```
\documentclass{article}
\def\version{$Id: cdrom-standard.tex,v 1.9 1997/12/28 15:42:49 david Exp $}
\newcommand{\newsection}[1]{\newpage\section{#1}}

\evensidemargin=0pt
\oddsidemargin=0pt
\topmargin=-\headheight \advance\topmargin by -\headsep
\textwidth=15.99cm \textheight=24.62cm % normal A4, 1'' margin

\def\linux{{\sc Linux}}
\def\cdrom{{\sc cd-rom}}
\def\UCD{{\sc Uniform cd-rom Driver}}
\def\cdromc{{\tt {cdrom.c}}}
\def\cdromh{{\tt {cdrom.h}}}
\def\fo{\sl}                    % foreign words
\def\ie{{\fo i.e.}}
\def\eg{{\fo e.g.}}

\everymath{\it} \everydisplay{\it}
\catcode`\_=\active \def_{\_\penalty100 }
\catcode`\<=\active \def<#1>{{\langle\hbox{\rm#1}\rangle}}

\begin{document}
\title{A \linux\ \cdrom\ standard}
\author{David van Leeuwen\\{\normalsize\tt david@ElseWare.cistron.nl}
\\{\footnotesize updated by Erik Andersen {\tt(andersee@debian.org)}}
\\{\footnotesize updated by Jens Axboe {\tt(axboe@image.dk)}}}
\date{12 March 1999}

\maketitle

\newsection{Introduction}

\linux\ is probably the Unix-like operating system that supports
the widest variety of hardware devices. The reasons for this are
presumably
\begin{itemize}
\item
  The large list of hardware devices available for the many platforms
  that \linux\ now supports (\ie, i386-PCs, Sparc Suns, etc.)
\item
  The open design of the operating system, such that anybody can write a
  driver for \linux.
\item
  There is plenty of source code around as examples of how to write a driver.
\end{itemize}
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 \cdrom\
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\ \cdrom\
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\ \cdrom\ drivers.

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

When the \cdrom\ was developed, the interface between the \cdrom\ drive and the computer was not specified in the standards. As a result, many different \cdrom\ 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 \cdrom\ support for many of these different interfaces. Nowadays, almost all new \cdrom\ 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 \cdromh, it appeared to be a rather wild set of commands and data formats.\footnote{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.} It seemed that many features of the software interface had been added to accommodate the capabilities of a particular drive, in an {\fo ad hoc\/} manner. More importantly, it appeared that the behavior of the `standard' commands was different for most of the different drivers: \eg, 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\ \cdrom\ drivers behave more uniformly. I began by contacting the developers of the many \cdrom\ drivers found in the \linux\ kernel. Their reactions encouraged me to write the \UCD\ which this document is intended to describe. The implementation of the \UCD\ is in the file \cdromc. This driver is intended to be an additional software layer that sits on top of the low-level device drivers for each \cdrom\ drive. By adding this additional layer, it is possible to have all the different \cdrom\ devices behave {\em exactly\/} the same (insofar as the underlying hardware will allow).

The goal of the \UCD\ is {\em not\/} to alienate driver developers who have not yet taken steps to support this effort. The goal of \UCD\ is

simply to give people writing application programs for \cdrom\ drives {\em one\/} \linux\ \cdrom\ interface with consistent behavior for all \cdrom\ 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 \cdromh. This guide was written to help \cdrom\ driver developers adapt their code to use the \UCD\ code defined in \cdromc.

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 \cdrom\ drive, possibly of mixed types. It is important that these drives behave in the same way. In December 1994, one of the cheapest \cdrom\ 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 {\em find} anything less than a 16 speed \cdrom\ drive, and 24 speed drives are common.

\newsection{Standardizing through another software level}
\label{cdrom.c}

At the time this document was conceived, all drivers directly implemented the \cdrom\ $ioctl()$ 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 \UCD\ was created to enforce consistent \cdrom\ drive behavior, and to provide a common set of services to the various low-level \cdrom\ device drivers. The \UCD\ 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 \cdrom\ 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 \cdromh.

