ps** command with the **-d** option to get the process identifier for the **hs** driver process, then unload the driver process by using the **kill** command. Note that the **hs** driver process will not unload until all **open**ed ports have been closed. For example (user input shown in bold):

```
$ ps -d
TTY         PID
-------       0  <idle>
-------      38  <hs>
...
$ kill kill 38
```

The present version of COHERENT limits "polled" operation to one device driver at a time. Therefore, if any of the **com** family of devices is used in polled mode, **hs** devices cannot be used. Conversely, **/dev/com1pl** through **/dev/com4pl** and **/dev/com1pr** through **/dev/com4pr** cannot be used if the **hs** driver is in use. Both drivers can be present at the same time, but polled devices may not be **open** under both drivers at the same time. Note that enabling a port via **/etc/enable** keeps it open continuously.

Port Configuration

The default configuration for the **hs** driver is for four ports, at hexadecimal addresses 0x3F8, 0x2F8, 0x3E8, and 0x2E8, at a speed of 9600 baud. The driver is configured by setting the following parameters:

1. The number of ports.
2. The I/O address for each port.
3. The default speed of each port.

All steps in the configuration must be done as the superuser **root**. Patch the number of ports into driver variable **HSNUM_**. For example, if you wish to support three ports, enter:

```
/conf/patch /drv/hs HSNUM_=3
```

Address and speed information are stored sequentially starting at variable **HS_PORTS_**. The speed for each port is indicated by the corresponding value found in **<sgtty.h>**, from one, corresponding to 50 baud, to 16, corresponding to 9600 baud. If the three ports in the example above are at hexadecimal adresses of 0x2A0, 0x2B0, and 0x2C0, with speeds of 2400, 2400, and 9600 baud, respectively, then the following three patches must be performed:

```
/conf/patch /drv/hs HS_PORTS_=0x2A0 HS_PORTS_+2=12
/conf/patch /drv/hs HS_PORTS_+4=0x2B0 HS_PORTS_+6=12
/conf/patch /drv/hs HS_PORTS_+8=0x2C0 HS_PORTS_+10=16
```

Finally, nodes must be created for each port using the **mknod** command. The major device number is 7; the minor number will range from 0 through 7 for ports **/dev/hs00** through **/dev/hs07**, respectively, with 128 added to the device minor number if modem control is desired. The following commands will make nodes in **/dev** for local and remote versions of the three ports in the example:

```
/etc/mknod -f /dev/hs00  c  7  0
/etc/mknod -f /dev/hs01  c  7  1
/etc/mknod -f /dev/hs02  c  7  2
/etc/mknod -f /dev/hs00r c  7  128
/etc/mknod -f /dev/hs01r c  7  129
/etc/mknod -f /dev/hs02r c  7  130
```

See Also

com, device drivers, drvld

Diagnostics

An attempt to open a non-existent device will generate error messages. This can occur if hardware is absent or not turned on.

Notes

Note that if any **com** device driver is used in polling mode, the **hs** driver cannot be used, and vice versa.

i8086.h — Header File

Machine-dependent information
#include <sys/i8086.h>

The header file **i8086.h** holds manifest constants and definitions that are useful with device drivers run on computers built around the Intel 8086 family of microprocessors. The definitions include manifest constants for magic locations in memory, trap codes, saved registers, and various memory segments.

See Also

device drivers, header files

inb() — Accessible Kernel Routine

Read a byte from an I/O port
int
inb(*port***)**
unsigned *port***;**

inb reads a byte from *port*.

See Also

accessible kernel routines

ins8250.h — Header File

Definitions used with i8250 chip
#include <sys/ins8250.h>

The header file **ins8250.h** holds definitions that are useful to device drivers that manipulate the Intel 8250 chip. The definitions include manifest constants to describe the states of the interrupt-enable register, the line-control register, the modem-control register, the line-status register, and the modem-status register.

See Also

device drivers, header files

interrupt-handler routines — Overview

The following routines can be used by device drivers to handle interrupts:

clrivec	Clear interrupt vector
setivec	Set an interrupt vector
sphi	Disable interrupts
spl	Adjust interrupt mask
splo	Enable interrupts

See Also

device drivers

I/O routines — Overview

The following functions can be used by device drivers to perform input/output (I/O):

devmsg	Write major/minor device numbers and message to console
iogetc	Get a character from I/O segment
ioputc	Put a character into I/O segment
ioread	Read from I/O segment
ioreq	Request I/O through block routine
iowrite	Write to I/O segment
printf	Write message directly to console

See Also

device drivers

iogetc() — I/O 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, **iogetc** sets **u.u_error** to **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

I/O routines

ioputc() — I/O Routine

Put a character into I/O segment

```
int
#include <sys/io.h>
ioputc(c, iop)
char c;
IO *iop;
```

ioputc write character *c* into the I/O segment referenced by *iop*. If an address fault occurs, **ioputc** sets **u.u_error** to **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

I/O routines

ioread() — I/O Routine

Read from I/O segment
void
#include <sys/io.h>
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 sets **u.u_error** to **EFAULT**.

See Also

I/O routines

ioreq() — I/O Routine

Re-queue I/O request through block routine
void
#include <sys/io.h>
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** is normally called from the read/write routines of a block device that does not support DMA.

See Also

dmareq, I/O routines

iowrite() — I/O Routine

Write to I/O segment
void
#include <sys/io.h>
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** sets **u.u_error** to **EFAULT**.

See Also

I/O routines

kalloc() — Memory-Manipulation 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 will contain garbage. Use **kclear()** if needed. Space allocated with **kalloc()** must be deallocated with **kfree()**.

See Also

kfree(), memory-manipulation routines

kclear() — Memory-Manipulation Routine

Clear kernel memory
void
kclear(*k*, *n*)
char *k;
unsigned *n*;

kclear clears *n* bytes in the kernel's data segment, starting at offset *k*.

See Also

memory-manipulation routines

kernel variables — Technical Information

Variables set within COHERENT kernel

The following describes variables set within the COHERENT kernel. Each variable is described, and its default setting given. The clock rate is defined as the manifest constant **HZ** (hertz), which is set in header file **sys/const.h**. Normally, this value is set to 100, which translates into 100 ticks per second, or approximately 10 milliseconds per tick.

By using the debugger **db** 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 = 16*1024;
```

ALLSIZE gives the number of bytes in the kernel's memory allocation pool. This pool is manipulated by the functions **kalloc** and **kfree**.

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.

Note that the stack size of individual programs can be changed by using the command

fixstack.

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.**

KRUNCH — Time in ticks between krunch attempts

```
int KRUNCH = 200;
```

KRUNCH specifies the number of clock ticks between attempts to coalesce (or "krunch") free memory to reduce memory fragmentation. It only operates if swapping is disabled and the **KRUNCH** varable is non-zero.

NBUF — Number of blocks in buffer cache

```
int NBUF = 32;
```

NBUF specifies the number of blocks in the buffer cache.

NCLIST — Number of clists

```
int NCLIST = 24;
```

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 = 64;
```

NINODE specifies the maximum number of i-nodes that can be opened simultaneously.

NMSC — Number of characters per message

```
int NMSC = 640;
```

NMSC gives the maximum number of characters per message. This variable is **kalloc**'d.

NMSG — Number of message buffers

```
int NMSG = 10;
```

NMSG gives the number of message buffers allocated. This variable is **kalloc**'d. You should increase variable **ALLSIZE** by 16 bytes per message buffer.

NMSQB — Maximum characters per message queue

```
int NMSQB = 2048;
```

NMSQB gives the default maximum number of bytes of messages on any one message queue. This variable is **kalloc**'d. You should increase variable **ALLSIZE** by 64 bytes per message queue.

NMSQID — Maximum number of message queues

```
int NMSQID = 9;
```

NMSQID specifies the maximum number of message queues in the system. This variable is **kalloc**'d. You should increase variable **ALLSIZE** by 64 bytes per message queue.

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. This variable is **kalloc**'d. You increase variable **ALLSIZE** by eight bytes per pending poll.

NSLOT — Number of loadable driver data slots

```
int NSLOT = 64;
```

NSLOT specifies the number of 64-kilobyte slots available to data associated with loadable drivers.

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.

cs:cds — Kernel's core copy of kernel data selector core copy of kernel data selector'>=29

```
saddr_t cds;
```

cds is a variable that resides in kernel code space. It contains a selector through which a function can access the kernel's data space. This variable is accessible only by assembly-language subroutines.

