STMicroelectronics

USB for OSPlus

User manual

8018282 Rev G

February 2011

www.st.com

BLANK

User manual

USB for OSPlus

Introduction

The OSPlus device layer presents applications with a simple API for accessing devices. This follows the conventional open, close, read, write, ioctl style seen in many other systems. For full details, refer to the *OSPlus User Manual (ADCS 7702033)*.

Contents

Preface

Document identification and control

Each book carries a unique ADCS identifier of the form:

ADCS *nnnnnnnx*

where *nnnnnnn* is the document number, and *x* is the revision.

Whenever making comments on a document, the complete identification ADCS *nnnnnnnx* should be quoted.

Conventions used in this guide

General notation

The notation in this document uses the following conventions:

- `Sample code`, `keyboard input` and `file names`,
- *Variables* and *`code variables`*,
- *`code comments`*,
- **Screens**, **windows** and **dialog boxes**,
- **Instructions**.

Hardware notation

The following conventions are used for hardware notation:

- REGISTER NAMES and FIELD NAMES,
- PIN NAMES and SIGNAL NAMES.

Software notation

Syntax definitions are presented in a modified Backus-Naur Form (BNF). Briefly:

1. Terminal strings of the language, that is, strings not built up by rules of the language, are printed in teletype font. For example, `void`.
2. Nonterminal strings of the language, that is, strings built up by rules of the language, are printed in italic teletype font. For example, *`name`*.
3. If a nonterminal string of the language starts with a nonitalicized part, it is equivalent to the same nonterminal string without that nonitalicized part. For example, `vspace-`*`name`*.
4. Each phrase definition is built up using a double colon and an equals sign to separate the two sides (‘`::=`’).
5. Alternatives are separated by vertical bars (‘`|`’).
6. Optional sequences are enclosed in square brackets (‘`[`’ and ‘`]`’).
7. Items which may be repeated appear in braces (‘`{`’ and ‘`}`’).

Acknowledgements

Maxtor One Touch^{TM} is a trademark of Seagate Technology LLC.

Lite-On^{TM} is a trademark of Lite-On IT Corporation.

1 OSPlus USB device model

1.1 Device driver application interface

The OSPlus device layer presents applications with a simple API for accessing devices. This follows the conventional open, close, read, write, ioctl style seen in many other systems. For full details, refer to the *OSPlus User Manual (ADCS 7702033)*.

The OSPlus USBLINK library makes USB attached devices appear within the system as standard OSPlus devices. Once the USB stack is initialized with the `USBLINK_Initialize()` call, any USB attached devices appear within the system. For instance, a USB attached hard disk might appear to the system as OSPlus device `/usb/hdisk0`. The USB stack supports hot plugging, so devices appear and disappear in the system as they are plugged and unplugged. Refer to *Section 2.7: Class driver overview on page 14*.

1.1.1 USB device types

All devices have a type, used to help distinguish between classes of devices, and therefore what a given device can do. Device types are known within OSPlus by a unique name and an associated unique numeric value.

OSPlus defines several standard types. The USB types are shown in *Table 1*.

Table 1. Standard OSPlus USB device types

Device type symbolic name	Device type name
`OSPLUS_DEVICE_TYPE_USB_CONTROLLER`	`"USB controller"`
`OSPLUS_DEVICE_TYPE_USB_COMM`	`"USB comm"`
`OSPLUS_DEVICE_TYPE_USB_GADGET`	`"USB gadget"`
`OSPLUS_DEVICE_TYPE_USB_H2H`	`"USB host-to-host"`
`OSPLUS_DEVICE_TYPE_USB_HID`	`"USB HID"`
`OSPLUS_DEVICE_TYPE_USB_HUB`	`"USB hub"`
`OSPLUS_DEVICE_TYPE_USB_JOYSTICK`	`"USB joystick"`
`OSPLUS_DEVICE_TYPE_USB_UNKNOWN`	`"USB unknown device"`
`OSPLUS_DEVICE_TYPE_USB_AUDIO`	`"USB audio"`
`OSPLUS_DEVICE_TYPE_USB_AUDIO_USB`	`"USB audio usb"`
`OSPLUS_DEVICE_TYPE_USB_AUDIO_STREAMING`	`"USB audio streaming"`
`OSPLUS_DEVICE_TYPE_USB_AUDIO_INPUT`	`"USB audio input"`
`OSPLUS_DEVICE_TYPE_USB_AUDIO_MICROPHONE`	`"USB audio microphone"`
`OSPLUS_DEVICE_TYPE_USB_AUDIO_OUTPUT`	`"USB audio output"`

Table 1. **Standard OSPlus USB device types**

Device type symbolic name	Device type name
`OSPLUS_DEVICE_TYPE_USB_AUDIO_SPEAKERS`	`"USB audio speakers"`
`OSPLUS_DEVICE_TYPE_USB_AUDIO_BIDIR`	`"USB audio bidir"`
`OSPLUS_DEVICE_TYPE_USB_AUDIO_TELEPHONY`	`"USB audio telephony"`
`OSPLUS_DEVICE_TYPE_USB_AUDIO_EXTERNAL`	`"USB audio external "`
`OSPLUS_DEVICE_TYPE_USB_AUDIO_ANALOG`	`"USB audio analog"`
`OSPLUS_DEVICE_TYPE_USB_AUDIO_DIGITAL`	`"USB audio digital"`
`OSPLUS_DEVICE_TYPE_USB_AUDIO_LINE`	`"USB audio line"`
`OSPLUS_DEVICE_TYPE_USB_AUDIO_SPDIF`	`"USB audio spdif"`
`OSPLUS_DEVICE_TYPE_USB_AUDIO_EMBEDDED`	`"USB audio embedded"`
`OSPLUS_DEVICE_TYPE_USB_AUDIO_MIXER`	`"USB audio mixer"`
`OSPLUS_DEVICE_TYPE_USB_AUDIO_SELECTOR`	`"USB audio selector"`
`OSPLUS_DEVICE_TYPE_USB_AUDIO_PROCESSOR`	`"USB audio processor"`
`OSPLUS_DEVICE_TYPE_USB_AUDIO_EXTENSION`	`"USB audio extension"`
`OSPLUS_DEVICE_TYPE_USB_VOIP`	`"USB VOIP"`
`OSPLUS_DEVICE_TYPE_USB_VOIP_YEALINK`	`"USB VOIP Yealink"`

The type for a particular device can be queried from two standard device attributes for that device, as shown in *Table 2*.

Table 2. **Device type attributes**

Device type symbolic name	Device type name
`"type_name"`	Type name for the device
`"type"`	Bound type value for the device

1.1.2 Starting and stopping USB devices

The USB standard supports hot plugging (the ability to insert or remove a device while a system is running).

The `device_stop()` and `device_stop_timeout()` calls allow for safe extraction to take place. These calls prevent further opens of the device in question, and return when all open handles on the device have been closed (or a timeout occurs using `device_stop_timeout()`). Following the successful return of one of these calls, the device can be safely removed. While in the stopped state, it is not possible to open the device. It can be restarted with the `device_start()` call, which permits subsequent opens of the device to succeed.

1.1.3 Device attributes

Information about a device is held in the form of device attributes. These are stored as arbitrary name/value pairs in the registry. Refer to the *OSPlus User Manual (ADCS 7702033)* for full details.

2 OSPlus USB driver

2.1 USB driver overview

The OSPlus USB driver is based on version 3.4 of the SoftConnex USBLINK product. The USB driver allows supported devices to be hot plugged to and from the USB on supported platforms. The devices are then registered with the device library so they can be opened and used by the target application.

2.2 USB driver host controller support

The USB driver currently supports multiple on-chip USB host controllers based on the open host controller interface specification for USB Release 1.0a (OHCI) and enhanced host controller interface specification for USB Revision 1.0 (EHCI).

2.3 USB driver device support

The USB driver currently supports only a limited subset of the available USB devices. The types of supported device include:

- audio devices (for example a microphone or speakers)
- communications devices (for example cable modem)
- ethernet adapter
- HID device (for example joystick, keyboard, mouse)
- host-to-host networking adaptor
- hub
- mass storage device (for example, memory sticks, memory card readers, hard disks)
- serial adapter
- VOIP phone

Note: 1 *Not all devices of a particular type identified above are supported. This is because not all manufacturers keep to the standard USB class specifications, and some devices require custom class drivers.*

2 *It is possible for users to write their own class drivers for the USB stack. Refer to Section 2.7.10 on page 21, Section 2.9.10 on page 36* and *Section 3.1 on page 68.*

2.4 USB driver configuration

The USB driver supports configuration by the user. These configuration options are limited to specifying the priority of the tasks created by the USB driver and selectively enabling support for specific USB class drivers.

The USB driver contains two tasks. The first task handles enumeration and performs the actions needed when inserting or removing a device on the USB. The second task handles the notifications involved with data transfers. By default, the tasks execute at `MAX_USER_PRIORITY`. It is recommended the tasks remain running at this priority, but if the priority absolutely must be changed, it can be done by writing to particular registry keys.

By default, no class drivers included with the USB stack are enabled. This is in order to restrict functionality or to allow a class driver to be overridden by another user supplied class driver. To enable a class driver, create an entry in the registry. The name of the entry must be unique. The value of the entry is actually a function pointer to the class driver register module function. All OSPlus registries for supported platforms contain the required registry keys to register all class drivers. If you are using a custom registry, this may not be the case. Be careful to ensure that you have selected at least one class driver to register. If you do not, then the USB stack will be useless.

All configuration options must be written to the registry before the USB driver is initialized. Writing to the registry keys after the USB driver is initialized has no effect on the driver.

The path to the registry keys is provided as a define named `USB_CONFIG_PATH`. The name of the keys that hold the priority for the USB driver tasks and the keys that disable class drivers can be found in *Table 52: Configuration macros defined in usblink.h on page 60*.

Figure 1 shows an example of using the configuration options to set user-defined task priorities for the two USB tasks before initializing the USB driver.

Figure 1. Example configuration options

```
registry_key_set (
        USB_CONFIG_PATH,
        USB_CONFIG_ENU_TASK_PRI,
        (void *) MAX_USER_PRIORITY);

registry_key_set (
        USB_CONFIG_PATH,
        USB_CONFIG_URB_TASK_PRI,
        (void *) MAX_USER_PRIORITY);

registry_key_set (
        USB_CONFIG_CLASS_PATH,
        "Hub Class Driver",
        (void *) HUB_RegisterModule);

registry_key_set (
        USB_CONFIG_CLASS_PATH,
        "Mass Storage: CBI",
        (void *) CBI_RegisterModule);

if (USBLINK_Initialize (uncached_mem, uncached_mem_size) == -1)
{
   return -1;
}
```

2.4.1 Power management

OS21 R3.4 and later (that is, OS21 distributed with ST40 toolset R4.4 and later, or ST200 toolset R6.4 and later) supports a power management API.

If the USBLINK library is linked with OS21 R3.4 or later, then, by default, it registers a handler to take the USB hardware into and out of low power mode when instructed by OS21. To change this behavior, set the following key:

```
int disable = 1;

registry_key_set (USB_CONFIG_PATH,
                  "disable power management",
                  (void **)disable);
```

If this key is not set, or if the value stored is zero, the USB stack participates in power management events. If the value stored is 1, then the USB stack does not participate in power management events.

When registering a power management handler with OS21, a callback order must be specified. By default, the USB stack uses `OS21_POWER_CALLBACK_ORDER_LAST-1` as the order. This can be overridden by setting a registry key before initializing the stack, as follows:

```
int value = OS21_POWER_CALLBACK_ORDER_LAST;

registry_key_set (USB_CONFIG_PATH,
                 "power management callback order",
                 (void **)value);
```

where `value` is in the range `OS21_POWER_CALLBACK_ORDER_FIRST` to `OS21_POWER_CALLBACK_ORDER_LAST`.

2.5 USB driver initialization

Before the USB driver can be initialized, the target application must setup and reserve a block of uncached memory for sole use by the USB driver. This block of uncached memory is used to build internal data structures and as intermediate buffers during data transfers. The recommended absolute minimum size for this block of uncached memory is provided by the define `SCC_OSPLUS_NONCACHED_MEM_POOL_SIZE`. A smaller block of uncached memory can be used, but is not recommended as the USB driver's ability to perform multiple data transfers can be greatly diminished. Debug versions of the USB driver output a warning if the uncached memory block is smaller than recommended. The way this block of uncached memory is created varies depending on the target platform. An example of how to create an uncached block of memory can be found in the OSPlus `hotplug` example.

The starting address of the uncached block of memory, and the size of the memory block, are passed as parameters to the `USBLINK_Initialize()` function to initialize the USB driver.

During the initialization of the USB driver, several actions are performed. First, all USB class drivers are registered with the USB driver, then the registry is queried to determine the number of on-chip EHCI and OHCI controllers supported by the platform. The settings for each controller including base address, interrupt information and any hardware enable code is read from the registry. Starting with the EHCI controllers then finishing with the OHCI controllers, each controller is initialized. This involves building all required internal data structures, resetting the controller, installing all interrupt handlers and finally setting the controller to its working state.

Each USB host controller is registered with the OSPlus device library. They are given a name which has the type of the host controller as a prefix, and an instance number as a postfix. Currently only EHCI and OHCI host controllers are supported, so names of the form `/usb/ehcin` and `/usb/ohcin` are used. The USB host controller devices provide a simple mechanism to switch USB root hub ports in to and out of test modes for validation purposes.

After the USB driver has been successfully initialized, it can handle the insertion and extraction of devices to and from the USB.

2.6 USB driver device enumeration

When a device is attached to the USB, it is queried to determine what type of device it is and what class driver should be used to manage the device while it is attached to the USB. This whole process is termed enumeration.

When registering a class driver with the USB driver, the class driver is registered as being able to control a device based on either a combination of a unique vendor and device identifiers, or a specific device class and sub-class, or a specific communication protocol. When a device is first attached to the USB, the USB driver issues a standard command to the device to retrieve the device descriptor. The device descriptor is then parsed to determine what interfaces the device supports, and any additional descriptors are read as required. This information then identifies the specific class driver to be used to manage the device. If the identified class driver is registered with the USB driver, it is called to set up the device and register it with the device library. It can then be used as any other device. If no class driver is found to manage the device, the USB driver registers the device as an unknown device handled by the proxy class driver. This built-in class driver allows a user written class driver to handle the device. Refer to *Section 2.7.10 on page 21, Section 2.9.10 on page 36* and *Section 3.1 on page 68*.

When a device is removed from the USB, the class driver that manages the device is called to perform any clean-up required. This can involve aborting any input or output operations and unregistering the device from the device library. If the device is removed while input or output is in progress, then, under most circumstances, errors result. The device library contains an API that prevents any further input and output to a device so allows it to be removed from the USB without resulting in errors. The function to call is `device_stop_timeout()`. See *Section 1.1.2: Starting and stopping USB devices on page 9* for further details. The only operation allowed on a device that has been extracted but has open handles is a close.

2.7 Class driver overview

The USB driver supports several different devices through the use of class drivers. Each class driver associates attributes with the devices it manages. One attribute common to all USB devices is the attribute `connection`. This attribute is set to `usb` for each USB device. To read the attributes associated with a device, the `device_attribute()` API should be used. See *Section 1.1.3: Device attributes on page 9* for more details.

The supported devices and the class drivers they use are described in the following sections.

2.7.1 USB audio class devices

Before reading this section, we recommend that you read the USB audio class specification documents from *www.usb.org* in order to familiarize yourself with the terminology and concepts described here.

An USB audio device is a composite device that consists of one or more audio entities; these entities are named *units* and *terminals*. An audio unit is an entity that performs some manipulation of the audio (for example, volume control) carried on one or more input pins and outputs the result via a single output pin. A pin represents one or more logical audio channels. Audio units are wired together by connecting the input and output pins to other units and terminals in order to create the desired topology.

Audio terminals can be input or output. An input terminal can be thought of as a entry point for audio in a device (for example, a microphone) that is then wired to a unit's input pin. Similarly, an output terminal can be thought of as an exit point for audio that is wired to a unit's output pin.

The USB audio class driver implements a subset of the USB audio class specification v1.0 that provides enough support for the most common audio units and terminals so as to support basic USB audio devices, including microphones and speakers. The audio class driver supports only audio control and audio streaming interfaces; it does not support the audio class MIDI control interface.

An audio device is a composite device consisting of a number of audio units or terminals. We register a single device instance with the OSPlus device library to represent this composite device using the prefix `/usb/audio` and the postfix of the device instance number (for example, `/usb/audio0`). Each audio unit or terminal is then represented and manipulated as an individual device registered with the device library using the prefix `/usb/audio/` and the postfix of the unit or terminal type and a device instance number (for example, for an audio device with a mixer unit, the mixer unit is named `/usb/audio/mixer0`). Audio devices are neither character nor block devices. As such they do not support read or write functions, only ioctls.

Table 3 describes the audio units and terminals that the audio class driver supports.

Table 3. Audio units and terminals supported by audio class driver

Audio Device	Description
Mixer Unit	A mixer unit is supported as a device with the prefix `/usb/audio/mixer`. It is the job of a mixer to combine two or more audio channels from one or more inputs to a single output.
Selector Unit	A selector unit is supported as a device with the prefix `/usb/audio/selector`. It is the job of a selector to switch between two or more inputs
Feature Unit	A feature unit is supported. It does not appear as stand alone device, but instead as a set of controls attached to another unit or terminal. A feature unit allows you to modify things such as volume, bass, treble etc. The audio class driver does not support feature unit graphic equalizer controls, however.
Processing Unit	Processing units are not supported at this time.
Extension Unit	Extension units are not supported at this time.
USB Terminal Type	Only USB streaming terminals are supported. USB streaming terminals provide an interface to stream audio between the audio device and your application. A USB audio streaming terminal may only stream audio one way and is supported as a device with the prefix `/usb/audio/pcm`.
Input Terminal Types	Input terminal types are various types of microphones. The audio class driver supports microphones as devices with the prefix `/usb/audio/microphone.`
Output Terminal Types	Output terminal types are various types of speaker. The audio class driver supports speakers as devices with the prefix `/usb/audio/speakers`.
Bi-directional Terminal Types	Bi-directional terminal types can be essentially classified as headsets or types of speaker phone. All are registered as devices with the prefix `/usb/audio/bidir`.
Telephony Terminal Types	Telephony terminal types consist of phone lines and telephones. The audio class driver supports them with the prefix `/usb/audio/telephony`.

Table 3. Audio units and terminals supported by audio class driver (continued)

Audio Device	Description
External Terminal Types	External terminal types consist of items such as line in/out analog/digital connectors etc. The audio class driver supports them with various prefixes such as `/usb/audio/external` (for generic external terminal types), `/usb/audio/analog` (for analog connector), `/usb/audio/digital` (for digital connector), `/usb/audio/line` (for line in/out connector) and `/usb/audio/spdif`.(fort S/PDIF connectors).
Embedded Terminal Types	Embedded terminal types consist of connections to audio sources or sinks in a device such as a CD player or DAT player. The audio class driver supports them with the prefix `/usb/audio/embedded`

2.7.2 USB communications class devices

The USB communication class driver supports several communications class devices that keep to the abstract control model (ACM) and the ethernet networking control model (ENCM). For more information regarding the ACM and ENCM see the document *Universal Serial Bus Class Definitions for Communication Devices v1.1* available from www.usb.org.

Generic communication class devices are identified based on the device interface class and subclass. Specific communications class devices are identified based on vendor and product identifiers.

To enhance the usability of the communications class driver, it is not used directly but is instead used with a thin-class driver wrapper. Currently there are wrappers for ACM and ENCM devices.

USB communications class ACM wrapper

The ACM class driver wrapper manages generic communications class devices that keep to the ACM protocol. The wrapper ensures the device has a name and type that clearly identifies the device as an ACM device. All other management of the device is handled by the communications class driver.

The ACM class driver wrapper causes the communications class device to be registered with the device library as a character device with a name prefix `/usb/acm` and a postfix of the device instance number (for example, `/usb/acm0`).

All ACM communications class devices have an attribute named `protocol` associated with them that has a value of `acm`.

USB communications class RNDIS wrapper

The RNDIS class driver wrapper manages generic communications class devices that use the Microsoft RNDIS communication protocol (through the ACM protocol). The wrapper ensures the device has a name and type that clearly identifies the device as an RNDIS device. All other management of the device is handled by the communications class driver.

Specific devices supported include the Hitron BRG-35302-ED Broadband Cable Modem.

The RNDIS/ACM class driver wrapper causes the communications class device to be registered with the device library as a character device with a name prefix `/usb/rndis` and a postfix of the device instance number (for example, `/usb/rndis0`).

All ACM communications class devices have an attribute named `protocol` associated with them that has a value of `rndis`.

USB communications class ENCM wrapper

The ENCM class driver wrapper manages generic communications class devices that keep to the ENCM protocol. It also manages specific cable modem communication class devices that have specific vendor and product identifiers. Specific devices supported include the Thomson TCM 420 and D-Link DCM-201 DOCSIS compliant cable modems. Both of these cable modems adhere to the ENCM communications class protocol.

As with the ACM class driver wrapper, the ENCM wrapper ensures the device has a name and type that clearly identifies the device as an ENCM device. As with the ACM class driver wrapper, all other management of the device is handled by the communications class driver.

The ENCM class driver wrapper causes the communications class device to be registered with the device library as character device with a name prefix `/usb/encm` and a postfix of the device instance number (for example, `/usb/encm0`).

All ENCM communications class devices have an attribute named `protocol` associated with them that has a value of `encm`.

2.7.3 USB ethernet devices

USB ethernet devices based on various chip sets are supported by the USB stack. Supported devices include:

- Aztel USB 10/100M Fast Ethernet Adapter
- D-Link DUB-E100
- Edimax EU-4202 USB to Ethernet Adapter
- Level One 10/100/1000 Mbit/s Ethernet Adapter
- RealTek 8150 10/100 Mbit/s USB to Ethernet Adapter

Each supported ethernet device is registered with the device library as a character device with the prefix `/usb/ethernet` and the postfix of the actual device instance number (for example, `/usb/ethernet0`).

All ethernet devices have an attribute named `protocol` associated with them. This attribute has the value `osplus` indicating the device does not keep to a supported communications class control model.

Most USB ethernet devices are able to support asynchronous packet reads and writes. The USB stack needs to reserve enough resources to be able to efficiently process these asynchronous requests. This is done when the device is discovered and enumerated. The USB stack queries two registry keys to discover the maximum expected number of asynchronous reads and writes. The keys are located in USB_CONFIG_PATH, and are named `USB_CONFIG_ETHERNET_RX_ASYNCS` and `USB_CONFIG_ETHERNET_TX_ASYNCS`. If not set the stack will assume reasonable default values.

2.7.4 USB HID devices

The USB human interface device (HID) class driver supports many devices including joysticks, keyboards and mice. If an HID device is not recognized as either a joystick, keyboard or mouse it is handled as a generic HID device.

USB HID joystick/gamepad devices

USB HID joysticks and gamepads are supported to a limited degree through the USB HID class joystick module. Currently support is limited to X, Y, Z, rX, rY, rZ axis controls, a slider control, a hat switch control and up to 32 buttons. Not all HID joysticks support each control, so it is important to check what features a HID joystick supports before it is used. Force feedback (rumble) is supported on some models too.

HID joystick/gamepad devices that have been tested and are known to work include:

- Logitech Dual Action gamepad
- Logitech Wingman Precision
- Logitech Cordless Rumblepad2 (including force feedback)
- Saitek ST50 Action Stick
- Saitek P220 Digital gamepad

Each device is registered with the device library using the prefix `/usb/joystick` and the postfix of the actual device instance number (for example, `/usb/joystick0`). A USB HID joystick device supports communication using ioctls only.

USB HID keyboard devices

USB HID keyboards are be supported by the HID class driver keyboard module.

Note: Pressing a key on the keyboard produces an equivalent ASCII code.

When there is no ASCII code for a key press (such as for the function keys or any multimedia keys) the USB usage identifier is returned. For details on these identifiers please see the Universal Serial Bus HID Usage Tables v1.11 document available from *www.usb.org*.

HID keyboard devices that have been tested with and that are known to work include:

- Microsoft Office Keyboard
- Microsoft Internet Keyboard Pro
- Logitech Internet Navigator
- Cherry RS 6000
- Macally iWebkey

When a keyboard device is connected to the system, a new device is registered with the device library using the prefix `/usb/keyboard` and the postfix of the actual device instance number (for example, `/usb/keyboard0`).

To allow input from multiple keyboards to be handled efficiently, while at least one keyboard device is connected to the system, a keyboard device is registered with the device library using the name `/usb/keyboard`.This device combines the key presses from each attached keyboard into one input stream. Each keyboard still functions independently of the others, meaning that if the `CAPS LOCK` key is pressed on one keyboard, then it affects only that keyboard. This keyboard device is unregistered with the device library when the last keyboard is removed from the USB.

A USB keyboard device supports communication using ioctls only.

USB HID mouse devices

USB HID mouse devices are supported by the HID class driver mouse module.

Note: Not all features of every HID mouse are supported.

Currently supported mouse features include X and Y axis, wheel, tilt wheel and up to 32 buttons. Not all HID mice can support every control, so it is important to check the features supported by an HID mouse before it is used.

The following HID mouse devices have been tested and shown to work:

- Microsoft Wheel Mouse Optical
- Microsoft Wireless Optical Mouse
- Macally iWebkey

When a mouse device is connected to the system, a new device is registered with the device library using the prefix `/usb/mouse` and the postfix of the actual device instance number (for example, `/usb/mouse0`).

As with USB keyboard devices, to allow input from multiple mice to be handled efficiently, while at least one mouse device is connected to the system, a mouse device is registered with the device library using the name `/usb/mouse`.This device combines the the movement and button press data from each mouse into one definitive position and button status. This mouse device is unregistered with the device library when the last mouse is removed from the USB.

A USB mouse device supports communication using ioctls only.

USB HID generic devices

USB HID generic devices are handled by the HID class driver proxy module.

Each generic HID device is registered with the device library using the prefix `/usb/hid` and the postfix of the actual device instance number (for example, `/usb/hid0`).

2.7.5 USB host-to-host devices

The USB host-to-host class driver supports devices based on the Prolific 2301 and Prolific 2302 chip sets commonly found in most generic host-to-host adapters. Specific branded devices supported include the Vivanco USB2.0 Network-Link Cable.

Each host-to-host device is registered with the device library as a character device with the prefix `/usb/h2h` and the postfix of the actual device instance number (for example, `/usb/h2h0`).

A host-to-host device is a raw transport device that does not guarantee delivery of the data sent over it. A protocol (such as TCP/IP) should be layered over the raw device.

The host-to-host device is able to support asynchronous packet reads and writes. The USB stack needs to reserve enough resources to be able to efficiently process these asynchronous requests. This is done when the device is discovered and enumerated. The USB stack queries two registry keys to discover the maximum expected number of asynchronous reads and writes. The keys are located in `USB_CONFIG_PATH`, and are named `USB_CONFIG_H2H_RX_ASYNCS` and `USB_CONFIG_H2H_TX_ASYNCS`. If not set the stack will assume reasonable default values.

USB host-to-host devices ethernet wrapper

The USB host-to-host class driver can be treated as a USB ethernet device by using the h2heth ethernet wrapper driver. This wrapper presents the host-to-host device as a USB ethernet device, and allows it to be used as a standard USB ethernet device.

Once initialized (by calling `h2heth_initialize()`) any USB host-to-host device attached to the USB is detected and a new device is registered with the device library. This is a character device with the prefix `/usb/ethernet` and the postfix of the actual device number (for example, `/usb/ethernet0`).