\cdrom\ drives are specific enough (\ie, different from other block-devices such as floppy or hard disc drives), to define a set of common {\em \cdrom\ device operations}, $<cdrom-device>_dops$. These operations are different from the classical block-device file operations, $<block-device>_fops$.

The routines for the \UCD\ interface level are implemented in the file \cdromc. In this file, the \UCD\ interfaces with the kernel as a block device by registering the following general $struct\ file_operations$:
$$
\halign{$#$\ \hfil&$#$\ \hfil&$/*$ \rm# $*/$\hfil\cr
struct& file_operations\ cdrom_fops = \{\hidewidth\cr
        &NULL,   & lseek \cr
        &block_read,   & read---general block-dev read \cr

```
        &block_write,           & write---general block-dev write \cr
        &NULL,                  & readdir \cr
        &NULL,                  & select \cr
        &cdrom_ioctl,           & ioctl \cr
        &NULL,                  & mmap \cr
        &cdrom_open,            & open \cr
        &cdrom_release,         & release \cr
        &NULL,                  & fsync \cr
        &NULL,                  & fasync \cr
        &cdrom_media_changed,   & media change \cr
        &NULL                   & revalidate \cr
\};\cr
}
$$
```

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

Registration of a low-level \cdrom\ device driver is now done through the general routines in \cdromc, not through the Virtual File System (VFS) any more. The interface implemented in \cdromc\ 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:

\begin{description}
\item[$cdrom_device_ops$]
  This structure contains information about the low-level driver for a \cdrom\ 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).
\item[$cdrom_device_info$]
  This structure contains information about a particular \cdrom\ drive, such as its device name, speed, etc. This structure is conceptually connected to the minor number of the device.
\end{description}

Registering a particular \cdrom\ drive with the \UCD\ 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 \cdrom\ 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 \cdromc\ accesses a \cdrom\ device, it does it through the functions in this structure. It is impossible to know all

```
the capabilities of future \cdrom\ 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:
$$
\halign{$#$\ \hfil&$#$\ \hfil&\hbox to 10em{$#$\hss}&
  $/*$ \rm# $*/$\hfil\cr
struct& cdrom_device_ops\ \{ \hidewidth\cr
  &int& (* open)(struct\ cdrom_device_info *, int)\cr
  &void& (* release)(struct\ cdrom_device_info *);\cr
  &int& (* drive_status)(struct\ cdrom_device_info *, int);\cr
  &int& (* media_changed)(struct\ cdrom_device_info *, int);\cr
  &int& (* tray_move)(struct\ cdrom_device_info *, int);\cr
  &int& (* lock_door)(struct\ cdrom_device_info *, int);\cr
  &int& (* select_speed)(struct\ cdrom_device_info *, int);\cr
  &int& (* select_disc)(struct\ cdrom_device_info *, int);\cr
  &int& (* get_last_session) (struct\ cdrom_device_info *,
        struct\ cdrom_multisession *{});\cr
  &int& (* get_mcn)(struct\ cdrom_device_info *, struct\ cdrom_mcn *{});\cr
  &int& (* reset)(struct\ cdrom_device_info *);\cr
  &int& (* audio_ioctl)(struct\ cdrom_device_info *, unsigned\ int,
        void *{});\cr
  &int& (* dev_ioctl)(struct\ cdrom_device_info *, unsigned\ int,
        unsigned\ long);\cr
\noalign{\medskip}
  &const\ int& capability;& capability flags \cr
  &int& n_minors;& number of active minor devices \cr
\};\cr
}
$$
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
\cdrom\ hardware and/or low-level \cdrom\ driver when a \cdrom\ drive
is registered with the \UCD. The value $n_minors$ should be a positive
value indicating the number of minor devices that are supported by
the low-level device driver, normally~1. Although these two variables
are `informative' rather than `operational,' they are included in
$cdrom_device_ops$ because they describe the capability of the {\em
driver\/} rather than the {\em drive}. Nomenclature has always been
difficult in computer programming.