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 devices **console** and **lp** 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.

depth — Interrupt depth

```
char depth;
```

depth specifies the user/kernel depth. A setting of one indicates user mode; zero indicates a system call or an interrupt from user mode; and a negative value indicates a nested interrupt or an interrupt from system mode. System calls are illegal unless **depth** is set to one. The **defend** routine should be called only when **depth** is set to zero.

drvl — Device driver list

```
#include <sys/con.h>
#include <sys/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.

gdtsel — Global descriptor table selector

```
saddr_t gdtsel;
```

gdtsel is a virtual selector that references the global descriptor table. For further information, see the manual for the Intel iAPX-286.

idtsel — Interrupt descriptor table selector

```
saddr_t idtsel;
```

idtsel is a virtual selector referencing the interrupt descriptor table, or zero in real mode. For further information, see the manual for the Intel iAPX-286.

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).

realmode — Indicate mode of CPU

```
int realmode = 0;
```

realmode is set to a non-zero value if the CPU is operating in real mode. It is zero if the CPU is operating in protected mode.

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.

sds — Kernel data selector

```
saddr_t    sds;
```

sds contains a selector through which kernel data space can be accessed.

swapbot — Bottom of swap memory

```
daddr_t swapbot = 0;
```

swapbot gives the first block in the swap region. A partition can be shared by a file system and a swap region by using the first part of the partition for the file system, and setting **swapbot** and **swaptop** accordingly.

swapdev — Swap device

```
dev_t    swapdev = makedev(0,0);
```

swapdev gives the device to be used for swapping. It is zero if swapping is disabled.

swaptop — Top of swap memory

```
daddr_t swaptop = 0;
```

swaptop specifies the block just past the end of the swap region. A partition can be shared by a file system and a swap region by using the first part of the partition for the file system, and setting **swapbot** and **swaptop** accordingly.

uasa — User area selector

```
saddr_t uasa;
```

uasa specifies the selector for the user area segment of the currently executing process. The **u** structure and the kernel stack are transferred to the user area segment during a context switch.

ucl — User code limit

```
char *  ucl;
```

ucl specifies the offset of the last character within the code segment of the currently executing process.

ucs — User code selector

```
saddr_t ucs;
```

ucs specifies the selector of the code segment of the currently executing process.

udl — User data limit

```
char *  udl;
```

udl specifies the offset of the last character within the data segment of the currently executing process.

uds — User data segment

```
saddr_t uds;
```

uds specifies the selector of the data segment of the currently executing process.

See Also

device drivers

keyboard tables — Technical Information

How to write a keyboard table

The COHERENT device-driver **nkb** supports industry-standard 83-, 101-, and 102-key AT-protocol keyboards attached as the computer console.

nkb lets you define both the layout of the keyboard and the values returned by function keys. You can change layout and function-key bindings by using the special keyboard mapping programs kept in directory **/conf/kbd**. This directory contains the C source code for the mapping tables, as well as a **Makefile** that helps you rebuild the mapping programs.

Before you begin to write or modify an existing keyboard table, be sure to read throroughly this article and the Lexicon article on **nkb**. If you do not, you may foul up the keyboard so thoroughly that it will not work well enough for you to undo your mistake!

Operational Overview

The device driver **nkb** provides the system's portion of the interface to the console keyboard. It handles hardware-specific details, such as initializing the keyboard and internal state, handling keyboard interrupts, processing key scan codes, and queueing characters.

The user half of the keyboard interface is provided by a set of stand-alone utilities. With these, you can program the **nkb** driver via specialized **ioctl()** calls. These utilities differ from each other only in the keyboard binding or mapping tables each uses. You can re-construct the interface to the **nkb** driver by modifying a keyboard-mapping file and then using a support module to link that file to the driver.

The keyboard-mapping file is a C program that consists of initialized tables and strings. In addition, several header files provide the scan codes and other constants required for the key tables. This format makes the file easy to edit, and also lets you enter characters in several different formats.

The support module, in turn, performs several tasks. These include scanning the keyboard-mapping file for errors, reformatting the table for use by the device driver, and passing the reformatted table to the driver.

Key Mapping Files

By convention, directory **/conf/kbd** contains the keyboard-mapping files, executables, and a **Makefile** that you use to construct the executables from the corresponding source files.

A keyboard-mapping source file consists primarily of three data structures that you must modify to support a given keyboard mapping. The first, and simplest, of the structures is **tbl_name**. This is a character string that describes the keyboard. For example, the stock 101-key US AT keyboard mapping file **/conf/kbd/us.c** initializes this string to:

```
"U.S. AT keyboard table"
```

The second data structure, **kbtbl**, is an array of key-mapping entries. It has one entry (or row) for each possible key location. Each entry in this structure consists of 11 fields, which hold, respectively, the key number, nine possible mapping values, and a mode field. The following example is for physical key location 3 from key-mapping source file **/conf/kbd/belgian.c**:

```
{ K_3, 0x82, '2', none, none, 0x82, '2', '-', none, '-', O|T },
```

Field 1 contains the *scan code set 3* code value for the desired key. Header file **<sys/kbscan.h>** contains symbolic constants of the form **K_*nnn*** that map the AT keyboard's *physical* key number *nnn* to the corresponding scan code set 3 value generated by the keyboard. In the above example, **K_3** corresponds to key location three.

Fields 2 through 10 contain the key mappings corresponding to the following shift states, as follows:

2	base or unshifted
3	**SHIFT**
4	**CONTROL**
5	**CONTROL+SHIFT**
6	**ALT**
7	**ALT+SHIFT**
8	**ALT+CONTROL**
9	**ALT+CONTROL+SHIFT**
10	**ALT_GRAPHIC**

For "regular" keys, the values for these nine fields are eight-bit characters; for "function" or "shift" keys, they are special values. The symbolic constant **none** indicates that you want no output when the key is pressed in the specified shift state.

In the case of a function key, the value specified is the number of the desired function key. Header file **<sys/kb.h>** defines a set of symbolic constants of the form **f***n*, where *n* is the desired function key number. You should use these constants; they will improve the readability of your code, and they will protect your keyboard mapping source files from any future changes in the structure of the keyboard driver.

In the case of a "shift" key, all nine entries must be identical and must consist of one of the following symbolic constants: **scroll**, **num**, **caps**, **lalt**, **ralt**, **lshift**, **rshift**, **lctrl**, **rctrl**, or **altgr**. These are defined in the **<sys/kb.h>** header file. Note that 83-key XT-layout keyboards only have one "control" and "alt" key, so not all shift-key combinations may be possible on your target keyboard.

The last (11th) field in the key entry is the "mode" field. The following symbolic constants specify the mode of the current key:

C The **caps** lock key affects this key.

F The specified key is a "function" or special key. The value of all mapping entries must name function keys. See header file **<kb.h>** for a list of predefined function keys.

M Make: use this mode with keys that do not repeat. Note that accidentally using this mode with "shift" keys will stop you from being able to "unshift" upon releasing the key!

MB Make/Break: use this mode with "shift" keys.

N The **num** lock key affects this key.

O The specified key is "regular" and requires no special processing.

S The specified key is a "shift" or "lock" key. Note that all mapping entries for a given key must be identical for a "shift" or "lock" key to work correctly.

T Typematic: this type is usually associated with a "regular" key.

TMB Typematic/Make/Break.

The above example specifies a mode field of **O|T**, which corresponds to a "regular" key with Typematic repeat, and no special handling of the "lock" keys.

The last data structure, **funkey**, consists of an array of function-key initializers, one per function key. The initializers are simple quoted character strings delimited by either hexadecimal value **0xFF**, octal value **\377**, or symbolic constant **DELIM**. Note that any other value can be used as part of a function-key binding. Function keys are numbered starting at zero. By convention, function key 0, when enabled, reboots your computer. For traditional reasons, this function key is usually bound to the key sequence **<ctrl><alt><del>**.

Function keys are useful not only in the classical sense of the programmable function keys on the keyboard, but also as a general purpose mechanism for binding arbitrary length character sequences to a given key. For example, physical key location sixteen is usually associated with the **<tab>** and **<back tab>** on the AT keyboard. For example, **/conf/kbd/us.c** sets the key mapping table entry for key 16 as follows:

```
{ K_16, f42, f43, none, none, f42, f43, none, none, none, F|T },
```

For traditional reasons, the **<back tab>** key outputs the sequence **<esc>[Z** whereas the **<tab>** key simply outputs the horizontal-tab character **<ctrl-I>**. Because at least one of the mapping values for this key is more than one character long, the key must be defined as a "function" key and all entries for the the key must correspond to function-key numbers. In this example, function key number 42 was chosen for **<tab>**, and function key number 43 was chosen for **<back tab>**. The constant **none** indicates that you want no output when the key is pressed in the specified shift state. The corresponding **funkey** initialization entries for function keys **f42** and **f43** are as follows:

```
/* 42 */        "\t\377",          /* Tab */
/* 43 */        "\033[Z\377",      /* Back Tab */
```

We strongly recommend that you comment your function-key bindings.

You can also change function-key bindings via the command **fnkey**. This command lets you temporarily alter one or more function-key mappings without changing your key-mapping sources.

Building New Binaries

After you have modified an existing keyboard-mapping table, use the following commands to rebuild the corresponding executables:

```
cd /conf/kbd
su root
make
```

If you have created a new keyboard mapping table, you must edit **/conf/kbd/Makefile**. Duplicate an existing entry from the **Makefile**, and change the duplicated name to match the name of your new keyboard-mapping table. After you have finished your editing, build an executable from your source file by simply executing the above series of commands.

To load your new keyboard table, simply type the name of the executable that corresponds to your keyboard-mapping file. For example, if you just built executable **french** from source file **french.c**, type the following command:

```
/conf/kbd/french
```

If the keyboard-support module finds an error, it will print an appropriate message. If it finds no errors, it will update the internal tables of the **nkb** keyboard driver, reprogram the keyboard, and print a message of the form:

```
Loaded French AT keyboard table
```

Examples

Prior to the release of the 101- and 102-key, enhanced-layout AT keyboards, the **<ctrl>** key was positioned to the left of 'A' key. Most terminals also locate the **<ctrl>** key there. The first example shows how to swap the left **<ctrl>** key and the **<caps-lock>** key on a 101- and 102-key keyboard. The **<caps-lock>** key is physical key 30, whereas the left **<ctrl>** key is physical key 58. Their respective entries in file **/conf/kbd/us.c** source file are as follows:

```
{ K_30, caps, caps, caps, caps, caps, caps, caps, caps, caps,  S|M },
{ K_58, lctrl,lctrl,lctrl,lctrl,lctrl,lctrl,lctrl,lctrl,lctrl, S|MB },
```

Note that the **<caps-lock>** key is defined with mode **M** as it is a "lock" key. The keyboard will interrupt only on key depressions, because releasing a "lock" key has no effect. The left **<ctrl>** key is defined with mode **MB** as it is a "shift" key. The keyboard generates an interrupt on both key depression and key release, because the driver must track the state of this key.

To swap the aforementioned keys, simply change all occurrences of **caps** to **lctrl** and vice-versa, as well as swapping the mode fields. After making the changes, the entries now appear as:

```
{ K_30, lctrl,lctrl,lctrl,lctrl,lctrl,lctrl,lctrl,lctrl,lctrl, S|MB },
{ K_58, caps, caps, caps, caps, caps, caps, caps, caps, caps,  S|M  },
```

The second example converts a 101- or 102-key keyboard table to support an XT-style 83-key keyboard layout. The following section summarizes the "typical" differences found when comparing the two keyboard layouts. Needless to say, given the extreme variety in keyboard designs, your mileage may vary.

Physical Location	*101/102 Value*	*83-key Value*	*Comments*
14	**none**	*various*	Keyboard specific
30	**caps**	**lctrl**	
58	**lctrl**	**lalt**	
64	**rctrl**	**caps**	
65	**none**	**f2**	Function Key
66	**none**	**f4**	Function Key
67	**none**	**f6**	Function Key
68	**none**	**f8**	Function Key
69	**none**	**f10**	Function Key
70	**none**	**f1**	Function Key
71	**none**	**f3**	Function Key
72	**none**	**f5**	Function Key
73	**none**	**f7**	Function Key
74	**none**	**f9**	Function Key
90	**num**	**esc**	
95	**'/'**	**num**	
100	**'*'**	**scroll**	
105	**'-'**	**none**	**<SysReq>** not used
106	**'+'**	**'*'**	
107	**none**	**'-'**	
108	**<enter>**	**'+'**	
110	**esc**	**none**	Not on XT layout
112-123	**F1-F12**	**none**	Not on XT layout
124	**none**	**none**	**<PrtScr>** not used
125	**scroll**	**none**	Not on XT layout
126	**none**	**none**	**<Pause>** not used

See Also

device drivers, fnkey, nkb

Notes

Key 14, if used, varies considerably among keyboard models.

The location of the key that contains characters '\' and '|' varies among 101-key US-layout keyboards.

When designing keyboard tables for keyboards that use the **ALT_GRAPHIC** shift key, for reasons of backwards compatibility you should allow the use of combination shift **ALT+CTRL** as a synonym for **ALT_GRAPHIC**.

kfcopy() — Memory-Manipulation Routine

Copy data from kernel to far address

#include <sys/types.h>
unsigned
kfcopy(*k***,** *fp***,** *n***)**
char ****k*;**
faddr_t *fp***;**
unsigned *n***;**

kfcopy copies *n* bytes from offset *k* in the kernel's data segment to far address *f*. It returns the number of bytes copied.

See Also

memory-manipulation routines

kfree() — Memory-Manipulation 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

memory-manipulation routines

kkcopy() — Memory-Manipulation Routine

Kernel to kernel data copy
int
kkcopy(*src*, *dst*, *n*)
char *src;
char *dst;
unsigned *n*;

kkcopy copies *n* bytes from *src* to *dst* within kernel's data segment. It returns the number of bytes copied.

See Also

memory-manipulation routines

kpcopy() — Memory-Manipulation Routine

Copy from kernel to physical memory
unsigned
kpcopy(*k*, *p*, *n*)
char *k;
paddr_t *p;
unsigned *n*;

kpcopy copies *n* bytes from offset *k* in the kernel's data segment to offset *p* in physical memory. It returns the number of bytes copied.

See Also

memory-manipulation routines

ktty.h — Header File

Kernel portion of tty structure
#include <sys/ktty.h>

The header file **ktty.h** defines the kernel's portion of the teletypewriter (tty) structure. It also defines a set of test macros that can be used to test for specific conditions.

See Also

device drivers, header files

kucopy() — Memory-Manipulation 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** sets **u.u_error** to **EFAULT** and returns zero.

See Also

memory-manipulation routines

ldconfig — Command

Build one or more loadable device drivers
ldconfig [*swap*] [*DRV* ...]

ldconfig creates one or more loadable device drivers in directory **/usr/sys/ldrv**.

Each *DRV* argument names a device driver to create. The driver must exist as an archive of object modules in directory **/usr/sys/lib**. Option *swap* tells **ldconfig** to generate a loadable driver for the swapper into file **/usr/sys/ldrv/swap**. Note that unlike other systems, COHERENT does not require the use of a swapper in order to run. Some releases of COHERENT do not include support for swapping. See the Lexicon entry for **swap** for further details.

By convention, a loadable device driver should be kept in directory **/drv**, not directory **/dev**. To load the driver into memory, use the command **drvld**.

See Also

config, drvld, device drivers, kernel variables

lock() — Accessible 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** must *never* be called from an interrupt handler, block routine, deferred function, or timed function.

See Also

accessible kernel routines

locked() — Accessible Kernel Routine

See if a gate is locked

```
#include <sys/proc.h>
#include <sys/types.h>
int
locked(g)
GATE g;
```

locked is a macro that determines if the specified gate is locked.

See Also

accessible kernel routines

lp — Device Driver

Line printer driver

Files **/dev/lp*** access the line-printer's device drivers for IBM AT COHERENT. The drivers are assigned major device number 3. The COHERENT system supports three printers, in both cooked and raw modes. The following gives the device name, minor device, and I/O port:

/dev/lpt1	0	0x3BC	(/etc/mknod /dev/lpt1 c 3 0)
/dev/lpt2	1	0x378	(/etc/mknod /dev/lpt2 c 3 1)
/dev/lpt3	2	0x278	(/etc/mknod /dev/lpt3 c 3 2)
/dev/rlpt1	128	0x3BC	(/etc/mknod /dev/rlpt1 c 3 128)
/dev/rlpt2	129	0x378	(/etc/mknod /dev/rlpt2 c 3 129)
/dev/rlpt3	130	0x278	(/etc/mknod /dev/rlpt3 c 3 130)

"Cooked" processing processes the special characters BS (backspace), HT (horizontal tab), LF (line feed), FF (form feed), and CR (carriage return) appropriately; raw processing simply passes them on to the printer.

The driver uses a hybrid busy-wait/timeout discipline to support printers efficiently that have varying buffer sizes in a multi-tasking environment.

The kernel variable **LPWAIT_** is the time during which the processor waits for the printer to accept the next character. If the printer is not ready within the **LPWAIT_** time period, the then processor resumes normal processing for the number of ticks set by **LPTIME_**. Thus, setting **LPWAIT_** to a very large number (e.g., 3,000) and **LPTIME_** to a very small number (e.g., one) results in a fast printer, but slow processing on other tasks. Conversely, setting **LPWAIT_** to a small number (e.g., 50) and **LPTIME_** to a large number (e.g., five) result in efficient multi-tasking, but also results in a slow printer unless the printer itself contains a buffer (as is presently normal with all except the least expensive printers). By default, **LPWAIT_** is set to 400 and **LPTIME_** to four. We recommend that you set **LPWAIT_** to no less than 50, and **LPTIME_** to no less than one. The kernel variable **LPTEST_** determines whether or not the device driver checks for the printer being in an "on-line" condition before allowing the device to be used. Users of poorly designed printers which do not support this signal must set kernel variable **LPTEST_** to zero.

Files

/dev/lp* — "Cooked" printer interfaces
/dev/rlp* — Raw printer interfaces

See Also

ascii, db, device drivers, epson, lpr

major() — Driver-Access Routine

Extract major device
#include <sys/stat.h>
#include <sys/types.h>
int
major(*dev*)
dev_t *dev*;

major is a macro that returns a device's major number.

See Also

driver-access routine

memory-manipulation routines — Overview

The following functions can be used by device drivers to manipulate memory:

fclear	Clear far memory
ffbyte	Fetch a far byte
ffword	Fetch a far word
fkcopy	Copy from far address to kernel
getubd	Get a byte from user data space
getuwd	Get a word from user data space
getuwi	Get a word from user code space
kalloc	Allocate kernel memory
kclear	Clear kernel memory
kfcopy	Copy data from kernel to far address
kfree	Free kernel memory
kkcopy	Kernel to kernel data copy
kpcopy	Kernel to physical data copy
kucopy	Kernel to user data copy
pclear	Clear physical memory
pkcopy	Physical to kernel data copy
plrcopy	Left to right physical copy
prlcopy	Right to left physical copy
ptov	Translate from physical to virtual address
pucopy	Copy data from physical to user memory
putubd	Store a byte into user data space
putuwd	Store a word into user data space
putuwi	Put a word into user code space
sfbyte	Set a far byte
sfword	Set a far word
ukcopy	User to kernel data copy
upcopy	User to physical data copy
vrelse	Release virtual address
vremap	Adjust virtual address associated with a segment
vtop	Translate virtual address to physical address

See Also

device drivers

minor() — Driver-Access Routine

Extract minor device
#include <sys/stat.h>
int
minor(*dev*)
dev_t *dev*;

minor is a macro that returns a device's minor number.

See Also

driver-access routines

mmu.h — Header File

Definitions for memory-management unit
#include <sys/mmu.h>

The header file **mmu.h** defines functions that are useful to device drivers that manipulate the memory-management unit (MMU) of the Intel 80X86 family of microprocessors.

See Also

device drivers, header files

ms.h — Header File

Header for Microsoft Mouse driver
#include <sys/ms.h>

The header file **ms.h** holds definitions used by the device driver for the Microsoft Mouse.

See Also

device drivers, header files

ms — Device Driver

Driver for the Microsoft mouse

/dev/mouse is a low-level interface to the traditional Microsoft bus mouse. It does not currently support the Microsoft InPort series of mice. It is assigned major device 10, and is accessed as a character-special device.

The following **ioctl** routines provide access to the mouse:

```
#include <sys/ms.h>
struct msparms parm;
struct mspos   mick;
struct msbuts  buts;
struct mspos   pos;
int st;

ioctl( fd, MS_SETUP,    &parm );
ioctl( fd, MS_SETCRS,   &pos  );
ioctl( fd, MS_GETCRS,   &pos  );
ioctl( fd, MS_READBTNS, &buts );
ioctl( fd, MS_READSTAT, &st   );
ioctl( fd, MS_SETMICK,  &mick );
ioctl( fd, MS_GETMICK,  &mick );
```

The **ioctl** call **MS_SETUP** defines the initial setup for the mouse. The field **accel_t** gives the incremental movement threshold at which the speed of movement will double. The fields **h_cmin** and **h_cmax** give the allowable range of horizontal movement. The fields **v_cmin** and **v_cmax** give the allowable range of vertical movement. The fields **h_mpr** and **v_mpr** specify multipliers to be applied to movement. A movement multipler of zero or one provides single-tick resolution.

The **ioctl** call **MS_SETCRS** changes the active position of the mouse, whereas the call **MS_GETCRS** retrieves the mouse's current position.

The **ioctl** call **MS_READBTNS** retrieves the status of the mouse buttons. It returns the positions at which buttons were pressed and released, and clears the button status.

The **ioctl** call **MS_READSTAT** identifies recently occurring mouse events. If the **MS_S_MOVE** bit is set, the mouse has been moved and the new position can be obtained by the **ioctl** call **MS_GETCRS**. The bits **MS_S_L_PRESS** and **MS_S_L_RELEASE** indicate that the left button has been, respectively, pressed or released. Likewise, the bits **MS_S_R_PRESS** and **MS_S_R_RELEASE** indicate that the right button has been, respectively, pressed or released. The position at which a button was pressed or released can be obtained by the **ioctl** call **MS_READBTNS**.

Finally, the **ioctl** call **MS_SETMICK** changes the mouse-movement multiplers.

Files

/dev/mouse — Character-special file
<sys/ms.h> — Include file

See Also

device drivers

Notes

All mouse support uses the same **/usr/include** file. However, each type of mouse requires its own driver.

nkb — Device Driver

Device driver for console keyboard

The COHERENT device-driver **nkb** supports industry-standard 83-, 101-, and 102-key AT-protocol keyboards attached as the computer console.

nkb lets you define both the layout of the keyboard and the values returned by function keys. You can change layout and function-key bindings by using the special keyboard mapping programs kept in directory **/conf/kbd**. This directory contains the C source code for the mapping tables, as well as a **Makefile** that helps you rebuild the mapping programs. See the Lexicon article **keyboard tables** for details.

nkb understands the following "shift" and "lock" keys:

scroll	Scroll lock
num	Keypad NUM lock
caps	Shift or CAPS lock
lalt	Left ALT key
ralt	Right ALT key
lshift	Left SHIFT key
rshift	Right SHIFT key
lctrl	Left CTRL key
rctrl	Right CTRL key
altgr	ALT Graphic key (non-US keyboards)

nkb records an internal shift state, as defined by the current positions of the shift and lock keys. The shift state is a logical combination of internal states **SHIFT**, **CTRL**, **ALT**, and **ALT_GR**. The **lshift** and **rshift** keys combine to form the current **SHIFT** state for non-alphabetic keys. Alphabetic keys generally use the current state of the **caps** lock key in addition to **lshift** and **rshift**. Numeric keys found on the keypad generally use the state of the **num** lock key combined with **lshift** and **rshift**. The two "control" keys, **lctrl** and **rctrl**, form the internal **CTRL** state. In a similar manner, the two "alt" keys, **lalt** and **ralt**, form the internal **ALT** state. Note that 102-key keyboards generally replace the **ralt** key with the **altgr** key, to allow access to the alternate graphics characters found on some keyboards.

nkb lets you configure or read the internal mapping tables via the following **ioctl()** requests, as defined in header file **<sgtty.h>**:

TIOCGETF	Get function key bindings
TIOCSETF	Set function key bindings
TIOCGETKBT	Get keyboard table bindings
TIOCSETKBT	Set keyboard table bindings

Requests **TIOCGETF** and **TIOCSETF** reference a data structure of type **FNKEY**, which is a **typedef** defined in header file **<sys/kb.h>**. Structure member **k_fnval** is a character array that contains a series of contiguous function key/value bindings; the end of the bindings is marked by manifest constant **DELIM**. You can use any value other than **DELIM** as part of a function-key binding. Structure member **k_nfkeys** indicates how many function keys have associated entries in **k_fnval**. Function keys are numbered from zero through **k_nfkeys**-1.

By convention, function-key 0, when enabled, causes the computer system to reboot. This function key is usually bound to the key sequence **<ctrl><alt><del>**, but you can disable it by setting the value of driver-variable **KBBOOT_** to zero.

Requests **TIOCGETKBT** and **TIOCSETKBT** reference an array that contains **MAX_KEYS** occurrences of data structure **KBTBL**, which is a **typedef** defined in header file **<sys/kb.h>**. Structure member **k_key** contains the *scan code set three* code value for the desired key. Header file **<sys/kbscan.h>** contains manifest (symbolic) constants of the form **K_***nnn*, which map AT keyboard *physical* key number *nnn* to the corresponding scan-code set-three value generated by the keyboard. Note that the **nkb** driver disables the scan-code translation that the keyboard controller normally performs, as well as setting the keyboard to scan code set three.

Structure member **k_val** is a nine-element array that contains the key mappings that correspond to the following index values and shift states:

0	**BASE**
1	**SHIFT**
2	**CTRL**
3	**CTRL_SHIFT**
4	**ALT**
5	**ALT_SHIFT**
6	**ALT_CTRL**
7	**ALT_CTRL_SHIFT**
8	**ALT_GR**

Structure member **k_flags** contains mode information for the given key. One field in **k_flags** indicates the *class* of key. This sub-field lets you specify whether a key is a "shift" key (as defined above), a special or programmable "function" key, or a "regular" key. The following symbolic constants specify the *class* of key:

S The specified key is a "shift" or "lock" key. Note that all entries in array **k_val** must be identical for a "shift" or "lock" key to work correctly.

F The specified key is a "function" or special key. The value of all elements of array **k_val** must specify a function key number. See header file **<kb.h>** for a list of predefined function keys.

O The specified key is "regular" and requires no special processing.

The next sub-field of **k_flags** specifies the *type* of key, as specified in the AT keyboard technical reference. The *type* sub-field specifies under what conditions a given key will generate an interrupt. The possible choices are:

M Make: generate an interrupt only upon key "make" (i.e., when the key is depressed). This mode is useful for keys which do not repeat. Note that using this mode with "shift" keys stops you from unshifting upon release of the key!

T Typematic: generate an interrupt when the key is depressed, and generate subsequent key-depression interrupts while the key is depressed. The rate at which interrupts are generated is specified by the typematic rate of the keyboard. This type is usually associated with a "regular" key.

MB Make/Break: generate an interrupt when the key is depressed, and when it is released. No additional interrupts are generated no matter how long the key is depressed. This mode is used for "shift" keys.

TMB Typematic/Make/Break: generate an interrupt when the key is first depressed; generate subsequent key depression interrupts while the key remains depressed; and generate an interrupt when the key is released.

The last sub-field of **k_flags** specifies the *lock* keys, if any, that affect the specified key:

C The **caps** lock key that affects this key. If the specified key is depressed while **caps** lock is active, it is equivalent to having used either of the SHIFT keys with this key. When **caps** lock is in effect, use of either of the SHIFT keys temporarily toggles the state of the **caps** lock.

N The **num** lock key affects this key. If the specified key is depressed while **num** lock is active, it is equivalent to having used either of the SHIFT keys in conjunction with the specified key. When **num** lock is in effect, use of either of the SHIFT keys temporarily toggles the state of the **num** lock.

References

Technical Reference for the IBM Personal Computer AT, IBM Corporation, 1984.

Multi-Function Keyboards: Layouts, Cherry Electrical Products Corp.

See Also

device drivers, fnkey, keyboard tables

nondsig() — Signal-Handler 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

signal-handler routines

nonedev() — Driver-Access Routine

Illegal device request
void
nonedev()

nonedev sets the field **u.u_error** to **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

driver-access routines

nulldev() — Driver-Access Routine

Ignored device request
void
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

driver-access routines

outb() — Accessible Kernel Routine

Output a byte to an I/O port
int
outb(*port*, *c***)**
unsigned *port*;
char *c*;

outb writes character *c* to *port*.

See Also

accessible kernel routines

panic() — Accessible 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

accessible kernel routine, printf

pclear() — Memory-Manipulation Routine

Clear physical memory

