```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook XML V4.1.2//EN"
        "http://www.oasis-open.org/docbook/xml/4.1.2/docbookx.dtd" []>

<book id="Linux-USB-API">
 <bookinfo>
  <title>The Linux-USB Host Side API</title>

  <legalnotice>
   <para>
     This documentation is free software; you can redistribute
     it and/or modify it under the terms of the GNU General Public
     License as published by the Free Software Foundation; either
     version 2 of the License, or (at your option) any later
     version.
   </para>

   <para>
     This program is distributed in the hope that it will be
     useful, but WITHOUT ANY WARRANTY; without even the implied
     warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
     See the GNU General Public License for more details.
   </para>

   <para>
     You should have received a copy of the GNU General Public
     License along with this program; if not, write to the Free
     Software Foundation, Inc., 59 Temple Place, Suite 330, Boston,
     MA 02111-1307 USA
   </para>

   <para>
     For more details see the file COPYING in the source
     distribution of Linux.
   </para>
  </legalnotice>
 </bookinfo>

<toc></toc>

<chapter id="intro">
    <title>Introduction to USB on Linux</title>

    <para>A Universal Serial Bus (USB) is used to connect a host,
    such as a PC or workstation, to a number of peripheral
    devices.  USB uses a tree structure, with the host as the
    root (the system's master), hubs as interior nodes, and
    peripherals as leaves (and slaves).
    Modern PCs support several such trees of USB devices, usually
    one USB 2.0 tree (480 Mbit/sec each) with
    a few USB 1.1 trees (12 Mbit/sec each) that are used when you
    connect a USB 1.1 device directly to the machine's "root hub".
    </para>

    <para>That master/slave asymmetry was designed-in for a number of
    reasons, one being ease of use.  It is not physically possible to
    assemble (legal) USB cables incorrectly:  all upstream "to the host"
    connectors are the rectangular type (matching the sockets on
    root hubs), and all downstream connectors are the squarish type
    (or they are built into the peripheral).
    Also, the host software doesn't need to deal with distributed
    auto-configuration since the pre-designated master node manages all that.
    And finally, at the electrical level, bus protocol overhead is reduced by
    eliminating arbitration and moving scheduling into the host software.
    </para>

    <para>USB 1.0 was announced in January 1996 and was revised
    as USB 1.1 (with improvements in hub specification and
    support for interrupt-out transfers) in September 1998.
    USB 2.0 was released in April 2000, adding high-speed
    transfers and transaction-translating hubs (used for USB 1.1
    and 1.0 backward compatibility).
    </para>

    <para>Kernel developers added USB support to Linux early in the 2.2 kernel
    series, shortly before 2.3 development forked.  Updates from 2.3 were
    regularly folded back into 2.2 releases, which improved reliability and
    brought <filename>/sbin/hotplug</filename> support as well more drivers.
    Such improvements were continued in the 2.5 kernel series, where they added
    USB 2.0 support, improved performance, and made the host controller drivers
    (HCDs) more consistent.  They also simplified the API (to make bugs less
    likely) and added internal "kerneldoc" documentation.
    </para>

    <para>Linux can run inside USB devices as well as on
    the hosts that control the devices.
    But USB device drivers running inside those peripherals
    don't do the same things as the ones running inside hosts,
    so they've been given a different name:
    <emphasis>gadget drivers</emphasis>.
    This document does not cover gadget drivers.
    </para>

    </chapter>

<chapter id="host">
    <title>USB Host-Side API Model</title>

    <para>Host-side drivers for USB devices talk to the "usbcore" APIs.
    There are two.  One is intended for
    <emphasis>general-purpose</emphasis> drivers (exposed through
    driver frameworks), and the other is for drivers that are
    <emphasis>part of the core</emphasis>.
    Such core drivers include the <emphasis>hub</emphasis> driver
    (which manages trees of USB devices) and several different kinds
    of <emphasis>host controller drivers</emphasis>,
    which control individual busses.
    </para>

    <para>The device model seen by USB drivers is relatively complex.
    </para>
```

```
<itemizedlist>

    <listitem><para>USB supports four kinds of data transfers
    (control, bulk, interrupt, and isochronous).  Two of them (control
    and bulk) use bandwidth as it's available,
    while the other two (interrupt and isochronous)
    are scheduled to provide guaranteed bandwidth.
    </para></listitem>

    <listitem><para>The device description model includes one or more
    "configurations" per device, only one of which is active at a time.
    Devices that are capable of high-speed operation must also support
    full-speed configurations, along with a way to ask about the
    "other speed" configurations which might be used.