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
\cdrom\ 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
\cdromc, currently contains the following fields:
$$
```

```
\halign{$#$\ \hfil&$#$\ \hfil&\hbox to 10em{$#$\hss}&
  $/*$ \rm# $*/$\hfil\cr
struct& cdrom_device_info\ \{ \hidewidth\cr
  & struct\ cdrom_device_ops *& ops;& device operations for this major\cr
  & struct\ cdrom_device_info *& next;& next device_info for this major\cr
  & void *&  handle;& driver-dependent data\cr
\noalign{\medskip}
  & kdev_t&  dev;& device number (incorporates minor)\cr
  & int& mask;& mask of capability: disables them \cr
  & int& speed;& maximum speed for reading data \cr
  & int& capacity;& number of discs in a jukebox \cr
\noalign{\medskip}
  &int& options : 30;& options flags \cr
  &unsigned& mc_flags : 2;& media-change buffer flags \cr
  & int& use_count;& number of times device is opened\cr
  & char& name[20];& name of the device type\cr
\}\cr
}$$
```

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 $ops\to capability$, 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 (176\,kB/sec raw data or 150\,kB/sec file system data). The value n_discs should reflect the number of discs the drive can hold simultaneously, if it is designed as a juke-box, or otherwise~1. 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 \cdrom\ drive. The flags $options$ are used to specify how the general \cdrom\ routines should behave. These various flags registers should provide enough flexibility to adapt to the different users' wishes (and {\em 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 \cdromc\ 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 {\em 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.

\subsection{$Int\ open(struct\ cdrom_device_info * cdi, int\ purpose)$}

$Open()$ should try to open the device for a specific $purpose$, which
can be either:
\begin{itemize}
\item[0] Open for reading data, as done by {\tt {mount()}} (2), or the
user commands {\tt {dd}} or {\tt {cat}}.
\item[1] Open for $ioctl$ commands, as done by audio-CD playing
programs.
\end{itemize}
Notice that any strategic code (closing tray upon $open()$, etc.)\ is
done by the calling routine in \cdromc, so the low-level routine
should only be concerned with proper initialization, such as spinning
up the disc, etc. % and device-use count

\subsection{$Void\ release(struct\ cdrom_device_info * cdi)$}

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 $void$.

\subsection{$Int\ drive_status(struct\ cdrom_device_info * cdi, int\ slot_nr)$}
\label{drive status}

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 \cdromh\ the possibilities are listed:
$$
\halign{$#$\ \hfil&$/*$ \rm# $*/$\hfil\cr
CDS_NO_INFO& no information available\cr
CDS_NO_DISC& no disc is inserted, tray is closed\cr
CDS_TRAY_OPEN& tray is opened\cr
CDS_DRIVE_NOT_READY& something is wrong, tray is moving?\cr
CDS_DISC_OK& a disc is loaded and everything is fine\cr
}
$$

\subsection{$Int\ media_changed(struct\ cdrom_device_info * cdi, int\ disc_nr)$}

This function is very similar to the original function in $struct\
file_operations$. It returns 1 if the medium of the device $cdi\to
dev$ has changed since the last call, and 0 otherwise. The parameter
$disc_nr$ identifies a specific slot in a juke-box, it should be
ignored for single-disc drives. Note that by `re-routing' this

function through $cdrom_media_changed()$, we can implement separate queues for the VFS and a new $ioctl()$ function that can report device changes to software (\eg, an auto-mounting daemon).

\subsection{$Int\ tray_move(struct\ cdrom_device_info * cdi, int\ position)$}

This function, if implemented, should control the tray movement. (No other function should control this.) The parameter $position$ controls the desired direction of movement:
\begin{itemize}
\item[0] Close tray
\item[1] Open tray
\end{itemize}
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.

\subsection{$Int\ lock_door(struct\ cdrom_device_info * cdi, int\ lock)$}

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:
\begin{itemize}
\item[0] Unlock door, manual opening is allowed
\item[1] Lock door, tray cannot be ejected manually
\end{itemize}
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.

\subsection{$Int\ select_speed(struct\ cdrom_device_info * cdi, int\ speed)$}

Some \cdrom\ drives are capable of changing their head-speed. There are several reasons for changing the speed of a \cdrom\ drive. Badly pressed \cdrom s may benefit from less-than-maximum head rate. Modern \cdrom\ drives can obtain very high head rates (up to $24\times$ 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. %Finally, %although the audio-low-pass filters probably aren't designed for it, %more than real-time playback of audio might be used for high-speed %copying of audio tracks.

