Linux Cdrom Documentation

The kernel development community

Jun 10, 2024

CONTENTS

CHAPTER

ONE

A LINUX CD-ROM STANDARD

Author
David van Leeuwen <david@ElseWare.cistron.nl>

Date
12 March 1999

Updated by
Erik Andersen (andersee@debian.org)

Updated by
Jens Axboe (axboe@image.dk)

1.1 Introduction

Linux is probably the Unix-like operating system that supports the widest variety of hardware devices. The reasons for this are presumably

- The large list of hardware devices available for the many platforms that Linux now supports (i.e., i386-PCs, Sparc Suns, etc.)
- The open design of the operating system, such that anybody can write a driver for Linux.
- There is plenty of source code around as examples of how to write a driver.

The openness of Linux, and the many different types of available hardware has allowed Linux to support many different hardware devices. Unfortunately, the very openness that has allowed Linux to support all these different devices has also allowed the behavior of each device driver to differ significantly from one device to another. This divergence of behavior has been very significant for CD-ROM devices; the way a particular drive reacts to a *standard ioctl()* call varies greatly from one device driver to another. To avoid making their drivers totally inconsistent, the writers of Linux CD-ROM drivers generally created new device drivers by understanding, copying, and then changing an existing one. Unfortunately, this practice did not maintain uniform behavior across all the Linux CD-ROM drivers.

This document describes an effort to establish Uniform behavior across all the different CD-ROM device drivers for Linux. This document also defines the various *ioctl()*' *s*, and how the low-level CD-ROM device drivers should implement them. Currently (as of the Linux 2.1.*x* development kernels) several low-level CD-ROM device drivers, including both IDE/ATAPI and SCSI, now use this Uniform interface.

When the CD-ROM was developed, the interface between the CD-ROM drive and the computer was not specified in the standards. As a result, many different CD-ROM interfaces were developed. Some of them had their own proprietary design (Sony, Mitsumi, Panasonic, Philips), other manufacturers adopted an existing electrical interface and changed the functionality (CreativeLabs/SoundBlaster, Teac, Funai) or simply adapted their drives to one or more of the already existing electrical interfaces (Aztech, Sanyo, Funai, Vertos, Longshine, Optics Storage and most of the *NoName* manufacturers). In cases where a new drive really brought its own interface or used its own command set and flow control scheme, either a separate driver had to be written, or an existing driver had to be enhanced. History has delivered us CD-ROM support for many of these different interfaces. Nowadays, almost all new CD-ROM drives are either IDE/ATAPI or SCSI, and it is very unlikely that any manufacturer will create a new interface. Even finding drives for the old proprietary interfaces is getting difficult.

When (in the 1.3.70’ s) I looked at the existing software interface, which was expressed through *cdrom.h*, it appeared to be a rather wild set of commands and data formats[1]. It seemed that many features of the software interface had been added to accommodate the capabilities of a particular drive, in an *ad hoc* manner. More importantly, it appeared that the behavior of the *standard* commands was different for most of the different drivers: e. g., some drivers close the tray if an *open()* call occurs when the tray is open, while others do not. Some drivers lock the door upon opening the device, to prevent an incoherent file system, but others don’ t, to allow software ejection. Undoubtedly, the capabilities of the different drives vary, but even when two drives have the same capability their drivers’ behavior was usually different.

I decided to start a discussion on how to make all the Linux CD-ROM drivers behave more uniformly. I began by contacting the developers of the many CD-ROM drivers found in the Linux kernel. Their reactions encouraged me to write the Uniform CD-ROM Driver which this document is intended to describe. The implementation of the Uniform CD-ROM Driver is in the file *cdrom.c*. This driver is intended to be an additional software layer that sits on top of the low-level device drivers for each CD-ROM drive. By adding this additional layer, it is possible to have all the different CD-ROM devices behave **exactly** the same (insofar as the underlying hardware will allow).

The goal of the Uniform CD-ROM Driver is **not** to alienate driver developers who-have not yet taken steps to support this effort. The goal of Uniform CD-ROM Driver is simply to give people writing application programs for CD-ROM drives **one** Linux CD-ROM interface with consistent behavior for all CD-ROM devices. In addition, this also provides a consistent interface between the low-level device driver code and the Linux kernel. Care is taken that 100% compatibility exists with the data structures and programmer’ s interface defined in *cdrom.h*. This guide was written to help CD-ROM driver developers adapt their code to use the Uniform CD-ROM Driver code defined in *cdrom.c*.

Personally, I think that the most important hardware interfaces are the IDE/ATAPI drives and, of course, the SCSI drives, but as prices of hardware drop continuously, it is also likely that people may have more than one CD-ROM drive, possibly of mixed types. It is important that these drives behave in the same way. In December

[1] I cannot recollect what kernel version I looked at, then, presumably 1.2.13 and 1.3.34 —the latest kernel that I was indirectly involved in.

1994, one of the cheapest CD-ROM drives was a Philips cm206, a double-speed proprietary drive. In the months that I was busy writing a Linux driver for it, proprietary drives became obsolete and IDE/ATAPI drives became the standard. At the time of the last update to this document (November 1997) it is becoming difficult to even **find** anything less than a 16 speed CD-ROM drive, and 24 speed drives are common.

1.2 Standardizing through another software level

At the time this document was conceived, all drivers directly implemented the CD-ROM *ioctl()* calls through their own routines. This led to the danger of different drivers forgetting to do important things like checking that the user was giving the driver valid data. More importantly, this led to the divergence of behavior, which has already been discussed.

For this reason, the Uniform CD-ROM Driver was created to enforce consistent CD-ROM drive behavior, and to provide a common set of services to the various low-level CD-ROM device drivers. The Uniform CD-ROM Driver now provides another software-level, that separates the *ioctl()* and *open()* implementation from the actual hardware implementation. Note that this effort has made few changes which will affect a user’ s application programs. The greatest change involved moving the contents of the various low-level CD-ROM drivers' header files to the kernel’ s cdrom directory. This was done to help ensure that the user is only presented with only one cdrom interface, the interface defined in *cdrom.h*.

CD-ROM drives are specific enough (i. e., different from other block-devices such as floppy or hard disc drives), to define a set of common **CD-ROM device operations**, *<cdrom-device>_dops*. These operations are different from the classical block-device file operations, *<block-device>_fops*.

The routines for the Uniform CD-ROM Driver interface level are implemented in the file *cdrom.c*. In this file, the Uniform CD-ROM Driver interfaces with the kernel as a block device by registering the following general *struct file_operations*:

```
struct file_operations cdrom_fops = {
        NULL,                   /∗ lseek ∗/
        block _read ,           /∗ read—general block-dev read ∗/
        block _write,           /∗ write—general block-dev write ∗/
        NULL,                   /∗ readdir ∗/
        NULL,                   /∗ select ∗/
        cdrom_ioctl,            /∗ ioctl ∗/
        NULL,                   /∗ mmap ∗/
        cdrom_open,             /∗ open ∗/
        cdrom_release,          /∗ release ∗/
        NULL,                   /∗ fsync ∗/
        NULL,                   /∗ fasync ∗/
        NULL                    /∗ revalidate ∗/
};
```

Every active CD-ROM device shares this *struct*. The routines declared above are all implemented in *cdrom.c*, since this file is the place where the behavior of all CD-ROM-devices is defined and standardized. The actual interface to the various types of CD-ROM hardware is still performed by various low-level CD-ROM-device drivers. These routines simply implement certain **capabilities** that are common to all CD-ROM (and really, all removable-media devices).

Registration of a low-level CD-ROM device driver is now done through the general routines in *cdrom.c*, not through the Virtual File System (VFS) any more. The interface implemented in *cdrom.c* is carried out through two general structures that contain information about the capabilities of the driver, and the specific drives on which the driver operates. The structures are:

cdrom_device_ops
: This structure contains information about the low-level driver for a CD-ROM device. This structure is conceptually connected to the major number of the device (although some drivers may have different major numbers, as is the case for the IDE driver).

cdrom_device_info
: This structure contains information about a particular CD-ROM drive, such as its device name, speed, etc. This structure is conceptually connected to the minor number of the device.