```
#include <sys/types.h>
void
pclear(p, n)
paddr_t p;
fsize_t n;
```

pclear clears *n* bytes of memory at physical address *p*.

See Also

memory-manipulation routines

pkcopy() — Memory-Manipulation Routine

Physical to kernel data copy

```
unsigned
pkcopy(p, k, n)
paddr_t p;
char *k;
unsigned n;
```

pkcopy copies *n* bytes from address *p* in physical memory to address *k* in the kernel's data segment. It returns the number of bytes copied.

See Also

memory-manipulation routines

plrcopy() — Memory-Manipulation Routine

Left to right physical copy

```
#include <sys/types.h>
plrcopy(p1, p2, n)
paddr_t p1, p2;
fsize_t n;
```

plrcopy copies *n* bytes from address *p1* to address *p2*. As its name implies, it copies from left to right. Note that this routine can copy no more than 64 kilobytes of data.

See Also

memory-manipulation routines, prlcopy()

pollopen() — Accessible Kernel Routine

Initiate driver polled event

```
void
pollopen(eventp)
event_t *eventp;
```

pollopen creates a polled event on the event structure pointed to by *eventp*. The event structure must reside in static kernel data space.

See Also

accessible kernel routines

pollwake() — Accessible 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 **pollwake** does not need to be called.

See Also

accessible kernel routines

printf() — Accessible Kernel Routine

Formatted print

```
void
printf(format, arg, ... )
char *format;
```

The kernel's version of **printf** is 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 the standard-I/O (STDIO) **printf** library function.

See Also

accessible kernel routines, printf()

Notes

Note that 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.

prlcopy() — Memory-Manipulation Routine

Right to left physical copy

```
#include <sys/types.h>
prlcopy(p1, p2, n)
paddr_t p1, p2;
int n;
```

prlcopy copies *n* bytes from address *p1* to address *p2*. As its name implies, it copies data from right to left. Note that this function can copy no more than 64 kilobytes of data.

See Also

memory-manipulation routines, plrcopy()

ptov() — Memory-Manipulation Routine

Translate from physical to virtual address

```
#include <sys/mmu.h>
#include <sys/types.h>
faddr_t
ptov(paddr, len)
```

paddr_t *paddr***;**
fsize_t *len***;**

ptov initializes a virtual address to access physical memory at location *paddr*, of size *len* bytes. It provides read and write (but not execute) access. At most, 8,191 virtual addresses are available simultaneously. When no longer required, a virtual address should be released by **vrelse**.

See Also

memory-allocation routines

Notes

If space is not available for a descriptor, a system panic will occur.

ptrace.h — Header File

Process trace
#include <sys/ptrace.h>

The header file **ptrace.h** holds definitions used by routines that perform process tracing. Among other things, it defines the structure **ptrace**.

See Also

device drivers, header files

pucopy() — Memory-Allocation Routine

Copy data from physical to user memory
#include <sys/types.h>
unsigned
pucopy(*p***,** *u***,** *n***)**
paddr_t *p***;**
char ****u*;**
unsigned *n***;**

pucopy copies *n* bytes from address *p* in physical memory to address *u* in the user's data segment. It returns the number of bytes copied. If an address fault occurs, **pucopy** sets **u.u_error** to **EFAULT** and returns zero.

See Also

memory-allocation routines

putq() — Terminal-Device Routine

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

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

See Also

terminal-device routines

putubd() — Memory-Manipulation Routine

Store a byte into user data space

putubd(*u***,** *b***)**
char ****u*;**
char *b***;**

putubd stores byte *b* at address *u* in the user's data segment. If an address fault occurs, it sets field **u.u_error** to **EFAULT**.

See Also

memory-manipulation routines

putuwd() — Memory-Manipulation 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 sets field **u.u_error** to **EFAULT**.

See Also

memory-manipulation routines

putuwi() — Memory-Manipulation 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 sets field **u.u_error** to **EFAULT**.

See Also

memory-manipulation routines

race condition — Definition

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( &some_variable_in_the_kernel_data_area )
restore interrupt mask
```

If an interrupt were to occur between the **while** statement and the call to **sleep()**, the driver would never wake up because the event it was waiting for (sleeping on) will have already occurred. To avoid this situation, your code must this block of code with calls to the kernel functions **sphi()/spl()**. This will ensure that interrupts cannot occur until after **sleep()** has been called. The system will re-enable interrupts when the driver calls **sleep()**, but it is guaranteed to have the same interrupt level (mask) when it awakens, thus preserving the lockout of the interrupt handler.

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

device drivers

ram — Device Driver

Driver for manipulating RAM

The COHERENT **ram** devices let you allocate and use the random access memory (RAM) of the computer system directly. A typical use is for a RAM disk, which is a COHERENT file system kept in memory rather than on a floppy disk or hard disk.

The COHERENT RAM device driver has major number 8. It can be accessed either as a block-special device or as a character-special device. The high-order bit of the minor number gives a RAM device number (0 or 1), which lets you use up to two RAM devices simultaneously. The low-order seven bits specify the device size in 64-kilobyte increments. The first **open** call on a RAM device with nonzero size (1 to 127) allocates memory for the device; the system call **open** fails if sufficient memory is not available. Accessing a RAM device with a minor number specifying size zero frees the allocated memory, provided all earlier **open** calls have been closed.

Initially, COHERENT includes two block-special devices for RAM disks: the 512-kilobyte device **/dev/ram0** (8, 8) and the 192-kilobyte device **/dev/ram1** (8, 131). It also includes the devices **/dev/ram0close** (8, 0) and **/dev/ram1close** (8, 128). You should change the RAM devices to sizes appropriate for the amount of memory available on your system.

Examples

The following example formats and mounts a 512-kilobyte RAM disk on directory **/fast**.

```
mkdir /fast
/etc/mkfs /dev/ram0 1024
/etc/mount /dev/ram0 /fast
```

When the RAM disk is no longer needed, its allocated memory can be freed as follows:

```
/etc/umount /dev/ram0
cat /dev/null >/dev/ram0close
```

The next example replaces the default **/dev/ram0** with a one-megabyte device containing a COHERENT file system. The new minor number 16 specifies RAM device 0 and size 16 times 64 kilobytes (i.e., one megabyte). The new RAM device contains 2,048 blocks of 512 bytes each.

```
rm /dev/ram0
/etc/mknod /dev/ram0 b 8 16
/etc/mkfs /dev/ram0 2048
```

Files

/dev/ram*

See Also

compress, device drivers, fsck, mkfs, mount, umount, uncompress, zcat

Notes

Moving frequently used commands or files to a RAM disk can improve system performance substantially. However, the contents of a RAM device are lost if the system loses power, reboots, or crashes, files kept on a RAM disk should frequently be copied the hard disk or floppy disk.

If a RAM device uses most but not all available system memory, its **open** call will succeed but subsequent commands may fail because insufficient memory remains for the system.