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 (176\,kB/sec raw data or 150\,kB/sec file system data). So to request that a \cdrom\ drive operate at 300\,kB/sec you would call the CDROM_SELECT_SPEED $ioctl$ with $speed=2$. The special value `0' means `auto-selection', \ie, 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.

\subsection{$Int\ select_disc(struct\ cdrom_device_info * cdi, int\ number)$}

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.

\subsection{$Int\ get_last_session(struct\ cdrom_device_info * cdi, struct\ cdrom_multisession * ms_info)$}

This function should implement the old corresponding $ioctl()$. For device $cdi\to dev$, the start of the last session of the current disc should be returned in the pointer argument ms_info. Note that routines in \cdromc\ have sanitized this argument: its requested format will {\em 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 $ms_info\rightarrow addr_format$ field appropriately, of course) and the routines in \cdromc\ will make the transformation if necessary. The return value is 0 upon success.

\subsection{$Int\ get_mcn(struct\ cdrom_device_info * cdi, struct\ cdrom_mcn * mcn)$}

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.

\subsection{$Int\ reset(struct\ cdrom_device_info * cdi)$}

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.

\subsection{$Int\ audio_ioctl(struct\ cdrom_device_info * cdi, unsigned\ int\ cmd, void * arg)$}

Some of the \cdrom-$ioctl$s defined in \cdromh\ 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

$cm206_audio_ioctl()$ in {\tt {cm206.c}} that should be updated with this documentation.

An unimplemented ioctl should return $-ENOSYS$, but a harmless request (\eg, $CDROMSTART$) 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 \UCD\ 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.)

\subsection{$Int\ dev_ioctl(struct\ cdrom_device_info * cdi, unsigned\ int\ cmd, unsigned\ long\ arg)$}

Some $ioctl$s seem to be specific to certain \cdrom\ 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 \cdromc.

Because there are so many $ioctl$s that seem to be introduced to satisfy certain drivers,\footnote{Is there software around that actually uses these? I'd be interested!} 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 \cdrom\ $ioctl$ number, {\tt {0x53}}. Currently the non-supported $ioctl$s are: {\it CDROMREADMODE1, CDROMREADMODE2, CDROMREADAUDIO, CDROMREADRAW, CDROMREADCOOKED, CDROMSEEK, CDROMPLAY\-BLK and CDROM\-READALL}.

\subsection{\cdrom\ capabilities}
\label{capability}

Instead of just implementing some $ioctl$ calls, the interface in \cdromc\ supplies the possibility to indicate the {\em capabilities\/} of a \cdrom\ drive. This can be done by ORing any number of capability-constants that are defined in \cdromh\ at the registration phase. Currently, the capabilities are any of:
$$
\halign{$#$\ \hfil&$/*$ \rm# $*/$\hfil\cr
CDC_CLOSE_TRAY& can close tray by software control\cr
CDC_OPEN_TRAY& can open tray\cr
CDC_LOCK& can lock and unlock the door\cr
CDC_SELECT_SPEED& can select speed, in units of $\sim$150\,kB/s\cr
CDC_SELECT_DISC& drive is juke-box\cr

CDC_MULTI_SESSION& can read sessions $>\rm1$\cr
CDC_MCN& can read Media Catalog Number\cr
CDC_MEDIA_CHANGED& can report if disc has changed\cr
CDC_PLAY_AUDIO& can perform audio-functions (play, pause, etc)\cr
CDC_RESET& hard reset device\cr
CDC_IOCTLS& driver has non-standard ioctls\cr
CDC_DRIVE_STATUS& driver implements drive status\cr
}
$$
The capability flag is declared $const$, to prevent drivers from
accidentally tampering with the contents. The capability fags actually
inform \cdromc\ 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 \cdrom\
driver has implemented the code for loading and ejecting \cdrom's, and
hence its corresponding flags in $capability$ will be set. But a SCSI
\cdrom\ 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 \cdromc\ you will encounter many constructions of the type
$$\it
if\ (cdo\rightarrow capability \mathrel\& \mathord{\sim} cdi\rightarrow mask
   \mathrel{\&} CDC_<capability>) \ldots
$$
There is no $ioctl$ to set the mask\dots The reason is that
I think it is better to control the {\em behavior\/} rather than the
{\em capabilities}.