Registering a particular CD-ROM drive with the Uniform CD-ROM Driver is done by the low-level device driver though a call to:

```
register_cdrom(struct cdrom_device_info * <device>_info)
```

The device information structure, *<device>_info*, contains all the information needed for the kernel to interface with the low-level CD-ROM device driver. One of the most important entries in this structure is a pointer to the *cdrom_device_ops* structure of the low-level driver.

The device operations structure, *cdrom_device_ops*, contains a list of pointers to the functions which are implemented in the low-level device driver. When *cdrom.c* accesses a CD-ROM device, it does it through the functions in this structure. It is impossible to know all the capabilities of future CD-ROM drives, so it is expected that this list may need to be expanded from time to time as new technologies are developed. For example, CD-R and CD-R/W drives are beginning to become popular, and support will soon need to be added for them. For now, the current *struct* is:

```
struct cdrom_device_ops {
        int (*open)(struct cdrom_device_info *, int)
        void (*release)(struct cdrom_device_info *);
        int (*drive_status)(struct cdrom_device_info *, int);
        unsigned int (*check_events)(struct cdrom_device_info *,
                                     unsigned int, int);
        int (*media_changed)(struct cdrom_device_info *, int);
        int (*tray_move)(struct cdrom_device_info *, int);
        int (*lock_door)(struct cdrom_device_info *, int);
        int (*select_speed)(struct cdrom_device_info *, int);
        int (*select_disc)(struct cdrom_device_info *, int);
        int (*get_last_session) (struct cdrom_device_info *,
                                 struct cdrom_multisession *);
```

(continues on next page)

(continued from previous page)

```
        int (*get_mcn)(struct cdrom_device_info *, struct cdrom_mcn␣
↪*);
        int (*reset)(struct cdrom_device_info *);
        int (*audio_ioctl)(struct cdrom_device_info *,
                           unsigned int, void *);
        const int capability;           /* capability flags */
        int (*generic_packet)(struct cdrom_device_info *,
                              struct packet_command *);
};
```

When a low-level device driver implements one of these capabilities, it should add a function pointer to this *struct*. When a particular function is not implemented, however, this *struct* should contain a NULL instead. The *capability* flags specify the capabilities of the CD-ROM hardware and/or low-level CD-ROM driver when a CD-ROM drive is registered with the Uniform CD-ROM Driver.

Note that most functions have fewer parameters than their *blkdev_fops* counterparts. This is because very little of the information in the structures *inode* and *file* is used. For most drivers, the main parameter is the *struct cdrom_device_info*, from which the major and minor number can be extracted. (Most low-level CD-ROM drivers don' t even look at the major and minor number though, since many of them only support one device.) This will be available through *dev* in *cdrom_device_info* described below.

The drive-specific, minor-like information that is registered with *cdrom.c*, currently contains the following fields:

```
struct cdrom_device_info {
      const struct cdrom_device_ops * ops;    /* device operations␣
↪for this major */
      struct list_head list;                  /* linked list of all␣
↪device_info */
      struct gendisk * disk;                  /* matching block␣
↪layer disk */
      void *  handle;                         /* driver-dependent␣
↪data */

      int mask;                               /* mask of␣
↪capability: disables them */
      int speed;                              /* maximum speed for␣
↪reading data */
      int capacity;                           /* number of discs in␣
↪a jukebox */

      unsigned int options:30;                /* options flags */
      unsigned mc_flags:2;                    /*  media-change␣
↪buffer flags */
      unsigned int vfs_events;                /*  cached events for␣
↪vfs path */
      unsigned int ioctl_events;              /*  cached events for␣
```

(continues on next page)

(continued from previous page)

```
↪ioctl path */
     int use_count;                          /*  number of times␣
↪device is opened */
     char name[20];                          /*  name of the␣
↪device type */

     __u8 sanyo_slot : 2;                    /*  Sanyo 3-CD␣
↪changer support */
     __u8 keeplocked : 1;                    /*  CDROM_LOCKDOOR␣
↪status */
     __u8 reserved : 5;                      /*  not used yet */
     int cdda_method;                        /*  see CDDA_* flags␣
↪*/
     __u8 last_sense;                        /*  saves last sense␣
↪key */
     __u8 media_written;                     /*  dirty flag,␣
↪DVD+RW bookkeeping */
     unsigned short mmc3_profile;            /*  current MMC3␣
↪profile */
     int for_data;                           /*  unknown:TBD */
     int (*exit)(struct cdrom_device_info *);/*  unknown:TBD */
     int mrw_mode_page;                      /*  which MRW mode␣
↪page is in use */
};
```

Using this *struct*, a linked list of the registered minor devices is built, using the *next* field. The device number, the device operations struct and specifications of properties of the drive are stored in this structure.

The *mask* flags can be used to mask out some of the capabilities listed in *ops->capability*, if a specific drive doesn't support a feature of the driver. The value *speed* specifies the maximum head-rate of the drive, measured in units of normal audio speed (176kB/sec raw data or 150kB/sec file system data). The parameters are declared *const* because they describe properties of the drive, which don't change after registration.

A few registers contain variables local to the CD-ROM drive. The flags *options* are used to specify how the general CD-ROM routines should behave. These various flags registers should provide enough flexibility to adapt to the different users' wishes (and **not** the *arbitrary* wishes of the author of the low-level device driver, as is the case in the old scheme). The register *mc_flags* is used to buffer the information from *media_changed()* to two separate queues. Other data that is specific to a minor drive, can be accessed through *handle*, which can point to a data structure specific to the low-level driver. The fields *use_count*, *next*, *options* and *mc_flags* need not be initialized.

The intermediate software layer that *cdrom.c* forms will perform some additional bookkeeping. The use count of the device (the number of processes that have the device opened) is registered in *use_count*. The function *cdrom_ioctl()* will verify the appropriate user-memory regions for read and write, and in case a location on the CD is transferred, it will *sanitize* the format by making requests to the low-level drivers in a standard format, and translating all formats between the user-software and low level drivers. This relieves much of the drivers' memory checking and format checking and translation. Also, the necessary structures will be declared on the program stack.

The implementation of the functions should be as defined in the following sections. Two functions **must** be implemented, namely *open()* and *release()*. Other functions may be omitted, their corresponding capability flags will be cleared upon registration. Generally, a function returns zero on success and negative on error. A function call should return only after the command has completed, but of course waiting for the device should not use processor time.

```
int open(struct cdrom_device_info *cdi, int purpose)
```

Open() should try to open the device for a specific *purpose*, which can be either:

- Open for reading data, as done by *mount()* (2), or the user commands *dd* or *cat*.
- Open for *ioctl* commands, as done by audio-CD playing programs.

Notice that any strategic code (closing tray upon *open()*, etc.) is done by the calling routine in *cdrom.c*, so the low-level routine should only be concerned with proper initialization, such as spinning up the disc, etc.

```
void release(struct cdrom_device_info *cdi)
```

Device-specific actions should be taken such as spinning down the device. However, strategic actions such as ejection of the tray, or unlocking the door, should be left over to the general routine *cdrom_release()*. This is the only function returning type *void*.

```
int drive_status(struct cdrom_device_info *cdi, int slot_nr)
```

The function *drive_status*, if implemented, should provide information on the status of the drive (not the status of the disc, which may or may not be in the drive). If the drive is not a changer, *slot_nr* should be ignored. In *cdrom.h* the possibilities are listed:

```
CDS_NO_INFO             /* no information available */
CDS_NO_DISC             /* no disc is inserted, tray is closed */
CDS_TRAY_OPEN           /* tray is opened */
CDS_DRIVE_NOT_READY     /* something is wrong, tray is moving? */
CDS_DISC_OK             /* a disc is loaded and everything is fine␣
↪*/
```

```
int tray_move(struct cdrom_device_info *cdi, int position)
```

This function, if implemented, should control the tray movement. (No other function should control this.) The parameter *position* controls the desired direction of movement:

- 0 Close tray
- 1 Open tray

This function returns 0 upon success, and a non-zero value upon error. Note that if the tray is already in the desired position, no action need be taken, and the return value should be 0.

```
int lock_door(struct cdrom_device_info *cdi, int lock)
```

This function (and no other code) controls locking of the door, if the drive allows this. The value of *lock* controls the desired locking state:

- 0 Unlock door, manual opening is allowed
- 1 Lock door, tray cannot be ejected manually

This function returns 0 upon success, and a non-zero value upon error. Note that if the door is already in the requested state, no action need be taken, and the return value should be 0.

```
int select_speed(struct cdrom_device_info *cdi, int speed)
```

Some CD-ROM drives are capable of changing their head-speed. There are several reasons for changing the speed of a CD-ROM drive. Badly pressed CD-ROM s may benefit from less-than-maximum head rate. Modern CD-ROM drives can obtain very high head rates (up to *24x* is common). It has been reported that these drives can make reading errors at these high speeds, reducing the speed can prevent data loss in these circumstances. Finally, some of these drives can make an annoyingly loud noise, which a lower speed may reduce.