The COHERENT installation program **/etc/build** uses RAM device **/dev/ram1** as a RAM disk during installation. Commands **compress**, **uncompress**, **zcat**, and **fsck** sometimes use **/dev/ram1** as a temporary storage device. Users should avoid using **/dev/ram1** as a RAM disk because of these programs. In addition, users of **compress**, **uncompress**, and **zcat** may have to change the size of **/dev/ram1** from the default size of 192 to 512 kilobytes, to handle files compressed to 16 bits. The following script makes this change; note that it must be run by the superuser **root**:

```
cat /dev/null >/dev/ram1close
rm /dev/ram1 /dev/rram1
mknod /dev/ram1  b 8 136
mknod /dev/rram1 c 8 136
```

Please note that increasing the size of **/dev/ram1** to 512 kilobytes requires a system with at least one megabyte of RAM.

rs — Device Driver

Raw serial device driver

/dev/rs1 and **/dev/rs2** are the raw serial-line drivers. They are assigned major devices 5 and 6, and are accessed by character-special files. The following lists the available interfaces

/dev/rs0	(serial port 0)	mknod /dev/rs0	c 5 0
/dev/rs1	(serial port 1)	mknod /dev/rs1	c 6 0
/dev/rs0r	(modem port 0)	mknod /dev/rs0r	c 5 128
/dev/rs1r	(modem port 1)	mknod /dev/rs1r	c 6 128

The driver supports the following System-V **termio ioctl()** calls. Note well that this device driver is not compatible with the **ioctl()** calls found in header file **<sgtty.h>**. See the header file **<termio.h>** for details:

```
#include <termio.h>
struct termio tb;

ioctl( fno, TCGETA,  &tb  );
ioctl( fno, TCSETA,  &tb  );
ioctl( fno, TCSETAW, &tb  );
ioctl( fno, TCSETAF, &tb  );
ioctl( fno, TCXONC,  0..1 );
ioctl( fno, TCFLSH,  0..2 );
ioctl( fno, TCSBRK,  0..n );
```

The driver recognizes the following flags:

c_iflag: ISTRIP, IXON, IXANY, INPCK, IGNPAR, PARMRK, IGNBRK.

c_cflag: CBAUD, CSIZE, CSTOPB, CREAD, PARENB, PARODD, HUPCL, CLOCAL.

c_oflag: OPOST, ONLCR, ONLRET, TAB3.

The **/dev/rs*** devices provide fast communications (up to 19.2K baud) standard IBM AT serial ports. They are intended for protocol support and so implement only the following System-V-compatible features:

- Baud rates from 50 to 19.2K baud.
- Strip input character to 7 bits.
- XON/XOFF output flow control.
- Hardware output flow control using CTS handshaking.
- Modem control.
- Input parity check.
- Character size of 5, 6, 7, or 8 bits.
- One or two stop bits.
- Hangup on last close.
- Local or dial-up line.
- Map newline to newline/carriage return.
- Map tab to an appropriate number of spaces.

Reads are atomic. A read either transferrs some data (1 ... n) from the input buffer and returns a code that indicates success, or it transfers no data and it returns -1 and sets **errno** to **EINTR**.

Writes of 512 bytes or less are atomic. Either the driver transfers all data into an output buffer and returns a code that indicates success, or it transfers no data and it returns -1 and sets **errno** **EINTR**.

Modem control provides carrier monitoring and hardware flow control.

Carrier monitoring uses the Data-Carrier-Detect (DCD) signal to control processes attached to the port. An open on the modem line blocks until a carrier is present or a signal is sent to the blocked process. Loss of carrier generates a hangup signal to all attached processes.

Hardware flow control utilizes CTS handshaking. Transmission does not start until CTS becomes true, and stops if CTS becomes false. This feature should be enabled when using specific printers (i.e., the Texas Intruments 810 or 850) or high speed modems (i.e., the Telebit Trailblazer).

To enable modem control, access **/dev/rs0m** or **/dev/rs1m** instead of **/dev/rs0** or **/dev/rs1**, respectively. Alternatively, the **CLOCAL** bit in the **termio** field **c_cflag** can be cleared, as follows:

```
#include <termio.h>
struct termio tb;

ioctl( fno, TCGETA, &tb );
tb.c_cflag &= -CLOCAL;
ioctl( fno, TCSETA, &tb );
```

Files

<termio.h>
/dev/rs* — Character-special files

See Also

device drivers, termio.h

Notes

In general, it is not possible to run these drivers simultaneously at maximum speed.

Some COHERENT commands (e.g., **ksh**, **more**, **vi**, **stty** and **login**) do not work with these drivers as they are Version-7 (i.e., **<sgtty.h>**) rather than System-V (i.e., **<termio.h>**) compatible.

salloc() — Segment-Manipulation Routine

Allocate a segment

```
#include <sys/seg.h>
SEG *
salloc(len, flag)
fsize_t len;
int flag;
```

salloc allocates a segment 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 will call **sfree** when done), the device driver should accordingly increment the fields **s_urefc** and **s_refc** in the **seg** structure.

flag can be set to one or more of the following values:

SFSYST The segment is to be a system segment, and will not be associated with a user process.

SFHIGH The segment is to be allocated from the high end of memory.

SFNSWP The segment must be memory resident.

SFNCLR The segment does not have to be initialized to zero.

Device drivers should normally use **SFSYST**, **SFHIGH**, and **SFNSWP**. These constants are defined in header file **seg.h**.

See Also

segment-manipulation routines

SCSI — Device Driver

SCSI device drivers

The COHERENT SCSI series of device drivers lets you use SCSI-interface devices attached to host adapters from several vendors.

All COHERENT SCSI device drivers use major number 13, thus allowing all SCSI devices to be accessed via standard device-naming conventions. Peripherals can be accessed as either block- or character-special devices. The minor number specifies the device and partition number for disk-type devices; this allows the use of up to eight SCSI identifiers (SCSI-ID's), with up to four logical unit numbers (LUNs) per SCSI-ID and up to four partitions per LUN. Tape and other special devices decode the minor number to perform special operations such as "rewind on close" or "no rewind on close".

The first **open** call on a SCSI disk device allocates memory for the partition table and reads it into memory.

See the release notes for further information regarding supported host adapters and peripherals.

Files

/dev/sd* — block-special devices
/dev/rsd* — character-special devices

See Also

aha154x, device drivers, drvld, ss

Notes

The Mark Williams Company's bulletin board makes available loadable device drivers for various SCSI host adapters, as well as device driver updates. See the release notes for further information.

seggrow() — Segment-Manipulation Routine

Adjust segment size

```
#include <sys/seg.h>
int
seggrow(sp, len)
SEG *sp;
fsize_t len;
```

seggrow tries to change the size of segment *sp* to *len* bytes. It returns one for success, and zero for failure. The segment may be moved in memory, or swapped out and back in.

See Also

segment-manipulation routines

segment-manipulation routines — Overview

The following routines can be used by device drivers to manipulate segments:

salloc	Allocate a segment
seggrow	Adjust segment size
sfree	Free a segment

See Also

device drivers

sendsig() — Signal-Handler 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

signal-handler routines

setivec() — Interrupt-Handler Routine

Set an interrupt vector
void
setivec(*level*, *function***)**
int *level*;
int (**function***)();**

setivec establishes the routine pointed to by *function* as the handler for interrupt vector *level*. If the interrupt vector is already in use, it sets field **u.u_error** to **EDBUSY**.

See Also

clrivec(), interrupt-handler routines

Notes

You must call **setivec** from the load or unload routines in your driver. If you call it from any other entry point within the driver, a panic will occur.

sfbyte() — Memory-Manipulation Routine

Set a far byte
#include <sys/types.h>
void
sfbyte(*fp*, *b***)**
faddr_t *fp*;
char *b*;

sfbyte writes byte *b* to address *fp*. Note that an address fault will cause the system to panic.

See Also

memory-manipulation routines

sfree() — Segment-Manipulation Routine

Free a segment
void
sfree(*sp***)**
**SEG ****sp*;

sfree decrements the reference count for *sp*. It frees the segment if it is no longer referenced.

See Also

segment-manipulation routines

sfword() — Memory-Manipulation Routine

Set a far word
#include <sys/types.h>
void
sfword(*fp*, *w***)**
faddr_t *fp*;
int *w*;

sfword writes word *w* to address *fp*. Note that an address fault cause the system to panic.

See Also

memory-manipulation routines

sigdump() — Signal-Handler Routine

Generate core dump
void
sigdump()

sigdump writes a dump of the current process into file *core* in the current directory. It does not return.