    </para></listitem>

    <listitem><para>Configurations have one or more "interfaces", each
    of which may have "alternate settings".  Interfaces may be
    standardized by USB "Class" specifications, or may be specific to
    a vendor or device.</para>

    <para>USB device drivers actually bind to interfaces, not devices.
    Think of them as "interface drivers", though you
    may not see many devices where the distinction is important.
    <emphasis>Most USB devices are simple, with only one configuration,
    one interface, and one alternate setting.</emphasis>
    </para></listitem>

    <listitem><para>Interfaces have one or more "endpoints", each of
    which supports one type and direction of data transfer such as
    "bulk out" or "interrupt in".  The entire configuration may have
    up to sixteen endpoints in each direction, allocated as needed
    among all the interfaces.
    </para></listitem>

    <listitem><para>Data transfer on USB is packetized; each endpoint
    has a maximum packet size.
    Drivers must often be aware of conventions such as flagging the end
    of bulk transfers using "short" (including zero length) packets.
    </para></listitem>

    <listitem><para>The Linux USB API supports synchronous calls for
    control and bulk messages.
    It also supports asynchnous calls for all kinds of data transfer,
    using request structures called "URBs" (USB Request Blocks).
    </para></listitem>

</itemizedlist>

<para>Accordingly, the USB Core API exposed to device drivers
covers quite a lot of territory.  You'll probably need to consult
the USB 2.0 specification, available online from www.usb.org at
no cost, as well as class or device specifications.
</para>

<para>The only host-side drivers that actually touch hardware
```

```
    (reading/writing registers, handling IRQs, and so on) are the HCDs.
    In theory, all HCDs provide the same functionality through the same
    API.  In practice, that's becoming more true on the 2.5 kernels,
    but there are still differences that crop up especially with
    fault handling.  Different controllers don't necessarily report
    the same aspects of failures, and recovery from faults (including
    software-induced ones like unlinking an URB) isn't yet fully
    consistent.
    Device driver authors should make a point of doing disconnect
    testing (while the device is active) with each different host
    controller driver, to make sure drivers don't have bugs of
    their own as well as to make sure they aren't relying on some
    HCD-specific behavior.
    (You will need external USB 1.1 and/or
    USB 2.0 hubs to perform all those tests.)
    </para>

    </chapter>

<chapter id="types"><title>USB-Standard Types</title>

    <para>In <filename>&lt;linux/usb/ch9.h&gt;</filename> you will find
    the USB data types defined in chapter 9 of the USB specification.
    These data types are used throughout USB, and in APIs including
    this host side API, gadget APIs, and usbfs.
    </para>

!Iinclude/linux/usb/ch9.h

    </chapter>

<chapter id="hostside"><title>Host-Side Data Types and Macros</title>

    <para>The host side API exposes several layers to drivers, some of
    which are more necessary than others.
    These support lifecycle models for host side drivers
    and devices, and support passing buffers through usbcore to
    some HCD that performs the I/O for the device driver.
    </para>


!Iinclude/linux/usb.h

    </chapter>

    <chapter id="usbcore"><title>USB Core APIs</title>

    <para>There are two basic I/O models in the USB API.
    The most elemental one is asynchronous:  drivers submit requests
    in the form of an URB, and the URB's completion callback
    handle the next step.
    All USB transfer types support that model, although there
    are special cases for control URBs (which always have setup
    and status stages, but may not have a data stage) and
    isochronous URBs (which allow large packets and include
    per-packet fault reports).
    Built on top of that is synchronous API support, where a
    driver calls a routine that allocates one or more URBs,
    submits them, and waits until they complete.
    There are synchronous wrappers for single-buffer control
    and bulk transfers (which are awkward to use in some
    driver disconnect scenarios), and for scatterlist based
    streaming i/o (bulk or interrupt).
    </para>

    <para>USB drivers need to provide buffers that can be
    used for DMA, although they don't necessarily need to
    provide the DMA mapping themselves.
    There are APIs to use used when allocating DMA buffers,
    which can prevent use of bounce buffers on some systems.
    In some cases, drivers may be able to rely on 64bit DMA
    to eliminate another kind of bounce buffer.
    </para>

!Edrivers/usb/core/urb.c
!Edrivers/usb/core/message.c
!Edrivers/usb/core/file.c
!Edrivers/usb/core/driver.c
!Edrivers/usb/core/usb.c
!Edrivers/usb/core/hub.c
    </chapter>

    <chapter id="hcd"><title>Host Controller APIs</title>

    <para>These APIs are only for use by host controller drivers,
    most of which implement standard register interfaces such as
    EHCI, OHCI, or UHCI.