This function specifies the speed at which data is read or audio is played back. The value of *speed* specifies the head-speed of the drive, measured in units of standard cdrom speed (176kB/sec raw data or 150kB/sec file system data). So to request that a CD-ROM drive operate at 300kB/sec you would call the CDROM_SELECT_SPEED *ioctl* with *speed=2*. The special value *0* means *auto-selection*, i. e., maximum data-rate or real-time audio rate. If the drive doesn't have this *auto-selection* capability, the decision should be made on the current disc loaded and the return value should be positive. A negative return value indicates an error.

```
int select_disc(struct cdrom_device_info *cdi, int number)
```

If the drive can store multiple discs (a juke-box) this function will perform disc selection. It should return the number of the selected disc on success, a negative value on error. Currently, only the ide-cd driver supports this functionality.

```
int get_last_session(struct cdrom_device_info *cdi,
                     struct cdrom_multisession *ms_info)
```

This function should implement the old corresponding *ioctl()*. For device *cdi->dev*, the start of the last session of the current disc should be returned in the pointer argument *ms_info*. Note that routines in *cdrom.c* have sanitized this argument: its requested format will **always** be of the type *CDROM_LBA* (linear block addressing mode), whatever the calling software requested. But sanitization goes even further: the low-level implementation may return the requested information in *CDROM_MSF* format if it wishes so (setting the *ms_info->addr_format* field appropriately, of course) and the routines in *cdrom.c* will make the transformation if necessary. The return value is 0 upon success.

```
int get_mcn(struct cdrom_device_info *cdi,
            struct cdrom_mcn *mcn)
```

Some discs carry a *Media Catalog Number* (MCN), also called *Universal Product Code* (UPC). This number should reflect the number that is generally found in the bar-code on the product. Unfortunately, the few discs that carry such a number on the disc don' t even use the same format. The return argument to this function is a pointer to a pre-declared memory region of type *struct cdrom_mcn*. The MCN is expected as a 13-character string, terminated by a null-character.

```
int reset(struct cdrom_device_info *cdi)
```

This call should perform a hard-reset on the drive (although in circumstances that a hard-reset is necessary, a drive may very well not listen to commands anymore). Preferably, control is returned to the caller only after the drive has finished resetting. If the drive is no longer listening, it may be wise for the underlying low-level cdrom driver to time out.

```
int audio_ioctl(struct cdrom_device_info *cdi,
                unsigned int cmd, void *arg)
```

Some of the CD-ROM-*ioctl()* 's defined in *cdrom.h* can be implemented by the routines described above, and hence the function *cdrom_ioctl* will use those. However, most *ioctl()* 's deal with audio-control. We have decided to leave these to be accessed through a single function, repeating the arguments *cmd* and *arg*. Note that the latter is of type *void*, rather than *unsigned long int*. The routine *cdrom_ioctl()* does do some useful things, though. It sanitizes the address format type to *CDROM_MSF* (Minutes, Seconds, Frames) for all audio calls. It also verifies the memory location of *arg*, and reserves stack-memory for the argument. This makes implementation of the *audio_ioctl()* much simpler than in the old driver scheme. For example, you may look up the function *cm206_audio_ioctl() cm206.c* that should be updated with this documentation.

An unimplemented ioctl should return *-ENOSYS*, but a harmless request (e. g., *CDROMSTART*) may be ignored by returning 0 (success). Other errors should be according to the standards, whatever they are. When an error is returned by the low-level driver, the Uniform CD-ROM Driver tries whenever possible to return the error code to the calling program. (We may decide to sanitize the return value in *cdrom_ioctl()* though, in order to guarantee a uniform interface to the audio-player software.)

```
int dev_ioctl(struct cdrom_device_info *cdi,
              unsigned int cmd, unsigned long arg)
```

Some *ioctl()' s* seem to be specific to certain CD-ROM drives. That is, they are introduced to service some capabilities of certain drives. In fact, there are 6 different *ioctl()' s* for reading data, either in some particular kind of format, or audio data. Not many drives support reading audio tracks as data, I believe this is because of protection of copyrights of artists. Moreover, I think that if audio-tracks are supported, it should be done through the VFS and not via *ioctl()' s*. A problem here could be the fact that audio-frames are 2352 bytes long, so either the audio-file-system should ask for 75264 bytes at once (the least common multiple of 512 and 2352), or the drivers should bend their backs to cope with this incoherence (to which I would be opposed). Furthermore, it is very difficult for the hardware to find the exact frame boundaries, since there are no synchronization headers in audio frames. Once these issues are resolved, this code should be standardized in *cdrom.c*.

Because there are so many *ioctl()*' *s* that seem to be introduced to satisfy certain drivers[2], any non-standard *ioctl()*s are routed through the call *dev_ioctl()*. In principle, *private ioctl()* 's should be numbered after the device' s major number, and not the general CD-ROM *ioctl* number, *0x53*. Currently the non-supported *ioctl()*' *s* are:

> CDROMREADMODE1, CDROMREADMODE2, CDROMREADAUDIO, CDROMREADRAW, CDROMREADCOOKED, CDROMSEEK, CDROMPLAY-BLK and CDROM-READALL

1.2.1 CD-ROM capabilities

Instead of just implementing some *ioctl* calls, the interface in *cdrom.c* supplies the possibility to indicate the **capabilities** of a CD-ROM drive. This can be done by ORing any number of capability-constants that are defined in *cdrom.h* at the registration phase. Currently, the capabilities are any of:

```
CDC_CLOSE_TRAY          /* can close tray by software control */
CDC_OPEN_TRAY           /* can open tray */
CDC_LOCK                /* can lock and unlock the door */
CDC_SELECT_SPEED        /* can select speed, in units of * sim*150 ,
↪kB/s */
CDC_SELECT_DISC         /* drive is juke-box */
CDC_MULTI_SESSION       /* can read sessions *> rm1* */
CDC_MCN                 /* can read Media Catalog Number */
CDC_MEDIA_CHANGED       /* can report if disc has changed */
CDC_PLAY_AUDIO          /* can perform audio-functions (play, pause,
↪ etc) */
CDC_RESET               /* hard reset device */
CDC_IOCTLS              /* driver has non-standard ioctls */
CDC_DRIVE_STATUS        /* driver implements drive status */
```

The capability flag is declared *const*, to prevent drivers from accidentally tampering with the contents. The capability flags actually inform *cdrom.c* of what the driver can do. If the drive found by the driver does not have the capability, is can be masked out by the *cdrom_device_info* variable *mask*. For instance, the SCSI CD-ROM driver has implemented the code for loading and ejecting CD-ROM' s, and hence its corresponding flags in *capability* will be set. But a SCSI CD-ROM drive might be a caddy system, which can' t load the tray, and hence for this drive the *cdrom_device_info* struct will have set the *CDC_CLOSE_TRAY* bit in *mask*.

In the file *cdrom.c* you will encounter many constructions of the type:

[2] Is there software around that actually uses these? I' d be interested!

```
if (cdo->capability & ~cdi->mask & CDC _⟨capability⟩) ...
```

There is no *ioctl* to set the mask…The reason is that I think it is better to control the **behavior** rather than the **capabilities**.