2.7.6 USB hub devices

Attachment of generic USB v1.1 and USB v2.0 hubs is supported with the USB hub class driver. A hub allows the attachment of more USB devices than the platform has ports. Specific branded devices supported include Belkin ExpressBus^{TM} and NEWlink^{TM} Hi Speed hubs.

Each USB hub is registered with the device library with the prefix `/usb/hub` and the postfix of the actual device instance number (for example, `/usb/hub0`). A USB hub supports communication using ioctls only.

2.7.7 USB mass storage devices

The USB mass storage class driver supports a variety of mass storage devices including hard disks, CD/DVD-ROM, memory card readers, and pen drives. If the mass storage device contains multiple logical unit numbers (such as memory card readers with support for compact flash, smart media, SD cards, and memory sticks), each logical unit is treated as a separate USB device and is registered with the device library as a separate device.

Specific branded devices supported include:

- Maxtor One Touch^{TM} range of USB hard disks
- LiteOn^{TM} LDW-851SX DVD drive
- Buffalo^{TM} RUF-C512M/U2 PenDrive
- Sony USB Floppy Disk Drive

Mass storage devices are treated as two distinct types of devices: removable media and fixed media. Each type is registered with the device library under a different sequential name: for example a hard disk is a fixed media device and is named `/usb/hdisk0`, `/usb/hdisk1`, and so on, whereas a DVD drive is a removable media device and is named `/usb/rdisk0`, `/usb/rdisk1`, and so on.

All USB mass storage devices are registered with the device library as block devices.

USB mass storage devices have an attribute named `protocol` and a value of `scsi` associated with them. This attribute determines the command protocol used by the devices.

Limiting SCSI transfer size

Some USB mass storage devices place limits on the number of blocks of data that can be transferred in a single SCSI operation. This is in violation of the USB specification, which places no such limit. If this limit is exceeded, the device behaves unpredictably, and may report I/O errors.

The USB stack is aware of many such devices, and automatically reduces the size of transfers to these devices accordingly. However, new USB devices are always appearing, and the USB stack will inevitably not be aware of every such device. To take account of this, OSPlus has introduced a USB configuration key in the registry called `USB_CONFIG_SCSI_MAX_TRANSFER`. If set, this key specifies the maximum number of blocks that can be requested in a single SCSI transaction.

For instance, to limit all SCSI transfers to at most 128 blocks:

```
registry_key_set (USB_CONFIG_PATH,
                  USB_CONFIG_SCSI_MAX_TRANSFER,
                  (void**)128);
```

If you discover a USB disk drive which is persistently giving I/O errors, then setting this key to limit the transfer size to 64 or 128 blocks may be beneficial.

2.7.8 USB serial devices

The USB serial class driver supports USB to serial adapters based on the Prolific 2303 chip set, used in a large number of generic USB-to-serial adapters. Specific branded devices supported include the NEWlink USB Serial Adapter.

Each supported serial device is registered with the device library as a character device with the prefix `/usb/serial` and the postfix of the device instance number (for example, `/usb/serial0`).

2.7.9 USB VOIP devices

The USB VOIP class driver supports USB VOIP phones based on the Yealink USB-P8DH and USB-P1KH. VOIP phones are component devices that consist of a screen and keypad with a microphone and speakers. The VOIP class driver merely provides support for the phone display and keypad. The microphone and speakers are handled by the USB audio class driver, just as if they were any other audio device.

Each supported Yealink device is registered with the device library with the prefix `/usb/voip/yealink` and the postfix of the device instance number (for example, `/usb/voip/yealink0`). The audio components of the VOIP phones will be registered in accordance with the USB audio class driver, for example the phone microphone will be registered with the prefix `/usb/audio/microphone` etc.

2.7.10 USB unsupported devices

Devices that are not specifically supported by the USB driver are handled by a USB proxy class driver. This proxy class driver exposes an interface to the raw device in order to perform control, bulk and interrupt transfers. Using this raw device interface it is possible for a third party to add support for any device that is currently not supported directly be the USB driver.

Each device handled by the proxy class driver is registered with the device library as a character device with the prefix `/usb/dev` and the postfix of the device instance number (for example, `/usb/dev0`).

2.8 USB driver host controller usage model

The USB driver presents all host controllers as devices with names prefixed with either `/usb/ehci` or `/usb/ohci`.

The ioctl function `OSPLUS_IOCTL_USB_HOST_GET_PORTS` sent to a host controller device passes back the number of root hub ports supported by that controller. The number of ports is written to the integer pointed to by the ioctl `param` argument.

The ioctl `OSPLUS_IOCTL_USB_HOST_TEST_MODE` is used to put a root hub port in to a USB test mode. This can be very useful when validating a board design (for instance generating an eye diagram to check USB signal conformance). The ioctl `param` argument should point to a `usb_host_test_mode_params_t` structure which describes the port to use, and the required test mode.

Note: Only EHCI host controllers support putting USB ports into test modes. An EHCI port can only be put into test mode if a high speed device is connected to it. When a controller has a port in a test mode, it should not be used for I/O on other ports.

Use the ioctls `OSPLUS_IOCTL_USB_HOST_SUSPEND_PORT` and `OSPLUS_IOCTL_USB_HOST_RESUME_PORT` to suspend and resume the given host controller port. The ioctl parameter argument is the integer number of the affected port, and should be in the range of 0 to one less than the total number of ports (as returned by the `OSPLUS_IOCTL_USB_HOST_GET_PORTS` ioctl.)

2.9 USB driver device usage model

As with the USB driver, all associated USB class drivers are written to conform to the OSPlus device model. Therefore when a USB device is attached and registered with the device library, the device can be accessed using the standard device library open, close, read, write and ioctl functions. See *Chapter 1: OSPlus USB device model on page 7* for more details on using the device library.

Some USB devices provide multiple functions, and consequently rely on more than one class driver to operate correctly. An example of such a device is an audio device with a physical volume control, where typically there are multiple functions such as microphone and mixer (handled by the `audio` class) and the volume control (handled by the `HID` class). Such devices are known as composite devices. Use the ioctl `OSPLUS_IOCTL_USB_COMPOSITE_INFO` to check if a device is a composite device and to get the name and type of each device that is part of that composite.

The ioctl `OSPLUS_IOCTL_USB_TREE_NODE_INFO` can be used to get information about a USB device. This information consists of the device and port to which the device is attached, and if the device is a hub the devices plugged into each hub port. With this information, it is possible to create a graphical tree of the USB devices attached to the system. For an example of how to do this see the `usbtree` example.

2.9.1 USB audio class devices

An example named audio is supplied with the OSPlus USB stack. This example shows how to capture audio from a USB attached microphone, save it to a .WAV file and play back the contents of the .WAV file through USB attached speakers.

Audio device topology

Audio peripherals are composite devices, so they can be quite complicated to use. The first thing that we need to know in order to use an audio device correctly is the device topology, which describes how the audio units and terminals are connected together. If a device supports both audio capture and playback, then usually there is a capture topology and a playback topology.

For the purpose of illustration, the example uses a generic audio peripheral. This peripheral has a microphone and analog line-in inputs, and one output for connecting to speakers. The peripheral supports both recording from the microphone and analog line inputs to a user application (although only one input may be selected at a time) and also supports playback through the speakers of audio from both the microphone and analog line in inputs and from a user application.

Figure 2 shows the tree topography for this peripheral when it is used for capturing audio:

Figure 2. Peripheral topology when capturing audio

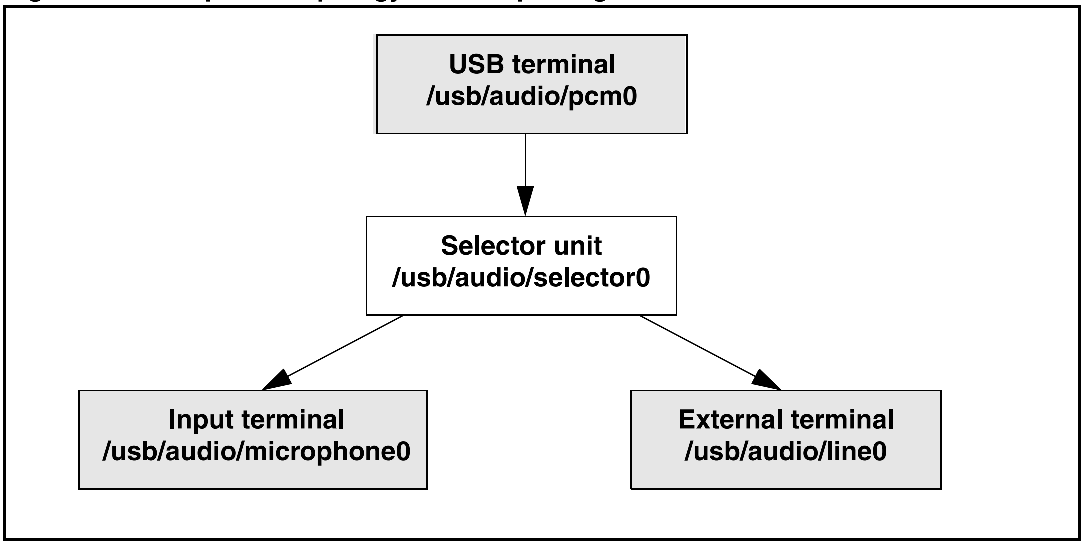
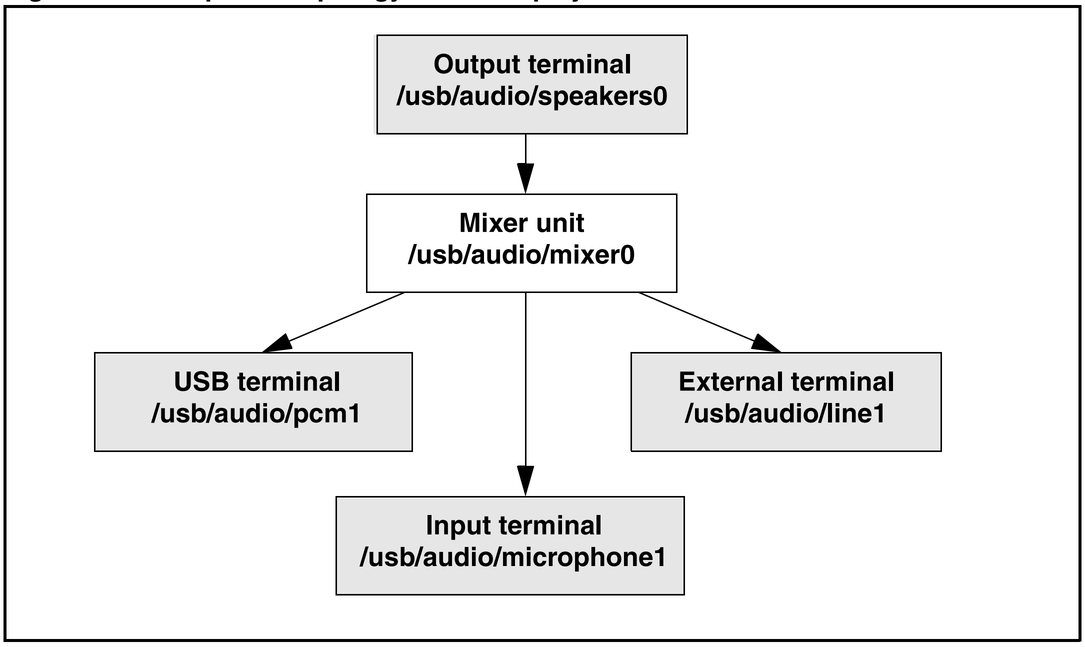


Figure 3 shows the tree topography that the peripheral uses for audio playback.

Figure 3. Peripheral topology for audio playback



In both the above diagrams, audio terminals have a gray background and audio units have a white background.

An application normally opens the audio device that it wants to use for audio capture or playback. For example, if an application intends to record from a microphone input, it opens a microphone capture device; and an application intending to playback audio to a set of speakers opens the speakers device.

It is not possible to read audio from a microphone device or write it to a speakers device directly. Audio data can only be read or written by an application that uses an USB terminal device that exists in the tree topology. These devices provide either an entry or exit point for audio in each tree topology. A USB terminal only allows audio to be read from or written to it (depending on whether the topology is for capture or playback); they are not bidirectional. If a topology does not contain a USB terminal, then audio cannot be read or written in that particular topology, although any controls attached to any other audio devices in the topology may still be modified. When an application opens an audio device in a particular topology, and the application wants to read or write audio, the application must walk the tree until it finds a USB terminal that can be used either for reading or writing audio. Generally, in a capture topology, the USB terminal is the root node of the tree.

Walking audio device tree topology

When the application has selected the particular device it wishes to use, it must determine the USB terminal to use for reading/writing audio by walking the tree topology. For capture topologies, the application finds the USB terminal by walking up the tree, and for playback topologies the application walks down the tree.

The easiest way for the application to walk the tree topology is by using a recursive function. The application first calls the recursive function passing in the name of the audio device that it wishes to use (i.e. the starting node in the tree topology). The recursive function opens the audio device and issues the ioctl `OSPLUS_IOCTL_USB_AUDIO_GET_UNIT_INFO` to get the audio device information. This information tells the application if the device is for capture or playback, how many audio channels it has and the channel configuration, amongst other things.

In order to move up or down the tree, the recursive function needs to get a list of audio devices (that is, nodes in the tree) that provide an audio input to the current node or to which the current node outputs audio; this is done with the ioctls `OSPLUS_IOCTL_USB_AUDIO_GET_UNIT_INPUTS` and `OSPLUS_IOCTL_USB_AUDIO_GET_UNIT_OUTPUTS`. The recursive function then calls itself for each output node (to move up the tree for capture topology) or each input node (to move down the tree for playback topology). This process is repeated until the recursive function has found the USB terminal it requires or there are no more inputs/outputs to consider.

On finding the correct USB terminal, the recursive function returns the USB terminal name and starts to unwind. As the recursive function unwinds, it visits each node in the tree that forms part of the path between the USB terminal and original device that the application wanted to open. As it visits each node on the path, the function must to ensure that if the node contains any feature controls such as mute or volume, then these controls are programmed to allow the audio to pass through. Similarly, if any node is an audio mixer device, then the mixer should be set to mix the correct input channels to the mixer output channels. If the node is an audio selector device, then it should be configured to select the correct input device.

Finally, when the recursive function exits, it must return the name of the USB terminal attached to the audio device the application wishes to use. At this step, every audio device on the path to the USB terminal must be configured for use. If the recursive function does not return the name of the USB terminal, then the terminal could not be found.

Setting audio mixer values

The purpose of audio mixer units is to mix the audio channels of a given mixer input to the mixer output channels. To do this, the application needs to have certain information about the mixer, such as what inputs it has and the mapping of input channels to input pins. By issuing the ioctl `OSPLUS_IOCTL_USB_AUDIO_GET_UNIT_INPUTS`, the application obtains a list of all mixer inputs, in the order in which they are connected to the mixer. The application can then issue the ioctl `OSPLUS_IOCTL_USB_AUDIO_GET_MIXER_INFO` to obtain a list of the mixer input channels that are mapped to a given input.

When the application has this information, it can determine which mixer input channels map to which mixer output channel. To do this, the application issues the ioctl `OSPLUS_IOCTL_USB_AUDIO_GET_MIXER` to return the mapping of mixer output channels to mixer input channels, and the current volume setting for each channel. The application is then free to set the required volume for each of the mapped channels using the ioctl `OSPLUS_IOCTL_USB_AUDIO_SET_MIXER`.

Setting audio selector values

Audio selector units are designed to allow an application to switch between two or more audio inputs. As with any other audio unit, the application can obtain a list of inputs to the selector unit by issuing the ioctl `OSPLUS_IOCTL_USB_AUDIO_GET_UNIT_INPUTS`. When the required input has been chosen from the list of inputs, the selector unit switches to a given input by issuing the ioctl `OSPLUS_IOCTL_USB_AUDIO_SET_SELECTOR`. The application can retrieve the currently selected input by issuing the ioctl `OSPLUS_IOCTL_USB_AUDIO_GET_SELECTOR`.

Setting audio feature control values

Audio feature controls can manipulate audio features such as mute, volume, bass, treble and so forth. When the ioctl `OSPLUS_IOCTL_USB_AUDIO_GET_UNIT_INFO` is issued, part of the information returned is a bitmap of all feature controls supported for either capture or playback devices. If an audio device supports both capture and playback, then it may have separate capture and playback feature controls.

Feature controls are always attached to an audio device and are not treated as individual units in their own right. Also, feature controls can work either on the master audio channel (channel 0) or on any or all of the logical audio channels of an audio device output.

To get the current value of a feature control, the application must first specify the feature control and if the control is for capture or playback. It can then issue the ioctl `OSPLUS_IOCTL_USB_AUDIO_GET_FEATURE`. The structure returned by this ioctl provides the minimum and maximum values for the control, and the resolution of the control (for example, if the control is a volume control, then this would specify if an increase in volume occurs in single units or in units of two or any other value). The returned structure also includes a bitmap of the channels that support the feature control, and, for each supported channel, it provides the current value of the feature control. Based on this information, an application can set the current feature control value for one or all supported channels by issuing the ioctl `OSPLUS_IOCTL_USB_AUDIO_SET_FEATURE`.

Setting the audio format and sample rate

To transfer audio data to or from the audio device, the application must always use a USB terminal and specify the format of the audio data before the transfer can begin. Typically, the format is 16-bit stereo PCM samples at a 48KHz sampling rate. Because the USB terminal device can only provide a limited number of audio formats, the device must first be queried to return a list of supported formats so that one can be chosen from the list.

Before retrieving the list of audio formats, the application needs to determine the size of the list; this is returned by the ioctl `OSPLUS_IOCTL_USB_AUDIO_GET_FORMAT_COUNT`. The application can use this value to allocate sufficient buffer space to hold all the audio formats in one single list. The list itself is returned by the ioctl

OSPLUS_IOCTL_USB_AUDIO_GET_FORMAT_LIST. The application can then search this list to find an appropriate format for transferring the data.

The audio formats listed may either be individual audio formats (such as 8-bit mono PCM) at specific discrete sampling rates (such as 8KHz or 16KHz), or the audio format may simply specify a minimum and maximum sample rate. In this latter case, almost any sampling rate between the minimum and maximum is valid.

When the application has chosen an audio format from the list, it can set the audio format by issuing the ioctl OSPLUS_IOCTL_USB_AUDIO_SET_FORMAT. At any point, the application can get the audio format currently set by issuing the ioctl OSPLUS_IOCTL_USB_AUDIO_GET_FORMAT.

At this time only type 1 audio formats as outlined in the USB class definition for Audio Data Formats are supported. Normally, most simple audio devices only support the type 1 PCM audio format.

Transferring audio to and from audio devices

The application must set the audio format to use before audio data can be transferred to or from a USB terminal. This enables some basic calculations to be performed so the application knows the frequency and the amount of data that it needs to send to the USB terminal.

Audio can be transferred to and from an USB terminal in blocks of any size because the data is coalesced internally in a staging buffer. The application uses the functions `device_read()` and `device_write()` to transfer the data to and from the staging buffer. Care should be taken to ensure that, when playing back audio, the staging buffer is continually fed with data, and when recording audio the staging buffer is continually being drained. Failure to do this can cause the staging buffer to completely empty or to fill and cause glitches in the audio. As a general indication, the staging buffer holds approximately 500ms of audio at the currently set sampling rate. The status of the audio transfer engine can be monitored at any time and the buffer state obtained by using the ioctl OSPLUS_IOCTL_USB_AUDIO_TRANSFER_STATUS.

Before audio playback, it is wise to start buffering audio first to ensure smooth playback using the `device_read` function as the audio data drains quickly. After some initial audio data has been buffered, the audio transfer engine should be started to begin playback using the ioctl OSPLUS_IOCTL_USB_AUDIO_TRANSFER_START. In the case of audio capture, just issue the ioctl.

When the application has completed all audio capture or playback, it issues the ioctl OSPLUS_IOCTL_USB_AUDIO_TRANSFER_STOP. After issuing this ioctl, outstanding audio data may remain in the audio transfer engine internal buffers, and the application may still buffer more audio in the case of playback. If the application wishes to purge the internal buffer then it must issue the ioctl OSPLUS_IOCTL_USB_AUDIO_TRANSFER_PURGE. This ioctl removes all audio from the audio transfer engine internal buffers.

The application can get the current status of the progress of the audio transfer at any time with the ioctl OSPLUS_IOCTL_USB_AUDIO_TRANSFER_STATUS. This ioctl returns the audio transfer engine status, and internal buffer statistics.

All of the audio class ioctls are described in *Section 3.2: USB audio ioctl definitions on page 90*.

2.9.2 USB communications class devices

The usage model for all of the supported communication class devices varies depending on whether the device keeps to either the ACM or the ENCM communications class protocol. Despite this there are several commonalities in the use of each of the device types, through the support of some OSPlus communications class ioctls.

Each device can be opened by multiple tasks, as either, read, write, or read write. Once open, a callback can be registered with the OSPlus communications class ioctl `OSPLUS_IOCTL_USB_COMM_SET_LISTENER` that is executed whenever the state of the communications device changes, such as a network connection speed change, and so on.

Device-specific formatted commands can be sent to the communications class device using the ioctl `OSPLUS_IOCTL_USB_COMM_SEND_COMMAND` and a response obtained with the `OSPLUS_IOCTL_USB_COMM_GET_RESPONSE` ioctl.

The capabilities of the communications class device can be queried using the `OSPLUS_IOCTL_USB_COMM_GET_CAPABILITY` ioctl. The structures passed-in to the command are specific to the device protocol.

Each communications class device is read from or written to using the standard OSPlus `device_read()` or `device_write()` functions. By default, reads and writes are blocking with a timeout of five seconds. The standard OSPlus ioctls allow this to be changed to non-blocking or a different timeout as required.

After each read or write it is important to check the return code for error. On error `errno` is set. If `errno` is set to `ENODEV`, the communications class device has been unplugged from the USB and all further reads and writes are prevented. Only then can the device be closed.

Before the communications class device is closed, all reads and writes should be completed and any new reads or writes prevented. Attempting to close the device while a read or write is in progress aborts the read or write.

All reads from and writes to a communications class device are staged through internal buffers. There are therefore no alignment or cache line restrictions on the buffers passed to `device_read()` and `device_write()`. However, all reads and writes incur a small overhead for copying the data.

All of the communication class ioctls are described in *Section 3.3: USB communications ioctl definitions on page 102*.

USB communications class ACM devices

Communication class ACM devices can be thought of as devices that are controlled through serial interfaces, such as analogue modems.

Once successfully opened, a communications class ACM device is configured with a baud rate, data format, parity and number of data bits. This is done by using the OSPlus ACM ioctls.

After being configured, a communications class ACM device is then read or written to much like any other serial interface. Note that ACM devices only read in 64-byte multiples.

For information on what ioctls can be issued to communication class ACM devices, see *Section 3.4: USB ACM ioctl definitions on page 106*.

USB communications class ENCM devices

Communication class ENCM devices are devices primarily intended for use with a TCP/IP stack. They are similar in use to USB ethernet devices and the OSPlus LAN91 and LAN9x1x drivers, but with some differences.

A small amount of setup is needed for communications class ENCM devices to use them effectively with a TCP/IP stack. The device has a hardware ethernet MAC address associated with it that can be read using the OSPlus ENCM ioctl `OSPLUS_IOCTL_USB_ENCM_GET_MAC` or the Ethernet ioctl `OSPLUS_IOCTL_ETH_GET_MAC`. A TCP/IP stack also usually needs to know how big a packet the device can send. This information is returned, along with filtering information when querying the device capabilities.

To efficiently read and write TCP/IP data to the communications class ENCM device, all reads and writes must be a full TCP/IP data packet. When reading a TCP/IP packet, a buffer large enough to support the maximum device packet size must be supplied.

When reading ethernet packets from a communications class ENCM device for processing by a TCP/IP stack, STMicroelectronics recommend the read timeout is set to the OSPlus timeout value of `TIMEOUT_INFINITY` and a dedicated task used to perform the read. This way packets are read as they arrive without the need to poll the device and consume valuable CPU time.

Communications class ENCM devices have various filtering options. These include multicast filtering, power management filtering and packet filtering. Not all communication class ENCM devices support all options.

Communications class ENCM devices also support a limited subset of the OSPlus ethernet ioctls. This allows them to be used as if they were USB ethernet devices. For more details see *Section 2.9.3: USB ethernet devices*.

For a list of the ioctls that can be issued, see *Table 58: Communications ENCM/RNDIS device supported OSPlus ethernet ioctls on page 63*.

USB communications class RNDIS devices

Communication class RNDIS devices are devices primarily intended for use with a TCP/IP stack. They are similar in use to USB ethernet devices and the OSPlus LAN91 and LAN9x1x drivers, but with some differences.

A small amount of setup is needed for communications class RNDIS devices to use them effectively with a TCP/IP stack. The device has a hardware ethernet MAC address associated with it that can be read using the OSPlus Ethernet ioctl `OSPLUS_IOCTL_ETH_GET_MAC`. A TCP/IP stack also usually needs to know how big a packet the device can send. This information is returned, along with filtering information when querying the device capabilities.

To efficiently read and write TCP/IP data to the communications class RNDIS device, all reads and writes must be a full TCP/IP data packet. When reading a TCP/IP packet, a buffer large enough to support the maximum device packet size must be supplied.

When reading ethernet packets from a communications class RNDIS device for processing by a TCP/IP stack, STMicroelectronics recommend the read timeout is set to the OSPlus timeout value of `TIMEOUT_INFINITY` and a dedicated task used to perform the read. This way packets are read as they arrive without the need to poll the device and consume valuable CPU time.

Communications class RNDIS devices have various filtering options. These include multicast filtering, power management filtering and packet filtering. Not all communication class RNDIS devices support all options.

Communications class RNDIS devices also support a limited subset of the OSPlus ethernet ioctls. This allows them to be used as if they were USB ethernet devices. For more details see *Section 2.9.3: USB ethernet devices*.

For a list of the ioctls that can be issued, see *Table 58: Communications ENCM/RNDIS device supported OSPlus ethernet ioctls on page 63*.

2.9.3 USB ethernet devices

The USB ethernet devices are intended for use with a TCP/IP stack. The usage model for USB ethernet devices is almost identical to that for the OSPlus LAN91 and LAN9x1x ethernet drivers, but with some minor differences. As the ethernet device cannot yet be attached to the USB, it is useful to block waiting on the device before attempting to open it. This can be done using the `device_open_timeout()` function and passing a timeout value of `TIMEOUT_INFINITY`.

To use the driver effectively, a small amount of setup is usually performed. This involves reading the hardware MAC address using the OSPlus ethernet ioctl `OSPLUS_IOCTL_ETH_GET_MAC` and registering it with the TCP/IP stack for use when processing TCP/IP packets. Alternatively, the ethernet device can support user specified MAC hardware addresses and an address can be set using the OSPlus ethernet ioctl `OSPLUS_IOCTL_ETH_SET_MAC`.

The user can also specify read and write timeout values. These are set using the OSPlus common ioctls `OSPLUS_IOCTL_SET_RD_TIMEOUT` and `OSPLUS_IOCTL_SET_WR_TIMEOUT` respectively. By default, the read and write functions are set to block with a timeout of five seconds.