    UHCI was one of the first interfaces, designed by Intel and
    also used by VIA; it doesn't do much in hardware.
    OHCI was designed later, to have the hardware do more work
    (bigger transfers, tracking protocol state, and so on).
    EHCI was designed with USB 2.0; its design has features that
    resemble OHCI (hardware does much more work) as well as
    UHCI (some parts of ISO support, TD list processing).
    </para>

    <para>There are host controllers other than the "big three",
    although most PCI based controllers (and a few non-PCI based
    ones) use one of those interfaces.
    Not all host controllers use DMA; some use PIO, and there
    is also a simulator.
    </para>

    <para>The same basic APIs are available to drivers for all
    those controllers.
    For historical reasons they are in two layers:
    <structname>struct usb_bus</structname> is a rather thin
    layer that became available in the 2.2 kernels, while
    <structname>struct usb_hcd</structname> is a more featureful
    layer (available in later 2.4 kernels and in 2.5) that
    lets HCDs share common code, to shrink driver size
    and significantly reduce hcd-specific behaviors.
    </para>

!Edrivers/usb/core/hcd.c
!Edrivers/usb/core/hcd-pci.c
!Idrivers/usb/core/buffer.c
    </chapter>

    <chapter id="usbfs">
        <title>The USB Filesystem (usbfs)</title>

        <para>This chapter presents the Linux <emphasis>usbfs</emphasis>.
        You may prefer to avoid writing new kernel code for your
        USB driver; that's the problem that usbfs set out to solve.
        User mode device drivers are usually packaged as applications
        or libraries, and may use usbfs through some programming library
        that wraps it.  Such libraries include
        <ulink url="http://libusb.sourceforge.net">libusb</ulink>
        for C/C++, and
        <ulink url="http://jUSB.sourceforge.net">jUSB</ulink> for Java.
        </para>

        <note><title>Unfinished</title>
            <para>This particular documentation is incomplete,
            especially with respect to the asynchronous mode.
            As of kernel 2.5.66 the code and this (new) documentation
            need to be cross-reviewed.
            </para>
            </note>

        <para>Configure usbfs into Linux kernels by enabling the
        <emphasis>USB filesystem</emphasis> option (CONFIG_USB_DEVICEFS),
        and you get basic support for user mode USB device drivers.
        Until relatively recently it was often (confusingly) called
        <emphasis>usbdevfs</emphasis> although it wasn't solving what
        <emphasis>devfs</emphasis> was.
        Every USB device will appear in usbfs, regardless of whether or
        not it has a kernel driver.
        </para>

        <sect1 id="usbfs-files">
            <title>What files are in "usbfs"?</title>

            <para>Conventionally mounted at
            <filename>/proc/bus/usb</filename>, usbfs
            features include:
            <itemizedlist>
                <listitem><para><filename>/proc/bus/usb/devices</filename>
                    ... a text file
                    showing each of the USB devices on known to the kernel,
                    and their configuration descriptors.
                    You can also poll() this to learn about new devices.
                    </para></listitem>
                <listitem><para><filename>/proc/bus/usb/BBB/DDD</filename>
                    ... magic files
                    exposing the each device's configuration descriptors, and
                    supporting a series of ioctls for making device requests,
                    including I/O to devices.  (Purely for access by programs.)
                    </para></listitem>
            </itemizedlist>
            </para>

            <para> Each bus is given a number (BBB) based on when it was
            enumerated; within each bus, each device is given a similar
            number (DDD).
            Those BBB/DDD paths are not "stable" identifiers;
            expect them to change even if you always leave the devices
            plugged in to the same hub port.
            <emphasis>Don't even think of saving these in application
            configuration files.</emphasis>
            Stable identifiers are available, for user mode applications
            that want to use them.  HID and networking devices expose
            these stable IDs, so that for example you can be sure that
            you told the right UPS to power down its second server.
            "usbfs" doesn't (yet) expose those IDs.
            </para>

        </sect1>

        <sect1 id="usbfs-fstab">
            <title>Mounting and Access Control</title>

            <para>There are a number of mount options for usbfs, which will
            be of most interest to you if you need to override the default
            access control policy.
            That policy is that only root may read or write device files
            (<filename>/proc/bus/BBB/DDD</filename>) although anyone may read
            the <filename>devices</filename>
            or <filename>drivers</filename> files.
            I/O requests to the device also need the CAP_SYS_RAWIO capability,
            </para>

            <para>The significance of that is that by default, all user mode
            device drivers need super-user privileges.