1.2.2 Options

A final flag register controls the **behavior** of the CD-ROM drives, in order to satisfy different users' wishes, hopefully independently of the ideas of the respective author who happened to have made the drive's support available to the Linux community. The current behavior options are:

```
CDO_AUTO_CLOSE  /* try to close tray upon device open() */
CDO_AUTO_EJECT  /* try to open tray on last device close() */
CDO_USE_FFLAGS  /* use file_pointer->f_flags to indicate purpose␣
↪for open() */
CDO_LOCK        /* try to lock door if device is opened */
CDO_CHECK_TYPE  /* ensure disc type is data if opened for data */
```

The initial value of this register is *CDO_AUTO_CLOSE* | *CDO_USE_FFLAGS* | *CDO_LOCK*, reflecting my own view on user interface and software standards. Before you protest, there are two new *ioctl()*' *s* implemented in *cdrom.c*, that allow you to control the behavior by software. These are:

```
CDROM_SET_OPTIONS       /* set options specified in (int)arg */
CDROM_CLEAR_OPTIONS     /* clear options specified in (int)arg */
```

One option needs some more explanation: *CDO_USE_FFLAGS*. In the next new-section we explain what the need for this option is.

A software package *setcd*, available from the Debian distribution and *sunsite.unc.edu*, allows user level control of these flags.

1.3 The need to know the purpose of opening the CD-ROM device

Traditionally, Unix devices can be used in two different *modes*, either by reading/writing to the device file, or by issuing controlling commands to the device, by the device's *ioctl()* call. The problem with CD-ROM drives, is that they can be used for two entirely different purposes. One is to mount removable file systems, CD-ROM's, the other is to play audio CD's. Audio commands are implemented entirely through *ioctl()'s*, presumably because the first implementation (SUN?) has been such. In principle there is nothing wrong with this, but a good control of the *CD player* demands that the device can **always** be opened in order to give the *ioctl* commands, regardless of the state the drive is in.

On the other hand, when used as a removable-media disc drive (what the original purpose of CD-ROM s is) we would like to make sure that the disc drive is ready for operation upon opening the device. In the old scheme, some CD-ROM drivers don't do any integrity checking, resulting in a number of i/o errors reported by the VFS to the kernel when an attempt for mounting a CD-ROM on an empty drive occurs. This is not a particularly elegant way to find out that there is no CD-ROM inserted; it more-or-less looks like the old IBM-PC trying to read an empty floppy drive for a couple of seconds, after which the system complains it can' t read from it. Nowadays we can **sense** the existence of a removable medium in a drive, and we believe we should exploit that fact. An integrity check on opening of the device, that verifies the availability of a CD-ROM and its correct type (data), would be desirable.

These two ways of using a CD-ROM drive, principally for data and secondarily for playing audio discs, have different demands for the behavior of the *open()* call. Audio use simply wants to open the device in order to get a file handle which is needed for issuing *ioctl* commands, while data use wants to open for correct and reliable data transfer. The only way user programs can indicate what their *purpose* of opening the device is, is through the *flags* parameter (see *open(2)*). For CD-ROM devices, these flags aren' t implemented (some drivers implement checking for write-related flags, but this is not strictly necessary if the device file has correct permission flags). Most option flags simply don' t make sense to CD-ROM devices: *O_CREAT*, *O_NOCTTY*, *O_TRUNC*, *O_APPEND*, and *O_SYNC* have no meaning to a CD-ROM.

We therefore propose to use the flag *O_NONBLOCK* to indicate that the device is opened just for issuing *ioctl* commands. Strictly, the meaning of *O_NONBLOCK* is that opening and subsequent calls to the device don' t cause the calling process to wait. We could interpret this as don' t wait until someone has inserted some valid data-CD-ROM. Thus, our proposal of the implementation for the *open()* call for CD-ROM s is:

- If no other flags are set than *O_RDONLY*, the device is opened for data transfer, and the return value will be 0 only upon successful initialization of the transfer. The call may even induce some actions on the CD-ROM, such as closing the tray.
- If the option flag *O_NONBLOCK* is set, opening will always be successful, unless the whole device doesn' t exist. The drive will take no actions whatsoever.

1.3.1 And what about standards?

You might hesitate to accept this proposal as it comes from the Linux community, and not from some standardizing institute. What about SUN, SGI, HP and all those other Unix and hardware vendors? Well, these companies are in the lucky position that they generally control both the hardware and software of their supported products, and are large enough to set their own standard. They do not have to deal with a dozen or more different, competing hardware configurations[3].

[3] Incidentally, I think that SUN' s approach to mounting CD-ROM s is very good in origin: under Solaris a volume-daemon automatically mounts a newly inserted CD-ROM under */cdrom/*<volume-name>**.

In my opinion they should have pushed this further and have **every** CD-ROM on the local area network be mounted at the similar location, i. e., no matter in which particular machine you insert a CD-ROM, it will always appear at the same position in the directory tree, on every system. When I wanted to implement such a user-program for Linux, I came across the differences in behavior of the various drivers, and the need for an *ioctl* informing about media changes.

We believe that using *O_NONBLOCK* to indicate that a device is being opened for *ioctl* commands only can be easily introduced in the Linux community. All the CD-player authors will have to be informed, we can even send in our own patches to the programs. The use of *O_NONBLOCK* has most likely no influence on the behavior of the CD-players on other operating systems than Linux. Finally, a user can always revert to old behavior by a call to *ioctl(file_descriptor, CDROM_CLEAR_OPTIONS, CDO_USE_FFLAGS)*.

1.3.2 The preferred strategy of *open()*

The routines in *cdrom.c* are designed in such a way that run-time configuration of the behavior of CD-ROM devices (of **any** type) can be carried out, by the *CDROM_SET/CLEAR_OPTIONS ioctls*. Thus, various modes of operation can be set:

CDO_AUTO_CLOSE | CDO_USE_FFLAGS | CDO_LOCK
: This is the default setting. (With *CDO_CHECK_TYPE* it will be better, in the future.) If the device is not yet opened by any other process, and if the device is being opened for data (*O_NONBLOCK* is not set) and the tray is found to be open, an attempt to close the tray is made. Then, it is verified that a disc is in the drive and, if *CDO_CHECK_TYPE* is set, that it contains tracks of type *data mode 1*. Only if all tests are passed is the return value zero. The door is locked to prevent file system corruption. If the drive is opened for audio (*O_NONBLOCK* is set), no actions are taken and a value of 0 will be returned.

CDO_AUTO_CLOSE | CDO_AUTO_EJECT | CDO_LOCK
: This mimics the behavior of the current sbpcd-driver. The option flags are ignored, the tray is closed on the first open, if necessary. Similarly, the tray is opened on the last release, i. e., if a CD-ROM is unmounted, it is automatically ejected, such that the user can replace it.

We hope that these option can convince everybody (both driver maintainers and user program developers) to adopt the new CD-ROM driver scheme and option flag interpretation.

1.4 Description of routines in *cdrom.c*

Only a few routines in *cdrom.c* are exported to the drivers. In this new section we will discuss these, as well as the functions that *take over* the CD-ROM interface to the kernel. The header file belonging to *cdrom.c* is called *cdrom.h*. Formerly, some of the contents of this file were placed in the file *ucdrom.h*, but this file has now been merged back into *cdrom.h*.

```
struct file_operations cdrom_fops
```

The contents of this structure were described in *cdrom_api*. A pointer to this structure is assigned to the *fops* field of the *struct gendisk*.

```
int register_cdrom(struct cdrom_device_info *cdi)
```

This function is used in about the same way one registers *cdrom_fops* with the kernel, the device operations and information structures, as described in cdrom_api, should be registered with the Uniform CD-ROM Driver:

```
register_cdrom(&<device>_info);
```

This function returns zero upon success, and non-zero upon failure. The structure *<device>_info* should have a pointer to the driver's *<device>_dops*, as in:

```
struct cdrom_device_info <device>_info = {
        <device>_dops;
        ...
}
```

Note that a driver must have one static structure, *<device>_dops*, while it may have as many structures *<device>_info* as there are minor devices active. *Register_cdrom()* builds a linked list from these.

```
void unregister_cdrom(struct cdrom_device_info *cdi)
```

Unregistering device *cdi* with minor number *MINOR(cdi->dev)* removes the minor device from the list. If it was the last registered minor for the low-level driver, this disconnects the registered device-operation routines from the CD-ROM interface. This function returns zero upon success, and non-zero upon failure.

```
int cdrom_open(struct inode * ip, struct file * fp)
```

This function is not called directly by the low-level drivers, it is listed in the standard *cdrom_fops*. If the VFS opens a file, this function becomes active. A strategy is implemented in this routine, taking care of all capabilities and options that are set in the *cdrom_device_ops* connected to the device. Then, the program flow is transferred to the device_dependent *open()* call.

```
void cdrom_release(struct inode *ip, struct file *fp)
```

This function implements the reverse-logic of *cdrom_open()*, and then calls the device-dependent *release()* routine. When the use-count has reached 0, the allocated buffers are flushed by calls to *sync_dev(dev)* and *invalidate_buffers(dev)*.

```
int cdrom_ioctl(struct inode *ip, struct file *fp,
                unsigned int cmd, unsigned long arg)
```

This function handles all the standard *ioctl* requests for CD-ROM devices in a uniform way. The different calls fall into three categories: *ioctl()*'*s* that can be directly implemented by device operations, ones that are routed through the call *audio_ioctl()*, and the remaining ones, that are presumable device-dependent. Generally, a negative return value indicates an error.