The ethernet device supports packet filtering options of promiscuous mode, accept all broadcasts, accept all multicasts, and accept multicasts matching the multicast filters. The filtering options can be set using the OSPlus ethernet ioctl `OSPLUS_IOCTL_ETH_SET_PKT_FILTER`. By default the ethernet device opens in promiscuous mode.

The multicast filter hash tables can be set directly by passing a pre-calculated hash table as a parameter to the OSPlus ethernet ioctl `OSPLUS_IOCTL_ETH_SET_MCAST_HASH_TABLE`. Alternatively, the multicast filter hash table can be set by passing a list of multicast addresses to the OSPlus ethernet ioctl `OSPLUS_IOCTL_ETH_SET_MCAST_FILTERS`. This ioctl calculates the hash table from the list of multicast addresses passed-in, then writes the table to the hardware.

STMicroelectronics recommend that a dedicated task is used to read TCP/IP packets from the USB ethernet device, and that the read operation blocks indefinitely. This ensures all TCP/IP packets are read as they arrive without polling the device and consuming valuable CPU time. On each read operation there is no way to determine the size of the TCP/IP packet that is read. Therefore, the `device_read()` function must be supplied with a buffer large enough to hold the maximum sized packet. To determine the maximum sized packet, issue the OSPlus ethernet ioctl `OSPLUS_IOCTL_ETH_GET_INFO`. On a successful, read the size of the packet is returned. The task then passes the packet to the TCP/IP stack for processing, before attempting to read the next packet.

All TCP/IP packets are written to the USB ethernet device using the `device_write()` function. Before the ethernet device can be closed, all further reads and writes should be prevented. Attempting to close the ethernet device while a read or write is in progress causes the read or write to fail.

All reads from and writes to an ethernet class device are staged through internal buffers. This causes no alignment or cache line restrictions on the buffers passed to `device_read()` and `device_write()`. However, these reads and writes incur a small overhead for copying the data.

After any read from, or write to the ethernet device, it is important to check for an error (`errno` is set). When `errno` is set to `ENODEV` the ethernet device has been unplugged from the USB and all further reads and writes are prevented and the ethernet device is closed.

The status of the ethernet driver can be retrieved using the OSPlus ethernet ioctl `OSPLUS_IOCTL_ETH_GET_STATUS`. The status indicates whether the ethernet device is enabled, has a valid link and if it is in loopback mode. On any read or write error it is advisable to query the status in order to determine if the error was due to the link being down, in which case all further reads and writes can be suspended until the link is back up.

For details of supported USB ethernet ioctls, see *Section 2.20.2: USB ethernet device ioctls on page 62*.

2.9.4 USB HID devices

Each USB HID device is used in a way specific to the device HID type.

USB HID joystick devices

USB HID joystick devices can only be opened for exclusive access by only one task at a time. Attempting to open a HID joystick device more than once returns an error through `errno`.

When a HID joystick device has been opened, the features it supports should be queried using the OSPlus common ioctl `OSPLUS_IOCTL_GET_FEATURES`. The returned features bitmap can then determine the data to query and process.

Normally when using the HID joystick, the task that reads the HID joystick data should block while waiting for that data to change. To do this the OSPlus common ioctl `OSPLUS_IOCTL_WAIT` is supported. This ioctl should be passed a standard OSPlus timeout value. When the ioctl returns with the success code, each supported HID joystick axis, hat switch, or slider can be queried and the buttons read using the OSPlus joystick ioctls.

The HID joystick data can be polled for changes, but this is not as efficient as using task and blocking until there are changes to process.

For details of all supported USB HID joystick ioctls, see *USB HID joystick device ioctls on page 63*.

USB HID keyboard devices

USB HID keyboard devices can only be opened for exclusive access by only one task at a time. Attempting to open the HID keyboard device more than once results in an error being returned through `errno`.

When the USB keyboard device has been opened, key presses are read using the OSPlus keyboard ioctls. If the key press can be represented with an ASCII code, the ASCII code of the key press is returned. If the key press is for a function key or can be a multimedia key, the USB usage identifier is returned.

To prevent continuous polling of the keyboard, the OSPlus common ioctl `OSPLUS_IOCTL_WAIT` is supported. This ioctl should be passed a standard OSPlus timeout value. When the ioctl returns with the success code, the number of key presses in the internal buffers can be queried using the OSPlus keyboard ioctl `OSPLUS_IOCTL_KEYBOARD_COUNT` and each one required read using the OSPlus keyboard ioctl `OSPLUS_IOCTL_KEYBOARD_GETKEY`.

Unwanted key presses in the internal buffers can be flushed using the OSPlus common ioctl `OSPLUS_IOCTL_FLUSH`.

For details of all supported USB HID keyboard ioctls, see *USB HID keyboard device ioctls on page 64*.

USB HID mouse devices

USB HID mouse devices can only be opened for exclusive access by one task at a time. Attempting to open the HID mouse device more than once results in an error returned through `errno`.

When an HID mouse device has been opened, the features it supports should be queried using the OSPlus common ioctl `OSPLUS_IOCTL_GET_FEATURES`. The features bitmap returned can then be used to determine the data to query and process.

Normally when using the HID mouse, to prevent continuous mouse cursor updates to the screen, the HID mouse data should block waiting for that data to change. To do this the OSPlus common ioctl `OSPLUS_IOCTL_WAIT` is supported. This ioctl should be passed a standard OSPlus timeout value. When the ioctl returns with the success code, each supported HID mouse axis, or wheel can then be queried and the buttons read using the OSPlus mouse ioctls. The data obtained is then be processed and any cursor update to the screen performed.

The HID mouse data can be polled for changes, but this is not as efficient as using task and blocking until there are changes to process.

When the HID mouse device is opened for the first time, the minimum and maximum range for each axis or wheel control are set to that specified in the USB HID mouse descriptors (usually -127 to 127). This can be overridden using set bounds ioctls.

For details of all supported USB mouse ioctls see *USB HID mouse device ioctls on page 64*.

USB HID generic devices

Due to the limited functionality of USB HID generic devices, they are not restricted to being opened only once. When a generic HID device has been opened, only the common ioctls for returning vendor and product information for the device are supported.

For details of all supported USB generic ioctls see *USB HID generic device ioctls on page 66*.

2.9.5 USB host-to-host devices

Assuming a USB host-to-host device has both ends plugged in, several handshake operations must be performed before data can be successfully sent and received using the device.

On first opening the host-to-host device, a “peer connected” handshake request is issued to inform the other end of the device (the peer) that something is connected. Immediately following this, a further “ready to transfer data” handshake request is issued, indicating this end of the cable is ready to transmit and receive. At this point, the open call issues reset requests to the device to clear the input and output buffers.

After successfully opening the host-to-host device, the application should issue the OSPlus host-to-host ioctl `OSPLUS_IOCTL_USB_H2H_WAIT_PEER_E` passing in a standard OSPlus timeout. This blocks the calling task until a peer connected to the other end of the host-to-host cable has acknowledged the “peer connected” handshake request.

The application should then next issue OSPlus host-to-host ioctl `OSPLUS_IOCTL_USB_H2H_WAIT_TX_RDY` ioctl passing in a standard OSPlus timeout. This blocks the calling task until the connected peer has acknowledged the “ready to transfer data” handshake. The input and output buffers are reset automatically as required.

A peer is now connected and ready to transmit data. Data is read and written using the standard device read and write function calls. By default, all read and write operations are blocking with a timeout of five seconds. This can be changed to non-blocking or to a different timeout using the standard OSPlus ioctls.

After each read or write it is important to check the return code. On error, `errno` is set. If `errno` is set to `ENODEV`, the host-to-host class device has been unplugged from the USB and all further reads and writes are prevented. The device can now only be closed.

If the peer is disconnected, or the “transferred all data” handshake is issued while data transfers are still in progress, any outstanding read or write is aborted.

All reads from and writes to a host-to-host device are handled in the most efficient manner. When the buffer passed-in is determined to be in uncached memory and 32-bit aligned, it is used directly with no staging. When the buffer is not in uncached memory, but is cache line aligned, the buffer is purged from the cache and used directly without staging. In all other cases, the buffer must be staged through an internal buffer. So for maximum efficiency, all read and write buffers should be 32-bit aligned and located in a region of uncached memory, registered with the OSPlus uncached memory manager. If this is not possible, buffers should at least be allocated with the cache safe allocator. Refer to the *Cache management* section in the *OSPlus support library* chapter of the *OSPlus User Manual (ADCS 7702033)*.

When the host-to-host device is connected to a USB v1.1 system, the application writer should be aware that writes of exactly 64 bytes have an extra byte of padding added onto the end of the transferred data. This is a workaround to a hardware problem with the Prolific 2301 and 2302 chip sets. Similarly on USB v2.0 systems writes of exactly 512 bytes also have a byte of padding added on the end of the data transfer.

When all reads and writes are completed, the device should be closed. If the peer is still connected, a “transferred all data” handshake is issued to inform the peer that no more data is to be transferred.

To obtain a greater throughput over a host-to-host device, larger packets should be sent. Small packets of 1 to 2 Kbytes drastically reduce the achievable transfer rate.

For details of all supported USB host-to-host ioctls. See *Section 3.6: USB host-to-host ioctl definitions on page 120*.

2.9.6 USB hub devices

USB hub devices function without any form of user intervention. The only time a hub device would be used directly, is when performing low-level port testing. In this case, when a hub device has been opened, only those ioctls that set a hub or hub port feature, clear a hub or hub port feature, or return a hub or hub port status can be issued.

For details of the supported USB hub ioctls, see *Section 3.8: USB hub ioctl definitions on page 123*.

2.9.7 USB mass storage devices

USB mass storage devices are logically separated into two types - fixed and removable. The mass storage device name prefix `/usb/hdisk` identifies it as fixed. The name prefix `/usb/rdisk` identifies it as removable.

After successfully opening the mass storage device, the number of data blocks the device contains and the block size are required. These are retrieved using the OSPlus common ioctls `OSPLUS_IOCTL_NUM_BLOCKS` and `OSPLUS_IOCTL_BLOCK_SIZE` respectively.

Data can then be read or written using the `device_block_read()` and `device_block_write()` functions. The number of blocks on the device is used to ensure the read or write does not exceed the number of blocks on the device.

To work effectively with removable media devices, the media status of the device must be obtained. It is queried using the OSPlus common ioctl `OSPLUS_IOCTL_CHECK_MEDIA`. The result determines whether the removable device contains any media.

When writing to mass storage devices, if the media is write protected, the write fails. The media is checked for write protection using the OSPlus common ioctl `OSPLUS_IOCTL_WR_PROTECTED`.

All reads from and writes to a mass storage device must be a multiple of the device block size. Failure to ensure this will result in the read or write operation failing.

After each read or write it is important to check the return code for error. On error `errno` is set. If `errno` is set to `ENODEV`, the mass storage class device has been unplugged from the USB and all further reads and writes are prevented. Only then can the device be closed.

For details of all supported USB mass storage ioctls see *Section 2.20.4: USB mass storage device ioctls on page 66*.

2.9.8 USB serial devices

The USB serial class devices are used in an almost identical manner to the OSPlus serial driver. Most of the same ioctls are supported by the USB serial class driver with the following exceptions. The USB serial class driver does not support flushing or purging of buffers and does not provide internal buffering.

For details of all supported USB serial class ioctls see *Section 2.20.5: USB serial device ioctls on page 67*. Refer to the *OSPlus User Manual (ADCS 7702033)* for details on non-USB serial devices.

2.9.9 USB VOIP devices

USB VOIP (voice over IP) devices are composite devices that typically consist of an audio component to handle audio capture and playback, and other components to manage the keypad and display. The USB VOIP class driver leaves the audio component management to the USB audio class driver and takes care of the display and keypad only.

When opening the VOIP device, the application gets the device information by executing the ioctl `OSPLUS_IOCTL_USB_VOIP_GET_INFO`. The information returned includes:

- the VOIP phone model
- the type of display
- the name of the audio capture device for the microphone
- the name of the audio playback device for the speaker

The phone microphone and speaker are handled as any other USB audio device, as described in *Section 2.9.1: USB audio class devices on page 22*.

When the device is open, the application can customize how the user interacts with the VOIP phone. A default ring tone is already set for the device, and the volume of this ring tone can be changed by supplying a value between 0 and 255 to the ioctl `OSPLUS_IOCTL_USB_VOIP_SET_RING_VOLUME`. If required, a new ring tone can be set on the phone by passing a structure containing a list of frequencies to play and the duration that each frequency should be played to the ioctl `OSPLUS_IOCTL_USB_SET_RING_STYLE`. The application can turn the phone ringer on and off by issuing the ioctl `OSPLUS_IOCTL_USB_VOIP_SET_RING_TONE`. The application can also silently alert the user by flashing the phone LED on and off; this is done with the ioctl `OSPLUS_IOCTL_USB_VOIP_SET_LED`.

The application can accept keypresses from the phone. Each keypress is returned to the user as both a decoded value for the key (for example, pickup key, hang-up key and so forth) and also as a raw value received from the phone. The purpose of returning this raw keypress is so that even if the attached VOIP phone is not supported, it can still be possible to get some meaningful keypress data. The application, however, would need to be able to correctly decode the keypress. Keypresses are read by the ioctl `OSPLUS_IOCTL_USB_VOIP_GET_KEYPRESS`.

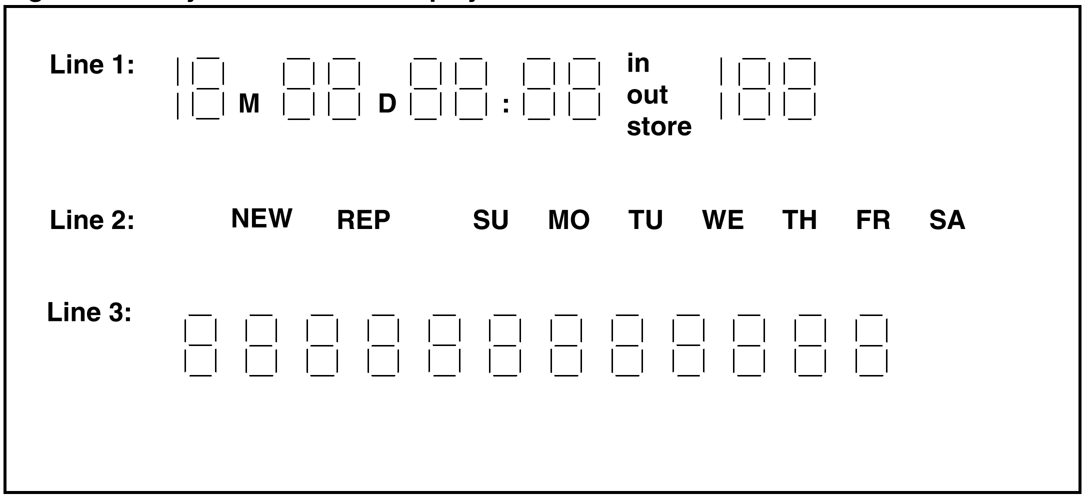
The VOIP phones currently support two different types of display: a seven segment LCD monochrome display or a 104x48 pixel monochrome display. The type of display the phone possesses is returned as part of the information obtained with the ioctl `OSPLUS_IOCTL_USB_VOIP_GET_INFO`.

Seven-segment LCD display

The Seven-segment LCD display consists of three lines to display alphanumeric characters and hard coded symbols. For a comprehensive description of seven-segment displays, see *http://en.wikipedia.org/wiki/Seven_segment_display*.

The LCD display consists of three lines. The first line is capable of displaying a combination of limited alphanumeric characters and symbols, the second line only displays symbols, and the third line only displays alphanumeric characters. The layout of the display is shown in *Figure 4*:

Figure 4. Layout of the LCD display



To program the LCD display, the application supplies a string and the offset at which it starts on the display. Line 1 starts at offset 0, line 2 at offset 17, and line 3 at offset 26. Each character in the string provided represents either one seven-segment display item or a symbol. The LCD display attempts to map each character in the string to the corresponding seven-segment display item or symbol. In cases where the character in the string represents a symbol, use a period character to display the symbol. Any other character clears the symbol.

For example, if the application has to write the date Monday 25th December 15:00 and 20 seconds, it writes the string "`12.25.15.00      20`" to line 1 and string "`     .        `" to line 2.

Monochrome graphics display

Certain models of Yealink VOIP phones have a 104x48 pixel monochrome display. This display should be treated as one large bitmapped buffer, with each bit of the buffer corresponding to a pixel on the screen. The screen is set up so that pixel 0,0 is at the bottom right of the phone display and pixel 103,47 is at the top left of the phone display. To simplify management of the display, there are only two graphic display functions. The first, using ioctl `OSPLUS_IOCTL_USB_VOIP_LCD_CLEAR`, completely clears the display; and the second, using `OSPLUS_IOCTL_USB_VOIP_LCD_PIXEL_BLIT`, carries out a blit (bit block transfer) on a buffer, sending the contents directly to the display.

To use this facility efficiently, the application maintains an offscreen buffer and updates it as required. When the buffer has been updated, the application blits either part or all of this buffer to the phone.

Two macros are available to help set and clear pixels in the offscreen buffer. These are `OSPLUS_USB_VOIP_PIXEL_SET()` and `OSPLUS_USB_VOIP_PIXEL_CLR()`.

All of the VOIP class ioctls are described in *Section 3.9: USB VOIP ioctl definitions on page 125*.

2.9.10 USB unsupported devices

Unsupported USB devices are managed by the USB proxy class driver to allow third party drivers to be developed. This section outlines the basic steps involved in creating a third party driver for an unsupported device. Examples of third party drivers are shipped with the OSPlus USBLINK package, such as `disk_driver`, `mouse_driver` and `serial_driver`.

Third party class drivers usually consist of three main functions and various other ancillary functions. These main functions are a probe function, a start function and a stop function.

The probe function

The entry point to a third party driver is the probe function. This should be called each time a new device is created in the OSPlus device library for any devices that have a type of `OSPLUS_DEVICE_TYPE_USB_UNKNOWN` and a name beginning with the prefix `/usb/dev`. The easiest way to call the probe function is to register a device library callback to be executed each time the device is registered or unregistered with the OSPlus device library. Then, call the probe function when the appropriate device creation messages are identified.

The probe function is used to determine if the device is a supported USB device and if the third party driver is capable of supporting it. The probe function should determine this by opening the USB device and retrieving whatever device, configuration, interface or endpoint details are required using the ioctls `OSPLUS_IOCTL_USB_GET_DEVICE_INFO`, `OSPLUS_IOCTL_USB_GET_CONFIGURATION`, `OSPLUS_IOCTL_USB_GET_INTERFACE` and `OSPLUS_IOCTL_USB_GET_ENDPOINT`.

Determining if the driver can support a USB device is often as simple as checking the vendor and product identifiers in the device information. Alternatively, all configurations and interfaces supported by the device can be searched.

The third party driver start function can be called, when the USB device has been identified as supported.

The start function

The start function has the following main responsibilities.

1. Allocate and initialize any third party driver internal state.
2. Correctly configure the USB device.
3. Setup any transfers to be performed on particular endpoints.
4. Register an instance of the third party class driver with the OSPlus device library.
5. Register an extraction callback with the USB driver when the USB device is removed.

The third party driver internal state should consist of at least a device handle for the USB device. This allows commands to be sent to the USB device without opening the USB device every time.

Correctly configuring the USB device can involve setting a USB configuration with the ioctl `OSPLUS_IOCTL_USB_SET_CONFIGURATION` or setting a USB interface with the ioctl `OSPLUS_IOCTL_USB_SET_INTERFACE`. Alternatively, control transfers can be sent to the USB device to enable some functionality.

Setting up any transfers to be performed on particular endpoints usually involves searching through the USB device interface looking for a matching endpoint. When found a transfer is created with the necessary parameters set. An example of this is searching for an interrupt in endpoint on the USB device. When found a new interrupt transfer is created and hooked to the interrupt endpoint with the ioctl `OSPLUS_IOCTL_USB_INTR_TRANSFER`.

To allow a user application to use the USB device, the third party driver should be registered with the OSPlus device library using `dev_instance_create()`. The third party driver should export an appropriate interface to the USB device. This can consist of an ioctl interface by itself that maps ioctls to USB transfers performed on the USB device. Additionally, it can include device read and write operations that map onto bulk transfers supported by the USB device.

When the USB device is removed, it is important that an extraction callback is registered with the USB driver using ioctl `OSPLUS_IOCTL_USB_SET_EXTRACT_CB`. This is the third party driver stop function.

The stop function

The third party driver stop function has four main responsibilities.

1. Unregister the instance of the third party driver with the OSPlus device library.
2. Abort any outstanding transfers.
3. Free any internal driver state.
4. Close the USB device.

By unregistering the third party driver with the OSPlus device library with the `dev_instance_delete()` function, further attempts to open and use the third party driver instance are prevented.

Aborting outstanding transfers results in an error being propagated to the user application, therefore the user application must always check return codes. Additionally, it is useful for the third party driver to set `errno` to an appropriate value. For example, to indicate the USB device was extracted. The third party driver must also free any internal state that it has allocated. Failure to do this causes memory leaks.

Finally, the stop function must close the USB device. Failure to do so results in a ghost device persisting through the life of the system. Whilst this is not harmful, if a third party driver is being continually started and stopped, the memory consumed by each ghost device can add up to a significant amount.

For details of all supported USB proxy class ioctls see *Section 2.20.6: USB unsupported device ioctls on page 67*.

2.10 Device library header file: ioctls/usb.h

The device library defines a set of ioctls that are used by USB devices. These ioctls are in the file `ioctls/usb.h`. For non USB ioctl definitions refer to the *OSPlus User Manual (ADCS 7702033)*.

Table 4. Ioctls in ioctls/usb.h

Ioctls	Description
`OSPLUS_IOCTL_USB_SET_EXTRACT_CB`	Install a USB device extract callback
`OSPLUS_IOCTL_USB_GET_EXTRACT_CB`	Retrieve an installed USB device extract callback
`OSPLUS_IOCTL_USB_CLR_EXTRACT_CB`	Remove an installed USB device extract callback
`OSPLUS_IOCTL_USB_GET_DEVICE_INFO`	Get the USB device information
`OSPLUS_IOCTL_USB_GET_CONFIGURATION`	Get the USB device specified configuration information
`OSPLUS_IOCTL_USB_GET_INTERFACE`	Get the USB device specified interface information
`OSPLUS_IOCTL_USB_GET_ENDPOINT`	Get the USB device specified endpoint information

Table 4. Ioctls in ioctls/usb.h

Ioctls	Description
`OSPLUS_IOCTL_USB_SET_CONFIGURATION`	Set the USB device configuration
`OSPLUS_IOCTL_USB_SET_INTERFACE`	Set the USB device interface
`OSPLUS_IOCTL_USB_CTRL_TRANSFER`	Perform a USB control transfer
`OSPLUS_IOCTL_USB_ISOC_INFO`	Get the neccessary information to perform a USB isochronous transfer
`OSPLUS_IOCTL_USB_ISOC_TRANSFER`	Perform a USB isochronous transfer
`OSPLUS_IOCTL_USB_BULK_TRANSFER`	Perform a USB bulk transfer
`OSPLUS_IOCTL_USB_INTR_TRANSFER`	Perform a USB interrupt transfer
`OSPLUS_IOCTL_USB_ABORT_TRANSFER`	Abort a USB transfer
`OSPLUS_IOCTL_USB_HOST_GET_PORTS`	Return the number of root hub ports supported by the controller
`OSPLUS_IOCTL_USB_HOST_TEST_MODE`	Put a root hub port in to a USB test mode
`OSPLUS_IOCTL_USB_PORT_INFO`	Return information about the host controller and root hub port to which the device is connected
`OSPLUS_IOCTL_USB_TEST_PORT`	Place the root hub port, to which the device is connected, into the specified test mode
`OSPLUS_IOCTL_USB_COMPOSITE_INFO`	Return composite device name and type information
`OSPLUS_IOCTL_USB_TREE_NODE_INFO`	Return device parent device/port and child device/port information

Table 5. Types defined in ioctls/usb.h

Types	Description
`usb_extract_callback_t`	USB extract callback structure
`usb_device_info_t`	USB device information structure
`usb_port_info_t`	USB port information structure
`usb_configuration_t`	USB device configuration structure
`usb_interface_t`	USB device interface structure
`usb_endpoint_t`	USB device endpoint structure
`usb_ctrl_transfer_t`	USB control transfer structure
`usb_isoc_info_t`	USB isochronous information structure
`usb_isoc_transfer_t`	USB isochronous transfer structure
`usb_bulk_transfer_t`	USB bulk transfer structure
`usb_intr_transfer_t`	USB interrupt transfer structure
`usb_abort_transfer_t`	USB transfer abort structure
`usb_port_info_t`	USB port information structure

Table 5. Types defined in ioctls/usb.h

Types	Description
`usb_composite_info_t`	USB composite device information structure
`usb_tree_node_info_t`	USB tree node information structure

Table 6. USB defines in ioctls/usb.h

Define	Description
Device information defines	
`USB_SPEED_LOW`	USB low speed (1.5 Mbps) device
`USB_SPEED_FULL`	USB full speed (12 Mbps) device
`USB_SPEED_HIGH`	USB high speed (480 Mbps) device
Device configuration defines	
`USB_SELF_POWERED`	USB device is self-powered
`USB_REMOTE_WAKEUP`	USB device supports remote wake-up
Device endpoint direction defines	
`USB_DIRECTION_IN`	USB device in endpoint
`USB_DIRECTION_OUT`	USB device out endpoint
Device endpoint transfer type defines	
`USB_TRANSFER_TYPE_CONTROL`	Control transfer
`USB_TRANSFER_TYPE_ISOCHRONOUS`	Isochronous transfer
`USB_TRANSFER_TYPE_BULK`	Bulk transfer
`USB_TRANSFER_TYPE_INTERRUPT`	Interrupt transfer
Device endpoint usage type defines	
`USB_USAGE_TYPE_DATA`	Data usage type
`USB_USAGE_TYPE_FEEDBACK`	Feedback usage type
`USB_USAGE_TYPE_IMPLICIT_FEEDBACK`	Implicit feedback usage type
`USB_USAGE_TYPE_RESERVED`	Reserved usage type
Device endpoint sync type defines	
`USB_SYNC_TYPE_NONE`	No isochronous sync type
`USB_SYNC_TYPE_ASYNCHRONOUS`	Asynchronous isochronous sync type
`USB_SYNC_TYPE_ADAPTIVE`	Adaptive isochronous sync type
`USB_SYNC_TYPE_SYNCHRONOUS`	Synchronous isochronous sync type
Request type defines	
`USB_REQUEST_TYPE_STANDARD`	Standard request type
`USB_REQUEST_TYPE_CLASS`	Class specific request type
`USB_REQUEST_TYPE_VENDOR`	Vendor specific request type
`USB_REQUEST_TYPE_RESERVED`	Reserved request type

Table 6. USB defines in ioctls/usb.h (continued)

Define	Description
Recipient type defines	
`USB_RECIPIENT_DEVICE`	USB device is transfer recipient
`USB_RECIPIENT_INTERFACE`	USB interface is transfer recipient
`USB_RECIPIENT_ENDPOINT`	USB endpoint is transfer recipient
`USB_RECIPIENT_OTHER`	Other transfer recipient
Request defines	
`USB_REQUEST_GET_STATUS`	USB get status request
`USB_REQUEST_CLEAR_FEATURE`	USB clear feature request
`USB_REQUEST_SET_FEATURE`	USB set feature request
`USB_REQUEST_SET_ADDRESS`	USB set address request
`USB_REQUEST_GET_DESCRIPTOR`	USB get descriptor request
`USB_REQUEST_SET_DESCRIPTOR`	USB set descriptor request
`USB_REQUEST_GET_CONFIGURATION`	USB get configuration request
`USB_REQUEST_SET_CONFIGURATION`	USB set configuration request
`USB_REQUEST_GET_INTERFACE`	USB get interface request
`USB_REQUEST_SET_INTERFACE`	USB set interface request
`USB_REQUEST_SYNCH_FRAME`	USB synch frame request
Descriptor defines	
`USB_DESCRIPTOR_DEVICE`	USB device descriptor
`USB_DESCRIPTOR_CONFIGURATION`	USB configuration descriptor
`USB_DESCRIPTOR_STRING`	USB string descriptor
`USB_DESCRIPTOR_INTERFACE`	USB interface descriptor
`USB_DESCRIPTOR_ENDPOINT`	USB endpoint descriptor
Feature defines	
`USB_FEATURE_ENDPOINT_HALT`	USB endpoint halt feature
`USB_FEATURE_DEVICE_REMOTE_WAKEUP`	USB device remote walkup feature
`USB_FEATURE_TEST_MODE`	USB test mode feature
Transfer status code defines	
`USB_ERROR_NONE`	No error
`USB_ERROR_CRC`	CRC error
`USB_ERROR_BIT_STUFFING`	Bit stuffing error
`USB_ERROR_TOGGLE_MISMATCH`	Toggle mismatch error
`USB_ERROR_STALL`	Stall error
`USB_ERROR_DEV_NO_ANSWER`	Device not answering error
`USB_ERROR_PID_FAILURE`	PID failure error

Table 6. USB defines in ioctls/usb.h (continued)

Define	Description
`USB_ERROR_BAD_PID`	Bad PID error
`USB_ERROR_DATA_OVERRUN`	Data overrun error
`USB_ERROR_DATA_UNDERRUN`	Data underrun error
`USB_ERROR_BUFFER_OVERRUN`	Buffer overrun error
`USB_ERROR_BUFFER_UNDERRUN`	Buffer underrun error
`USB_ERROR_NOT_ACCESSED`	Not accessed error
`USB_ERROR_NAK`	NAK error
`USB_ERROR_BABBLE`	Babble error
`USB_ERROR_ISOC_SYNC_OUT`	Isochronous sync out error
Transfer return code defines	
`USB_TRANSFER_DONE`	USB transfer is complete
`USB_TRANSFER_REHOOK`	USB transfer should be re-hooked

Table 7. USB control transfer macros in ioctls/usb.h

Macro	Description
`USB_VALUE_INDEX`	Return the USB control transfer index value
`USB_VALUE_DESCRIPTOR`	Return the USB control transfer descriptor value
`USB_VALUE`	Build the USB control transfer value from an index and an descriptor value

2.11 Device library header file: ioctls/usb/audio.h

The device library defines a set of ioctls that are used by USB audio class devices. These ioctls are in the file `ioctls/usb/audio.h`. For the ioctl definitions, see *Section 3.2: USB audio ioctl definitions on page 90*.