            You can change modes or ownership in a driver setup
            when the device hotplugs, or maye just start the
            driver right then, as a privileged server (or some activity
            within one).
            That's the most secure approach for multi-user systems,
            but for single user systems ("trusted" by that user)
            it's more convenient just to grant everyone all access
            (using the <emphasis>devmode=0666</emphasis> option)
            so the driver can start whenever it's needed.
            </para>

            <para>The mount options for usbfs, usable in /etc/fstab or
            in command line invocations of <emphasis>mount</emphasis>, are:

            <variablelist>
                <varlistentry>
                    <term><emphasis>busgid</emphasis>=NNNNN</term>
                    <listitem><para>Controls the GID used for the
                    /proc/bus/usb/BBB
                    directories.  (Default: 0)</para></listitem></varlistentry>
                <varlistentry><term><emphasis>busmode</emphasis>=MMM</term>
                    <listitem><para>Controls the file mode used for the
                    /proc/bus/usb/BBB
                    directories.  (Default: 0555)
                    </para></listitem></varlistentry>
                <varlistentry><term><emphasis>busuid</emphasis>=NNNNN</term>
                    <listitem><para>Controls the UID used for the
                    /proc/bus/usb/BBB
                    directories.  (Default: 0)</para></listitem></varlistentry>

                <varlistentry><term><emphasis>devgid</emphasis>=NNNNN</term>
                    <listitem><para>Controls the GID used for the
                    /proc/bus/usb/BBB/DDD
                    files.  (Default: 0)</para></listitem></varlistentry>
                <varlistentry><term><emphasis>devmode</emphasis>=MMM</term>
                    <listitem><para>Controls the file mode used for the
                    /proc/bus/usb/BBB/DDD
                    files.  (Default: 0644)</para></listitem></varlistentry>
                <varlistentry><term><emphasis>devuid</emphasis>=NNNNN</term>
                    <listitem><para>Controls the UID used for the
                    /proc/bus/usb/BBB/DDD
                    files.  (Default: 0)</para></listitem></varlistentry>

                <varlistentry><term><emphasis>listgid</emphasis>=NNNNN</term>
                    <listitem><para>Controls the GID used for the
                    /proc/bus/usb/devices and drivers files.
                    (Default: 0)</para></listitem></varlistentry>
                <varlistentry><term><emphasis>listmode</emphasis>=MMM</term>
                    <listitem><para>Controls the file mode used for the
                    /proc/bus/usb/devices and drivers files.
                    (Default: 0444)</para></listitem></varlistentry>
                <varlistentry><term><emphasis>listuid</emphasis>=NNNNN</term>
                    <listitem><para>Controls the UID used for the
                    /proc/bus/usb/devices and drivers files.
                    (Default: 0)</para></listitem></varlistentry>
            </variablelist>

            </para>

            <para>Note that many Linux distributions hard-wire the mount options
            for usbfs in their init scripts, such as
            <filename>/etc/rc.d/rc.sysinit</filename>,
            rather than making it easy to set this per-system
            policy in <filename>/etc/fstab</filename>.
            </para>

        </sect1>

        <sect1 id="usbfs-devices">
            <title>/proc/bus/usb/devices</title>

            <para>This file is handy for status viewing tools in user
            mode, which can scan the text format and ignore most of it.
            More detailed device status (including class and vendor
            status) is available from device-specific files.
```

```
            For information about the current format of this file,
            see the
            <filename>Documentation/usb/proc_usb_info.txt</filename>
            file in your Linux kernel sources.
            </para>

            <para>This file, in combination with the poll() system call, can
            also be used to detect when devices are added or removed:
<programlisting>int fd;
struct pollfd pfd;

fd = open("/proc/bus/usb/devices", O_RDONLY);
pfd = { fd, POLLIN, 0 };
for (;;) {
        /* The first time through, this call will return immediately. */
        poll(&amp;pfd, 1, -1);

        /* To see what's changed, compare the file's previous and current
           contents or scan the filesystem.  (Scanning is more precise.) */
}</programlisting>
            Note that this behavior is intended to be used for informational
            and debug purposes.  It would be more appropriate to use programs
            such as udev or HAL to initialize a device or start a user-mode
            helper program, for instance.
            </para>
        </sect1>

        <sect1 id="usbfs-bbbddd">
            <title>/proc/bus/usb/BBB/DDD</title>

            <para>Use these files in one of these basic ways:
            </para>

            <para><emphasis>They can be read,</emphasis>
            producing first the device descriptor
            (18 bytes) and then the descriptors for the current configuration.