1.4.1 Directly implemented *ioctl()*’ *s*

The following *old* CD-ROM *ioctl()* ‘s are implemented by directly calling device-operations in *cdrom_device_ops*, if implemented and not masked:

CDROMMULTISESSION
Requests the last session on a CD-ROM.

CDROMEJECT
Open tray.

CDROMCLOSETRAY
Close tray.

CDROMEJECT_SW
If *argnot=0*, set behavior to auto-close (close tray on first open) and auto-eject (eject on last release), otherwise set behavior to non-moving on *open()* and *release()* calls.

CDROM_GET_MCN
Get the Media Catalog Number from a CD.

1.4.2 *Ioctl*s routed through *audio_ioctl()*

The following set of *ioctl()*’ *s* are all implemented through a call to the *cdrom_fops* function *audio_ioctl()*. Memory checks and allocation are performed in *cdrom_ioctl()*, and also sanitization of address format (*CDROM_LBA*/*CDROM_MSF*) is done.

CDROMSUBCHNL
Get sub-channel data in argument *arg* of type *struct cdrom_subchnl* *.

CDROMREADTOCHDR
Read Table of Contents header, in *arg* of type *struct cdrom_tochdr* *.

CDROMREADTOCENTRY
Read a Table of Contents entry in *arg* and specified by *arg* of type *struct cdrom_tocentry* *.

CDROMPLAYMSF
Play audio fragment specified in Minute, Second, Frame format, delimited by *arg* of type *struct cdrom_msf* *.

CDROMPLAYTRKIND
Play audio fragment in track-index format delimited by *arg* of type *struct cdrom_ti* *.

CDROMVOLCTRL
Set volume specified by *arg* of type *struct cdrom_volctrl* *.

CDROMVOLREAD
Read volume into by *arg* of type *struct cdrom_volctrl* *.

CDROMSTART
Spin up disc.

CDROMSTOP
Stop playback of audio fragment.

CDROMPAUSE
: Pause playback of audio fragment.

CDROMRESUME
: Resume playing.

1.4.3 New *ioctl()*' *s* in *cdrom.c*

The following *ioctl()*' *s* have been introduced to allow user programs to control the behavior of individual CD-ROM devices. New *ioctl* commands can be identified by the underscores in their names.

CDROM_SET_OPTIONS
: Set options specified by *arg*. Returns the option flag register after modification. Use *arg* = *rm0* for reading the current flags.

CDROM_CLEAR_OPTIONS
: Clear options specified by *arg*. Returns the option flag register after modification.

CDROM_SELECT_SPEED
: Select head-rate speed of disc specified as by *arg* in units of standard cdrom speed (176,kB/sec raw data or 150kB/sec file system data). The value 0 means *auto-select*, i. e., play audio discs at real time and data discs at maximum speed. The value *arg* is checked against the maximum head rate of the drive found in the *cdrom_dops*.

CDROM_SELECT_DISC
: Select disc numbered *arg* from a juke-box.

  First disc is numbered 0. The number *arg* is checked against the maximum number of discs in the juke-box found in the *cdrom_dops*.

CDROM_MEDIA_CHANGED
: Returns 1 if a disc has been changed since the last call. For juke-boxes, an extra argument *arg* specifies the slot for which the information is given. The special value *CDSL_CURRENT* requests that information about the currently selected slot be returned.

CDROM_DRIVE_STATUS
: Returns the status of the drive by a call to *drive_status()*. Return values are defined in *cdrom_drive_status*. Note that this call doesn' t return information on the current playing activity of the drive; this can be polled through an *ioctl* call to *CDROMSUBCHNL*. For juke-boxes, an extra argument *arg* specifies the slot for which (possibly limited) information is given. The special value *CDSL_CURRENT* requests that information about the currently selected slot be returned.

CDROM_DISC_STATUS
: Returns the type of the disc currently in the drive. It should be viewed as a complement to *CDROM_DRIVE_STATUS*. This *ioctl* can provide *some* information about the current disc that is inserted in the drive. This functionality used to be implemented in the low level drivers, but is now carried out entirely in Uniform CD-ROM Driver.

The history of development of the CD' s use as a carrier medium for various digital information has lead to many different disc types. This *ioctl* is useful only in the case that CDs have emph {only one} type of data on them. While this is often the case, it is also very common for CDs to have some tracks with data, and some tracks with audio. Because this is an existing interface, rather than fixing this interface by changing the assumptions it was made under, thereby breaking all user applications that use this function, the Uniform CD-ROM Driver implements this *ioctl* as follows: If the CD in question has audio tracks on it, and it has absolutely no CD-I, XA, or data tracks on it, it will be reported as *CDS_AUDIO*. If it has both audio and data tracks, it will return *CDS_MIXED*. If there are no audio tracks on the disc, and if the CD in question has any CD-I tracks on it, it will be reported as *CDS_XA_2_2*. Failing that, if the CD in question has any XA tracks on it, it will be reported as *CDS_XA_2_1*. Finally, if the CD in question has any data tracks on it, it will be reported as a data CD (*CDS_DATA_1*).

This *ioctl* can return:

```
CDS_NO_INFO     /* no information available */
CDS_NO_DISC     /* no disc is inserted, or tray is opened */
CDS_AUDIO       /* Audio disc (2352 audio bytes/frame) */
CDS_DATA_1      /* data disc, mode 1 (2048 user bytes/frame) */
CDS_XA_2_1      /* mixed data (XA), mode 2, form 1 (2048 user␣
↪bytes) */
CDS_XA_2_2      /* mixed data (XA), mode 2, form 1 (2324 user␣
↪bytes) */
CDS_MIXED       /* mixed audio/data disc */
```

For some information concerning frame layout of the various disc types, see a recent version of *cdrom.h*.

CDROM_CHANGER_NSLOTS
Returns the number of slots in a juke-box.

CDROMRESET
Reset the drive.

CDROM_GET_CAPABILITY
Returns the *capability* flags for the drive. Refer to section *cdrom_capabilities* for more information on these flags.

CDROM_LOCKDOOR
Locks the door of the drive. *arg == 0* unlocks the door, any other value locks it.

CDROM_DEBUG
Turns on debugging info. Only root is allowed to do this. Same semantics as CDROM_LOCKDOOR.

1.4.4 Device dependent *ioctl()*' *s*

Finally, all other *ioctl()*' *s* are passed to the function *dev_ioctl()*, if implemented. No memory allocation or verification is carried out.