Table 8. Ioctls in ioctls/usb/audio.h

Ioctls	Description
`OSPLUS_IOCTL_USB_AUDIO_GET_UNIT_INFO`	Return information about an audio unit.
`OSPLUS_IOCTL_USB_AUDIO_GET_UNIT_INPUTS`	Return a list of all units that provide an input to the current audio unit.
`OSPLUS_IOCTL_USB_AUDIO_GET_UNIT_OUTPUTS`	Return a list of all units to which the current audio unit provides inputs.
`OSPLUS_IOCTL_USB_AUDIO_GET_FEATURE`	Return the minimum, maximum, resolution and current value of a feature control.
`OSPLUS_IOCTL_USB_AUDIO_SET_FEATURE`	Set the current value of a feature control.
`OSPLUS_IOCTL_USB_AUDIO_GET_MIXER_INFO`	Return information about a mixer unit including what input channels are used for a particular mixer input.
`OSPLUS_IOCTL_USB_AUDIO_GET_MIXER`	Return the minimum, maximum, resolution and current value of a mixer input channel.
`OSPLUS_IOCTL_USB_AUDIO_SET_MIXER`	Set the current value of a mixer input channel.
`OSPLUS_IOCTL_USB_AUDIO_GET_SELECTOR`	Return the minimum, maximum, resolution and current value of selector control.
`OSPLUS_IOCTL_USB_AUDIO_SET_SELECTOR`	Set the current value of selector control.
`OSPLUS_IOCTL_USB_AUDIO_GET_FORMAT_COUNT`	Get the number of formats supported by audio device.
`OSPLUS_IOCTL_USB_AUDIO_GET_FORMAT_LIST`	Get a list of formats supported by audio device.
`OSPLUS_IOCTL_USB_AUDIO_GET_FORMAT`	Get the currently selected format on audio device.
`OSPLUS_IOCTL_USB_AUDIO_SET_FORMAT`	Set the audio device format.
`OSPLUS_IOCTL_USB_AUDIO_TRANSFER_STATUS`	Get the audio transfer engine status.
`OSPLUS_IOCTL_USB_AUDIO_TRANSFER_START`	Start the audio transfer engine.
`OSPLUS_IOCTL_USB_AUDIO_TRANSFER_STOP`	Stop the audio transfer engine.
`OSPLUS_IOCTL_USB_AUDIO_PURGE`	Purge the audio transfer engine internal buffers.

Table 9. Audio terminal attribute defines in ioctls/usb/audio.h

Defines	Description
`OSPLUS_USB_AUDIO_ATTRIBUTE_MUTE`	Mute feature control
`OSPLUS_USB_AUDIO_ATTRIBUTE_VOLUME`	Volume feature control
`OSPLUS_USB_AUDIO_ATTRIBUTE_BASS`	Bass feature control
`OSPLUS_USB_AUDIO_ATTRIBUTE_MID`	Mid feature control
`OSPLUS_USB_AUDIO_ATTRIBUTE_TREBLE`	Treble feature control
`OSPLUS_USB_AUDIO_ATTRIBUTE_GRAPHIC_EQ`	Graphic equalizer feature control (not currently supported)
`OSPLUS_USB_AUDIO_ATTRIBUTE_AUTO_GAIN`	Auto-gain feature control
`OSPLUS_USB_AUDIO_ATTRIBUTE_DELAY`	Delay feature control
`OSPLUS_USB_AUDIO_ATTRIBUTE_BASS_BOOST`	Bass boost feature control
`OSPLUS_USB_AUDIO_ATTRIBUTE_LOUDNESS`	Loudness feature control
`OSPLUS_USB_AUDIO_ATTRIBUTE_FEATURES`	Mask of all possible feature control attributes
`OSPLUS_USB_AUDIO_ATTRIBUTE_CAPTURE`	Capture feature control
`OSPLUS_USB_AUDIO_ATTRIBUTE_PLAYBACK`	Playback feature control
`OSPLUS_USB_AUDIO_ATTRIBUTE_DIRECTION`	Mask of capture/playback feature control attributes
`OSPLUS_USB_AUDIO_ATTRIBUTE_S8`	Feature control value is signed 8-bits
`OSPLUS_USB_AUDIO_ATTRIBUTE_U8`	Feature control values is unsigned 8-bits
`OSPLUS_USB_AUDIO_ATTRIBUTE_S16`	Feature control values is signed 16-bits
`OSPLUS_USB_AUDIO_ATTRIBUTE_U16`	Feature control values is unsigned 16-bits
`OSPLUS_USB_AUDIO_ATTRIBUTE_8BIT`	Feature control values are 8-bit mask
`OSPLUS_USB_AUDIO_ATTRIBUTE_16BIT`	Feature control values are 16-bit mask
`OSPLUS_USB_AUDIO_ATTRIBUTE_SIZES`	Feature control values size mask
`OSPLUS_USB_AUDIO_ATTRIBUTE_CUR`	Feature control supports getting/setting current value
`OSPLUS_USB_AUDIO_ATTRIBUTE_MIN`	Feature control supports getting minimum value
`OSPLUS_USB_AUDIO_ATTRIBUTE_MAX`	Feature control supports getting maximum value
`OSPLUS_USB_AUDIO_ATTRIBUTE_VALUES`	Feature control values mask

Table 10. Audio channel defines in ioctls/usb/audio.h

Defines	Description
`OSPLUS_USB_AUDIO_CHANNEL_FRONT_LEFT`	Front left channel
`OSPLUS_USB_AUDIO_CHANNEL_FRONT_RIGHT`	Front right channel
`OSPLUS_USB_AUDIO_CHANNEL_FRONT_CENTER`	Front center channel
`OSPLUS_USB_AUDIO_CHANNEL_SUB_WOOFER`	Sub woofer channel
`OSPLUS_USB_AUDIO_CHANNEL_REAR_LEFT`	Rear left channel
`OSPLUS_USB_AUDIO_CHANNEL_REAR_RIGHT`	Rear right channel
`OSPLUS_USB_AUDIO_CHANNEL_CENTER_LEFT`	Center left channel
`OSPLUS_USB_AUDIO_CHANNEL_CENTER_RIGHT`	Center right channel
`OSPLUS_USB_AUDIO_CHANNEL_REAR_CENTER`	Rear center channel

Table 11. Audio format defines in ioctls/usb/audio.h

Defines	Description
`OSPLUS_USB_AUDIO_FORMAT_TAG_PCM`	PCM audio format
`OSPLUS_USB_AUDIO_FORMAT_TAG_PCM8`	PCM8 audio format
`OSPLUS_USB_AUDIO_FORMAT_TAG_IEEE_FLOAT`	IEEE float audio format
`OSPLUS_USB_AUDIO_FORMAT_TAG_ALAW`	A-law audio format
`OSPLUS_USB_AUDIO_FORMAT_TAG_ULAW`	µ-law audio format

Table 12. Audio transfer defines in ioctls/usb/audio.h

Defines	Description
`OSPLUS_USB_AUDIO_TRANSFER_STATUS_STOPPED`	Audio transfer engine stopped
`OSPLUS_USB_AUDIO_TRANSFER_STATUS_RUNNING`	Audio transfer engine running
`OSPLUS_USB_AUDIO_TRANSFER_STATUS_SHUTDOWN`	Audio transfer engine terminating

Table 13. Types defined in ioctls/usb/audio.h

Defines	Description
`usb_audio_unit_t`	Type to return information about an audio unit
`usb_audio_names_t`	Type to return a list of audio unit input or output names
`usb_audio_feature_t`	Type to get and set audio unit feature control
`usb_audio_mixer_info_t`	Type to return input channel information for audio mixer unit

Table 13. Types defined in ioctls/usb/audio.h (continued)

Defines	Description
`usb_audio_mixer_t`	Type to get and set audio unit mixer channel volume
`usb_audio_selector_t`	Type to get and set audio selector unit values
`usb_audio_format_t`	Type to get and set a audio format
`usb_audio_transfer_status_t`	Type to get the audio transfer engine status
`usb_audio_transfer_t`	Type to encapsulate an audio transfer

2.12 Device library header file: ioctls/usb/comm.h

The device library defines a set of ioctls that are used by USB communications class devices. These ioctls are in the file `ioctls/usb/comm.h`. For the ioctl definitions, see *Section 3.3 on page 102*.

Table 14. Ioctls in ioctls/usb/comm.h

Ioctls	Description
`OSPLUS_IOCTL_USB_COMM_SET_LISTENER`	Register a callback to be executed on device status change
`OSPLUS_IOCTL_USB_COMM_GET_LISTENER`	Return the installed callback to be executed on device status change details
`OSPLUS_IOCTL_USB_COMM_CLR_LISTENER`	Remove the installed callback to be executed on device status change
`OSPLUS_IOCTL_USB_COMM_SEND_COMMAND`	Send an encapsulated command to the device
`OSPLUS_IOCTL_USB_COMM_GET_RESPONSE`	Get the device response to an encapsulated command
`OSPLUS_IOCTL_USB_COMM_GET_CAPABILITY`	Get the device capabilities

Table 15. Communication callback event defines in ioctls/usb/comm.h

Defines	Description
`OSPLUS_USB_COMM_EVENT_NETWORK_CONNECTION`	Network connection event
`OSPLUS_USB_COMM_EVENT_RESPNSE_AVAILABLE`	Response to encapsulated command available event
`OSPLUS_USB_COMM_EVENT_AUX_JACK_HOOK_STATE`	Change in the auxiliary jack hook state event
`OSPLUS_USB_COMM_EVENT_RING_DETECT`	Ring detected event
`OSPLUS_USB_COMM_EVENT_SERIAL_STATE`	Serial state change event
`OSPLUS_USB_COMM_EVENT_CALL_STATE_CHANGE`	Call state change event

Table 15. Communication callback event defines in ioctls/usb/comm.h (continued)

Defines	Description
OSPLUS_USB_COMM_EVENT_LINE_ STATE_CHANGE	Line state change event
OSPLUS_USB_COMM_EVENT_ CONNECTION_SPEED_CHANGE	Connection speed change

Table 16. Communication network connection event defines in ioctls/usb/comm.h

Defines	Description
OSPLUS_USB_COMM_VALUE_DISCONNECT	Network is disconnected
OSPLUS_USB_COMM_VALUE_CONNECT	Network is connected

Table 17. Communication auxiliary jack hook event defines in ioctls/usb/comm.h

Defines	Description
OSPLUS_USB_COMM_VALUE_OFF_HOOK	Off hook
OSPLUS_USB_COMM_VALUE_ON_HOOK	On hook

Table 18. Communication serial state event defines in ioctls/usb/comm.h

Defines	Description
OSPLUS_USB_COMM_VALUE_RX_CARRIER	RX carrier
OSPLUS_USB_COMM_VALUE_TX_CARRIER	TX carrier
OSPLUS_USB_COMM_VALUE_BREAK	Break
OSPLUS_USB_COMM_VALUE_RING_SIGNAL	Ring signal
OSPLUS_USB_COMM_VALUE_FRAMING	Framing error
OSPLUS_USB_COMM_VALUE_PARITY	Parity error
OSPLUS_USB_COMM_VALUE_OVERRUN	Overrun error

Table 19. Communication call state event defines in ioctls/usb/comm.h

Defines	Description
`OSPLUS_USB_COMM_VALUE_RESERVED`	Reserved
`OSPLUS_USB_COMM_VALUE_IDLE`	Call has become idle
`OSPLUS_USB_COMM_VALUE_DIALING`	Dialing
`OSPLUS_USB_COMM_VALUE_RINGBACK`	Ring back
`OSPLUS_USB_COMM_VALUE_CONNECTION`	Connected
`OSPLUS_USB_COMM_VALUE_INCOMING`	Incoming call

Table 20. Communication call state ring back defines in ioctls/usb/comm.h

Defines	Description
`OSPLUS_USB_COMM_VALUE_RINGBACK_NORMAL`	Normal
`OSPLUS_USB_COMM_VALUE_RINGBACK_BUSY`	Busy
`OSPLUS_USB_COMM_VALUE_RINGBACK_FAST_BUSY`	Fast busy
`OSPLUS_USB_COMM_VALUE_RINGBACK_UNKNOWN`	Unkown ring back type

Table 21. Communications call state connection defines in ioctls/usb/comm.h

Defines	Description
`OSPLUS_USB_COMM_VALUE_CONNECTION_VOICE`	Voice connection
`OSPLUS_USB_COMM_VALUE_CONNECTION_ANS_MACHINE`	Answering machine connection
`OSPLUS_USB_COMM_VALUE_CONNECTION_FAX`	Fax connection
`OSPLUS_USB_COMM_VALUE_CONNECTION_DATA`	Data connection
`OSPLUS_USB_COMM_VALUE_CONNECTION_UNKNOWN`	Unknown connection

Table 22. Communications line state event defines in ioctls/usb/comm.h

Defines	Description
`OSPLUS_USB_COMM_VALUE_LINE_IDLE`	Line idle
`OSPLUS_USB_COMM_VALUE_LINE_HOLD`	Line hold
`OSPLUS_USB_COMM_VALUE_LINE_OFF_HOOK`	Line off hook
`OSPLUS_USB_COMM_VALUE_LINE_ON_HOOK`	Line on hook

Table 23. Communications connection speed change event defines in ioctls/usb/comm.h

Defines	Description
`OSPLUS_USB_COMM_VALUE_US_BITRATE_MASK`	Upstream bit rate mask
`OSPLUS_USB_COMM_VALUE_US_BITRATE_SHIFT`	Upstream bit rate shift
`OSPLUS_USB_COMM_VALUE_DS_BITRATE_MASK`	Downstream bit rate mask
`OSPLUS_USB_COMM_VALUE_DS_BITRATE_SHIFT`	Downstream bit rate shift

Table 24. Types defined in ioctls/usb/comm.h

Types	Description
`usb_comm_callback_t`	Device status change callback
`usb_comm_cmd_t`	Encapsulated command and response

2.13 Device library header file: ioctls/usb/acm.h

The device library defines a set of ioctls that are used by USB ACM devices. These ioctls are in the file `ioctls/usb/acm.h`. For the ioctl definitions, see *Section 3.4 on page 106*.

Table 25. Ioctls in ioctls/usb/acm.h

Ioctls	Description
`OSPLUS_IOCTL_USB_ACM_SET_ABSTRACT_STATE`	Set the ACM device abstract state
`OSPLUS_IOCTL_USB_ACM_GET_ABSTRACT_STATE`	Get the ACM device abstract state
`OSPLUS_IOCTL_USB_ACM_CLR_ABSTRACT_STATE`	Clear the ACM device abstract state
`OSPLUS_IOCTL_USB_ACM_SET_COUNTRY_CODE`	Set the ACM device country code
`OSPLUS_IOCTL_USB_ACM_GET_COUNTRY_CODE`	Get the ACM device country code
`OSPLUS_IOCTL_USB_ACM_CLR_COUNTRY_CODE`	Clear the ACM device country code
`OSPLUS_IOCTL_USB_ACM_SET_LINE_CODING`	Set the ACM device line coding configuration
`OSPLUS_IOCTL_USB_ACM_GET_LINE_CODING`	Get the ACM device line coding configuration
`OSPLUS_IOCTL_USB_ACM_SET_BAUD`	Set the ACM device line coding baud rate
`OSPLUS_IOCTL_USB_ACM_GET_BAUD`	Get the ACM device line coding baud rate
`OSPLUS_IOCTL_USB_ACM_SET_STOP`	Set the ACM device line coding stop bits

Table 25. Ioctls in ioctls/usb/acm.h (continued)

Ioctls	Description
`OSPLUS_IOCTL_USB_ACM_GET_STOP`	Get the ACM device line coding stop bits
`OSPLUS_IOCTL_USB_ACM_SET_PARITY`	Set the ACM device line coding parity
`OSPLUS_IOCTL_USB_ACM_GET_PARITY`	Get the ACM device line coding parity
`OSPLUS_IOCTL_USB_ACM_SET_DATA`	Set the ACM device line coding data bits
`OSPLUS_IOCTL_USB_ACM_GET_DATA`	Get the ACM device line coding data bits
`OSPLUS_IOCTL_USB_ACM_SET_CTRL_LINE_STATE`	Set the ACM device control line state
`OSPLUS_IOCTL_USB_ACM_SEND_BREAK`	Send an RS-232 style break
`OSPLUS_IOCTL_USB_ACM_SEND_BREAK_START`	Start sending RS-232 style breaks
`OSPLUS_IOCTL_USB_ACM_SEND_BREAK_STOP`	Stop sending RS-232 style breaks
`OSPLUS_IOCTL_USB_ACM_GET_LINE_STATUS`	Get the ACM device line status

Table 26. ACM call management capabilities defines in ioctls/usb/acm.h

Defines	Description
`OSPLUS_USB_ACM_CAP_CALL_MANAGEMENT`	ACM device handles call management
`OSPLUS_USB_ACM_CAP_USES_DATA_CLASS`	ACM device sends/receives call management data over a data class interface

Table 27. ACM capabilities defines in ioctls/usb/acm.h

Defines	Description
`OSPLUS_USB_ACM_CAP_SET_COMM_FEATURE`	ACM device can set a comm feature
`OSPLUS_USB_ACM_CAP_CLEAR_COMM_FEATURE`	ACM device can clear a comm feature
`OSPLUS_USB_ACM_CAP_GET_COMM_FEATURE`	ACM device can get a comm feature
`OSPLUS_USB_ACM_CAP_SET_LINE_CODING`	ACM device can set line coding
`OSPLUS_USB_ACM_CAP_SET_CONTROL_LINE_STATE`	ACM device can set control line state
`OSPLUS_USB_ACM_CAP_GET_LINE_CODING`	ACM device can get line coding
`OSPLUS_USB_ACM_CAP_SERIAL_STATE`	ACM device can get control line state
`OSPLUS_USB_ACM_CAP_SEND_BREAK`	ACM device can send a break
`OSPLUS_USB_ACM_CAP_NETWORK_CONNECTION`	ACM device can make a network connection

Table 28. ACM line coding baud rate defines in ioctls/usb/acm.h

Defines	Description
`OSPLUS_USB_ACM_BAUD_0`	Baud rate of 0
`OSPLUS_USB_ACM_BAUD_300`	Baud rate of 300
`OSPLUS_USB_ACM_BAUD_600`	Baud rate of 600
`OSPLUS_USB_ACM_BAUD_1200`	Baud rate of 1200
`OSPLUS_USB_ACM_BAUD_2400`	Baud rate of 2400
`OSPLUS_USB_ACM_BAUD_4800`	Baud rate of 4800
`OSPLUS_USB_ACM_BAUD_9600`	Baud rate of 9600
`OSPLUS_USB_ACM_BAUD_19200`	Baud rate of 19200
`OSPLUS_USB_ACM_BAUD_38400`	Baud rate of 38400
`OSPLUS_USB_ACM_BAUD_57600`	Baud rate of 57600
`OSPLUS_USB_ACM_BAUD_76800`	Baud rate of 76800
`OSPLUS_USB_ACM_BAUD_115200`	Baud rate of 115200
`OSPLUS_USB_ACM_BAUD_230400`	Baud rate of 230400
`OSPLUS_USB_ACM_BAUD_460800`	Baud rate of 460800

Table 29. ACM line coding stop bit defines in ioctls/usb/acm.h

Defines	Description
`OSPLUS_USB_ACM_STOP_1`	One stop bit
`OSPLUS_USB_ACM_STOP_1_5`	One and a half stop bits
`OSPLUS_USB_ACM_STOP_2`	Two stop bits

Table 30. ACM line coding parity defines in ioctls/usb/acm.h

Defines	Description
`OSPLUS_USB_ACM_PARITY_NONE`	No parity
`OSPLUS_USB_ACM_PARITY_ODD`	Odd parity
`OSPLUS_USB_ACM_PARITY_EVEN`	Even parity
`OSPLUS_USB_ACM_PARITY_MARK`	Mark parity
`OSPLUS_USB_ACM_PARITY_SPACE`	Space parity

Table 31. ACM line coding data bit defines in ioctls/usb/acm.h

Defines	Description
`OSPLUS_USB_ACM_DATA_5`	5 data bits
`OSPLUS_USB_ACM_DATA_6`	6 data bits
`OSPLUS_USB_ACM_DATA_7`	7 data bits
`OSPLUS_USB_ACM_DATA_8`	8 data bits

Table 32. Types defined in ioctls/usb/acm.h

Types	Description
`usb_acm_capabilities_t`	ACM device capabilities
`usb_acm_line_coding_t`	ACM device line coding

2.14 Device library header file: ioctls/usb/encm.h

The device library defines a set of ioctls that are used by USB ENCM devices. These ioctls are in the file `ioctls/usb/encm.h`. For the ioctl definitions, see *Section 3.5 on page 115*.