            See the USB 2.0 spec for details about those binary data formats.
            You'll need to convert most multibyte values from little endian
            format to your native host byte order, although a few of the
            fields in the device descriptor (both of the BCD-encoded fields,
            and the vendor and product IDs) will be byteswapped for you.
            Note that configuration descriptors include descriptors for
            interfaces, altsettings, endpoints, and maybe additional
            class descriptors.
            </para>

            <para><emphasis>Perform USB operations</emphasis> using
            <emphasis>ioctl()</emphasis> requests to make endpoint I/O
            requests (synchronously or asynchronously) or manage
            the device.
            These requests need the CAP_SYS_RAWIO capability,
            as well as filesystem access permissions.
            Only one ioctl request can be made on one of these
            device files at a time.
            This means that if you are synchronously reading an endpoint
            from one thread, you won't be able to write to a different
```

```
    endpoint from another thread until the read completes.
    This works for <emphasis>half duplex</emphasis> protocols,
    but otherwise you'd use asynchronous i/o requests.
    </para>

    </sect1>


<sect1 id="usbfs-lifecycle">
    <title>Life Cycle of User Mode Drivers</title>

    <para>Such a driver first needs to find a device file
    for a device it knows how to handle.
    Maybe it was told about it because a
    <filename>/sbin/hotplug</filename> event handling agent
    chose that driver to handle the new device.
    Or maybe it's an application that scans all the
    /proc/bus/usb device files, and ignores most devices.
    In either case, it should <function>read()</function> all
    the descriptors from the device file,
    and check them against what it knows how to handle.
    It might just reject everything except a particular
    vendor and product ID, or need a more complex policy.
    </para>

    <para>Never assume there will only be one such device
    on the system at a time!
    If your code can't handle more than one device at
    a time, at least detect when there's more than one, and
    have your users choose which device to use.
    </para>

    <para>Once your user mode driver knows what device to use,
    it interacts with it in either of two styles.
    The simple style is to make only control requests; some
    devices don't need more complex interactions than those.
    (An example might be software using vendor-specific control
    requests for some initialization or configuration tasks,
    with a kernel driver for the rest.)
    </para>

    <para>More likely, you need a more complex style driver:
    one using non-control endpoints, reading or writing data
    and claiming exclusive use of an interface.
    <emphasis>Bulk</emphasis> transfers are easiest to use,
    but only their sibling <emphasis>interrupt</emphasis> transfers
    work with low speed devices.
    Both interrupt and <emphasis>isochronous</emphasis> transfers
    offer service guarantees because their bandwidth is reserved.
    Such "periodic" transfers are awkward to use through usbfs,
    unless you're using the asynchronous calls.  However, interrupt
    transfers can also be used in a synchronous "one shot" style.
    </para>

    <para>Your user-mode driver should never need to worry
    about cleaning up request state when the device is
```

```
            disconnected, although it should close its open file
            descriptors as soon as it starts seeing the ENODEV
            errors.
            </para>

            </sect1>

        <sect1 id="usbfs-ioctl"><title>The ioctl() Requests</title>

            <para>To use these ioctls, you need to include the following
            headers in your userspace program:
<programlisting>#include &lt;linux/usb.h&gt;
#include &lt;linux/usbdevice_fs.h&gt;
#include &lt;asm/byteorder.h&gt;</programlisting>
            The standard USB device model requests, from "Chapter 9" of
            the USB 2.0 specification, are automatically included from
            the <filename>&lt;linux/usb/ch9.h&gt;</filename> header.
            </para>

            <para>Unless noted otherwise, the ioctl requests
            described here will
            update the modification time on the usbfs file to which
            they are applied (unless they fail).
            A return of zero indicates success; otherwise, a
            standard USB error code is returned.  (These are
            documented in
            <filename>Documentation/usb/error-codes.txt</filename>
            in your kernel sources.)
            </para>

            <para>Each of these files multiplexes access to several
            I/O streams, one per endpoint.
            Each device has one control endpoint (endpoint zero)
            which supports a limited RPC style RPC access.
            Devices are configured
            by khubd (in the kernel) setting a device-wide
            <emphasis>configuration</emphasis> that affects things
            like power consumption and basic functionality.
            The endpoints are part of USB <emphasis>interfaces</emphasis>,
            which may have <emphasis>altsettings</emphasis>
            affecting things like which endpoints are available.
            Many devices only have a single configuration and interface,
            so drivers for them will ignore configurations and altsettings.
            </para>


            <sect2 id="usbfs-mgmt">
                <title>Management/Status Requests</title>

                <para>A number of usbfs requests don't deal very directly
                with device I/O.