\subsection{Options}

A final flag register controls the {\em behavior\/} of the \cdrom\
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:
$$
\halign{$#$\ \hfil&$/*$ \rm# $*/$\hfil\cr
CDO_AUTO_CLOSE& try to close tray upon device $open()$\cr
CDO_AUTO_EJECT& try to open tray on last device $close()$\cr
CDO_USE_FFLAGS& use $file_pointer\rightarrow f_flags$ to indicate
 purpose for $open()$\cr
CDO_LOCK& try to lock door if device is opened\cr
CDO_CHECK_TYPE& ensure disc type is data if opened for data\cr
}
$$

The initial value of this register is $CDO_AUTO_CLOSE \mathrel|
CDO_USE_FFLAGS \mathrel| CDO_LOCK$, reflecting my own view on user
interface and software standards. Before you protest, there are two
new $ioctl$s implemented in \cdromc, that allow you to control the
behavior by software. These are:
$$
\halign{$#$\ \hfil&$/*$ \rm# $*/$\hfil\cr
CDROM_SET_OPTIONS& set options specified in $(int)\ arg$\cr
CDROM_CLEAR_OPTIONS& clear options specified in $(int)\ arg$\cr

}
$$
One option needs some more explanation: CDO_USE_FFLAGS. In the next newsection we explain what the need for this option is.

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

\newsection{The need to know the purpose of opening the \cdrom\ 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 \cdrom\ drives, is that they can be used for two entirely different purposes. One is to mount removable file systems, \cdrom 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 {\em 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 \cdrom 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 \cdrom\ 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 \cdrom\ on an empty drive occurs. This is not a particularly elegant way to find out that there is no \cdrom\ 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 {\em 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 \cdrom\ and its correct type (data), would be desirable.

These two ways of using a \cdrom\ 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 {\em purpose\/} of opening the device is, is through the $flags$ parameter (see {\tt {open(2)}}). For \cdrom\ 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 \cdrom\ devices: O_CREAT, O_NOCTTY, O_TRUNC, O_APPEND, and O_SYNC have no meaning to a \cdrom.

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-\cdrom.'' Thus, our proposal of the
implementation for the $open()$ call for \cdrom s is:
\begin{itemize}
\item 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 \cdrom, such as closing the tray.
\item 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.
\end{itemize}

\subsection{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.\footnote{Incidentally, I think that SUN's approach to
mounting \cdrom s is very good in origin: under Solaris a
volume-daemon automatically mounts a newly inserted \cdrom\ under {\tt
{/cdrom/$<volume-name>$/}}. In my opinion they should have pushed this
further and have {\em every\/} \cdrom\ on the local area network be
mounted at the similar location, \ie, no matter in which particular
machine you insert a \cdrom, 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 $ioctl(file_descriptor, CDROM_CLEAR_OPTIONS,
CDO_USE_FFLAGS)$.

\subsection{The preferred strategy of $open()$}

The routines in \cdromc\ are designed in such a way that run-time
configuration of the behavior of \cdrom\ devices (of {\em any\/} type)
can be carried out, by the $CDROM_SET/CLEAR_OPTIONS$ $ioctls$. Thus, various
modes of operation can be set:
\begin{description}
\item[$CDO_AUTO_CLOSE \mathrel| CDO_USE_FFLAGS \mathrel| CDO_LOCK$] 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.
\item[$CDO_AUTO_CLOSE \mathrel| CDO_AUTO_EJECT \mathrel| CDO_LOCK$] 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, \ie, if a \cdrom\ is unmounted, it is automatically ejected, such that the user can replace it.
\end{description}
We hope that these option can convince everybody (both driver maintainers and user program developers) to adopt the new \cdrom\ driver scheme and option flag interpretation.