Table 33. Ioctls in ioctls/usb/encm.h

Ioctls	Description
`OSPLUS_IOCTL_USB_ENCM_SET_MCAST`	Set the ENCM device multicast filters
`OSPLUS_IOCTL_USB_ENCM_GET_MCAST`	Get the ENCM device multicast filters
`OSPLUS_IOCTL_USB_ENCM_SET_PM_FILT`	Set the ENCM device power management filter
`OSPLUS_IOCTL_USB_ENCM_GET_PM_FILT`	Get the ENCM device power management filter
`OSPLUS_IOCTL_USB_ENCM_SET_PKT_FILTER`	Set the ENCM device packet filter
`OSPLUS_IOCTL_USB_ENCM_GET_PKT_FILTER`	Get the ENCM device packet filter
`OSPLUS_IOCTL_USB_ENCM_GET_STATS`	Get the ENCM device statistics
`OSPLUS_IOCTL_USB_ENCM_SET_MAC`	Set the ENCM device MAC address
`OSPLUS_IOCTL_USB_ENCM_GET_MAC`	Get the ENCM device MAC address
`OSPLUS_IOCTL_USB_ENCM_GET_CAPABILITY`	Get the ENCM device capabilities

Table 34. ENCM capabilities statistic bitmap defines in ioctls/usb/encm.h

Defines	Description
`OSPLUS_USB_ENCM_CAP_OSPLUS_USB_ENCM_CAP_XMIT_OK`	Frames transmitted without errors
`OSPLUS_USB_ENCM_CAP_RECV_OK`	Frames received without errors
`OSPLUS_USB_ENCM_CAP_XMIT_ERROR`	Frames not transmitted, or transmitted with errors
`OSPLUS_USB_ENCM_CAP_RECV_ERROR`	Frames received with errors
`OSPLUS_USB_ENCM_CAP_RCV_NO_BUFFER`	Frames not received due to lack of memory buffers
`OSPLUS_USB_ENCM_CAP_DIRECTED_BYTES_XMIT`	Directed bytes transmitted without errors
`OSPLUS_USB_ENCM_CAP_DIRECTED_FRAMES_XMIT`	Directed frames transmitted without errors
`OSPLUS_USB_ENCM_CAP_MULTICATS_BYTES_XMIT`	Multicast bytes transmitted without errors

Table 34. ENCM capabilities statistic bitmap defines in ioctls/usb/encm.h

Defines	Description
`OSPLUS_USB_ENCM_CAP_MULTICAST_FRAMES_XMIT`	Multicast frames transmitted without errors
`OSPLUS_USB_ENCM_CAP_BROADCAST_BYTES_XMIT`	Broadcast bytes transmitted without errors
`OSPLUS_USB_ENCM_CAP_BROADCAST_FRAMES_XMIT`	Broadcast frames transmitted without errors
`OSPLUS_USB_ENCM_CAP_DIRECTED_BYTES_RCV`	Directed bytes received without errors
`OSPLUS_USB_ENCM_CAP_DIRECTED_FRAMES_RCV`	Directed frames received without errors
`OSPLUS_USB_ENCM_CAP_MULTICATS_BYTES_RCV`	Multicast bytes received without errors
`OSPLUS_USB_ENCM_CAP_MULTICAST_FRAMES_RCV`	Multicast frames received without errors
`OSPLUS_USB_ENCM_CAP_BROADCAST_BYTES_RCV`	Broadcast bytes received without errors
`OSPLUS_USB_ENCM_CAP_BROADCAST_FRAMES_RCV`	Broadcast frames received without errors
`OSPLUS_USB_ENCM_CAP_RCV_CRC_ERROR`	Frames received with CRC errors
`OSPLUS_USB_ENCM_CAP_TRANSMIT_QUE_LENGTH`	Length of transmit queue

Table 35. ENCM packet filter type defines in ioctls/usb/encm.h

Defines	Description
`OSPLUS_USB_ENCM_PKT_TYPE_MULTICAST`	Multicast packet filter type
`OSPLUS_USB_ENCM_PKT_TYPE_BROADCAST`	Broadcast packet filter type
`OSPLUS_USB_ENCM_PKT_TYPE_DIRECTED`	Directed packet filter type
`OSPLUS_USB_ENCM_PKT_TYPE_ALL_MULTICAST`	All multicast packet filter type
`OSPLUS_USB_ENCM_PKT_TYPE_PROMISCUOUS`	Promiscuous packet filter type

Table 36. ENCM statistics code defines in ioctls/usb/encm.h

Defines	Description
`OSPLUS_USB_ENCM_STATS_XMIT_OK`	Frames transmitted without errors
`OSPLUS_USB_ENCM_STATS_RECV_OK`	Frames received without errors
`OSPLUS_USB_ENCM_STATS_XMIT_ERROR`	Frames not transmitted, or transmitted with errors
`OSPLUS_USB_ENCM_STATS_RECV_ERROR`	Frames received with errors
`OSPLUS_USB_ENCM_STATS_RCV_NO_BUFFER`	Frames not received due to lack of memory buffers

Table 36. ENCM statistics code defines in ioctls/usb/encm.h (continued)

Defines	Description
OSPLUS_USB_ENCM_STATS_ DIRECTED_BYTES_XMIT	Directed bytes transmitted without errors
OSPLUS_USB_ENCM_STATS_ DIRECTED_FRAMES_XMIT	Directed frames transmitted without errors
OSPLUS_USB_ENCM_STATS_ MULTICAST_BYTES_XMIT	Multicast bytes transmitted without errors
OSPLUS_USB_ENCM_STATS_ MULTICAST_FRAMES_XMIT	Multicast frames transmitted without errors
OSPLUS_USB_ENCM_STATS_ BROADCAST_BYTES_XMIT	Broadcast bytes transmitted without errors
OSPLUS_USB_ENCM_STATS_ BROADCAST_FRAMES_XMIT	Broadcast frames transmitted without errors
OSPLUS_USB_ENCM_STATS_ DIRECTED_BYTES_RCV	Directed bytes received without errors
OSPLUS_USB_ENCM_STATS_ DIRECTED_FRAMES_RCV	Directed frames received without errors
OSPLUS_USB_ENCM_STATS_ MULTICAST_BYTES_RCV	Multicast bytes received without errors
OSPLUS_USB_ENCM_STATS_ MULTICAST_FRAMES_RCV	Multicast frames received without errors
OSPLUS_USB_ENCM_STATS_ BROADCAST_BYTES_RCV	Broadcast bytes received without errors
OSPLUS_USB_ENCM_STATS_ BROADCAST_FRAMES_RCV	Broadcast frames received without errors
OSPLUS_USB_ENCM_STATS_ RCV_CRC_ERROR	Frames received with CRC errors
OSPLUS_USB_ENCM_STATS_ TRANSMIT_QUE_LENGTH	Length of transmit queue
OSPLUS_USB_ENCM_STATS_RCV_ ERROR_ALIGN	Frames received with alignment error
OSPLUS_USB_ENCM_STATS_XMIT_ ONE_COLLISION	Frames transmitted with one collision
OSPLUS_USB_ENCM_STATS_XMIT_ MORE_COLLISION	Frames transmitted with more than one collision
OSPLUS_USB_ENCM_STATS_XMIT_ DEFERRED	Frames transmitted after deferral
OSPLUS_USB_ENCM_STATS_XMIT_ MAX_COLLISIONS	Frames not transmitted due to collisions
OSPLUS_USB_ENCM_STATS_RCV_OVERRUN	Frames not received due to overrun
OSPLUS_USB_ENCM_STATS_XMIT_ UNDERRUN	Frames not transmitted due to underrun

Table 36. ENCM statistics code defines in ioctls/usb/encm.h (continued)

Defines	Description
OSPLUS_USB_ENCM_STATS_XMIT_ HEARTBEAT_FAIL	Frames transmitted with heartbeat failure
OSPLUS_USB_ENCM_STATS_XMIT_ TIMES_CRS_LOST	Times carrier sense lost during transmission
OSPLUS_USB_ENCM_STATS_XMIT_ LATE_COLLISIONS	Late collisions detected.

Table 37. Types defined in ioctls/usb/encm.h

Types	Description
usb_encm_capabilities_t	ENCM capabilities type
usb_encm_mcast_t	ENCM multicast filter type
usb_encm_pm_filter_t	ENCM power management filter type
usb_encm_statistic_t	ENCM statistic type
usb_encm_mac_t	ENCM MAC address type

2.15 Device library header file: ioctls/usb/h2h.h

The device library defines a set of ioctls that are used by USB host-to-host devices. These ioctls are in the file `ioctls/usb/h2h.h`. For the ioctl definitions, see *Section 3.6 on page 120*.

Table 38. Ioctls in ioctls/usb/h2h.h

Ioctls	Description
OSPLUS_IOCTL_USB_H2H_RESET_IN	Reset the host-to-host input pipe
OSPLUS_IOCTL_USB_H2H_RESET_OUT	Reset the host-to-host output pipe
OSPLUS_IOCTL_USB_H2H_GET_STATUS	Get the host-to-host status
OSPLUS_IOCTL_USB_H2H_WAIT_PEER_E	Wait until a peer is ready to connect
OSPLUS_IOCTL_USB_H2H_WAIT_TX_RDY	Wait until a peer is ready to transmit

Table 39. Defines in ioctls/usb/h2h.h

Defines	Description
OSPLUS_USB_H2H_PEER_E	Peer is connected
OSPLUS_USB_H2H_TX_REQ	Peer requests a transfer
OSPLUS_USB_H2H_TX_C	Transfer complete
OSPLUS_USB_H2H_RESET_IN	Waiting for a reset of input pipe
OSPLUS_USB_H2H_RESET_OUT	Waiting for a reset of output pipe
OSPLUS_USB_H2H_TX_RDY	Ready to transmit
OSPLUS_USB_H2H_RESERVED	Reserved

2.16 Device library header file: ioctls/usb/host.h

The device library defines a set of ioctls that are used by USB host controller devices. These are in the file `ioctls/usb/host.h`. For the ioctl definitions, see *Section 3.6 on page 120*.

Table 40. Ioctls in ioctls/usb/host.h

Ioctls	Description
`OSPLUS_IOCTL_USB_HOST_GET_PORTS`	Return the number of root hub ports supported by the controller
`OSPLUS_IOCTL_USB_HOST_TEST_MODE`	Put a root hub port in to a USB test mode
`OSPLUS_IOCTL_USB_HOST_RESUME_PORT`	Resume a suspended root hub port
`OSPLUS_IOCTL_USB_HOST_SUSPEND_PORT`	Suspend an active root hub port

Table 41. Types defined in ioctls/usb/host.h

Types	Description
`usb_host_test_mode_t`	Hi-Speed test modes
`usb_host_test_mode_params_t`	Test mode parameters

2.17 Device library header file: ioctls/usb/hub.h

The device library defines a set of ioctls that are used by USB hub devices. These are in the file `ioctls/usb/hub.h`. For the ioctl definitions, see *Section 3.8 on page 123*.

Table 42. Ioctls in ioctls/usb/hub.h

Ioctls	Description
`OSPLUS_IOCTL_USB_HUB_SET_FEAT`	Set the hub/port feature
`OSPLUS_IOCTL_USB_HUB_CLR_FEAT`	Clear the hub/port feature
`OSPLUS_IOCTL_USB_HUB_GET_STATUS`	Get the hub/port status

Table 43. USB hub feature defines in ioctls/usb/hub.h

Defines	Description
`OSPLUS_USB_HUB_C_HUB_LOCAL_POWER`	Hub local power feature
`OSPLUS_USB_HUB_C_HUB_OVER_POWER`	Hub over power feature

Table 44. USB hub port feature defines in ioctls/usb/hub.h

Defines	Description
`OSPLUS_USB_HUB_PORT_CONNECTION`	Port connection feature
`OSPLUS_USB_HUB_PORT_ENABLE`	Port enable feature
`OSPLUS_USB_HUB_PORT_SUSPEND`	Port suspend feature
`OSPLUS_USB_HUB_PORT_OVER_CURRENT`	Port over current feature
`OSPLUS_USB_HUB_PORT_RESET`	Port reset feature
`OSPLUS_USB_HUB_PORT_POWER`	Port power feature
`OSPLUS_USB_HUB_PORT_LOW_SPEED`	Port low speed feature
`OSPLUS_USB_HUB_C_PORT_CONNECTION`	Port clear connection feature
`OSPLUS_USB_HUB_C_PORT_ENABLE`	Port clear enable feature
`OSPLUS_USB_HUB_C_PORT_SUSPEND`	Port clear suspend feature
`OSPLUS_USB_HUB_C_PORT_OVER_CURRENT`	Port clear over current feature
`OSPLUS_USB_HUB_C_PORT_RESET`	Port clear reset feature
`OSPLUS_USB_HUB_PORT_TEST`	Port test feature
`OSPLUS_USB_HUB_PORT_INDICATOR`	Port indicator feature

Table 45. USB hub port test selector defines in ioctls/usb/hub.h

Defines	Description
`OSPLUS_USB_HUB_PORT_TEST_J`	Transmit a ’J’
`OSPLUS_USB_HUB_PORT_TEST_K`	Transmit a ’K’
`OSPLUS_USB_HUB_PORT_TEST_SEO_NAK`	Enter high speed receive state
`OSPLUS_USB_HUB_PORT_TEST_PACKET`	Transmit a test packet
`OSPLUS_USB_HUB_PORT_TEST_ FORCE_ENABLE`	Test force enable

Table 46. USB hub port status defines in ioctls/usb/hub.h

Defines	Description
`OSPLUS_USB_HUB_STATUS_PORT_CONNECTION`	Port connection status
`OSPLUS_USB_HUB_STATUS_PORT_ENABLE`	Port enable status
`OSPLUS_USB_HUB_STATUS_PORT_SUSPEND`	Port suspend status
`OSPLUS_USB_HUB_STATUS_PORT_OVER_CURRENT`	Port over current status
`OSPLUS_USB_HUB_STATUS_PORT_RESET`	Port reset status
`OSPLUS_USB_HUB_STATUS_PORT_POWER`	Port power status
`OSPLUS_USB_HUB_STATUS_PORT_LOW_SPEED`	Port low speed status
`OSPLUS_USB_HUB_STATUS_PORT_HIGH_SPEED`	Port high speed status
`OSPLUS_USB_HUB_STATUS_PORT_TEST`	Port test status
`OSPLUS_USB_HUB_STATUS_PORT_INDICATOR`	Port indicator status
`OSPLUS_USB_HUB_CHANGE_PORT_CONNECTION`	Port change connection status
`OSPLUS_USB_HUB_CHANGE_PORT_ENABLE`	Port change enable status
`OSPLUS_USB_HUB_CHANGE_PORT_SUSPEND`	Port change suspend status
`OSPLUS_USB_HUB_CHANGE_PORT_OVER_CURRENT`	Port change over current status
`OSPLUS_USB_HUB_CHANGE_PORT_RESET`	Port change reset status

Table 47. Types defined in ioctls/usb/hub.h

Types	Description
`usb_hub_feature_t`	Used for setting/clearing a hub/port feature
`usb_hub_status_t`	Used to get the hub/port status

2.18 Device library header file: ioctls/usb/voip.h

The device library defines a set of ioctls that are used by USB VOIP class devices. These ioctls are in the file `ioctls/usb/voip.h`. For the ioctl definitions, see *Section 3.9: USB VOIP ioctl definitions on page 125*.

Table 48. Ioctls in ioctls/usb/voip.h

Ioctls	Description
`OSPLUS_IOCTL_USB_VOIP_GET_INFO`	Return information about the VOIP phone
`OSPLUS_IOCTL_USB_VOIP_GET_KEYPRESS`	Return keypress information
`OSPLUS_IOCTL_USB_VOIP_SET_RING_VOLUME`	Set the phone ring volume
`OSPLUS_IOCTL_USB_VOIP_SET_RING_STYLE`	Set the phone ring style
`OSPLUS_IOCTL_USB_VOIP_SET_RING_TONE`	Turn the phone ring tone on or off
`OSPLUS_IOCTL_USB_VOIP_SET_DIAL_TONE`	Turn the phone dial tone on or off (if supported)
`OSPLUS_IOCTL_USB_VOIP_SET_LED`	Turn the phone LED on or off (on phones with a graphical display this also turns the backlight on and off)
`OSPLUS_IOCTL_USB_VOIP_LCD_CLEAR`	Clear the contents of the phone display
`OSPLUS_IOCTL_USB_VOIP_LCD_SEG7_WRITE`	Write text to a seven-segment display
`OSPLUS_IOCTL_USB_VOIP_LCD_PIXEL_BLIT`	Blit part of an off-screen buffer to the phone display.

Table 49. VOIP seven-segment display defines in ioctls/usb/voip.h

Defines	Description
`OSPLUS_USB_VOIP_SEG7_LINE_1_OFFSET`	Offset to line 1 of display
`OSPLUS_USB_VOIP_SEG7_LINE_1_LENGTH`	Length of line 1 in characters
`OSPLUS_USB_VOIP_SEG7_LINE_2_OFFSET`	Offset to line 2 of display
`OSPLUS_USB_VOIP_SEG7_LINE_2_LENGTH`	Length of line 2 in characters
`OSPLUS_USB_VOIP_SEG7_LINE_3_OFFSET`	Offset to line 3 of display
`OSPLUS_USB_VOIP_SEG7_LINE_3_LENGTH`	Length of line 3 in characters

Table 50. VOIP pixel display defines in ioctls/usb/voip.h

Defines	Description
`OSPLUS_USB_VOIP_PIXEL_MAX_X`	Maximum X-axis length
`OSPLUS_USB_VOIP_PIXEL_MAX_Y`	Maximum Y-axis length
`OSPLUS_USB_VOIP_PIXEL_BUFFER_SIZE`	Size of buffer required for graphics display
`OSPLUS_USB_VOIP_PIXEL_BYTE`	Macro to return offset to buffer for a given x, y
`OSPLUS_USB_VOIP_PIXEL_BIT`	Macro to return what bit to set for a given x, y
`OSPLUS_USB_VOIP_PIXEL_SET`	Macro to set a bit in a buffer for a given x, y
`OSPLUS_USB_VOIP_PIXEL_CLR`	Macro to clear a bit in a buffer for a given x, y

Table 51. **Types defined in ioctls/usb/voip.h**

Types	Description
`usb_voip_key_t`	Enumeration type to describe a key
`usb_voip_keypress_t`	Structure type to return keypress
`usb_voip_model_t`	Enumeration type to describe phone model
`usb_voip_lcd_t`	Enumeration type to describe phone LCD type
`usb_voip_info_t`	Structure type to return phone information
`usb_voip_ringtone_t`	Structure type to hold ringtone
`usb_voip_seg7_t`	Structure for writing text to a seven-segment LCD display
`usb_voip_blit_t`	Structure describing a rectangle of screen to blit.

2.19 USB driver header file: usblink.h

All the definitions relating to the USB driver are in the file `usblink.h`. The functions defined in `usblink.h` are explained in *Section 3.10: USB driver function definitions on page 132*.

Table 52. **Configuration macros defined in usblink.h**

Macro	Description
`SCC_OSPLUS_NONCACHED_MEM_POOL_SIZE`	Minimum amount of uncached memory needed by the USB stack
`USB_CONFIG_PATH`	Registry path to USB driver configuration options
`USB_CONFIG_CLASS_PATH`	Registry path to USB driver configuration options specifying what class drivers to register.
`USB_CONFIG_ENU_TASK_PRI`	Registry key to hold an integer containing the enumeration task priority. Default is `MAX_USER_PRIORITY`
`USB_CONFIG_URB_TASK_PRI`	Registry key to hold an integer containing the URB task priority. Default is `MAX_USER_PRIORITY`

Table 53. **Functions defined in usblink.h**

Function	Description
`USBLINK_Initialize()`	Initialize the USB driver

2.20 USB driver common ioctls

The ioctls in this section relate to the device library `device_ioctl()` function. For full details on using ioctls refer to the *OSPlus User Manua*l (ADCS 7702033).

All USB devices support the OSPlus common ioctls listed in *Table 54* to retrieve basic device information.

Table 54. OSPlus common ioctls supported by all USB devices

Ioctl	Description
`OSPLUS_IOCTL_GET_PID`	Query the device product ID
`OSPLUS_IOCTL_GET_PID_STR`	Query the device product string
`OSPLUS_IOCTL_GET_SERIAL`	Query the device serial number
`OSPLUS_IOCTL_GET_VID`	Query the device vendor ID
`OSPLUS_IOCTL_GET_VID_STR`	Query the device vendor string

Note: Details of all the common ioctls listed in this section can be found in the OSPlus user manual (ADCS 770203)

2.20.1 USB communications class device ioctls

In addition to the supported ioctls described in *Table 54: OSPlus common ioctls supported by all USB devices on page 61*, all communications class devices also support the OSPlus common ioctls as described in *Table 55*.

Table 55. Communications device supported OSPlus common ioctls

Ioctl	Description
`OSPLUS_IOCTL_ABORT_RD`	Abort any current read operation
`OSPLUS_IOCTL_ABORT_WR`	Abort any current write operation
`OSPLUS_IOCTL_GET_RD_MODE`	Return the current mode (blocking or non-blocking) for read operations
`OSPLUS_IOCTL_GET_RD_TIMEOUT`	Return the timeout value for blocking read operations
`OSPLUS_IOCTL_GET_WR_MODE`	Return the current mode (blocking or non-blocking) for write operations
`OSPLUS_IOCTL_GET_WR_TIMEOUT`	Return the timeout value for blocking write operations
`OSPLUS_IOCTL_SET_RD_MODE`	Set the mode for read operations to blocking or non-blocking
`OSPLUS_IOCTL_SET_RD_TIMEOUT`	Set the timeout value for blocking read operations
`OSPLUS_IOCTL_SET_WR_MODE`	Set the mode for write operations to blocking or non-blocking
`OSPLUS_IOCTL_SET_WR_TIMEOUT`	Set the timeout value for blocking write operations

2.20.2 USB ethernet device ioctls

In addition to the ioctls supported by all USB devices listed in *Table 54: OSPlus common ioctls supported by all USB devices on page 61*, all USB ethernet devices also support the further OSPlus common ioctls as described in *Table 56*.

Table 56. Ethernet device supported OSPlus common ioctls

Ioctl	Description
`OSPLUS_IOCTL_ABORT_RD`	Abort any current read operation
`OSPLUS_IOCTL_ABORT_WR`	Abort any current write operation
`OSPLUS_IOCTL_GET_RD_TIMEOUT`	Return the timeout value for blocking read operations
`OSPLUS_IOCTL_GET_WR_TIMEOUT`	Return the timeout value for blocking write operations
`OSPLUS_IOCTL_POST_WRITES`	Specify if driver performs synchronous or asynchronous writes
`OSPLUS_IOCTL_SET_RD_TIMEOUT`	Set the timeout value for blocking read operations
`OSPLUS_IOCTL_SET_WR_TIMEOUT`	Set the timeout value for blocking write operations

USB ethernet devices also support a subset of the OSPlus ethernet ioctls, as described in *Table 57*.

Table 57. Ethernet device supported OSPlus ethernet ioctls

Ioctl	Description
`OSPLUS_IOCTL_ETH_GET_INFO`	Return the ethernet device information (for example, MTU)
`OSPLUS_IOCTL_ETH_GET_MAC`	Get the MAC address
`OSPLUS_IOCTL_ETH_GET_MCAST_HASH_TABLE`	Get the multicast hash table
`OSPLUS_IOCTL_ETH_GET_PKT_FILTER`	Get the current ethernet packet filtering options
`OSPLUS_IOCTL_ETH_GET_STATUS`	Get the ethernet status
`OSPLUS_IOCTL_ETH_SET_MAC`	Set the MAC address
`OSPLUS_IOCTL_ETH_SET_MCAST_FILTERS`	Set the multicast filter addresses
`OSPLUS_IOCTL_ETH_SET_MCAST_HASH_TABLE`	Set the multicast hash table
`OSPLUS_IOCTL_ETH_SET_MAC`	Set the MAC address
`OSPLUS_IOCTL_ETH_SET_PKT_FILTER`	Set the ethernet packet filtering options

Communications class ENCM and RNDIS devices also support a limited subset of the OSPlus ethernet ioctls in *Table 58*.

Table 58. Communications ENCM/RNDIS device supported OSPlus ethernet ioctls

Ioctl	Description
`OSPLUS_IOCTL_ETH_GET_INFO`	Return the ethernet device information (for example, MTU)
`OSPLUS_IOCTL_ETH_GET_MAC`	Get the MAC address
`OSPLUS_IOCTL_ETH_GET_PKT_FILTER`	Get the current ethernet packet filtering options
`OSPLUS_IOCTL_ETH_GET_STATISTIC`	Get an ethernet statistic
`OSPLUS_IOCTL_ETH_GET_STATUS`	Get the ethernet status
`OSPLUS_IOCTL_ETH_SET_MAC`	Set the MAC address
`OSPLUS_IOCTL_ETH_SET_MCAST_FILTERS`	Set the multicast filter addresses
`OSPLUS_IOCTL_ETH_SET_PKT_FILTER`	Set the ethernet packet filtering options

2.20.3 USB HID device ioctls

USB HID joystick device ioctls

In addition to the ioctls supported by all USB devices listed in *Table 54: OSPlus common ioctls supported by all USB devices on page 61*, USB HID joystick devices support the further OSPlus common ioctls, described in *Table 59*.

Table 59. HID joystick device supported OSPlus common ioctls

Ioctl	Description
`OSPLUS_IOCTL_WAIT`	Block waiting for the device to receive an event
`OSPLUS_IOCTL_GET_FEATURES`	Get the supported joystick features

USB HID joystick devices also support all of the OSPlus joystick ioctls, described in *Table 60*.

Table 60. HID joystick device supported OSPlus joystick ioctls

Ioctl	Description
`OSPLUS_IOCTL_JOYSTICK_GET_X`	Get the joystick X value
`OSPLUS_IOCTL_JOYSTICK_GET_X_BND`	Get the range of joystick X value
`OSPLUS_IOCTL_JOYSTICK_GET_Y`	Get the joystick Y value
`OSPLUS_IOCTL_JOYSTICK_GET_Y_BND`	Get the range of the joystick Y value
`OSPLUS_IOCTL_JOYSTICK_GET_Z`	Get the joystick Zvalue
`OSPLUS_IOCTL_JOYSTICK_GET_Z_BND`	Get the range of joystick Zvalue
`OSPLUS_IOCTL_JOYSTICK_GET_RX`	Get the joystick rX value
`OSPLUS_IOCTL_JOYSTICK_GET_RX_BND`	Get the range of joystick rX value
`OSPLUS_IOCTL_JOYSTICK_GET_RY`	Get the joystick rYvalue

Table 60. HID joystick device supported OSPlus joystick ioctls (continued)

Ioctl	Description
`OSPLUS_IOCTL_JOYSTICK_GET_RY_BND`	Get the range of joystick rY value
`OSPLUS_IOCTL_JOYSTICK_GET_RZ`	Get the joystick rZ value
`OSPLUS_IOCTL_JOYSTICK_GET_RZ_BND`	Get the range of joystick rZ value
`OSPLUS_IOCTL_JOYSTICK_SLIDER`	Get the joystick slider value
`OSPLUS_IOCTL_JOYSTICK_SLIDER_BND`	Get the range of joystick slider value
`OSPLUS_IOCTL_JOYSTICK_HAT_SWITCH`	Get the joystick hat switch value
`OSPLUS_IOCTL_JOYSTICK_BTN_COUNT`	Get the number of buttons on the joystick
`OSPLUS_IOCTL_JOYSTICK_BTN_BITMAP`	Get the bitmap of what buttons are pressed
`OSPLUS_IOCTL_JOYSTICK_ALL_VALUES`	Fill an array with all joystick values
`OSPLUS_IOCTL_JOYSTICK_SET_RUMBLE`	Set the amount of force feedback (0...65535)

USB HID keyboard device ioctls

In addition to the ioctls supported by all USB devices listed in *Table 54: OSPlus common ioctls supported by all USB devices on page 61*, all USB HID keyboard devices support the further OSPlus common ioctls, described in *Table 61*.

Table 61. HID keyboard device supported OSPlus common ioctls

Ioctl	Description
`OSPLUS_IOCTL_FLUSH`	Perform a flush on a device
`OSPLUS_IOCTL_WAIT`	Block waiting for the device to receive an event

USB HID keyboard devices also support all of the OSPlus keyboard ioctls, described in *Table 62*.

Table 62. HID keyboard device supported OSPlus keyboard ioctls

Ioctl	Description
`OSPLUS_IOCTL_KEYBOARD_COUNT`	Return the number of key presses pending
`OSPLUS_IOCTL_KEYBOARD_GETKEY`	Get a keyboard key

USB HID mouse device ioctls

In addition to the OSPlus common ioctls supported by all USB devices listed in *Table 54: OSPlus common ioctls supported by all USB devices on page 61*, all USB mouse devices support the further OSPlus common ioctls, described in *Table 63*.

Table 63. HID mouse device supported OSPlus common ioctls

Ioctl	Description
`OSPLUS_IOCTL_WAIT`	Block waiting for the device to receive an event

USB mouse devices support all OSPlus mouse ioctls, described in *Table 64*.

Table 64. HID mouse device supported OSPlus mouse ioctls

Ioctl	Description
`OSPLUS_IOCTL_MOUSE_BTN_INFO`	Return mouse button information
`OSPLUS_IOCTL_MOUSE_GET_BND`	Return the current mouse bounds
`OSPLUS_IOCTL_MOUSE_GET_BTN`	Get mouse button status
`OSPLUS_IOCTL_MOUSE_GET_POS`	Return the mouse position
`OSPLUS_IOCTL_MOUSE_SET_BND`	Set the current mouse bounds
`OSPLUS_IOCTL_MOUSE_SET_POS`	Set the mouse position
`OSPLUS_IOCTL_MOUSE_GET_X`	Get the mouse X axis position
`OSPLUS_IOCTL_MOUSE_SET_X`	Set the mouse X axis position
`OSPLUS_IOCTL_MOUSE_GET_X_BND`	Get the mouse X axis bounds
`OSPLUS_IOCTL_MOUSE_SET_X_BND`	Set the mouse X axis bounds
`OSPLUS_IOCTL_MOUSE_GET_Y`	Get the mouse Y axis position
`OSPLUS_IOCTL_MOUSE_SET_Y`	Set the mouse Y axis position
`OSPLUS_IOCTL_MOUSE_GET_Y_BND`	Get the mouse Y axis bounds
`OSPLUS_IOCTL_MOUSE_SET_Y_BND`	Set the mouse Y axis bounds
`OSPLUS_IOCTL_MOUSE_GET_XY`	Get the mouse X and Y axis position
`OSPLUS_IOCTL_MOUSE_SET_XY`	Set the mouse X and Y axis position
`OSPLUS_IOCTL_MOUSE_GET_XY_BND`	Get the mouse X and Y axis bounds
`OSPLUS_IOCTL_MOUSE_SET_XY_BND`	Set the mouse X and Y axis bounds
`OSPLUS_IOCTL_MOUSE_GET_WHEEL`	Get the mouse wheel position
`OSPLUS_IOCTL_MOUSE_SET_WHEEL`	Set the mouse wheel position
`OSPLUS_IOCTL_MOUSE_GET_WHEEL_BND`	Get the mouse wheel bounds
`OSPLUS_IOCTL_MOUSE_SET_WHEEL_BND`	Set the mouse wheel bounds
`OSPLUS_IOCTL_MOUSE_GET_TWHEEL`	Get the mouse tilt wheel position
`OSPLUS_IOCTL_MOUSE_SET_TWHEEL`	Set the mouse tilt wheel position
`OSPLUS_IOCTL_MOUSE_GET_TWHEEL_BND`	Get the mouse tilt wheel bounds
`OSPLUS_IOCTL_MOUSE_SET_TWHEEL_BND`	Set the mouse tilt wheel bounds
`OSPLUS_IOCTL_MOUSE_GET_BTN_COUNT`	Get the mouse button count
`OSPLUS_IOCTL_MOUSE_GET_BTN_BITMAP`	Get the mouse button state bitmap
`OSPLUS_IOCTL_MOUSE_GET_BTN_DATA`	Get the mouse button press and release data
`OSPLUS_IOCTL_MOUSE_CLR_BTN_DATA`	Clear the mouse button press and release data
`OSPLUS_IOCTL_MOUSE_GET_BTN_PRESS`	Get the mouse button press data
`OSPLUS_IOCTL_MOUSE_CLR_BTN_PRESS`	Clear the mouse button press data
`OSPLUS_IOCTL_MOUSE_GET_BTN_RELEASE`	Get the mouse button release data

Table 64. HID mouse device supported OSPlus mouse ioctls (continued)

Ioctl	Description
`OSPLUS_IOCTL_MOUSE_CLR_BTN_RELEASE`	Clear the mouse button release data
`OSPLUS_IOCTL_MOUSE_GET_ALL_VALUES`	Get all mouse values for X, Y, wheel, tilt wheel and button bitmap

USB HID generic device ioctls

USB HID generic devices support only the ioctls listed in *Table 54: OSPlus common ioctls supported by all USB devices on page 61*.