                They mostly relate to device management and status.
                These are all synchronous requests.
                </para>

                <variablelist>
                <varlistentry><term>USBDEVFS_CLAIMINTERFACE</term>
                    <listitem><para>This is used to force usbfs to
                    claim a specific interface,
                    which has not previously been claimed by usbfs or any other
                    kernel driver.
                    The ioctl parameter is an integer holding the number of
                    the interface (bInterfaceNumber from descriptor).
                    </para><para>
                    Note that if your driver doesn't claim an interface
                    before trying to use one of its endpoints, and no
                    other driver has bound to it, then the interface is
                    automatically claimed by usbfs.
                    </para><para>
                    This claim will be released by a RELEASEINTERFACE ioctl,
                    or by closing the file descriptor.
                    File modification time is not updated by this request.
                    </para></listitem></varlistentry>

                <varlistentry><term>USBDEVFS_CONNECTINFO</term>
                    <listitem><para>Says whether the device is lowspeed.
                    The ioctl parameter points to a structure like this:
<programlisting>struct usbdevfs_connectinfo {
        unsigned int   devnum;
        unsigned char  slow;
}; </programlisting>
                    File modification time is not updated by this request.
                    </para><para>
                    <emphasis>You can't tell whether a "not slow"
                    device is connected at high speed (480 MBit/sec)
                    or just full speed (12 MBit/sec).</emphasis>
                    You should know the devnum value already,
                    it's the DDD value of the device file name.
                    </para></listitem></varlistentry>

                <varlistentry><term>USBDEVFS_GETDRIVER</term>
                    <listitem><para>Returns the name of the kernel driver
                    bound to a given interface (a string).  Parameter
                    is a pointer to this structure, which is modified:
<programlisting>struct usbdevfs_getdriver {
        unsigned int  interface;
        char          driver[USBDEVFS_MAXDRIVERNAME + 1];
};</programlisting>
                    File modification time is not updated by this request.
                    </para></listitem></varlistentry>

                <varlistentry><term>USBDEVFS_IOCTL</term>
                    <listitem><para>Passes a request from userspace through
                    to a kernel driver that has an ioctl entry in the
                    <emphasis>struct usb_driver</emphasis> it registered.
<programlisting>struct usbdevfs_ioctl {
        int     ifno;
        int     ioctl_code;
        void    *data;
};
```

```
/* user mode call looks like this.
 * 'request' becomes the driver->ioctl() 'code' parameter.
 * the size of 'param' is encoded in 'request', and that data
 * is copied to or from the driver->ioctl() 'buf' parameter.
 */
static int
usbdev_ioctl (int fd, int ifno, unsigned request, void *param)
{
        struct usbdevfs_ioctl   wrapper;

        wrapper.ifno = ifno;
        wrapper.ioctl_code = request;
        wrapper.data = param;

        return ioctl (fd, USBDEVFS_IOCTL, &amp;wrapper);
} </programlisting>
                    File modification time is not updated by this request.
                    </para><para>
                    This request lets kernel drivers talk to user mode code
                    through filesystem operations even when they don't create
                    a charactor or block special device.
                    It's also been used to do things like ask devices what
                    device special file should be used.
                    Two pre-defined ioctls are used
                    to disconnect and reconnect kernel drivers, so
                    that user mode code can completely manage binding
                    and configuration of devices.
                    </para></listitem></varlistentry>

                <varlistentry><term>USBDEVFS_RELEASEINTERFACE</term>
                    <listitem><para>This is used to release the claim usbfs
                    made on interface, either implicitly or because of a
                    USBDEVFS_CLAIMINTERFACE call, before the file
                    descriptor is closed.
                    The ioctl parameter is an integer holding the number of
                    the interface (bInterfaceNumber from descriptor);
                    File modification time is not updated by this request.
                    </para><warning><para>
                    <emphasis>No security check is made to ensure
                    that the task which made the claim is the one
                    which is releasing it.
                    This means that user mode driver may interfere
                    other ones.  </emphasis>
                    </para></warning></listitem></varlistentry>

                <varlistentry><term>USBDEVFS_RESETEP</term>
                    <listitem><para>Resets the data toggle value for an endpoint
                    (bulk or interrupt) to DATA0.
                    The ioctl parameter is an integer endpoint number
                    (1 to 15, as identified in the endpoint descriptor),
                    with USB_DIR_IN added if the device's endpoint sends
                    data to the host.
                    </para><warning><para>
                    <emphasis>Avoid using this request.