1.5 How to update your driver

- Make a backup of your current driver.
- Get hold of the files *cdrom.c* and *cdrom.h*, they should be in the directory tree that came with this documentation.
- Make sure you include *cdrom.h*.
- Change the 3rd argument of *register_blkdev* from *&<your-drive>_fops* to *&cdrom_fops*.
- Just after that line, add the following to register with the Uniform CD-ROM Driver:

  ```
  register_cdrom(&<your-drive>_info);*
  ```

  Similarly, add a call to *unregister_cdrom()* at the appropriate place.
- Copy an example of the device-operations *struct* to your source, e. g., from *cm206.c cm206_dops*, and change all entries to names corresponding to your driver, or names you just happen to like. If your driver doesn' t support a certain function, make the entry *NULL*. At the entry *capability* you should list all capabilities your driver currently supports. If your driver has a capability that is not listed, please send me a message.
- Copy the *cdrom_device_info* declaration from the same example driver, and modify the entries according to your needs. If your driver dynamically determines the capabilities of the hardware, this structure should also be declared dynamically.
- Implement all functions in your *<device>_dops* structure, according to prototypes listed in *cdrom.h*, and specifications given in *cdrom_api*. Most likely you have already implemented the code in a large part, and you will almost certainly need to adapt the prototype and return values.
- Rename your *<device>_ioctl()* function to *audio_ioctl* and change the prototype a little. Remove entries listed in the first part in *cdrom_ioctl*, if your code was OK, these are just calls to the routines you adapted in the previous step.
- You may remove all remaining memory checking code in the *audio_ioctl()* function that deals with audio commands (these are listed in the second part of *cdrom_ioctl*. There is no need for memory allocation either, so most *case*s in the *switch* statement look similar to:

  ```
  case CDROMREADTOCENTRY:
          get_toc_entry\bigl((struct cdrom_tocentry *) arg);
  ```

- All remaining *ioctl* cases must be moved to a separate function, *<device>_ioctl*, the device-dependent *ioctl()*' *s*. Note that memory checking and allocation must be kept in this code!
- Change the prototypes of *<device>_open()* and *<device>_release()*, and remove any strategic code (i. e., tray movement, door locking, etc.).
- Try to recompile the drivers. We advise you to use modules, both for *cdrom.o* and your driver, as debugging is much easier this way.

1.6 Thanks

Thanks to all the people involved. First, Erik Andersen, who has taken over the torch in maintaining *cdrom.c* and integrating much CD-ROM-related code in the 2.1-kernel. Thanks to Scott Snyder and Gerd Knorr, who were the first to implement this interface for SCSI and IDE-CD drivers and added many ideas for extension of the data structures relative to kernel~2.0. Further thanks to Heiko Eißfeldt, Thomas Quinot, Jon Tombs, Ken Pizzini, Eberhard Mönkeberg and Andrew Kroll, the Linux CD-ROM device driver developers who were kind enough to give suggestions and criticisms during the writing. Finally of course, I want to thank Linus Torvalds for making this possible in the first place.

CHAPTER

TWO

IDE-CD DRIVER DOCUMENTATION

Originally by
scott snyder <snyder@fnald0.fnal.gov> (19 May 1996)

Carrying on the torch is
Erik Andersen <andersee@debian.org>

New maintainers (19 Oct 1998)
Jens Axboe <axboe@image.dk>

2.1 1. Introduction

The ide-cd driver should work with all ATAPI ver 1.2 to ATAPI 2.6 compliant CDROM drives which attach to an IDE interface. Note that some CDROM vendors (including Mitsumi, Sony, Creative, Aztech, and Goldstar) have made both ATAPI-compliant drives and drives which use a proprietary interface. If your drive uses one of those proprietary interfaces, this driver will not work with it (but one of the other CDROM drivers probably will). This driver will not work with *ATAPI* drives which attach to the parallel port. In addition, there is at least one drive (CyCDROM CR520ie) which attaches to the IDE port but is not ATAPI; this driver will not work with drives like that either (but see the aztcd driver).

This driver provides the following features:

- Reading from data tracks, and mounting ISO 9660 filesystems.
- Playing audio tracks. Most of the CDROM player programs floating around should work; I usually use Workman.
- Multisession support.
- On drives which support it, reading digital audio data directly from audio tracks. The program cdda2wav can be used for this. Note, however, that only some drives actually support this.
- There is now support for CDROM changers which comply with the ATAPI 2.6 draft standard (such as the NEC CDR-251). This additional functionality includes a function call to query which slot is the currently selected slot, a function call to query which slots contain CDs, etc. A sample program which demonstrates this functionality is appended to the end of this file. The Sanyo 3-disc changer (which does not conform to the standard) is also now supported. Please note the driver refers to the first CD as slot # 0.

2.2 2. Installation

0. The ide-cd relies on the ide disk driver. See Documentation/ide/ide.rst for up-to-date information on the ide driver.
1. Make sure that the ide and ide-cd drivers are compiled into the kernel you're using. When configuring the kernel, in the section entitled "Floppy, IDE, and other block devices", say either *Y* (which will compile the support directly into the kernel) or *M* (to compile support as a module which can be loaded and unloaded) to the options:

   ```
   ATA/ATAPI/MFM/RLL support
   Include IDE/ATAPI CDROM support
   ```

   Depending on what type of IDE interface you have, you may need to specify additional configuration options. See Documentation/ide/ide.rst.
2. You should also ensure that the iso9660 filesystem is either compiled into the kernel or available as a loadable module. You can see if a filesystem is known to the kernel by catting /proc/filesystems.
3. The CDROM drive should be connected to the host on an IDE interface. Each interface on a system is defined by an I/O port address and an IRQ number, the standard assignments being 0x1f0 and 14 for the primary interface and 0x170 and 15 for the secondary interface. Each interface can control up to two devices, where each device can be a hard drive, a CDROM drive, a floppy drive, or a tape drive. The two devices on an interface are called *master* and *slave*; this is usually selectable via a jumper on the drive.

   Linux names these devices as follows. The master and slave devices on the primary IDE interface are called *hda* and *hdb*, respectively. The drives on the secondary interface are called *hdc* and *hdd*. (Interfaces at other locations get other letters in the third position; see Documentation/ide/ide.rst.)

   If you want your CDROM drive to be found automatically by the driver, you should make sure your IDE interface uses either the primary or secondary addresses mentioned above. In addition, if the CDROM drive is the only device on the IDE interface, it should be jumpered as *master*. (If for some reason you cannot configure your system in this manner, you can probably still use the driver. You may have to pass extra configuration information to the kernel when you boot, however. See Documentation/ide/ide.rst for more information.)
4. Boot the system. If the drive is recognized, you should see a message which looks like:

   ```
   hdb: NEC CD-ROM DRIVE:260, ATAPI CDROM drive
   ```

   If you do not see this, see section 5 below.
5. You may want to create a symbolic link /dev/cdrom pointing to the actual device. You can do this with the command:

   ```
   ln -s  /dev/hdX  /dev/cdrom
   ```

where X should be replaced by the letter indicating where your drive is installed.

6. You should be able to see any error messages from the driver with the *dmesg* command.

2.3 3. Basic usage

An ISO 9660 CDROM can be mounted by putting the disc in the drive and typing (as root):

```
mount -t iso9660 /dev/cdrom /mnt/cdrom
```

where it is assumed that /dev/cdrom is a link pointing to the actual device (as described in step 5 of the last section) and /mnt/cdrom is an empty directory. You should now be able to see the contents of the CDROM under the /mnt/cdrom directory. If you want to eject the CDROM, you must first dismount it with a command like:

```
umount /mnt/cdrom
```

Note that audio CDs cannot be mounted.

Some distributions set up /etc/fstab to always try to mount a CDROM filesystem on bootup. It is not required to mount the CDROM in this manner, though, and it may be a nuisance if you change CDROMs often. You should feel free to remove the cdrom line from /etc/fstab and mount CDROMs manually if that suits you better.

Multisession and photocd discs should work with no special handling. The hpcdtoppm package (ftp.gwdg.de:/pub/linux/hpcdtoppm/) may be useful for reading photocds.

To play an audio CD, you should first unmount and remove any data CDROM. Any of the CDROM player programs should then work (workman, workbone, cdplayer, etc.).

On a few drives, you can read digital audio directly using a program such as cdda2wav. The only types of drive which I've heard support this are Sony and Toshiba drives. You will get errors if you try to use this function on a drive which does not support it.

For supported changers, you can use the *cdchange* program (appended to the end of this file) to switch between changer slots. Note that the drive should be unmounted before attempting this. The program takes two arguments: the CDROM device, and the slot number to which you wish to change. If the slot number is -1, the drive is unloaded.