2.20.4 USB mass storage device ioctls

In addition to the ioctls supported by all USB devices listed in *Table 54: OSPlus common ioctls supported by all USB devices on page 61*, all USB mass storage devices support the OSPlus common ioctls, described in *Table 65*.

Table 65. Mass storage device supported OSPlus common ioctls

Ioctl	Description
`OSPLUS_IOCTL_BLOCK_SIZE`	Query the size in bytes of the logical blocks supported by the device.
`OSPLUS_IOCTL_CHECK_MEDIA`	Query the status of the media.
`OSPLUS_IOCTL_NUM_BLOCKS`	Query the number of logically addressable blocks the device supports.
`OSPLUS_IOCTL_WRITE_PROTECTED`	Query whether the media inserted is write protected.
`OSPLUS_IOCTL_GET_RD_TIMEOUT`	Return the timeout value for blocking read operations
`OSPLUS_IOCTL_GET_WR_TIMEOUT`	Return the timeout value for blocking write operations
`OSPLUS_IOCTL_SET_RD_TIMEOUT`	Set the timeout value for blocking read operations
`OSPLUS_IOCTL_SET_WR_TIMEOUT`	Set the timeout value for blocking write operations
`OSPLUS_IOCTL_SPIN_REQUESTS`	Query if device supports spin requests
`OSPLUS_IOCTL_SPIN_UP`	Request device to spin up
`OSPLUS_IOCTL_SPIN_DOWN`	Request device to spin down

2.20.5 USB serial device ioctls

In addition to the ioctls supported by all USB devices listed in *Table 54: OSPlus common ioctls supported by all USB devices on page 61*, all USB serial devices support the further OSPlus common ioctls, described in *Table 66*.

Table 66. Serial device supported OSPlus common ioctls

Ioctl	Description
`OSPLUS_IOCTL_ABORT_RD`	Abort any current read operation
`OSPLUS_IOCTL_ABORT_WR`	Abort any current write operation
`OSPLUS_IOCTL_FLUSH`	Perform a flush on a device
`OSPLUS_IOCTL_GET_RD_TIMEOUT`	Return the timeout value for blocking read operations
`OSPLUS_IOCTL_GET_WR_TIMEOUT`	Return the timeout value for blocking write operations
`OSPLUS_IOCTL_SET_RD_TIMEOUT`	Set the timeout value for blocking read operations
`OSPLUS_IOCTL_SET_WR_TIMEOUT`	Set the timeout value for blocking write operations

USB serial devices also support a subset of the OSPlus ethernet ioctls, described in *Table 67*.

Table 67. Serial device supported OSPlus serial ioctls

Ioctl	Description
`OSPLUS_IOCTL_SERIAL_GET_BAUD`	Get the serial port current baud rate
`OSPLUS_IOCTL_SERIAL_GET_CONFIG`	Get the serial port current configuration
`OSPLUS_IOCTL_SERIAL_GET_DATA`	Get the serial port current data bit size
`OSPLUS_IOCTL_SERIAL_GET_PARITY`	Get the serial port current data parity
`OSPLUS_IOCTL_SERIAL_GET_STOP`	Get the serial port current stop bits
`OSPLUS_IOCTL_SERIAL_SET_BAUD`	Set the serial port baud rate
`OSPLUS_IOCTL_SERIAL_SET_CONFIG`	Set the serial port configuration
`OSPLUS_IOCTL_SERIAL_SET_DATA`	Set the serial port data bit size
`OSPLUS_IOCTL_SERIAL_SET_PARITY`	Set the serial port data parity
`OSPLUS_IOCTL_SERIAL_SET_STOP`	Set the serial port stop bits

2.20.6 USB unsupported device ioctls

USB unsupported devices that are managed by the USB proxy class, support only those ioctls that are listed in *Chapter 3: USB ioctl and driver definitions on page 68*.

For full details on using ioctls refer to the *OSPlus User Manual (ADCS 7702033)*.

3 USB ioctl and driver definitions

3.1 USB proxy device ioctl definitions

This chapter provides the USB ioctl definitions. An alphabetical index of all ioctls, excluding USB, is provided in the *OSPlus User Manual (ADCS 7702033)*. The USB device specific ioctl definitions in this manual relate to the `device_ioctl()` call. Refer to the *OSPlus User Manual (ADCS 7702033)* for more details.

OSPLUS_IOCTL_USB_SET_EXTRACT_CB

Install a USB extract callback

Arguments: The address of an `usb_extract_callback_t` structure that holds the details of the callback to execute on extraction of the USB device.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl installs a callback that is executed after the USB device has been extracted, but before the device is unregistered from the device library. The callback data is supplied in a `usb_extract_callback_t` structure whose address is passed-in. Only one extract callback can be registered per device.

The `usb_extract_callback_t` structure has the following format:

```
typedef struct usb_extract_callback_s
{
     void (*callBack)(void *cookie);
     void *cookie;
} usb_extract_callback_t;
```

The fields are explained in *Table 68*.

Table 68. usb_extract_callback_t fields

Field name	Description
`callBack`	Actual callback function to execute
`cookie`	Cookie passed-in to function

See also: `OSPLUS_IOCTL_USB_GET_EXTRACT_CB`, `OSPLUS_IOCTL_USB_CLR_EXTRACT_CB`

OSPLUS_IOCTL_USB_GET_EXTRACT_CB

Return the installed USB extract callback information

Arguments: The address of an `usb_extract_callback_t` structure to hold the details of the installed extract callback.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl populates the `usb_extract_callback_t` structure whose address is passed-in with the details of the installed extract callback. If no extract callback is installed, each filed in the `usb_extract_callback_t` structure is set to `NULL`.

See also: `OSPLUS_IOCTL_USB_SET_EXTRACT_CB`, `OSPLUS_IOCTL_USB_GET_EXTRACT_CB`

OSPLUS_IOCTL_USB_CLR_EXTRACT_CB

Remove the installed USB extract callback

Arguments: The address of an `usb_extract_callback_t` structure that holds the details of the installed extraction callback.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl removes the currently installed extract callback. The `usb_extract_callback_t` structure passed-in must contain the currently installed callback details. If it does not, the function will fail.

See also: `OSPLUS_IOCTL_USB_SET_EXTRACT_CB`, `OSPLUS_IOCTL_USB_CLR_EXTRACT_CB`

OSPLUS_IOCTL_USB_GET_DEVICE_INFO

Return the USB device information

Arguments: The address of a `usb_device_info_t` structure to hold the device details returned.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl obtains the USB device information related to the information contained in a USB device descriptor, and populates the `usb_device_info_t` structure whose address is passed-in with the information.

The usb_device_info_t structure has the following format:

```
typedef struct usb_device_info_s
{
      unsigned short usb;
      unsigned char usb_class;
      unsigned char usb_subclass;
      unsigned char usb_protocol;
      unsigned short vid;
      unsigned short pid;
      unsigned short version;
      unsigned int address;
      unsigned char speed;
      unsigned int max_packet;
      unsigned int num_configurations;
      osplus_str_t manufacturer;
      osplus_str_t product;
      osplus_str_t serial;
} usb_device_info_t;
```

The fields are explained in *Table 69*.

Table 69. usb_device_info_t fields

Field name	Description
usb	USB revision to which the device is compliant
usb_class	USB class to which the device belongs
usb_subclass	USB subclass to which the device belongs
usb_protocol	USB class/subclass protocol the device uses
vid	Vendor identifier
pid	Product identifier
version	Binary coded decimal version of device
address	Device address on USB bus
speed	Device speed (see “Device information defines” in *Table 6: USB defines in ioctls/usb.h on page 40*)
max_packet	Maximum sized packet that can be sent down the device control endpoint
num_configurations	Number of configuration supported by the device
manufacturer	String describing the manufacturer (if exists)
product	String describing the product (if exists)
serial	String containing serial number (if exists)

See also: OSPLUS_IOCTL_USB_GET_CONFIGURATION, OSPLUS_IOCTL_USB_GET_INTERFACE, OSPLUS_IOCTL_USB_GET_ENDPOINT

OSPLUS_IOCTL_USB_GET_CONFIGURATION

Return details of a USB device configuration

Arguments: The address of a `usb_configuration_t` structure to hold the configuration details returned.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl returns the details of a specified USB device configuration related to the information contained in a USB configuration descriptor. The `usb_configuration_t` structure whose address is passed-in must be first initialized with the index of the configuration to retrieve. The index starts at zero. After successfully completing the ioctl, the remainder of the `usb_configuration_t` structure is populated with the relevant data.

The `usb_configuration_t` structure has the following format:

```
typedef struct usb_configuration_s
{
      unsigned int configuration;
      unsigned int num_interfaces;
      unsigned int value;
      unsigned int attributes;
      unsigned int max_power;
      osplus_str_t description
} usb_configuration_t;
```

The fields are explained in *Table 70*.

Table 70. usb_configuration_t fields

Field name	Description
`configuration`	Index of the configuration to return
`num_interfaces`	Number of interfaces supported by the configuration
`value`	The value used to select this configuration
`attributes`	Bitmask of attributes for this configuration (see “Device configuration defines” in *Table 6: USB defines in ioctls/usb.h on page 40*)
`max_power`	Maximum power consumed by this interface
`description`	String describing configuration (if exists)

See also: `OSPLUS_IOCTL_USB_GET_DEVICE_INFO`, `OSPLUS_IOCTL_USB_GET_INTERFACE`, `OSPLUS_IOCTL_USB_GET_ENDPOINT`, `OSPLUS_IOCTL_USB_SET_CONFIGURATION`

OSPLUS_IOCTL_USB_GET_INTERFACE

Return details of a USB device interface

Arguments: The address of a `usb_interface_t` structure to hold the interface details returned.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl returns the details of a specified USB device interface for a specified USB device configuration related to the information contained in a USB interface descriptor. The `usb_interface_t` structure whose address is passed-in must be first initialized with the index of the configuration to retrieve, the index of the interface to retrieve, and the index of the interface alternate setting to retrieve. All indexes start at zero. After successfully completing the ioctl, the remainder of the `usb_interface_t` structure is populated with the relevant data.

If the alternate setting to retrieve is zero, then the `usb_interface_t` structure member `num_alt_settings` is populated with the number of alternate settings for the interface, else this member is set to zero.

The `usb_interface_t` structure has the following format:

```
typedef struct usb_interface_s
{
      unsigned int configuration;
      unsigned int interface;
      unsigned int alt_setting;
      unsigned int num_alt_settings;
      unsigned int num_endpoints;
      unsigned char if_class;
      unsigned char if_subclass;
      unsigned char if_protocol;
      osplus_str_t description;
} usb_interface_t;
```

The fields are explained in *Table 71*.

Table 71. **usb_interface_t fields**

Field name	Description
`configuration`	Index of the configuration containing the interface to return
`interface`	Index of the interface within the configuration to return
`alt_setting`	Index of the interface alternate setting to return
`num_alt_settings`	Number of alternate settings for interface
`num_endpoints`	Number of endpoints for this particular interface
`if_class`	USB class for interface (if defined on an interface basis)
`if_subclass`	USB subclass for interface (if defined on an interface basis)

Table 71. usb_interface_t fields (continued)

Field name	Description
`if_protocol`	USB protocol for interface (if defined on an interface basis)
`description`	String describing interface (if exists)

See also: `OSPLUS_IOCTL_USB_GET_DEVICE_INFO`, `OSPLUS_IOCTL_USB_GET_CONFIGURATION`, `OSPLUS_IOCTL_USB_GET_ENDPOINT`, `OSPLUS_IOCTL_USB_SET_INTERFACE`

OSPLUS_IOCTL_USB_GET_ENDPOINT

Return details of a USB device endpoint

Arguments: The address of an `usb_endpoint_t` structure to hold the endpoint details returned.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl returns the details of a specified USB device endpoint for a specified USB device interface for a specified USB device configuration related to the information contained in a USB endpoint descriptor. The `usb_interface_t` structure whose address is passed-in must be first initialized with the index of the configuration to retrieve, the index of the interface to retrieve, the index of the interface alternate setting to retrieve and the index of the endpoint to retrieve. All indexes start at zero. After successfully completing the ioctl, the remainder of the `usb_endpoint_t` structure is populated with the relevant data.

The `usb_endpoint_t` structure has the following format:

```
typedef struct usb_endpoint_s
{
      unsigned int configuration;
      unsigned int interface;
      unsigned int alt_setting;
      unsigned int endpoint;
      unsigned int number;
      unsigned char direction;
      unsigned char type;
      unsigned char sync;
      unsigned char usage;
      unsigned short max_packet;
      unsigned int interval;
} usb_endpoint_t;
```

The fields are explained in *Table 72*.

Table 72. usb_endpoint_t fields

Field name	Description
`configuration`	Index of the configuration containing the endpoint to return
`interface`	Index of the interface within the configuration containing the endpoint to return
`alt_setting`	Index of the interface alternate setting containing the endpoint to return
`endpoint`	Index of the endpoint to return
`number`	Number assigned to this endpoint
`direction`	Direction of the endpoint (IN or OUT, see “Device endpoint direction defines” n *Table 6: USB defines in ioctls/usb.h on page 40*)
`type`	Type of endpoint (isochronous, bulk, interrupt, see “Device endpoint transfer type defines” in *Table 6: USB defines in ioctls/usb.h on page 40*)
`sync`	Sync for endpoint (see “Device endpoint sync type defines” in *Table 6: USB defines in ioctls/usb.h on page 40*)
`usage`	Usage of endpoint (see “Device endpoint usage type defines” in *Table 6: USB defines in ioctls/usb.h on page 40*)
`max_packet`	Maximum sized packet this endpoint can handle
`interval`	Interval associated with this endpoint

See also: `OSPLUS_IOCTL_USB_GET_DEVICE_INFO`, `OSPLUS_IOCTL_USB_GET_CONFIGURATION`, `OSPLUS_IOCTL_USB_GET_INTERFACE`

OSPLUS_IOCTL_USB_SET_CONFIGURATION

Set the USB device configuration

Arguments: An unsigned 32-bit value indicating the index of the configuration to set.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl sets the USB device configuration to that specified by the index passed-in. If the USB device has only one configuration, then this configuration is automatically set by the USB stack.

See also: `OSPLUS_IOCTL_USB_GET_CONFIGURATION`, `OSPLUS_IOCTL_USB_SET_INTERFACE`

OSPLUS_IOCTL_USB_SET_INTERFACE

Set the USB device interface to use

Arguments: The address of an `usb_interface_t` structure populated with the details of the interface to set.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl sets the USB device interface to that specified in the `usb_interface_t` structure whose address is passed-in. The structure must be initialized with the index of the configuration containing the interface, the index of the interface to set, and the index of the interface alternate setting to set. The configuration must be set before attempting to set the interface. If the configuration index in the `usb_interface_t` structure does not match the configuration already set then an error is returned.

See also: `OSPLUS_IOCTL_USB_GET_INTERFACE`,
`OSPLUS_IOCTL_USB_SET_CONFIGURATION`

OSPLUS_IOCTL_USB_CTRL_TRANSFER

Perform a USB control transfer

Arguments: The address of a `usb_ctrl_transfer_t` structure containing the control transfer details.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl performs a USB control transfer on the USB device using endpoint 0 using the details specified in the `usb_ctrl_transfer_t` structure whose address is passed-in. The transfer can optionally send or receive data from or to a specified user buffer.

The `usb_ctrl_transfer_t` structure has the following format:

```
typedef struct usb_ctrl_transfer_s
{
      unsigned char direction;
      unsigned char type;
      unsigned char recipient;
      unsigned char request;
      unsigned short value;
      unsigned short index;
      void *data;
      unsigned short length;
} usb_ctrl_transfer_t;
```

The fields are explained in *Table 73*.

Table 73. usb_ctrl_transfer_t fields

Field name	Description
`direction`	Direction of the transfer (IN or OUT, see *Device endpoint direction defines* in *Table 6: USB defines in ioctls/usb.h on page 53*)
`type`	Transfer request type (see *Request type defines* in *Table 6: USB defines in ioctls/usb.h on page 53*)
`recipient`	Recipient of transfer (device, interface, see *Recipient type defines* in *Table 6: USB defines in ioctls/usb.h on page 53*)
`request`	Transfer request (see *Request defines* in *Table 6: USB defines in ioctls/usb.h on page 53*)
`value`	Transfer value parameter
`index`	Transfer index parameter
`data`	Pointer to any data buffer
`length`	Length of transfer

See also: `OSPLUS_IOCTL_USB_ISOC_TRANSFER`, `OSPLUS_IOCTL_USB_BULK_TRANSFER`, `OSPLUS_IOCTL_USB_INTR_TRANSFER`, `OSPLUS_IOCTL_USB_ABORT_TRANSFER`

OSPLUS_IOCTL_USB_ISOC_INFO

Get the information required to do a USB isochronous transfer

Arguments: The address of a `usb_isoc_info_t` structure containing the isochronous transfer information details.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl queries the USB device to determine how to perform a USB isochronous transfer on the USB device, and fills in the `usb_isoc_info_t` structure whose address is passed in. The structure must specify the index of the interface of the currently selected configuration, the interface alternate setting and the index of the endpoint that is to be used for the transfer. The information returned specifies the maximum number of frames for each USB isochronous transfer and the size of each frame. This information can be used to allocate cache safe buffers and prevent the actual transfer staging the data.

The interface alternate setting and endpoint specifed must be an isochronous endpoint with a maximum packet size of greater than zero.

The `usb_isoc_info_t` structure has the following format:

```
typedef struct usb_isoc_info_s
{
      unsigned int interface;
      unsigned int alt_setting;
      unisgned int endpoint;
      unsigned int frame_count;
      unsigned int frame_length;
} usb_isoc_info_t;
```

The fields are described in *Table 74*.

Table 74. usb_isco_info_t fields

Field name	Description
`interface`	Index of interface
`alt_setting`	Index of interface alternate setting
`endpoint`	Index of endpoint
`frame_count`	Maximum number of frames of data allowed per transfer
`frame_length`	Length of each frame of data

See also: `OSPLUS_IOCTL_USB_ISOC_TRANSFER`

OSPLUS_IOCTL_USB_ISOC_TRANSFER

Perform a USB isochronous transfer

Arguments: The address of a `usb_isoc_transfer_t` structure containing the isochronous transfer details.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl performs a USB isochronous transfer on the USB device using the details specified in the `usb_isoc_transfer_t` structure whose address is passed-in. The structure must specify the index of the interface of the currently selected configuration and the index of the endpoint to use for the transfer along with a data buffer and data buffer length.

The endpoint to perform the transfer on must be an isochronous endpoint and the frame length specified must be less than or equal to the endpoint maximum packet size, otherwise an error is returned. Also the endpoint direction determines if the transfer is a read or write (an IN endpoint specifies a read, an OUT endpoint specifies a write).

After the transfer has completed or on error, the transfer callback is executed. The callback is supplied with the status of the transfer as defined in *Table 6: USB defines in ioctls/usb.h on page 40*, a pointer to the user data buffer, the actual length of data in the buffer and the user specified cookie.

The transfer must not exceed the maximum number of frames or the frame length as returned by the ioctl `OSPLUS_IOCTL_USB_ISOC_INFO`, otherwise the transfer will fail. The callback is executed when the transfer has completed and is expected to return `USB_TRANSFER_DONE`. The values of `frame_count` and `frame_length` in the `usb_isoc_transfer_t` structure must not exceed the coresponding values returned by `OSPLUS_IOCTL_USB_ISOC_INFO`.

The `usb_isoc_transfer_t` structure has the following format:

```
typedef struct usb_isoc_transfer_s
{
      unsigned int interface;
      unisgned int endpoint;
      void *data;
      unsigned int frame_count;
      unsigned int frame_length;
      unsigned int frame_number;
      void *cookie;
      int (*callback) (unsigned int status, void *data,
           unsigned int length, void *cookie);
} usb_isoc_transfer_t;
```

The fields are explained in *Table 75*.

Table 75. usb_isco_transfer_t fields

Field name	Description
`interface`	Index of interface to perform transfer over
`endpoint`	Index of endpoint to perform transfer over
`data`	Pointer to data buffer
`frame_count`	Number of frames of data. This must not exceed the number returned by ioctl `OSPLUS_IOCTL_USB_ISOC_INFO`.
`frame_length`	Length of each frame of data. This must not exceed the size returned by ioctl `OSPLUS_IOCTL_USB_ISOC_INFO`.
`frame_number`	First frame number to use for this transfer. `0xffffffff` indicates no frame number. This field will be updated with the next frame number to use after the ioctl completes.
`cookie`	User supplied cookie for use with callback
`callback`	User supplied callback issued after transfer completes

See also: `OSPLUS_IOCTL_USB_ISOC_INFO`, `OSPLUS_IOCTL_USB_CTRL_TRANSFER`, `OSPLUS_IOCTL_USB_BULK_TRANSFER`, `OSPLUS_IOCTL_USB_INTR_TRANSFER`, `OSPLUS_IOCTL_USB_ABORT_TRANSFER`

OSPLUS_IOCTL_USB_BULK_TRANSFER

Perform a USB bulk transfer

Arguments: The address of a `usb_bulk_transfer_t` structure containing the bulk transfer details.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl performs a USB bulk transfer on the USB device using the details specified in the `usb_bulk_transfer_t` structure whose address is passed-in. The structure must specify the index of the interface of the currently selected configuration and the index of the endpoint to use for the transfer along with a data buffer and data buffer length.

The endpoint to perform the transfer on must be a bulk endpoint, otherwise an error is returned. Also the endpoint direction determines if the bulk transfer is a read or write (an IN endpoint specifies a read, an OUT endpoint specifies a write).

The transfer can be executed synchronously or asynchronously.

– If the `callback` field is `NULL`, the transfer executes synchronously. This means that it must wait until all other transfers are complete before starting. After the transfer has completed, the actual number of bytes transferred is returned in the `actual_length` structure member.

  Synchronous transfers can be of arbitrary length. If the data buffer is capable of undergoing DMA transfer (correctly aligned in memory), then the transfer occurs in 128 KByte units. If the buffer is not capable of DMA transfer, then the data buffer is staged in units of up to 32 KBytes at a time.

– If a valid `callback` is specified, the callback is executed asynchronously. The bulk transfer is set up and the ioctl returns success. The transfer then executes at some future point. After it has executed, the callback is called with the status of the transfer as defined in *Table 6: USB defines in ioctls/usb.h on page 40*, a pointer to the user data buffer, the actual length of data in the buffer and the user specified cookie. The callback should always return `USB_TRANSFER_DONE`. Posted transfers can only contain up to 128 Kbytes of data.

  Asynchronous transfers must supply a DMA-capable buffer less than the maximum transfer size (128 KBytes). Asynchronous transfers cannot stage their data.

The `usb_bulk_transfer_t` structure has the following format:

```
typedef struct usb_bulk_transfer_s
{
      unsigned int interface;
      unisgned int endpoint;
      void *data;
      unsigned int length;
      unsigned int actual_length;
      void *cookie;
      int (*callback) (unsigned int status, void *data,
              unsigned int length, void *cookie);
} usb_bulk_transfer_t;
```

The fields are explained in *Table 76*.

Table 76. usb_bulk_transfer_t fields

Field name	Description
`interface`	Index of interface to perform transfer over
`endpoint`	Index of endpoint to perform transfer over
`data`	Pointer to data buffer
`length`	Length of data buffer
`actual_length`	Length of data transferred
`cookie`	User supplied cookie for use with callback
`callback`	User supplied callback issued after transfer completes

See also: `OSPLUS_IOCTL_USB_CTRL_TRANSFER`, `OSPLUS_IOCTL_USB_ISOC_TRANSFER`, `OSPLUS_IOCTL_USB_INTR_TRANSFER`, `OSPLUS_IOCTL_USB_ABORT_TRANSFER`

OSPLUS_IOCTL_USB_INTR_TRANSFER

Perform a USB interrupt transfer

Arguments: The address of an `usb_intr_transfer_t` structure containing the interrupt transfer details.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl performs a USB interrupt transfer on the USB device using the details specified in the `usb_intr_transfer_t` structure whose address is passed-in. The structure must specify the index of the interface of the currently selected configuration and the index of the endpoint to use for the transfer along with a data buffer, the data buffer length and a callback.

The endpoint to perform the transfer on must be an interrupt endpoint, otherwise an error is returned. Also the endpoint direction determines if the interrupt transfer is a read or write (an IN endpoint specifies a read, an OUT endpoint specifies a write).

After the transfer has completed or on error, the transfer callback is executed. The callback is supplied with the status of the transfer as defined in *Table 6: USB defines in ioctls/usb.h on page 40*, a pointer to the user data buffer, the actual length of data in the buffer and the user specified cookie.

After processing the data in the user callback, the callback should return one of two possible values as defined in *Table 6: USB defines in ioctls/usb.h on page 40*. If there are no more interrupt transfers to perform for the particular endpoint then the callback should return `USB_TRANSFER_DONE`, otherwise the callback should return `USB_TRANSFER_REHOOK`. On returning `USB_TRANSFER_REHOOK` the interrupt transfer is rehooked ready to send or receive more data.

The `usb_intr_transfer_t` structure has the following format:

```
typedef struct usb_intr_transfer_s
{
      unsigned int interface;
      unisgned int endpoint;
      void *data;
      unsigned int length;
      void *cookie;
      int (*callback) (unsigned int status, void *data, unsigned
int length, void *cookie);
} usb_intr_transfer_t;
```

The fields are explained in *Table 77*.