                    It should probably be removed.</emphasis>
                    Using it typically means the device and driver will lose
```

```
                        toggle synchronization.  If you really lost synchronization,
                        you likely need to completely handshake with the device,
                        using a request like CLEAR_HALT
                        or SET_INTERFACE.
                        </para></warning></listitem></varlistentry>

                    </variablelist>

                    </sect2>

                <sect2 id="usbfs-sync">
                    <title>Synchronous I/O Support</title>

                    <para>Synchronous requests involve the kernel blocking
                    until the user mode request completes, either by
                    finishing successfully or by reporting an error.
                    In most cases this is the simplest way to use usbfs,
                    although as noted above it does prevent performing I/O
                    to more than one endpoint at a time.
                    </para>

                    <variablelist>

                    <varlistentry><term>USBDEVFS_BULK</term>
                        <listitem><para>Issues a bulk read or write request to the
                        device.
                        The ioctl parameter is a pointer to this structure:
<programlisting>struct usbdevfs_bulktransfer {
        unsigned int  ep;
        unsigned int  len;
        unsigned int  timeout; /* in milliseconds */
        void          *data;
};</programlisting>
                        </para><para>The "ep" value identifies a
                        bulk endpoint number (1 to 15, as identified in an endpoint
                        descriptor),
                        masked with USB_DIR_IN when referring to an endpoint which
                        sends data to the host from the device.
                        The length of the data buffer is identified by "len";
                        Recent kernels support requests up to about 128KBytes.
                        <emphasis>FIXME say how read length is returned,
                        and how short reads are handled.</emphasis>.
                        </para></listitem></varlistentry>

                    <varlistentry><term>USBDEVFS_CLEAR_HALT</term>
                        <listitem><para>Clears endpoint halt (stall) and
                        resets the endpoint toggle.  This is only
                        meaningful for bulk or interrupt endpoints.
                        The ioctl parameter is an integer endpoint number
                        (1 to 15, as identified in an endpoint descriptor),
                        masked with USB_DIR_IN when referring to an endpoint which
                        sends data to the host from the device.
                        </para><para>
                        Use this on bulk or interrupt endpoints which have
                        stalled, returning <emphasis>-EPIPE</emphasis> status
                        to a data transfer request.
                        Do not issue the control request directly, since
                        that could invalidate the host's record of the
                        data toggle.
                        </para></listitem></varlistentry>

                    <varlistentry><term>USBDEVFS_CONTROL</term>
                        <listitem><para>Issues a control request to the device.
                        The ioctl parameter points to a structure like this:
    <programlisting>struct usbdevfs_ctrltransfer {
            __u8   bRequestType;
            __u8   bRequest;
            __u16  wValue;
            __u16  wIndex;
            __u16  wLength;
            __u32  timeout;  /* in milliseconds */
            void   *data;
    };</programlisting>
                        </para><para>
                        The first eight bytes of this structure are the contents
                        of the SETUP packet to be sent to the device; see the
                        USB 2.0 specification for details.
                        The bRequestType value is composed by combining a
                        USB_TYPE_* value, a USB_DIR_* value, and a
                        USB_RECIP_* value (from
                        <emphasis>&lt;linux/usb.h&gt;</emphasis>).
                        If wLength is nonzero, it describes the length of the data
                        buffer, which is either written to the device
                        (USB_DIR_OUT) or read from the device (USB_DIR_IN).
                        </para><para>
                        At this writing, you can't transfer more than 4 KBytes
                        of data to or from a device; usbfs has a limit, and
                        some host controller drivers have a limit.
                        (That's not usually a problem.)
                        <emphasis>Also</emphasis> there's no way to say it's
                        not OK to get a short read back from the device.
                        </para></listitem></varlistentry>

                    <varlistentry><term>USBDEVFS_RESET</term>
                        <listitem><para>Does a USB level device reset.
                        The ioctl parameter is ignored.
                        After the reset, this rebinds all device interfaces.
                        File modification time is not updated by this request.
                        </para><warning><para>
                        <emphasis>Avoid using this call</emphasis>
                        until some usbcore bugs get fixed,
                        since it does not fully synchronize device, interface,
                        and driver (not just usbfs) state.
                        </para></warning></listitem></varlistentry>

                    <varlistentry><term>USBDEVFS_SETINTERFACE</term>
                        <listitem><para>Sets the alternate setting for an
                        interface.  The ioctl parameter is a pointer to a
                        structure like this:
    <programlisting>struct usbdevfs_setinterface {
            unsigned int  interface;
            unsigned int  altsetting;
    }; </programlisting>
                        File modification time is not updated by this request.
                        </para><para>
                        Those struct members are from some interface descriptor
                        applying to the current configuration.