2.4 4. Common problems

This section discusses some common problems encountered when trying to use the driver, and some possible solutions. Note that if you are experiencing problems, you should probably also review Documentation/ide/ide.rst for current information about the underlying IDE support code. Some of these items apply only to earlier versions of the driver, but are mentioned here for completeness.

In most cases, you should probably check with *dmesg* for any errors from the driver.

a. Drive is not detected during booting.

   - Review the configuration instructions above and in Documentation/ide/ide.rst, and check how your hardware is configured.
   - If your drive is the only device on an IDE interface, it should be jumpered as master, if at all possible.
   - If your IDE interface is not at the standard addresses of 0x170 or 0x1f0, you' ll need to explicitly inform the driver using a lilo option. See Documentation/ide/ide.rst. (This feature was added around kernel version 1.3.30.)
   - If the autoprobing is not finding your drive, you can tell the driver to assume that one exists by using a lilo option of the form *hdX=cdrom*, where X is the drive letter corresponding to where your drive is installed. Note that if you do this and you see a boot message like:

     ```
     hdX: ATAPI cdrom (?)
     ```

     this does _not_ mean that the driver has successfully detected the drive; rather, it means that the driver has not detected a drive, but is assuming there' s one there anyway because you told it so. If you actually try to do I/O to a drive defined at a nonexistent or nonresponding I/O address, you' ll probably get errors with a status value of 0xff.
   - Some IDE adapters require a nonstandard initialization sequence before they' ll function properly. (If this is the case, there will often be a separate MS-DOS driver just for the controller.) IDE interfaces on sound cards often fall into this category.

     Support for some interfaces needing extra initialization is provided in later 1.3.x kernels. You may need to turn on additional kernel configuration options to get them to work; see Documentation/ide/ide.rst.

     Even if support is not available for your interface, you may be able to get it to work with the following procedure. First boot MS-DOS and load the appropriate drivers. Then warm-boot linux (i.e., without powering off). If this works, it can be automated by running loadlin from the MS-DOS autoexec.

b. Timeout/IRQ errors.

- If you always get timeout errors, interrupts from the drive are probably not making it to the host.

- IRQ problems may also be indicated by the message *IRQ probe failed (<n>)* while booting. If <n> is zero, that means that the system did not see an interrupt from the drive when it was expecting one (on any feasible IRQ). If <n> is negative, that means the system saw interrupts on multiple IRQ lines, when it was expecting to receive just one from the CDROM drive.
- Double-check your hardware configuration to make sure that the IRQ number of your IDE interface matches what the driver expects. (The usual assignments are 14 for the primary (0x1f0) interface and 15 for the secondary (0x170) interface.) Also be sure that you don’t have some other hardware which might be conflicting with the IRQ you’re using. Also check the BIOS setup for your system; some have the ability to disable individual IRQ levels, and I’ve had one report of a system which was shipped with IRQ 15 disabled by default.
- Note that many MS-DOS CDROM drivers will still function even if there are hardware problems with the interrupt setup; they apparently don’t use interrupts.
- If you own a Pioneer DR-A24X, you _will_ get nasty error messages on boot such as "irq timeout: status=0x50 { DriveReady SeekComplete }" The Pioneer DR-A24X CDROM drives are fairly popular these days. Unfortunately, these drives seem to become very confused when we perform the standard Linux ATA disk drive probe. If you own one of these drives, you can bypass the ATA probing which confuses these CDROM drives, by adding *append="hdX=noprobe hdX=cdrom"* to your lilo.conf file and running lilo (again where X is the drive letter corresponding to where your drive is installed.)

c. System hangups.

- If the system locks up when you try to access the CDROM, the most likely cause is that you have a buggy IDE adapter which doesn’t properly handle simultaneous transactions on multiple interfaces. The most notorious of these is the CMD640B chip. This problem can be worked around by specifying the *serialize* option when booting. Recent kernels should be able to detect the need for this automatically in most cases, but the detection is not foolproof. See Documentation/ide/ide.rst for more information about the *serialize* option and the CMD640B.
- Note that many MS-DOS CDROM drivers will work with such buggy hardware, apparently because they never attempt to overlap CDROM operations with other disk activity.

d. Can’t mount a CDROM.

- If you get errors from mount, it may help to check *dmesg* to see if there are any more specific errors from the driver or from the filesystem.
- Make sure there’s a CDROM loaded in the drive, and that’s it’s an ISO 9660 disc. You can’t mount an audio CD.
- With the CDROM in the drive and unmounted, try something like:

  ```
  cat /dev/cdrom | od | more
  ```

  If you see a dump, then the drive and driver are probably working OK, and the problem is at the filesystem level (i.e., the CDROM is not ISO 9660 or has errors in the filesystem structure).

- If you see *not a block device* errors, check that the definitions of the device special files are correct. They should be as follows:

```
brw-rw----   1 root     disk       3,   0 Nov 11 18:48 /dev/hda
brw-rw----   1 root     disk       3,  64 Nov 11 18:48 /dev/hdb
brw-rw----   1 root     disk      22,   0 Nov 11 18:48 /dev/hdc
brw-rw----   1 root     disk      22,  64 Nov 11 18:48 /dev/hdd
```

  Some early Slackware releases had these defined incorrectly. If these are wrong, you can remake them by running the script scripts/MAKEDEV.ide. (You may have to make it executable with chmod first.)

  If you have a /dev/cdrom symbolic link, check that it is pointing to the correct device file.

  If you hear people talking of the devices *hd1a* and *hd1b*, these were old names for what are now called hdc and hdd. Those names should be considered obsolete.

- If mount is complaining that the iso9660 filesystem is not available, but you know it is (check /proc/filesystems), you probably need a newer version of mount. Early versions would not always give meaningful error messages.

e. Directory listings are unpredictably truncated, and *dmesg* shows *buffer botch* error messages from the driver.

- There was a bug in the version of the driver in 1.2.x kernels which could cause this. It was fixed in 1.3.0. If you can't upgrade, you can probably work around the problem by specifying a blocksize of 2048 when mounting. (Note that you won' t be able to directly execute binaries off the CDROM in that case.)

  If you see this in kernels later than 1.3.0, please report it as a bug.

f. Data corruption.

- Random data corruption was occasionally observed with the Hitachi CDR-7730 CDROM. If you experience data corruption, using "hdx=slow" as a command line parameter may work around the problem, at the expense of low system performance.

2.5 5. cdchange.c

```
/*
 * cdchange.c  [-v]  <device>  [<slot>]
 *
 * This loads a CDROM from a specified slot in a changer, and␣
↪displays
 * information about the changer status.  The drive should be␣
↪unmounted before
 * using this program.
 *
```

(continues on next page)

(continued from previous page)

```
 * Changer information is displayed if either the -v flag is␣
↪specified
 * or no slot was specified.
 *
 * Based on code originally from Gerhard Zuber <zuber@berlin.snafu.
↪de>.
 * Changer status information, and rewrite for the new Uniform␣
↪CDROM driver
 * interface by Erik Andersen <andersee@debian.org>.
 */

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/cdrom.h>


int
main (int argc, char **argv)
{
      char *program;
      char *device;
      int fd;           /* file descriptor for CD-ROM device */
      int status;       /* return status for system calls */
      int verbose = 0;
      int slot=-1, x_slot;
      int total_slots_available;

      program = argv[0];

      ++argv;
      --argc;

      if (argc < 1 || argc > 3) {
              fprintf (stderr, "usage: %s [-v] <device> [<slot>]\n",
                       program);
              fprintf (stderr, "       Slots are numbered 1 -- n.\n
↪");
              exit (1);
      }

     if (strcmp (argv[0], "-v") == 0) {
              verbose = 1;
              ++argv;
              --argc;
```

(continues on next page)