Table 77. usb_intr_transfer_t fields

Field name	Description
`interface`	Index of interface to perform transfer over
`endpoint`	Index of endpoint to perform transfer over
`data`	Pointer to data buffer
`length`	Length of data buffer
`cookie`	User supplied cookie for use with callback
`callback`	User supplied callback issued after transfer completes

See also: `OSPLUS_IOCTL_USB_CTRL_TRANSFER`, `OSPLUS_IOCTL_USB_ISOC_TRANSFER`, `OSPLUS_IOCTL_USB_BULK_TRANSFER`, `OSPLUS_IOCTL_USB_ABORT_TRANSFER`

OSPLUS_IOCTL_USB_ABORT_TRANSFER

Abort all transfers on a specified endpoint

Arguments: The address of a `usb_abort_transfer_t` structure containing the details of the transfer to abort.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl aborts all outstanding transfers for the specified interface endpoint. The `usb_abort_transfer_t` structure whose address is passed-in must specify the interface index of the currently selected configuration and the endpoint index.

The `usb_abort_transfer_t` structure has the following format:

```
typedef struct usb_abort_transfer_s
{
      unsigned int interface;
      unisgned int endpoint;
} usb_abort_transfer_t;
```

The fields are explained in *Table 78*.

Table 78. usb_abort_transfer_t fields

Field name	Description
`interface`	Index of interface on which to abort the transfer
`endpoint`	Index of endpoint on which to abort transfer

See also: `OSPLUS_IOCTL_USB_CTRL_TRANSFER`, `OSPLUS_IOCTL_USB_ISOC_TRANSFER`, `OSPLUS_IOCTL_USB_BULK_TRANSFER`, `OSPLUS_IOCTL_USB_INTR_TRANSFER`

OSPLUS_IOCTL_USB_PORT_INFO

Query host controller connection information for a device

Arguments: The address of a `usb_port_info_t` structure which is filled in by the call.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl returns information about the host controller and root hub port to which the device is connected. The address of a `usb_port_info_t` structure is passed-in and is filled-in by this call. The `usb_port_info_t` structure has the following format:

```
typedef struct usb_port_info_s
{
          const char host_controller_name[16];
          unisgned int port;
} usb_port_info_t;
```

The fields are explained in *Table 79*.

Table 79. usb_port_info_t fields

Field name	Description
`host_controller_name`	The device name of the host controller to which the device is connected
`port`	Root hub port number on this host controller that is connected to the device

See also: `OSPLUS_IOCTL_USB_TEST_PORT`

OSPLUS_IOCTL_USB_TEST_PORT

Place the root hub port which connects to the device in to a test mode

Arguments: A `usb_host_test_mode_t` describing the test mode to select.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl places the root hub port, to which the device is connected, into the specified test mode. The `usb_host_test_mode_t` type has the following values:

```
typedef enum
{
      USB_HOST_TEST_MODE_OFF,
      USB_HOST_TEST_MODE_J_STATE,
      USB_HOST_TEST_MODE_K_STATE,
      USB_HOST_TEST_MODE_SE0_NAK,
      USB_HOST_TEST_MODE_PACKET,
      USB_HOST_TEST_MODE_FORCE_ENABLE,
      USB_HOST_TEST_MODE_INVALID
} usb_host_test_mode_t;
```

Note: Only EHCI host controllers support USB test modes, so this ioctl will fail if the device is not connected to an EHCI host controller.

For a discussion of USB test modes see the documents *Enhanced Host Controller Interface Specification for Universal Serial Bus* section 4.14, and *Universal Serial Bus Specification Revision 2.0* section 7.1.20. Both are available from *www.usb.org*.

See also: `OSPLUS_IOCTL_USB_PORT_INFO`

OSPLUS_IOCTL_USB_TEST_1A0A_0107

Test the root hub port that connects to the device

Arguments: None.

Returns: Returns 0 for success, -1 for failure.

Description: This ioctl is intended for USB certification and testing purposes. In accordance with the “On-The-Go Supplement to the USB 2.0 Specification Revision 1.3” document (available from www.usb.org) this ioctl performs steps (3) and (4) of the `SINGLE_STEP_GET_DEVICE_DESC` test. It should only be issued to a device with a vendor ID of 0x1A0A and a product ID of 0x0107). The port_test example shows how to use this ioctl, and ensures that the other aspects of the test are performed correctly.

See also: `OSPLUS_IOCTL_USB_TEST_1A0A_0108`

OSPLUS_IOCTL_USB_TEST_1A0A_0108

Test the root hub port that connects to the device

Arguments: None.

Returns: Returns 0 for success, -1 for failure.

Description: This ioctl is intended for USB certification and testing purposes. In accordance with the “On-The-Go Supplement to the USB 2.0 Specification Revision 1.3” document (available from www.usb.org) this ioctl performs steps (2) through (7) of the `SINGLE_STEP_GET_DEVICE_DESC_DATA` test. It should only be issued to a device with a vendor ID of 0x1A0A and a product ID of 0x0108). The port_test example shows how to use this ioctl, and ensures that the other aspects of the test are performed correctly.

See also: `OSPLUS_IOCTL_USB_TEST_1A0A_0107`

OSPLUS_IOCTL_USB_SUSPEND_PORT

Suspend the root hub port that connects to the device

Arguments: None.

Returns: Returns 0 for success, -1 for failure.

Description: This ioctl suspends the root hub port that connects to the device in accordance with the ‘Universal Serial Bus Specification Revision 2.0’ document available from www.usb.org.

See also: `OSPLUS_IOCTL_USB_RESUME_PORT`

OSPLUS_IOCTL_USB_RESUME_PORT

Resume the root hub port that connects to the device

Arguments: None.

Returns: Returns 0 for success, -1 for failure.

Description: This ioctl resumes the root hub port that connects to the device in accordance with the ‘Universal Serial Bus Specification Revision 2.0’ document available from www.usb.org.

See also: `OSPLUS_IOCTL_USB_RESUME_PORT`

OSPLUS_IOCTL_USB_COMPOSITE_INFO

Return the USB device composite information

Arguments: The address of a `usb_composite_info_t` structure which is filled in by the call.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl returns composite information about the device including the name and type of each device. The address of a `usb_composite_info_t` structure is passed in and is filled in by this call. The `usb_composite_info_t` structure has the following format:

```
typedef struct usb_composite_info_s
{
     unsigned int dev_id;
     const char *dev_name;
     const char *dev_type;
} usb_composite_info_t;
```

The fields are explained in *Table 80*.

If a device is currently enumerating, then this ioctl returns `-1` and sets `errno` to `EBUSY`.

Table 80. usb_composite_info_t fields

Field name	Description
`dev_id`	Get the name and type of the composite device with this index. If set to `0xffffffff`, then the ioctl sets this field to the number of composites the device contains.
`dev_name`	The device name of the composite.
`dev_type`	The device type of the composite.

OSPLUS_IOCTL_USB_TREE_NODE_INFO

Return the USB device tree node parent and child information

Arguments: The address of a usb_tree_node_info_t structure which is filled in by the call.

Returns: Returns 0 for success, -1 for failure.

Description: This ioctl returns tree node information about the device including the name and port of the parent device that it is attached to, and the name and port of each child device attached to it.

The address of a usb_composite_info_t structure is passed in and is filled in by this call. The usb_composite_info_t structure has the following format:

```
typedef struct usb_tree_node_info_s
{
      const char *device_name;
      const char *parent_name;
      unsigned int parent_port;
      const char *child_names[128];
      unsigned int child_ports;
} usb_tree_node_info_t;
```

The fields are explained in *Table 81*.

If a device is currently enumerating, then this ioctl returns -1 and sets errno to EBUSY.

Table 81. usb_tree_node_info_t fields

Field name	Description
device_name	The device name of this device.
parent_name	The name of the parent device attached to, or null if no parent device (as is the case with a host controller).
parent_port	The port of the parent device attached to. This is only valid when the parent_name is not null.
child_names	A pointer to the name of each device attached to a port on this device. This is only valid when child_ports is not 0. If the name for a port is null, then this implies there is no device attached to that port.
child_ports	The number of ports attached to this device. If device is not a hub or host controller then this is 0.

3.2 USB audio ioctl definitions

These USB audio class device specific ioctl definitions relate to the `device_ioctl()` call. For further information, refer to the *OSPlus User Manual* (ADCS 7702033).

OSPLUS_IOCTL_USB_AUDIO_GET_UNIT_INFO

Return basic information about an audio unit

Arguments: A pointer to an `usb_audio_unit_t` structure in memory to receive the unit information.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl returns the basic audio unit information including the type of audio unit, audio channel configuration and identifies the capture and playback controls. Pass the address of an `usb_audio_unit_t` structure to be filled in by this call. The `usb_audio_unit_t` structure has the following format:

```
typedef struct usb_audio_unit_s
{
     unsigned int   utype;
     unsigned int   attributes;
     unsigned char  channel_count;
     unsigned int   channel_config;
     unsigned int   capture_features;
     unsigned int   playback_features;
} usb_audio_unit_t;
```

The fields are explained in *Table 82*.

Table 82. usb_audio_unit_t fields

Field name	Description
`utype`	Unit/terminal type as specified in USB Class Definition for Terminal Types r1.0.
`attributes`	Unit attributes, such as if unit is for capture or playback.
`channel_count`	Number of output channels from unit.
`channel_config`	Bitmap defining the channel configuration. Each bit represents a particular channel as defined in *Table 9: Audio terminal attribute defines in ioctls/usb/audio.h on page 44*.
`capture_features`	Bitmap of capture feature controls supported. Each bit represents a particular control as defined in *Table 9: Audio terminal attribute defines in ioctls/usb/audio.h on page 44*.
`playback_features`	Bitmap of playback feature controls supported. Each bit represents a particular control as defined in *Table 9: Audio terminal attribute defines in ioctls/usb/audio.h on page 44*.

See also: `OSPLUS_IOCTL_USB_AUDIO_GET_UNIT_INPUTS`, `OSPLUS_IOCTL_USB_AUDIO_GET_UNIT_OUTPUTS`

OSPLUS_IOCTL_USB_AUDIO_GET_UNIT_INPUTS

Return a list containing the names of all units providing inputs to the current unit

Arguments: A pointer to an `usb_audio_names_t` structure in memory to receive the unit input names.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl returns a list of those audio units that provide an input source to the current audio unit. Pass the address of an `usb_audio_names_t` structure to be filled in to this call. The `usb_audio_names_t` structure has the following format:

```
typedef struct usb_audio_names_s {
     unsigned int  count;
     char
names[USB_AUDIO_MAX_OUTPUTS][USB_AUDIO_MAX_LABEL_SIZE];
} usb_audio_names_t;
```

The fields are explained in *Table 83*.

Table 83. usb_audio_name_t fields

Field name	Description
`count`	The number of entries for the names array.
`names`	An array of strings containing names.

See also: `OSPLUS_IOCTL_USB_AUDIO_GET_UNIT_INFO`, `OSPLUS_IOCTL_USB_AUDIO_GET_UNIT_OUTPUTS`

OSPLUS_IOCTL_USB_AUDIO_GET_UNIT_OUTPUTS

Return a list containing the names of all units to which the current unit provides outputs

Arguments: A pointer to an `usb_audio_names_t` structure in memory to receive the unit output names.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl returns a list of those audio units to which the current audio unit provides outputs. Pass the address of an `usb_audio_names_t` structure to be filled in by this call. The structure of `usb_audio_names_t` is described in the definition of *OSPLUS_IOCTL_USB_AUDIO_GET_UNIT_INPUTS*.

See also: `OSPLUS_IOCTL_USB_AUDIO_GET_UNIT_INFO`, `OSPLUS_IOCTL_USB_AUDIO_GET_UNIT_INPUTS`

OSPLUS_IOCTL_USB_AUDIO_GET_FEATURE

Get a specified feature control current value and other information

Arguments: A pointer to an `usb_audio_feature_t` structure in memory to receive the feature control information.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl returns information about a particular feature control for an audio unit. The information includes:

– the attributes of the feature

– the audio channels controlled by the feature

– the minimum and maximum permitted values for the feature

– the resolution of the feature

– the current values of the feature for each supported channel

Pass the address of an `usb_audio_feature_t` structure to be filled in by this call. The structure passed in must be initialized such that the `attributes` field indicates what feature control to return information for and if the control is for capture or playback.

When the ioctl completes, the `attributes` field contains the full set of the control attributes, such as the control size, and the operations it supports (e.g. setting the current value). The `cn_mask` field indicates the channels that the feature controls. If the `cn_mask` field is zero, then the feature controls the master audio channel, otherwise each bit set in the mask corresponds to a logical channel (e.g. bit 0 for logical channel 1, bit 1 for logical channel 2 etc.). The `value_cur` field is an array that holds the current value for each supported channel. In this array, index zero contains the master channel value, index 1 contains logical channel 1 value, and so forth.

The `usb_audio_feature_t` structure has the following format:

```
typedef struct usb_audio_feature_s
{
      unsigned int   attributes;
      unsigned short cn_mask;
      unsigned short value_cur[USB_AUDIO_MAX_CHANNELS + 1];
      unsigned short value_min;
      unsigned short value_max;
      unsigned short value_res;
} usb_audio_feature_t;
```

The fields are explained in *Table 84*.

Table 84. usb_audio_feature_t fields

Field name	Description
`attributes`	Bitmap of attributes as defined in *Table 9: Audio terminal attribute defines in ioctls/usb/audio.h on page 44*
`cn_mask`	Mask of channels for which there is a value. Bit 1 means logical channel 1, bit 2 logical channel 2 etc. If cn_mask is zero then this means the master channel.
`value_cur`	Array holding current value of feature for each possible channel.
`value_min`	Minimum value of feature control
`value_max`	Maximum value of feature control
`value_res`	Resolution by which the feature control increases and decreases in value.

See also: `OSPLUS_IOCTL_USB_AUDIO_SET_FEATURE`

OSPLUS_IOCTL_USB_AUDIO_SET_FEATURE

Set a specified feature control current value

Arguments: A pointer to an `usb_audio_feature_t` structure in memory containing the feature control current value to set.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl sets the value of a feature control. Pass the address of an `usb_audio_feature_t` structure that has been initialized with the values to set.

The `attributes` field of the structure indicates the control to set and if the control is for capture or playback, the `cn_mask` field specifies the channels to set (with zero indicating the master channel, bit 1 set to indicate logical channel 1, bit 2 set to indicate logical channel 2, and so forth) and the `value_cur` array field is set so that it contains the current value to set for each channel specified in the `cn_mask` field, where index zero indicates the master channel, index 1 logical channel 1 and so forth.

The structure `usb_audio_feature_t` is defined in the definition of *OSPLUS_IOCTL_USB_AUDIO_GET_FEATURE*.

See also: `OSPLUS_IOCTL_USB_AUDIO_GET_FEATURE`

OSPLUS_IOCTL_USB_AUDIO_GET_MIXER_INFO

Return information about a mixer's input channels

Arguments: A pointer to an `usb_audio_mixer_info_t` structure in memory to receive the mixer unit input channel information.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl returns the mixer unit attributes and input channel information. This information gives the number of inputs available and how the input channels map to the inputs. Pass the address of a `usb_audio_mixer_info_t` structure to be filled in by this call. The `usb_audio_mixer_info_t` structure has the following format:

```
typedef struct usb_audio_mixer_info_s
{
      unsigned int   attributes;
      unsigned char  inputs;
      unsigned char  icn_start[USB_AUDIO_MAX_INPUTS];
      unsigned char  icn_count[USB_AUDIO_MAX_INPUTS];
} usb_audio_mixer_info_t;
```

The fields are explained in *Table 85*.

Table 85. usb_audio_mixer_info_t fields

Field name	Description
`attributes`	Bitmap of attributes as defined in *Table 9: Audio terminal attribute defines in ioctls/usb/audio.h on page 44*
`inputs`	Number of mixer inputs
`icn_start`	Array containing the start channel for each mixer input
`icn_count`	Array containing the number of channels for each mixer input.

See also: `OSPLUS_IOCTL_USB_AUDIO_GET_MIXER`, `OSPLUS_IOCTL_USB_AUDIO_SET_MIXER`

OSPLUS_IOCTL_USB_AUDIO_GET_MIXER

Get a mixer input channel current value and other information

Arguments: A pointer to an `usb_audio_mixer_t` structure in memory to receive the mixer input to output channel mapping information.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl returns the mixer unit output channels mapped to a particular mixer input channel and the current volume values for those output channels. Pass the address of a `usb_audio_mixer_t` structure to be filled in by this call. Initialize the `icn` field in this structure to indicate the input channel for which the mappings are required.

On return, the `ocn_mask` field indicates which output channels the input channel maps to (with bit 1 set to indicate logical output channel 1, bit 2 set to indicate logical output channel 2, and so forth) and the `value_cur` array field is set to the current value for each channel specified in the `ocn_mask` field, where index 1 holds the value for logical output channel 1, index 2 for logical output 2 and so forth.

The `usb_audio_mixer_t` structure has the following format:

```
typedef struct usb_audio_mixer_s
{
      unsigned char  icn;
      unsigned short ocn_mask;
      unsigned short value_cur[USB_AUDIO_MAX_CHANNELS + 1];
      unsigned short value_min;
      unsigned short value_max;
      unsigned short value_res;
} usb_audio_mixer_t;
```

The fields are explained in *Table 86*.

Table 86. usb_audio_mixer_t fields

Field name	Description
`icn`	Mixer input channel.
`ocn_mask`	Mask of output channels that this input channel maps to. Bit 1 means logical channel 1, bit 2 logical channel 2 etc.
`value_cur`	Array holding current value of feature for each possible channel.
`value_min`	Minimum value of control
`value_max`	Maximum value of control
`value_res`	Resolution by which the control increases and decreases in value.

See also: `OSPLUS_IOCTL_USB_AUDIO_GET_MIXER_INFO`, `OSPLUS_IOCTL_USB_AUDIO_SET_MIXER`

OSPLUS_IOCTL_USB_AUDIO_SET_MIXER

Set a mixer input channel current value

Arguments: A pointer to an `usb_audio_mixer_t` structure in memory containing the mixer input to output channel mapping current value to set.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl sets the value for the volume of mapping from mixer input channel to mixer output channel. Pass the address of a `usb_audio_mixer_t` structure to be filled in by this call. Initialize the `icn` field in this structure to indicate the input channel for which the volume is to be set, and initialize the `ocn_mask` field to indicate which output channels to set (where bit 1 set indicates logical output channel 1, bit 2 set indicates logical output channel 2, and so forth.) Initialize the `value_cur` array field to indicate the volume to set for each channel specified in the `ocn_mask` field, where index 1 holds the value for logical output channel 1, index 2 the value for logical output channel 2, and so forth.

The structure `usb_audio_mixer_t` is described in the definition of *OSPLUS_IOCTL_USB_AUDIO_GET_MIXER*

See also: `OSPLUS_IOCTL_USB_AUDIO_GET_MIXER_INFO`, `OSPLUS_IOCTL_USB_AUDIO_GET_MIXER`

OSPLUS_IOCTL_USB_AUDIO_GET_SELECTOR

Get a selector unit current value and other information

Arguments: A pointer to an `usb_audio_selector_t` structure in memory to receive the selector unit information.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl gets the current value, minimum value and maximum value of a selector unit. Pass the address of a `usb_audio_selector_t` structure to be filled in by this call. The `usb_audio_selector_t` structure has the following format:

```
typedef struct usb_audio_selector_s
{
      unsigned short value_cur;
      unsigned short value_min;
      unsigned short value_max;
} usb_audio_selector_t;
```

The fields are explained in *Table 87*.

Table 87. usb_audio_selector_t fields

Field name	Description
`value_cur`	Current value of the selector unit
`value_min`	Minimum value of the selector unit
`value_max`	Maximum value of the selector unit

See also: `OSPLUS_IOCTL_USB_AUDIO_SET_SELECTOR`

OSPLUS_IOCTL_USB_AUDIO_SET_SELECTOR

Set a selector unit current value

Arguments: A pointer to an `usb_audio_selector_t` structure in memory containing the selector unit input to select.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl sets the current value of a selector unit. Initialize a `usb_audio_selector_t` structure with the `value_cur` set to the number of the input to select and pass in to the call. The structure `usb_audio_selector_t` is described in the definition of *OSPLUS_IOCTL_USB_AUDIO_GET_SELECTOR*.

See also: `OSPLUS_IOCTL_USB_AUDIO_GET_SELECTOR`

OSPLUS_IOCTL_USB_AUDIO_GET_FORMAT_COUNT

Return the number of audio formats for the current USB terminal

Arguments: The address of an unsigned integer to receive the number of audio formats the USB terminal supports.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl returns the number of audio formats for the current USB terminal device. Pass the address of an integer variable; this contains the number of audio formats when the ioctl returns.

See also: `OSPLUS_IOCTL_USB_AUDIO_GET_FORMAT_LIST`,
`OSPLUS_IOCTL_USB_AUDIO_GET_FORMAT`,
`OSPLUS_IOCTL_USB_AUDIO_SET_FORMAT`

OSPLUS_IOCTL_USB_AUDIO_GET_FORMAT_LIST

Return a list of audio formats for the current USB terminal

Arguments: The address of an `usb_audio_format_t` structure to receive the audio formats the USB terminal supports.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl returns a list of audio formats for the current USB terminal device. Pass the address of an array of `usb_audio_format_t` structures to be filled in by this call. The array contains an entry for each audio format the USB terminal device supports; use `OSPLUS_IOCTL_USB_AUDIO_GET_FORMAT_COUNT` to determine the number of elements that the array requires. The `usb_audio_format_t` structure has the following format:

```
typedef struct usb_audio_format_s
{
     unsigned short format_tag;
     unsigned char  format_type;
     unsigned char  channel_count;
     unsigned char  frame_delay;
     unsigned char  frame_size;
     unsigned short max_bitrate;
     unsigned short sample_frame;
     unsigned char  sample_size;
     unsigned int   sample_rate;
     unsigned int   sample_min;
     unsigned int   sample_max;
} usb_audio_format_t;
```

The fields are explained in *Table 88*.

Table 88. usb_audio_format_t fields

Field name	Description
`format_tag`	Format tag for audio as listed in *Table 11: Audio format defines in ioctls/usb/audio.h on page 45*
`format_type`	Integer specifying format type. Valid values are 1, 2, and 3.
`channel_count`	Number of channels this format supports
`frame_delay`	Delay introduced by the data path in audio frames.
`frame_size`	Frame size in bytes
`max_bitrate`	Maximum bitrate for Type II formats.
`sample_frame`	Samples per frame for Type II formats
`sample_size`	Number of bits per sample
`sample_rate`	Sample rate in Hz

Table 88. usb_audio_format_t fields (continued)

Field name	Description
`sample_min`	Minimum sample rate in Hz
`sample_max`	Maximum sample rate in Hz

See also: `OSPLUS_IOCTL_USB_AUDIO_GET_FORMAT_COUNT`, `OSPLUS_IOCTL_USB_AUDIO_GET_FORMAT`, `OSPLUS_IOCTL_USB_AUDIO_SET_FORMAT`

OSPLUS_IOCTL_USB_AUDIO_GET_FORMAT

Get the current audio format for the current USB terminal

Arguments: The address of an `usb_audio_format_t` structure to receive the current audio format the USB terminal has set.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl returns the currently set audio format for the USB terminal. Pass the address of an `usb_audio_format_t` structure to be filled in by this call. The structure `usb_audio_format_t` is described in the definition of `OSPLUS_IOCTL_USB_AUDIO_GET_FORMAT_LIST`.

See also: `OSPLUS_IOCTL_USB_AUDIO_GET_FORMAT_COUNT`, `OSPLUS_IOCTL_USB_AUDIO_GET_FORMAT_LIST`, `OSPLUS_IOCTL_USB_AUDIO_SET_FORMAT`

OSPLUS_IOCTL_USB_AUDIO_SET_FORMAT

Set the current audio format for the current USB terminal

Arguments: The address of an `usb_audio_format_t` structure containing the current audio format to set.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl sets the audio format for the current USB terminal device. Before calling this ioctl, initialize the structure `usb_audio_format_t` pointed to by the argument as follows:

– set the `format_tag` field to one of the types defined in *Table 11: Audio format defines in ioctls/usb/audio.h on page 45*

– set the field `channel_count` to the correct number of channels

– set the `sample_size` field to the sample size (e.g. 8-bit, 16-bit)

– set the `sample_rate` to the sampling frequency in Hertz

See also: `OSPLUS_IOCTL_USB_AUDIO_GET_FORMAT_COUNT`, `OSPLUS_IOCTL_USB_AUDIO_GET_FORMAT_LIST`, `OSPLUS_IOCTL_USB_AUDIO_GET_FORMAT`

OSPLUS_IOCTL_USB_AUDIO_TRANSFER_STATUS

Get the status of the audio transfer engine and other information

Arguments: The address of an `usb_audio_transfer_status_t` structure to receive the current audio transfer engine status.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl returns the audio transfer engine status. Pass the address of an `usb_audio_transfer_status_t` structure to be filled in by this call.

In addition to the transfer engine status, this call also returns the following information:

– the amount of free buffer space in the audio transfer engine staging buffer
– the amount of used buffer space in the audio transfer engine staging buffer
– the number of audio input bytes that have been dropped due to lack of buffer space

The `usb_audio_transfer_status_t` structure has the following format:

```
typedef struct usb_audio_transfer_status_s
{
      unsigned int status;
      unsigned int processing;
      unsigned int buffer_free;
      unsigned int buffer_used;
      unsigned int buffer_drop;
} usb_audio_transfer_status_t;
```

The fields are explained in *Table 89*.

Table 89. usb_audio_transfer_status_t fields

Field name	Description
`status`	Current status of the audio transfer engine as defined in *Table 12: Audio transfer defines in ioctls/usb/audio.h on page 45*.
`processing`	Indicates if the audio transfer engine is currently processing audio.
`buffer_free`	Amount of free space in the audio transfer engine internal buffers.
`buffer_used`	Amount of used space in the audio transfer engine internal buffers.
`buffer_drop`	Amount of bytes dropped when processing audio input due to lack of space in internal buffers.

See also: `OSPLUS_IOCTL_USB_AUDIO_TRANSFER_START`, `OSPLUS_IOCTL_USB_AUDIO_TRANSFER_STOP`, `OSPLUS_IOCTL_USB_AUDIO_TRANSFER_PURGE`

OSPLUS_IOCTL_USB_AUDIO_TRANSFER_START

Start the audio transfer engine

Arguments: None.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl starts the audio transfer engine. If any audio transfers are queued, they will be processed when this ioctl is called.

See also: `OSPLUS_IOCTL_USB_AUDIO_TRANSFER_STOP`, `OSPLUS_IOCTL_USB_AUDIO_TRANSFER_PURGE`

OSPLUS_IOCTL_USB_AUDIO_TRANSFER_STOP

Stop the audio transfer engine

Arguments: None.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl stops the audio transfer engine. Any transfers still in progress are allowed to complete, but any transfers still queued are not processed.

See also: `OSPLUS_IOCTL_USB_AUDIO_TRANSFER_START`, `OSPLUS_IOCTL_USB_AUDIO_TRANSFER_PURGE`

OSPLUS_IOCTL_USB_AUDIO_TRANSFER_PURGE

Purge all data from the audio transfer engine internal buffers

Arguments: None.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl purges all data from the audio transfer engine internal buffers.

See also: `OSPLUS_IOCTL_USB_AUDIO_TRANSFER_START`, `OSPLUS_IOCTL_USB_AUDIO_TRANSFER_STOP`

3.3 USB communications ioctl definitions

These USB communications class device specific ioctl definitions relate to the `device_ioctl()` call. For further information, refer to the *OSPlus User Manual (ADCS 7702033)*.