                        The interface number is the bInterfaceNumber value, and
                        the altsetting number is the bAlternateSetting value.
                        (This resets each endpoint in the interface.)
                        </para></listitem></varlistentry>

                    <varlistentry><term>USBDEVFS_SETCONFIGURATION</term>
                        <listitem><para>Issues the
                        <function>usb_set_configuration</function> call
                        for the device.
                        The parameter is an integer holding the number of
                        a configuration (bConfigurationValue from descriptor).
                        File modification time is not updated by this request.
                        </para><warning><para>
                        <emphasis>Avoid using this call</emphasis>
                        until some usbcore bugs get fixed,
                        since it does not fully synchronize device, interface,
                        and driver (not just usbfs) state.
                        </para></warning></listitem></varlistentry>

                    </variablelist>
                </sect2>

                <sect2 id="usbfs-async">
                    <title>Asynchronous I/O Support</title>

                    <para>As mentioned above, there are situations where it may be
                    important to initiate concurrent operations from user mode code.
                    This is particularly important for periodic transfers
                    (interrupt and isochronous), but it can be used for other
                    kinds of USB requests too.
                    In such cases, the asynchronous requests described here
                    are essential.  Rather than submitting one request and having
                    the kernel block until it completes, the blocking is separate.
                    </para>

                    <para>These requests are packaged into a structure that
                    resembles the URB used by kernel device drivers.
                    (No POSIX Async I/O support here, sorry.)
                    It identifies the endpoint type (USBDEVFS_URB_TYPE_*),
                    endpoint (number, masked with USB_DIR_IN as appropriate),
                    buffer and length, and a user "context" value serving to
                    uniquely identify each request.
                    (It's usually a pointer to per-request data.)
                    Flags can modify requests (not as many as supported for
                    kernel drivers).
                    </para>

                    <para>Each request can specify a realtime signal number
                    (between SIGRTMIN and SIGRTMAX, inclusive) to request a
                    signal be sent when the request completes.
                    </para>
                    <para>When usbfs returns these urbs, the status value
                    is updated, and the buffer may have been modified.
                    Except for isochronous transfers, the actual_length is
                    updated to say how many bytes were transferred; if the
                    USBDEVFS_URB_DISABLE_SPD flag is set
                    ("short packets are not OK"), if fewer bytes were read
                    than were requested then you get an error report.
                    </para>

    <programlisting>struct usbdevfs_iso_packet_desc {
            unsigned int                    length;
            unsigned int                    actual_length;
            unsigned int                    status;
    };

    struct usbdevfs_urb {
            unsigned char                   type;
            unsigned char                   endpoint;
            int                             status;
            unsigned int                    flags;
            void                            *buffer;
            int                             buffer_length;
            int                             actual_length;
            int                             start_frame;
            int                             number_of_packets;
            int                             error_count;
            unsigned int                    signr;
            void                            *usercontext;
            struct usbdevfs_iso_packet_desc iso_frame_desc[];
    };</programlisting>

                    <para> For these asynchronous requests, the file modification
                    time reflects when the request was initiated.
                    This contrasts with their use with the synchronous requests,
                    where it reflects when requests complete.
                    </para>

                    <variablelist>

                    <varlistentry><term>USBDEVFS_DISCARDURB</term>
                        <listitem><para>
                        <emphasis>TBS</emphasis>
                        File modification time is not updated by this request.
                        </para><para>
                        </para></listitem></varlistentry>

                    <varlistentry><term>USBDEVFS_DISCSIGNAL</term>
                        <listitem><para>
                        <emphasis>TBS</emphasis>
                        File modification time is not updated by this request.
                        </para><para>
                        </para></listitem></varlistentry>

                    <varlistentry><term>USBDEVFS_REAPURB</term>
                        <listitem><para>
                        <emphasis>TBS</emphasis>
                        File modification time is not updated by this request.
                        </para><para>
                        </para></listitem></varlistentry>

                    <varlistentry><term>USBDEVFS_REAPURBNDELAY</term>
                        <listitem><para>
                        <emphasis>TBS</emphasis>
                        File modification time is not updated by this request.
                        </para><para>
                        </para></listitem></varlistentry>

                    <varlistentry><term>USBDEVFS_SUBMITURB</term>
                        <listitem><para>
                        <emphasis>TBS</emphasis>
                        </para><para>
                        </para></listitem></varlistentry>

                    </variablelist>
                </sect2>

            </sect1>

        </chapter>

    </book>
    <!-- vim:syntax=sgml:sw=4
    -->
```