\newsection{Description of routines in \cdromc}

Only a few routines in \cdromc\ are exported to the drivers. In this new section we will discuss these, as well as the functions that `take over' the \cdrom\ interface to the kernel. The header file belonging to \cdromc\ is called \cdromh. Formerly, some of the contents of this file were placed in the file {\tt {ucdrom.h}}, but this file has now been merged back into \cdromh.

\subsection{$Struct\ file_operations\ cdrom_fops$}

The contents of this structure were described in section~\ref{cdrom.c}. A pointer to this structure is assigned to the $fops$ field of the $struct gendisk$.

\subsection{$Int\ register_cdrom(struct\ cdrom_device_info\ * cdi)$}

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 section~\ref{cdrom.c}, should be registered with the \UCD:
$$
register_cdrom(\&<device>_info));
$$
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
$$
\vbox{\halign{&$#$\hfil\cr
struct\ &cdrom_device_info\ <device>_info = \{\cr
& <device>_dops;\cr
&\ldots\cr
\}\cr
}}$$
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.

\subsection{$Void\ unregister_cdrom(struct\ cdrom_device_info * cdi)$}

Unregistering device cdi with minor number $MINOR(cdi\to dev)$ 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 \cdrom\ interface. This function returns zero upon success, and non-zero upon failure.

\subsection{$Int\ cdrom_open(struct\ inode * ip, struct\ file * fp)$}

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.

\subsection{$Void\ cdrom_release(struct\ inode *ip, struct\ file *fp)$}

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 $invalidate_buffers(dev)$.

\subsection{$Int\ cdrom_ioctl(struct\ inode *ip, struct\ file *fp, unsigned\ int\ cmd, unsigned\ long\ arg)$}
\label{cdrom-ioctl}

This function handles all the standard $ioctl$ requests for \cdrom\ 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.

\subsubsection{Directly implemented $ioctl$s}
\label{ioctl-direct}

The following `old' \cdrom-$ioctl$s are implemented by directly calling device-operations in $cdrom_device_ops$, if implemented and not masked:
\begin{description}
\item[CDROMMULTISESSION] Requests the last session on a \cdrom.
\item[CDROMEJECT] Open tray.
\item[CDROMCLOSETRAY] Close tray.
\item[CDROMEJECT_SW] If $arg\not=0$, 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.
\item[CDROM_GET_MCN] Get the Media Catalog Number from a CD.
\end{description}

\subsubsection{$Ioctl$s routed through $audio_ioctl()$}
\label{ioctl-audio}

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.
\begin{description}

\item[CDROMSUBCHNL] Get sub-channel data in argument arg of type $struct\ cdrom_subchnl *{}$.
\item[CDROMREADTOCHDR] Read Table of Contents header, in arg of type $struct\ cdrom_tochdr *{}$.
\item[CDROMREADTOCENTRY] Read a Table of Contents entry in arg and specified by arg of type $struct\ cdrom_tocentry *{}$.
\item[CDROMPLAYMSF] Play audio fragment specified in Minute, Second, Frame format, delimited by arg of type $struct\ cdrom_msf *{}$.
\item[CDROMPLAYTRKIND] Play audio fragment in track-index format delimited by arg of type $struct\ \penalty-1000 cdrom_ti *{}$.
\item[CDROMVOLCTRL] Set volume specified by arg of type $struct\ cdrom_volctrl *{}$.
\item[CDROMVOLREAD] Read volume into by arg of type $struct\ cdrom_volctrl *{}$.
\item[CDROMSTART] Spin up disc.
\item[CDROMSTOP] Stop playback of audio fragment.
\item[CDROMPAUSE] Pause playback of audio fragment.
\item[CDROMRESUME] Resume playing.
\end{description}

\subsubsection{New $ioctl$s in \cdromc}

The following $ioctl$s have been introduced to allow user programs to control the behavior of individual \cdrom\ devices. New $ioctl$ commands can be identified by the underscores in their names.
\begin{description}
\item[CDROM_SET_OPTIONS] Set options specified by arg. Returns the option flag register after modification. Use $arg = \rm0$ for reading the current flags.
\item[CDROM_CLEAR_OPTIONS] Clear options specified by arg. Returns the option flag register after modification.