See Also

signal-handler routines

signal-handler routines — Overview

The following functions can be used by device drivers to handle signals:

actvsig	Activate signal handler
nondsig	Non-default signal pending
sendsig	Send a signal
sigdump	Generate core dump

See Also

device drivers

sleep() — Accessible Kernel Routine

Wait for event or signal
#include <sys/sched.h>
void
sleep(*e*, *cv*, *iv*, *sv*)
**char ****e*;
int *cv*, *iv*, *sv*;

sleep suspends processing of a process until event *e* has completed. *e* normally represents a data item's address in the static kernel data space.

cv is the scheduling value set to obtain the CPU as soon as the process awakes. *iv* is the swap value obtained to keep the process in memory for the duration of the sleep. *sv* is the swap value that allows the process to be swapped in if it has been swapped out. The following table gives the manifest constants to use with *cv*, *iv*, and *sv* for normal processing tasks, as set in the header file **<sys/sched.h>**:

Child Process	**CVCHILD**	**IVCHILD**	**SVCHILD**
Swapper	**CVSWAP**	**IVSWAP**	**SVSWAP**
Wait for Block I/O to Complete	**CVBLKIO**	**IVBLKIO**	**SVBLKIO**
Wait for Gate to Open	**CVGATE**	**IVGATE**	**SVGATE**
Terminal Output	**CVTTOUT**	**IVTTOUT**	**SVTTOUT**
Wait for Free clists	**CVCLIST**	**IVCLIST**	**SVCLIST**
Process Trace	**CVPTSET**	**IVPTSET**	**SVPTSET**
Process Trace Stop	**CVPTRET**	**IVPTRET**	**SVPTRET**
Waiting for a Pipe	**CVPIPE**	**IVPIPE**	**SVPIPE**
Terminal Input	**CVTTIN**	**IVTTIN**	**SVTTIN**
Pause	**CVPAUSE**	**IVPAUSE**	**SVPAUSE**
Wait	**CVWAIT**	**IVWAIT**	**SVWAIT**

If *cv* is less than *CVNOSIG*, then signals may abort the process without returning from the sleep.

Please note the following caveats when using **sleep**. Disobeying these rules can jeopardize the health of your system.

First, your driver can **sleep** while it waits for some condition to be satisfied. However, the **sleep** may return prematurely; therefore, you must place the call to **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)
        sleep( &some_variable_in_the_kernel_data_area)
restore interrupt mask
```

The interrupt routine will, in turn, call **wakeup** or defer wakeup for later background processing if time is not an issue. This will cause the aforementioned code to return from the **sleep** call.

As you can see, there is an inherent race condition between the **while** and **sleep**. If the work is serviced while the driver is **sleeping**, the **while** loop will work correctly. However, should the last interrupt happen after the **while** but before the **sleep**, the driver will deadlock — it will, in effect, be waiting for Godot.

sleep returns for various reasons, but you cannot always depend on it to return for reasons other than a process calling **wakeup** on the variable that your driver fell asleep on. So, if your driver is waiting for something to happen based upon an interrupt, be sure to bracket the call to **sleep** with calls to the kernel routines **sphi** and **spl**.

See Also

accessible kernel routines, sphi(), spl(), wakeup()

Notes

Please note the following warnings:

- Do not call **sleep**, either directly or indirectly, from the block routine of a driver.
- Do not call **sleep**, either directly or indirectly, from with an interrupt handler. When the interrupt occurs, the driver does not know which process was running at the time, so it does not whose *u area* it will be sleeping on. Thus, calling **sleep** from within an interrupt handler will deadlock your driver.
- Calling **sleep** from the load routine of a driver linked to the kernel will cause a panic.

sphi() — Interrupt-Handler Routine

Disable interrupts
int
sphi()

sphi disables hardware interrupts. It returns a value that describes the previous hardware interrupt state. The return value can later be passed to function **spl** to restore the previous hardware interrupt state.

See Also

interrupt-handling routines, spl()

spl() — Interrupt-Handler Routine

Adjust interrupt mask
int
spl(*s*)
int *s*;

spl restores the hardware interrupt state to state *s*, which was returned by functions **sphi** or **spl**.

See Also

interrupt-handler routines, sphi(), splo()

splo() — Interrupt-Handler Routine

Enable interrupts
int
splo()

splo enables hardware interrupts. It returns a value that describes the previous hardware interrupt state. Using **splo** to enable interrupts unconditionally is undesirable, and may indeed corrupt the system state. Use **spl** to return to the previous interrupt mask level.

See Also

interrupt-handler routines, spl()

ss — Device Driver

Future Domain/Seagate SCSI device driver

The device driver **ss** lets you use SCSI interface devices attached to any of the following host adapters:

Future Domain TMC-845/850/860/875/885
Future Domain TMC-840/841/880/881
Seagate ST01/ST02

This driver has major number 13. It can be accessed either as a block-special device or as a character-special device. The minor number specifies the device and partition number for disk-type devices, letting you use up to eight SCSI-IDs, with one logical unit number (LUN), LUN 0, per SCSI-ID and up to four partitions per LUN. The present version does not support non-zero LUN's.

The first **open** call on a SCSI disk device reads the partition table into memory.

Controller Configuration

Your Future Domain or Seagate host adapter must be installed with interrupts enabled in order for it to work with COHERENT. If you have been running your host adapter with interrupts disabled, a good first choice for interrupt number is IRQ 5, unless you know that you have another device installed on your computer that already makes use of this interrupt. Consult the instructions provided with your host adapter, and the jumper settings, to determine the IRQ number.

The base address value used by the **ss** device driver is the four-digit hexadecimal memory segment number of the host adapter's starting address. This number is most often **CA00**; other common values are **C800**, **CC00**, **CE00**, **DC00**, and **DE00**. You must use the correct value, as specified by the jumper settings on your host adapter.

Device driver variables **SS_BASE_** and **SS_INT_** correspond to the base address and interrupt vector, respectively. Device driver variable **NSDRIVE_** must be patched before the driver is loaded. The low-order byte of this variable is a "bit map" indicating the SCSI-ID's of all installed target devices. The high-order byte indicates the type of host adapter. Labeling the bits in the low-order byte of **NSDRIVE_** as follows:

Bit number: 7 6 5 4 3 2 1 0 ← *least significant bit*

there should be a value of 1 for each installed target device. Do not set a value of 1 for the SCSI-ID of the host adapter. The high-order byte of **NSDRIVE_** is **0x00** for Seagate ST01 and ST02, **0x80** for TMC-845/850/860/875/885, and **0x40** for TMC-840/841/880/881. For example, if you are using a TMC-885 and a single hard drive with SCSI ID of zero, then set **NSDRIVE_** to 0x8001. See Lexicon article **hs** for an example of how to configure a device driver.

When processing BIOS I/O requests prior to booting COHERENT, SCSI host adapters use "translation-mode" drive parameters: number of heads, cylinders, and sectors per track. These numbers are called translation-mode parameters because they have nothing to do with physical drive geometry. The translation-mode parameters used by the BIOS code present on your host adapter can be obtained using the **dpb** utility found on the boot diskette of versions 3.2.0 and later of COHERENT.

The **ss** device driver has a table, **drv_parm_**, which contains eight two-word entries — one for each possible SCSI-ID. The first word of each entry must contain the number of cylinders for the drive. The high-order byte of the second word is the number of sectors per track; the low-order byte is the number of heads. Entries in **drv_parm_** should be patched for each drive which is accessible by the BIOS. Values need not be patched for drives inaccessible by the BIOS. Note that BIOS code is executed by COHERENT only during the initial bootstrap. After that, drive parameters are of no consequence since SCSI I/O requests are based upon logical block number, rather than on cylinder/head/sector addressing.

The installation procedure for COHERENT versions 3.2.0 and later patches all necessary variables for the accompanying version of the **ss** driver by executing the command:

```
/etc/mkdev scsi
```

Minor Device Numbers

The **ss** driver usually makes use of special files **/dev/sd*** and **/dev/rsd***. For information on the meaning of minor numbers with these special files, see the article on **aha154x**.

Loading the Driver

The **ss** loadable device driver must be loaded on a system that does not have a SCSI hard disk as the root device. To do so, use the command **/etc/drvld**, as follows:

```
/etc/drvld -r /drv/ss
```

Files

/dev/sd* — block-special devices
/dev/rsd* — character-special devices

See Also

device drivers, drvld, scsi

Notes

Current releases of the **ss** device driver support disk-type devices only. Zero is the only LUN allowed. A future version of the driver will add support for tape-type and other devices, as well as nonzero LUN's.

In version 3.2.0 of COHERENT, another variable, **SS_HOST_**, must be patched in the driver to be equal to the SCSI-ID of the host adapter. This value is 6 for Future Domain adapters, and 7 for Seagate. Variable **SS_HOST_** has been deleted from versions of the **ss** driver later than that shipped with COHERENT 3.2.0.

st — Device Driver

Archive SC-400 streaming-tape driver

The **/dev/rst*** devices provide access to the Archive SC-400 streaming tape controller. Each entry is assigned major device number 12, and may be accessed as a character-special device.

The **st** tape driver handles one 0.25-inch streaming-tape drive. Minor device 0 requests allocation of a 256-kilobyte tape cache and should be used unless the system has minimal memory (e.g., less than 640 kilobytes). Minor devices 1 through 127 request allocation of a tape cache of one to 127 kilobytes. These devices normally rewind the tape during the close; adding 128 to a minor-device number specifies non-rewind on close.

For an interface to be accessible from the COHERENT system, a device file must be present in directory **/dev** with the appropriate type, major, and minor device numbers, and permissions. The following gives an example form of the command **mknod** to creates a special file for a device:

```
/etc/mknod /dev/rst256 c 12 0
/etc/mknod /dev/nrst256 c 12 128
```

Tape-oriented commands under COHERENT (e.g., **tar**) normally the disk devices to store their output. The following sample commands associate the generic interface with the Archive streaming tape driver:

```
/bin/ln -f /dev/rst256 /dev/rmt
/bin/ln -f /dev/nrst256 /dev/nrmt
```

Depending on the amount of memory available, you may wish to restrict the amount of memory used to buffer tape data. This may be done by linking the appropriate **/dev/rst** entry to **/dev/rmt**. For example, **/dev/rst64** allocates 64 kilobytes during tape transfer whereas **/dev/rst32** allocates only 32 kilobytes.

Hardware

The following kernel variables define the hardware interface to streaming tape.

STIRQ Specify the interrupt vector (default, 3).

STPORT Specify the input/output port (default, 0x200).

STDMA Specify the DMA channel (default, 1).

Should these parameters conflict with other system hardware, you should use the command **/conf/patch** to rebuild the kernel appropriately. See the Lexicon article on **hs** for sample commands.

Files

/dev/rst* — Auto-rewind character-special file
/dev/nrst* — Non-rewinding character-special file
<sys/mtioctl.h>— Tape **ioctl** commands

See Also

device drivers, tar

Notes

As delivered, the Archive tape controller uses interrupt vector 3. If this interrupt is to be used, then the COHERENT kernel must be configured without the second serial line driver (e.g., **/dev/com2***).

super() — Accessible Kernel Routine

Verify super-user
super()

super checks whether the user has super-user privileges. It return one if the user has these priviliges (i.e., if **u.u_uid == 0**). Otherwise, it sets field **u.u_errer** to **EPERM** and returns zero.

See Also

accessible kernel routines

systab.h — Header File

System-call table
#include <sys/systab.h>

The header file **systab.h** holds definitions used by routines that manipulate the system-call table.

See Also

device drivers, header files

terminal-device routines — Overview

The following routines can be used by device drivers to access teletypewriter (tty) devices:

clrq	Clear character queue
getq	Get a char from a character queue
putq	Put a character onto a character queue
ttclose	Close tty
ttflush	Flush a tty
tthup	tty hangup
ttin	Pass character to tty input queue
ttioctl	Perform tty I/O control
ttopen	Open a tty
ttout	Get next character from tty output queue
ttread	Read from tty
ttsetgrp	Set tty process group
ttsignal	Send tty signal
ttstart	Start tty output
ttwrite	Write to tty

See Also

device drivers

timeout() — Accessible Kernel Routine

Defer function execution
#include <sys/timeout.h>
void
timeout(*tp*, *n*, *function*, *a***)**
TIM **tp*;
int *n*;
int (**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 must be a static structure located in the kernel's data segment. Any previous activation of a timer on the same timing structure will be cancelled.

Calling **timeout** with *function* set to NULL will cancel a timer. A timed function should never sleep or alter the contents of the **u** structure.

See Also

accessible kernel routines

tn — Device Driver

Tiac 236/238 ARCNET driver

/dev/tn* provides access to an ARCNET local area network via a Tiac 236 card, Tiac 238 card or equivalent (e.g., Pure Data ARCNET card). Each entry is assigned major device number 20, and may be accessed as a character-special device.

The **tn** driver supports up to four ARCNET cards in a single computer. Minor devices 0, 1, 2, and 3 refer to each card. For a card to work properly, it must have a unique interrupt, 64-kilobyte memory bank, and port number assigned to it. The driver must also be configured to the same interrupt, memory bank, and port number. You can use the command **/conf/patch** to build a properly configured version of the kernel; see the Lexicon article **hs** for sample commands. If loadable device drivers are used they may be configured in the identical fashion.

For an interface to be accessible from the COHERENT system, a device file must be present in directory **/dev** with the appropriate type, major and minor device numbers, and permissions. You can use the command **mknod** to creates a special file for a device, as follows:

```
/etc/mknod /dev/tn0  c 20  0
/etc/mknod /dev/tn1  c 20  1
```

It is usual to have a generic LAN interface **/dev/tn**. This is associated with a particular LAN card by the following command:

```
/bin/ln -f /dev/tn0  /dev/tn
```

This device driver provides a raw interface to the LAN. To communicate with other computers on the network, it is normally necessary to add some higher level protocol (e.g., XNS or TCP/IP).

Files

/dev/tn* — LAN network access special file
/dev/tn — Default LAN

See Also

device drivers, ln, mknod

Notes

As delivered, the LAN driver supports one card with interrupt 2, port 0x2E0, and bank 0xD000.

ttclose() — Terminal-Device 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

terminal-device routines

ttflush() — Terminal-Device Routine

Flush a tty
#include <sys/ttflush>
void
ttflush(*tp*)
TTY * *tp*;

ttflush clears the input and output queues, and resets most state flags.

See Also

terminal-device routines

tthup() — Terminal-Device 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

terminal-device routines

ttin() — Terminal-Device 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 must be called with interrupts disabled.

See Also

terminal-device routines

ttioctl() — Terminal-Device Routine

Perform tty I/O control
#include <sys/tty.h>
#include <sgtty.h>
void
ttioctl(*tp*, *com*, *vec*)
TTY **tp*;
int *com*;
struct sgttyb **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 sets field **u.u_error** to an appropriate value. It returns nothing.

See Also

terminal-device routines

ttopen() — Terminal-Device Routine

Open a tty
#include <sys/tty.h>
#include <sgtty.h>
void
ttopen(*tp*)
TTY *tp;

ttopen is called by a teletypewriter (tty) device driver on the first open. It sets up default parameters, and invokes **(*tp->t_param)(*tp*)** to initialize the hardware.

See Also

terminal-device routines

ttout() — Terminal-Device Routine

Get next character from tty output queue
#include <sys/tty.h>
int
ttout(*tp*)
TTY *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

terminal-device routines

ttread() — Terminal-Device Routine

Read from tty
#include <sys/io.h>
#include <sys/tty.h>
void
ttread(*tp*, *iop*, 0)
TTY *tp;
IO *iop;

ttread moves data from the input queue associated with *tp*, to the I/O segment referenced by *iop*. If an error occurs, **ttread** sets field **u.u_error** to an appropriate value.

See Also

terminal-device routines

ttsetgrp() — Terminal-Device 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

terminal-device routines

ttsignal() — Terminal-Device 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

terminal-device routines

ttstart() — Terminal-Device Routine

Start tty output
#include <sys/tty.h>
void
ttstart(*tp*)
TTY **tp*;

ttstart starts output on a teletypewriter (tty) device if output is not disabled.

See Also

terminal-device routines

ttwrite() — Terminal-Device Routine

Write to tty
#include <sys/io.h>
#include <sys/tty.h>
void
ttwrite(*tp*, *iop*, **0**)
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 sets field **u.u_error** to an appropriate value.

See Also

terminal-device routines

ukcopy() — Memory-Manipulation Routine

User to kernel data copy
unsigned
ukcopy(*u*, *k*, *n*)
char **u*;
char **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 sets field **u.u_error** to **EFAULT**, and returns zero.

See Also

memory-manipulation routines

unlock() — Accessible 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 will allow the kernel to reschedule processes that had previously been blocked.

See Also

accessible kernel routines, lock()

upcopy() — Memory-Manipulation Routine

User to physical data copy
#include <sys/types.h>
unsigned
upcopy(*u*, *p*, *n*)
char **u*;
paddr_t *p*;
unsigned *n*;

upcopy copies *n* bytes from address *u* in the user's data segment to address *p* in physical memory. It returns the number of bytes copied. If an address fault occurs, it sets field **u.u_error** to **EFAULT** and returns zero.

See Also

memory-manipulation routines

vrelse() — Memory-Manipulation Routine

Release virtual address
#include <sys/mmu.h>
#include <sys/types.h>
void
vrelse(*faddr*)
faddr_t *faddr*;

vrelse releases a virtual address that was previously obtained with functions **vremap** or **ptov**. It is a fatal error to release a virtual address more than once. Only 8,191 virtual addresses can be allocated at any one time.

See Also

memory-manipulation routines, ptov(), vremap()

vremap() — Memory-Manipulation Routine

Adjust virtual address associated with a segment
#include <sys/mmu.h>
#include <sys/seg.h>
void
vremap(*sp*)
**SEG ****sp*;

vremap allocates or adjusts the virtual address associated with the segment referenced by *sp*. If *sp*->**s_faddr** is zero, **vremap** allocates a new virtual address. The virtual address limit will be adjusted to *sp*->**s_size-1**. If field *sp*->**s_flags** contains value **SFCORE**, the virtual address will be memory resident. If field *sp*->**s_flags** contains value **SFTEXT**, the virtual address will be read-execute; otherwise, it will be read-write.

See Also

memory-manipulation routines

vtop() — Memory-Manipulation Routine

Translate virtual address to physical address
#include <sys/mmu.h>
#include <sys/types.h>
paddr_t
vtop(*faddr*)
faddr_t *faddr*;

vtop returns the current physical address associated with virtual address *faddr*.

See Also

memory-manipulation routines

wakeup() — Accessible 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

accessible kernel routines, sleep()

Index