(continued from previous page)

```
        }

        device = argv[0];

        if (argc == 2)
                slot = atoi (argv[1]) - 1;

        /* open device */
        fd = open(device, O_RDONLY | O_NONBLOCK);
        if (fd < 0) {
                fprintf (stderr, "%s: open failed for `%s`: %s\n",
                         program, device, strerror (errno));
                exit (1);
        }

        /* Check CD player status */
        total_slots_available = ioctl (fd, CDROM_CHANGER_NSLOTS);
        if (total_slots_available <= 1 ) {
                fprintf (stderr, "%s: Device `%s` is not an ATAPI "
                        "compliant CD changer.\n", program, device);
                exit (1);
        }

        if (slot >= 0) {
                if (slot >= total_slots_available) {
                        fprintf (stderr, "Bad slot number.  "
                                 "Should be 1 -- %d.\n",
                                 total_slots_available);
                        exit (1);
                }

                /* load */
                slot=ioctl (fd, CDROM_SELECT_DISC, slot);
                if (slot<0) {
                        fflush(stdout);
                                perror ("CDROM_SELECT_DISC ");
                        exit(1);
                }
        }

        if (slot < 0 || verbose) {

                status=ioctl (fd, CDROM_SELECT_DISC, CDSL_CURRENT);
                if (status<0) {
                        fflush(stdout);
                        perror (" CDROM_SELECT_DISC");
                        exit(1);
                }
                slot=status;
```

(continues on next page)

(continued from previous page)

```
                printf ("Current slot: %d\n", slot+1);
                printf ("Total slots available: %d\n",
                        total_slots_available);

                printf ("Drive status: ");
                status = ioctl (fd, CDROM_DRIVE_STATUS, CDSL_CURRENT);
                if (status<0) {
                  perror(" CDROM_DRIVE_STATUS");
                } else switch(status) {
                case CDS_DISC_OK:
                        printf ("Ready.\n");
                        break;
                case CDS_TRAY_OPEN:
                        printf ("Tray Open.\n");
                        break;
                case CDS_DRIVE_NOT_READY:
                        printf ("Drive Not Ready.\n");
                        break;
                default:
                        printf ("This Should not happen!\n");
                        break;
                }

                for (x_slot=0; x_slot<total_slots_available; x_
↪slot++) {
                        printf ("Slot %2d: ", x_slot+1);
                        status = ioctl (fd, CDROM_DRIVE_STATUS, x_
↪slot);
                        if (status<0) {
                             perror(" CDROM_DRIVE_STATUS");
                        } else switch(status) {
                        case CDS_DISC_OK:
                                printf ("Disc present.");
                                break;
                        case CDS_NO_DISC:
                                printf ("Empty slot.");
                                break;
                        case CDS_TRAY_OPEN:
                                printf ("CD-ROM tray open.\n");
                                break;
                        case CDS_DRIVE_NOT_READY:
                                printf ("CD-ROM drive not ready.\n");
                                break;
                        case CDS_NO_INFO:
                                printf ("No Information available.");
                                break;
                        default:
                                printf ("This Should not happen!\n");
```

(continues on next page)

(continued from previous page)

```
                             break;
                     }
               if (slot == x_slot) {
               status = ioctl (fd, CDROM_DISC_STATUS);
               if (status<0) {
                     perror(" CDROM_DISC_STATUS");
               }
               switch (status) {
                     case CDS_AUDIO:
                             printf ("\tAudio disc.\t");
                             break;
                     case CDS_DATA_1:
                     case CDS_DATA_2:
                             printf ("\tData disc type %d.\t",␣
↪status-CDS_DATA_1+1);
                             break;
                     case CDS_XA_2_1:
                     case CDS_XA_2_2:
                             printf ("\tXA data disc type %d.\t",␣
↪status-CDS_XA_2_1+1);
                             break;
                     default:
                             printf ("\tUnknown disc type 0x%x!\t",
↪ status);
                             break;
                     }
                     }
                     status = ioctl (fd, CDROM_MEDIA_CHANGED, x_
↪slot);
                     if (status<0) {
                             perror(" CDROM_MEDIA_CHANGED");
                     }
                     switch (status) {
                     case 1:
                             printf ("Changed.\n");
                             break;
                     default:
                             printf ("\n");
                             break;
                     }
             }
     }

     /* close device */
     status = close (fd);
     if (status != 0) {
             fprintf (stderr, "%s: close failed for `%s`: %s\n",
                      program, device, strerror (errno));
             exit (1);
```

(continues on next page)

(continued from previous page)

```
        }

        exit (0);
}
```

CHAPTER

THREE

PACKET WRITING

3.1 Getting started quick

- Select packet support in the block device section and UDF support in the file system section.
- Compile and install kernel and modules, reboot.
- You need the udftools package (pktsetup, mkudffs, cdrwtool). Download from http://sourceforge.net/projects/linux-udf/
- Grab a new CD-RW disc and format it (assuming CD-RW is hdc, substitute as appropriate):

  ```
  # cdrwtool -d /dev/hdc -q
  ```

- Setup your writer:

  ```
  # pktsetup dev_name /dev/hdc
  ```

- Now you can mount /dev/pktcdvd/dev_name and copy files to it. Enjoy:

  ```
  # mount /dev/pktcdvd/dev_name /cdrom -t udf -o rw,noatime
  ```

3.2 Packet writing for DVD-RW media

DVD-RW discs can be written to much like CD-RW discs if they are in the so called "restricted overwrite" mode. To put a disc in restricted overwrite mode, run:

```
# dvd+rw-format /dev/hdc
```

You can then use the disc the same way you would use a CD-RW disc:

```
# pktsetup dev_name /dev/hdc
# mount /dev/pktcdvd/dev_name /cdrom -t udf -o rw,noatime
```

3.3 Packet writing for DVD+RW media

According to the DVD+RW specification, a drive supporting DVD+RW discs shall implement "true random writes with 2KB granularity", which means that it should be possible to put any filesystem with a block size >= 2KB on such a disc. For example, it should be possible to do:

```
# dvd+rw-format /dev/hdc   (only needed if the disc has never
                            been formatted)
# mkudffs /dev/hdc
# mount /dev/hdc /cdrom -t udf -o rw,noatime
```

However, some drives don't follow the specification and expect the host to perform aligned writes at 32KB boundaries. Other drives do follow the specification, but suffer bad performance problems if the writes are not 32KB aligned.

Both problems can be solved by using the pktcdvd driver, which always generates aligned writes:

```
# dvd+rw-format /dev/hdc
# pktsetup dev_name /dev/hdc
# mkudffs /dev/pktcdvd/dev_name
# mount /dev/pktcdvd/dev_name /cdrom -t udf -o rw,noatime
```

3.4 Packet writing for DVD-RAM media

DVD-RAM discs are random writable, so using the pktcdvd driver is not necessary. However, using the pktcdvd driver can improve performance in the same way it does for DVD+RW media.

3.5 Notes

- CD-RW media can usually not be overwritten more than about 1000 times, so to avoid unnecessary wear on the media, you should always use the noatime mount option.
- Defect management (ie automatic remapping of bad sectors) has not been implemented yet, so you are likely to get at least some filesystem corruption if the disc wears out.
- Since the pktcdvd driver makes the disc appear as a regular block device with a 2KB block size, you can put any filesystem you like on the disc. For example, run:

  ```
  # /sbin/mke2fs /dev/pktcdvd/dev_name
  ```

  to create an ext2 filesystem on the disc.

3.6 Using the pktcdvd sysfs interface

Since Linux 2.6.20, the pktcdvd module has a sysfs interface and can be controlled by it. For example the "pktcdvd"tool uses this interface. (see http://tom.ist-im-web.de/download/pktcdvd)

"pktcdvd" works similar to "pktsetup" , e.g.:

```
# pktcdvd -a dev_name /dev/hdc
# mkudffs /dev/pktcdvd/dev_name
# mount -t udf -o rw,noatime /dev/pktcdvd/dev_name /dvdram
# cp files /dvdram
# umount /dvdram
# pktcdvd -r dev_name
```

For a description of the sysfs interface look into the file:

Documentation/ABI/testing/sysfs-class-pktcdvd

3.7 Using the pktcdvd debugfs interface

To read pktcdvd device infos in human readable form, do:

```
# cat /sys/kernel/debug/pktcdvd/pktcdvd[0-7]/info
```

For a description of the debugfs interface look into the file:

Documentation/ABI/testing/debugfs-pktcdvd

3.8 Links

See http://fy.chalmers.se/~appro/linux/DVD+RW/ for more information about DVD writing.