\item[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 150\,kB/sec file system data). The value 0 means `auto-select', \ie, 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 $cdrom_dops$.
\item[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 $cdrom_dops$.
\item[CDROM_MEDIA_CHANGED] Returns 1 if a disc has been changed since the last call. Note that calls to $cdrom_media_changed$ by the VFS are treated by an independent queue, so both mechanisms will detect a media change once. 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.
\item[CDROM_DRIVE_STATUS] Returns the status of the drive by a call to $drive_status()$. Return values are defined in section~\ref{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.
\item[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 \emph {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 \UCD.

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 \UCD\ 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:
$$
\halign{$#$\ \hfil&$/*$ \rm# $*/$\hfil\cr
CDS_NO_INFO& no information available\cr
CDS_NO_DISC& no disc is inserted, or tray is opened\cr
CDS_AUDIO& Audio disc (2352 audio bytes/frame)\cr
CDS_DATA_1& data disc, mode 1 (2048 user bytes/frame)\cr
CDS_XA_2_1& mixed data (XA), mode 2, form 1 (2048 user bytes)\cr
CDS_XA_2_2& mixed data (XA), mode 2, form 1 (2324 user bytes)\cr
CDS_MIXED& mixed audio/data disc\cr
}
$$
For some information concerning frame layout of the various disc types, see a recent version of \cdromh.

\item[CDROM_CHANGER_NSLOTS] Returns the number of slots in a juke-box.
\item[CDROMRESET] Reset the drive.
\item[CDROM_GET_CAPABILITY] Returns the $capability$ flags for the drive. Refer to section \ref{capability} for more information on these flags.
\item[CDROM_LOCKDOOR] Locks the door of the drive. $arg == \rm0$ unlocks the door, any other value locks it.
\item[CDROM_DEBUG] Turns on debugging info. Only root is allowed to do this. Same semantics as CDROM_LOCKDOOR.
\end{description}

\subsubsection{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.

```
\newsection{How to update your driver}

\begin{enumerate}
\item Make a backup of your current driver.
\item Get hold of the files \cdromc\ and \cdromh, they should be in
  the directory tree that came with this documentation.
\item Make sure you include \cdromh.
\item Change the 3rd argument of $register_blkdev$ from
$\&<your-drive>_fops$ to $\&cdrom_fops$.
\item Just after that line, add the following to register with the \UCD:
  $$register_cdrom(\&<your-drive>_info);$$
  Similarly, add a call to $unregister_cdrom()$ at the appropriate place.
\item Copy an example of the device-operations $struct$ to your
  source, \eg, from {\tt {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.
\item 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.
\item Implement all functions in your $<device>_dops$ structure,
  according to prototypes listed in \cdromh, and specifications given
  in section~\ref{cdrom.c}. 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.
\item Rename your $<device>_ioctl()$ function to $audio_ioctl$ and
  change the prototype a little. Remove entries listed in the first
  part in section~\ref{cdrom-ioctl}, if your code was OK, these are
  just calls to the routines you adapted in the previous step.
\item 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 section~\ref{cdrom-ioctl}). There is no
  need for memory allocation either, so most $case$s in the $switch$
  statement look similar to:
  $$
  case\ CDROMREADTOCENTRY\colon get_toc_entry\bigl((struct\
  cdrom_tocentry *{})\ arg\bigr);
  $$
\item 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!
\item Change the prototypes of $<device>_open()$ and
  $<device>_release()$, and remove any strategic code (\ie, tray
  movement, door locking, etc.).
\item Try to recompile the drivers. We advise you to use modules, both
  for {\tt {cdrom.o}} and your driver, as debugging is much easier this
  way.
\end{enumerate}

\newsection{Thanks}

Thanks to all the people involved.  First, Erik Andersen, who has
taken over the torch in maintaining \cdromc\ and integrating much
```

```
\cdrom-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 Ei{\sz}feldt,
Thomas Quinot, Jon Tombs, Ken Pizzini, Eberhard M\"onkeberg and Andrew
Kroll, the \linux\ \cdrom\ 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.

\vfill
$ \version\ $
\eject
\end{document}
```