OSPLUS_IOCTL_USB_COMM_SET_LISTENER

Install a communications class notification callback

Arguments: The address of an `usb_comm_callback_t` structure that holds the details of the callback to execute on status change.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl installs a callback that is executed each time the communications class device status changes. The callback data is supplied in `usb_comm_callback_t` structure whose address is passed-in. Only one notification callback can be registered per device.

The `usb_comm_callback_t` structure has the following format:

```
typedef struct usb_comm_callback_s
{
     void (*callback)(unsigned int event,
              unsigned int arg1,
              unsigned int arg2,
              void *cookie);
     void *cookie;
} usb_comm_callback_t;
```

The fields are explained in *Table 90*.

Table 90. usb_comm_callback_t fields

Field name	Description
`callback`	Actual callback function to execute
`cookie`	Cookie passed back to the callback

See also: `OSPLUS_IOCTL_USB_COMM_GET_LISTENER`, `OSPLUS_IOCTL_USB_COMM_CLR_LISTENER`

OSPLUS_IOCTL_USB_COMM_GET_LISTENER

Return an installed communications class notification callback

Arguments: The address of an `usb_comm_callback_t` structure to hold the details of the installed callback.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl populates the `usb_comm_callback_t` structure whose address is passed-in with the details of the installed notification callback. If no notification callback is installed, each filed in the `usb_comm_callback_t` structure is set to `NULL`.

See also: `OSPLUS_IOCTL_USB_COMM_SET_LISTENER`, `OSPLUS_IOCTL_USB_COMM_GLR_LISTENER`

OSPLUS_IOCTL_USB_COMM_CLR_LISTENER

Remove an installed communications class notification callback

Arguments: The address of an `usb_comm_callback_t` structure that holds the details of the installed callback.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl removes the currently installed notification callback. The `usb_comm_callback_t` structure passed-in must contain the currently installed notification callback details. If it does not the function will fail.

See also: `OSPLUS_IOCTL_USB_COMM_SET_LISTENER`, `OSPLUS_IOCTL_USB_COMM_CLR_LISTENER`

OSPLUS_IOCTL_USB_COMM_SEND_COMMAND

Send an encapsulated command to a communications class device

Arguments: The address of an `usb_comm_cmd_t` structure that holds the details of the encapsulated command to send.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl sends an encapsulated command to a communications class device. The command data is supplied in the `usb_comm_cmd_t` structure whose address is passed-in. Typically devices that are sent encapsulated commands are analogue modems.

The `usb_comm_cmd_t` structure has the following format:

```
typedef struct usb_comm_cmd_s
{
      unsigned int length;
      void *buffer;
      unsigned int return_length;
} usb_comm_cmd_t;
```

The fields are explained in *Table 91*.

Table 91. usb_comm_cmd_t fields

Field name	Description
`length`	Length of the encapsulated command
`buffer`	Buffer containing the encapsulated command
`return_length`	Length of command returned

For more details see Section 6.2.1 of *Universal Serial Bus Class Definitions for Communication Devices version 1.1*, available from *www.usb.org*.

See also: `OSPLUS_IOCTL_COMM_GET_RESPONSE`

OSPLUS_IOCTL_USB_COMM_GET_RESPONSE

Get the response to an encapsulated command from a communications class device

Arguments: The address of an `usb_comm_cmd_t` structure to holds the details of the response to the encapsulated command previously sent.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl gets the response to an encapsulated command sent to a communications class device. The response data is written to the `usb_comm_cmd_t` structure whose address is passed-in.

For more details see Section 6.2.2 of *Universal Serial Bus Class Definitions for Communication Devices version 1.1*, available from *www.usb.org*.

See also: `OSPLUS_IOCTL_COMM_GET_RESPONSE`

OSPLUS_IOCTL_USB_COMM_GET_CAPABILITY

Get the communications class device capabilities

Arguments: The address of a structure to be populated with the device capabilities.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl retrieves the communications class device capabilities and places them in the capabilities structure specific to the device type (ACM or ENCM).

For an ACM device the capabilities structure has the following format:

```
typedef struct usb_acm_capabilities_s
{
      unsigned char call;
      unsigned char acm;
} usb_acm_capabilities_t;
```

The fields are explained in *Table 92*.

Table 92. usb_acm_capabilities_t fields

Field name	Description
`call`	Bit field containing call management capabilities (see *Table 26: ACM call management capabilities defines in ioctls/usb/acm.h on page 50* for details)
`acm`	Bit field containing ACM device capabilities (see *Table 27: ACM capabilities defines in ioctls/usb/acm.h on page 50* for details)

For an ENCM device the capabilities structure has the following format:

```
typedef struct usb_encm_capabilities_t
{
      unsigned int statistics;
      unsigned short max_segment;
      unsigned short mc_filters;
      unsigned char power_filters;
} usb_encm_capabilities_t;
```

The fields are explained in *Table 93*.

Table 93. usb_encm_capabilities_t fields

Field name	Description
`statistics`	Bitmap of supported statistics codes (see *Table 34: ENCM capabilities statistic bitmap defines in ioctls/usb/encm.h on page 52* for details)
`max_segment`	Maximum transmissible segment size
`mc_filters`	Number of multicast filters
`power_filters`	Number of power filters

3.4 USB ACM ioctl definitions

These USB ACM device specific ioctl definitions relate to the `device_ioctl()` call. For further information, refer to the *OSPlus User Manual (ADCS 7702033)*.

OSPLUS_IOCTL_USB_ACM_SET_ABSTRACT_STATE

Set the ACM device abstract state

Arguments: The address of a 16-bit unsigned short containing the data multiplexed state and idle setting.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl sets the ACM device abstract state.

For more details see Section 6.2.3 of *Universal Serial Bus Class Definitions for Communication Devices version 1.1*, available from *www.usb.org*.

See also: `OSPLUS_IOCTL_USB_ACM_GET_ABSTRACT_STATE`, `OSPLUS_IOCTL_USB_ACM_CLR_ABSTRACT_STATE`

OSPLUS_IOCTL_USB_ACM_GET_ABSTRACT_STATE

Get the ACM device abstract state

Arguments: The address of a 16-bit unsigned short that is populated with the data multiplexed state and idle setting.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl gets the ACM device abstract state.

For more details see Section 6.2.4 of *Universal Serial Bus Class Definitions for Communication Devices version 1.1*, available from *www.usb.org*.

See also: `OSPLUS_IOCTL_USB_ACM_SET_ABSTRACT_STATE`, `OSPLUS_IOCTL_USB_ACM_CLR_ABSTRACT_STATE`

OSPLUS_IOCTL_USB_ACM_CLR_ABSTRACT_STATE

Clear the ACM device abstract state

Arguments: None.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl clears the ACM device abstract state.

For more details see Section 6.2.5 of *Universal Serial Bus Class Definitions for Communication Devices version 1.1*, available from *www.usb.org*.

See also: `OSPLUS_IOCTL_USB_ACM_SET_ABSTRACT_STATE`, `OSPLUS_IOCTL_USB_ACM_GET_ABSTRACT_STATE`

OSPLUS_IOCTL_USB_ACM_SET_COUNTRY_CODE

Set the ACM device country code

Arguments: The address of a 16-bit unsigned short containing a country code as defined in ISO 3166.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl sets the ACM device country code.

For more details see Section 6.2.3 of *Universal Serial Bus Class Definitions for Communication Devices version 1.1*, available from *www.usb.org*.

See also: `OSPLUS_IOCTL_USB_ACM_GET_COUNTRY_CODE`, `OSPLUS_IOCTL_USB_ACM_CLR_COUNTRY_CODE`

OSPLUS_IOCTL_USB_ACM_GET_COUNTRY_CODE

Get the ACM device country code

Arguments: The address of a 16-bit unsigned short to be populated with a country code as defined in ISO 3166.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl gets the ACM device country code.

For more details see Section 6.2.4 of *Universal Serial Bus Class Definitions for Communication Devices version 1.1*, available from *www.usb.org*.

See also: `OSPLUS_IOCTL_USB_ACM_SET_COUNTRY_CODE`, `OSPLUS_IOCTL_USB_ACM_CLR_COUNTRY_CODE`

OSPLUS_IOCTL_USB_ACM_CLR_COUNTRY_CODE

Clear the ACM device country code

Arguments: None.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl clears the ACM device country code.

For more details see Section 6.2.5 of *Universal Serial Bus Class Definitions for Communication Devices version 1.1*, available from *www.usb.org*.

See also: `OSPLUS_IOCTL_USB_ACM_SET_ABSTRACT_STATE`, `OSPLUS_IOCTL_USB_ACM_GET_ABSTRACT_STATE`

OSPLUS_IOCTL_USB_ACM_SET_LINE_CODING

Set the ACM device line coding configuration

Arguments: The address of a `usb_acm_line_coding_t` structure containing the line coding configuration to be used.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl sets the current line coding configuration for the ACM device to that supplied in the `usb_acm_line_coding_t` structure whose address is passed-in.

The `usb_acm_line_coding_t` structure has the following format:

```
typedef struct usb_acm_line_coding_s
{
      unsigned int baud;
      unsigned char stop;
      unsigned char parity;
      unsigned char data;
} usb_acm_line_coding_t;
```

The fields are explained in *Table 94*.

Table 94. usb_acm_line_coding_t fields

Field name	Description
`baud`	Baud rate (see *Table 28: ACM line coding baud rate defines in ioctls/usb/acm.h on page 51* for details)
`stop`	Number of stop bits (see *Table 29: ACM line coding stop bit defines in ioctls/usb/acm.h on page 51* for details)
`parity`	Type of parity (see *Table 30: ACM line coding parity defines in ioctls/usb/acm.h on page 51* for details)
`data`	Number of data bits (see *TTable 31: ACM line coding data bit defines in ioctls/usb/acm.h on page 51* for details)

For more details see Section 6.2.12 of *Universal Serial Bus Class Definitions for Communication Devices version 1.1*, available from *www.usb.org*.

See also: `OSPLUS_IOCTL_USB_ACM_GET_LINE_CODING`, `OSPLUS_IOCTL_USB_ACM_SET_BAUD`, `OSPLUS_IOCTL_USB_ACM_GET_BAUD`, `OSPLUS_IOCTL_USB_ACM_SET_STOP`, `OSPLUS_IOCTL_USB_ACM_GET_STOP`, `OSPLUS_IOCTL_USB_ACM_SET_PARITY`, `OSPLUS_IOCTL_USB_ACM_GET_PARITY`, `OSPLUS_IOCTL_USB_ACM_SET_DATA`, `OSPLUS_IOCTL_USB_ACM_GET_DATA`

OSPLUS_IOCTL_USB_ACM_GET_LINE_CODING

Get the ACM device line coding configuration

Arguments: The address of a `usb_acm_line_coding_t` structure to hold the line coding configuration in use.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl gets the current line coding configuration for the ACM device and populates the `usb_acm_line_coding_t` structure whose address is passed-in.

For more details see Section 6.2.13 of *Universal Serial Bus Class Definitions for Communication Devices version 1.1*, available from *www.usb.org*.

See also: `OSPLUS_IOCTL_USB_ACM_SET_LINE_CODING`, `OSPLUS_IOCTL_USB_ACM_SET_BAUD`, `OSPLUS_IOCTL_USB_ACM_GET_BAUD`, `OSPLUS_IOCTL_USB_ACM_SET_STOP`, `OSPLUS_IOCTL_USB_ACM_GET_STOP`, `OSPLUS_IOCTL_USB_ACM_SET_PARITY`, `OSPLUS_IOCTL_USB_ACM_GET_PARITY`, `OSPLUS_IOCTL_USB_ACM_SET_DATA`, `OSPLUS_IOCTL_USB_ACM_GET_DATA`

OSPLUS_IOCTL_USB_ACM_SET_BAUD

Set the ACM device line coding baud rate

Arguments: The address of a 32-bit unsigned integer containing the line coding baud rate to use.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl sets the current line coding baud rate for the ACM device to that supplied in the 32-bit unsigned integer whose address is passed-in. For valid baud rates see *Table 28: ACM line coding baud rate defines in ioctls/usb/acm.h on page 51*.

For more details see Section 6.2.12 of *Universal Serial Bus Class Definitions for Communication Devices version 1.1*, available from *www.usb.org*.

See also: `OSPLUS_IOCTL_USB_ACM_SET_LINE_CODING`, `OSPLUS_IOCTL_USB_ACM_GET_LINE_CODING`, `OSPLUS_IOCTL_USB_ACM_GET_BAUD`, `OSPLUS_IOCTL_USB_ACM_SET_STOP`, `OSPLUS_IOCTL_USB_ACM_GET_STOP`, `OSPLUS_IOCTL_USB_ACM_SET_PARITY`, `OSPLUS_IOCTL_USB_ACM_GET_PARITY`, `OSPLUS_IOCTL_USB_ACM_SET_DATA`, `OSPLUS_IOCTL_USB_ACM_GET_DATA`

OSPLUS_IOCTL_USB_ACM_GET_BAUD

Get the ACM device line coding baud rate

Arguments: The address of a 32-bit unsigned integer to be populated with the line coding baud rate in use.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl gets the current line coding baud rate for the ACM device and populates the 32-bit unsigned integer whose address is passed-in.

For more details see Section 6.2.13 of *Universal Serial Bus Class Definitions for Communication Devices version 1.1*, available from *www.usb.org*.

See also: `OSPLUS_IOCTL_USB_ACM_SET_LINE_CODING`, `OSPLUS_IOCTL_USB_ACM_GET_LINE_CODING`, `OSPLUS_IOCTL_USB_ACM_SET_BAUD`, `OSPLUS_IOCTL_USB_ACM_SET_STOP`, `OSPLUS_IOCTL_USB_ACM_GET_STOP`, `OSPLUS_IOCTL_USB_ACM_SET_PARITY`, `OSPLUS_IOCTL_USB_ACM_GET_PARITY`, `OSPLUS_IOCTL_USB_ACM_SET_DATA`, `OSPLUS_IOCTL_USB_ACM_GET_DATA`

OSPLUS_IOCTL_USB_ACM_SET_STOP

Set the ACM device line coding stop bit configuration

Arguments: The address of a 8-bit unsigned integer containing the line coding stop bit configuration to be used.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl sets the current line coding stop bit configuration for the ACM device to that supplied in the 8-bit unsigned integer whose address is passed-in. For valid stop bit configurations see *Table 29: ACM line coding stop bit defines in ioctls/usb/acm.h on page 51*.

For more details see Section 6.2.12 of *Universal Serial Bus Class Definitions for Communication Devices version 1.1*, available from *www.usb.org*.

See also: `OSPLUS_IOCTL_USB_ACM_SET_LINE_CODING`, `OSPLUS_IOCTL_USB_ACM_GET_LINE_CODING`, `OSPLUS_IOCTL_USB_ACM_SET_BAUD`, `OSPLUS_IOCTL_USB_ACM_GET_BAUD`, `OSPLUS_IOCTL_USB_ACM_GET_STOP`, `OSPLUS_IOCTL_USB_ACM_SET_PARITY`, `OSPLUS_IOCTL_USB_ACM_GET_PARITY`, `OSPLUS_IOCTL_USB_ACM_SET_DATA`, `OSPLUS_IOCTL_USB_ACM_GET_DATA`

OSPLUS_IOCTL_USB_ACM_GET_STOP

Get the ACM device line coding stop bit configuration

Description: The address of a 8-bit unsigned integer to be populated with the line coding stop bit configuration in use.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl gets the current line coding stop bit configuration for the ACM device and populates the 8-bit unsigned integer whose address is passed-in.

For more details see Section 6.2.13 of *Universal Serial Bus Class Definitions for Communication Devices version 1.1*, available from *www.usb.org*.

See also: `OSPLUS_IOCTL_USB_ACM_SET_LINE_CODING`, `OSPLUS_IOCTL_USB_ACM_GET_LINE_CODING`, `OSPLUS_IOCTL_USB_ACM_SET_BAUD`, `OSPLUS_IOCTL_USB_ACM_GET_BAUD`, `OSPLUS_IOCTL_USB_ACM_SET_STOP`, `OSPLUS_IOCTL_USB_ACM_SET_PARITY`, `OSPLUS_IOCTL_USB_ACM_GET_PARITY`, `OSPLUS_IOCTL_USB_ACM_SET_DATA`, `OSPLUS_IOCTL_USB_ACM_GET_DATA`

OSPLUS_IOCTL_USB_ACM_SET_PARITY

Set the ACM device line coding parity configuration

Arguments: The address of a 8-bit unsigned integer containing the line coding parity configuration to be used.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl sets the current line coding parity configuration for the ACM device to that supplied in the 8-bit unsigned integer whose address is passed-in. For valid parity configurations see *Table 30: ACM line coding parity defines in ioctls/usb/acm.h on page 51*.

For more details see Section 6.2.12 of *Universal Serial Bus Class Definitions for Communication Devices version 1.1*, available from *www.usb.org*.

See also: `OSPLUS_IOCTL_USB_ACM_SET_LINE_CODING`, `OSPLUS_IOCTL_USB_ACM_GET_LINE_CODING`, `OSPLUS_IOCTL_USB_ACM_SET_BAUD`, `OSPLUS_IOCTL_USB_ACM_GET_BAUD`, `OSPLUS_IOCTL_USB_ACM_SET_STOP`, `OSPLUS_IOCTL_USB_ACM_GET_STOP`, `OSPLUS_IOCTL_USB_ACM_GET_PARITY`, `OSPLUS_IOCTL_USB_ACM_SET_DATA`, `OSPLUS_IOCTL_USB_ACM_GET_DATA`

OSPLUS_IOCTL_USB_ACM_GET_PARITY

Get the ACM device line coding parity configuration

Arguments: The address of a 8-bit unsigned integer to be populated with the line coding parity configuration in use.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl gets the current line coding parity configuration for the ACM device and populates the 8-bit unsigned integer whose address is passed-in.

For more details see Section 6.2.13 of *Universal Serial Bus Class Definitions for Communication Devices version 1.1*, available from *www.usb.org*.

See also: `OSPLUS_IOCTL_USB_ACM_SET_LINE_CODING`, `OSPLUS_IOCTL_USB_ACM_GET_LINE_CODING`, `OSPLUS_IOCTL_USB_ACM_SET_BAUD`, `OSPLUS_IOCTL_USB_ACM_GET_BAUD`, `OSPLUS_IOCTL_USB_ACM_SET_STOP`, `OSPLUS_IOCTL_USB_ACM_GET_STOP`, `OSPLUS_IOCTL_USB_ACM_SET_PARITY`, `OSPLUS_IOCTL_USB_ACM_SET_DATA`, `OSPLUS_IOCTL_USB_ACM_GET_DATA`

OSPLUS_IOCTL_USB_ACM_SET_DATA

Set the ACM device line coding data bit configuration

Arguments: The address of a 8-bit unsigned integer containing the line coding data bit configuration to be used.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl sets the current line coding data bit configuration for the ACM device to that supplied in the 8-bit unsigned integer whose address is passed-in. For valid data bit configurations see *Table 31: ACM line coding data bit defines in ioctls/usb/acm.h on page 51.*

For more details see Section 6.2.12 of *Universal Serial Bus Class Definitions for Communication Devices version 1.1*, available from *www.usb.org*.

See also: `OSPLUS_IOCTL_USB_ACM_SET_LINE_CODING`, `OSPLUS_IOCTL_USB_ACM_GET_LINE_CODING`, `OSPLUS_IOCTL_USB_ACM_SET_BAUD`, `OSPLUS_IOCTL_USB_ACM_GET_BAUD`, `OSPLUS_IOCTL_USB_ACM_SET_STOP`, `OSPLUS_IOCTL_USB_ACM_GET_STOP`, `OSPLUS_IOCTL_USB_ACM_SET_PARITY`, `OSPLUS_IOCTL_USB_ACM_GET_PARITY`, `OSPLUS_IOCTL_USB_ACM_GET_DATA`

OSPLUS_IOCTL_USB_ACM_GET_DATA

Get the ACM device line coding data bit configuration

Arguments: The address of a 8-bit unsigned integer to be populated with the line coding data bit configuration to be used.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl gets the current line coding data bit configuration for the ACM device and populates the 8-bit unsigned integer whose address is passed-in.

For more details see Section 6.2.13 of *Universal Serial Bus Class Definitions for Communication Devices version 1.1*, available from *www.usb.org*.

See also: `OSPLUS_IOCTL_USB_ACM_SET_LINE_CODING`, `OSPLUS_IOCTL_USB_ACM_GET_LINE_CODING`, `OSPLUS_IOCTL_USB_ACM_SET_BAUD`, `OSPLUS_IOCTL_USB_ACM_GET_BAUD`, `OSPLUS_IOCTL_USB_ACM_SET_STOP`, `OSPLUS_IOCTL_USB_ACM_GET_STOP`, `OSPLUS_IOCTL_USB_ACM_SET_PARITY`, `OSPLUS_IOCTL_USB_ACM_GET_PARITY`, `OSPLUS_IOCTL_USB_ACM_SET_DATA`

OSPLUS_IOCTL_USB_ACM_SET_CTRL_LINE_STATE

Set the ACM device control line state

Arguments: A 16-bit unsigned integer containing the line control settings.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl sets the current control line state for the ACM device to that supplied in the 16-bit unsigned integer passed-in.

For more details see Section 6.2.14 of *Universal Serial Bus Class Definitions for Communication Devices version 1.1*, available from *www.usb.org*.

See also: `OSPLUS_IOCTL_USB_ACM_GET_LINE_STATUS`

OSPLUS_IOCTL_USB_ACM_SEND_BREAK

Cause the ACM device to send an RS-232 style break

Arguments: The address of a 16-bit unsigned integer specifying a time in milliseconds.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl causes the ACM device to send an RS-232 style break for the length of time in milliseconds specified by the 16-bit unsigned integer passed-in.

For more details see Section 6.2.15 of *Universal Serial Bus Class Definitions for Communication Devices version 1.1*, available from *www.usb.org*.

See also: `OSPLUS_IOCTL_USB_ACM_SEND_BREAK_START`, `OSPLUS_IOCTL_USB_ACM_SEND_BREAK_STOP`

OSPLUS_IOCTL_USB_ACM_SEND_BREAK_START

Cause the ACM device to start sending an RS-232 style break

Arguments: None.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl causes the ACM device to start sending an RS-232 style break.

For more details see Section 6.2.15 of *Universal Serial Bus Class Definitions for Communication Devices version 1.1*, available from *www.usb.org*.

See also: `OSPLUS_IOCTL_USB_ACM_SEND_BREAK`, `OSPLUS_IOCTL_USB_ACM_SEND_BREAK_STOP`

OSPLUS_IOCTL_USB_ACM_SEND_BREAK_STOP

Cause the ACM device to stop sending an RS-232 style break

Arguments: None.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl causes the ACM device to stop sending an RS-232 style break.

For more details see Section 6.2.15 of *Universal Serial Bus Class Definitions for Communication Devices version 1.1*, available from *www.usb.org*.

See also: `OSPLUS_IOCTL_USB_ACM_SEND_BREAK`, `OSPLUS_IOCTL_USB_ACM_SEND_BREAK_START`

OSPLUS_IOCTL_USB_ACM_GET_LINE_STATUS

Get the ACM device line status

Arguments: The address of a 16-bit unsigned integer to be populated with the line status.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl gets the ACM device line status and copies it to the 16-bit unsigned integer whose address is passed-in. For control line states see *Table 22: Communications line state event defines in ioctls/usb/comm.h on page 48*.

See also: `OPSLUS_IOCTL_USB_ACM_SET_CTRL_LINE_STATE`

3.5 USB ENCM ioctl definitions

These USB ENCM device specific ioctl definitions relate to the `device_ioctl()` call. Refer to the *OSPlus User Manual (ADCS 7702033)* for more details.

USB communications class cable modem device ioctls

USB cable modem devices support all of the OSPlus common, and all of the ENCM ioctls supported by the USB communication ENCM class driver.

OSPLUS_IOCTL_USB_ENCM_SET_MCAST

Set the ENCM device multicast filter

Arguments: The address of a `usb_encm_mcast_t` that contains the filter details.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl sets the ENCM device multicast filters to the contents supplied in the `usb_encm_mcast_t` structure whose address is passed-in.

An `usb_encm_mcast_t` structure has the following format:

```
typedef struct usb_encm_mcast_s
{
      unsigned short number;
      unsigned char *addresses;
} usb_encm_mcast_t;
```

The fields are explained in *Table 95*.

Table 95. usb_encm_mcast_t fields

Field name	Description
`number`	Number of sequential 48-bit ethernet multicast addresses in network byte order
`addresses`	Sequential 48-bit ethernet addresses

For more details see Section 6.2.27 of *Universal Serial Bus Class Definitions for Communication Devices version 1.1*, available from *www.usb.org*.

See also: `OSPLUS_IOCTL_USB_ENCM_GET_MCAST`

OSPLUS_IOCTL_USB_ENCM_GET_MCAST

Get the ENCM device multicast filter

Arguments: The address of a `usb_encm_mcast_t` to hold the filter details.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl gets the ENCM device multicast filters and populates the `usb_encm_mcast_t` structure whose address is passed-in with them.

See also: `OSPLUS_IOCTL_USB_ENCM_SET_MCAST`

OSPLUS_IOCTL_USB_ENCM_SET_PM_FILT

Set the ENCM device power management pattern filter

Arguments: The address of a `usb_encm_pm_filter_t` that contains the filter details.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl sets the ENCM device power management pattern filters to the contents supplied in the `usb_encm_pm_filter_t` structure whose address is passed-in.

An `usb_encm_pm_filter_t` structure has the following format:

```
typedef struct usb_encm_pm_filter_s
{
      unsigned short mask_size;
      unsigned char *mask;
      unsigned int pattern_size;
      unsigned char *pattern;
      unsigned short filter_number;
      unsigned short status;
} usb_encm_pm_filter_t;
```

The fields are explained in *Table 96*.

Table 96. usb_encm_pm_filter_t fields

Field name	Description
`mask_size`	Mask size
`mask`	Mask
`pattern_size`	Pattern size
`pattern`	Pattern
`filter_number`	Filter number to set/get
`status`	Status (only used when getting a PM pattern filter)

For more details see Section 6.2.28 of *Universal Serial Bus Class Definitions for Communication Devices version 1.1*, available from *www.usb.org*.

See also: `OSPLUS_IOCTL_USB_ENCM_GET_PM_FILT`

OSPLUS_IOCTL_USB_ENCM_GET_PM_FILT

Get the ENCM device power management pattern filter

Arguments: The address of a `usb_encm_pm_filter_t` to hold the filter details.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl gets the ENCM device power management pattern filters and populates the `usb_encm_pm_filter_t` structure whose address is passed-in with them.

For more details see Section 6.2.29 of *Universal Serial Bus Class Definitions for Communication Devices version 1.1*, available from *www.usb.org*.

See also: `OSPLUS_IOCTL_USB_ENCM_SET_PM_FILT`

OSPLUS_IOCTL_USB_ENCM_SET_PKT_FILTER

Set the ENCM device packet filter

Arguments: A 16-bit unsigned integer containing the bit mask of packet filter settings.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl sets the ENCM device packet filters to the bit mask specified in the 16-bit unsigned integer passed-in. For valid packet filter settings, see *Table 35: ENCM packet filter type defines in ioctls/usb/encm.h on page 53*.

For more details see Section 6.2.30 of *Universal Serial Bus Class Definitions for Communication Devices version 1.1*, available from *www.usb.org*.

See also: `OSPLUS_IOCTL_USB_ENCM_GET_PKT_FILTER`

OSPLUS_IOCTL_USB_ENCM_GET_PKT_FILTER

Get the ENCM device packet filter

Arguments: The address of a 16-bit unsigned integer to hold the packet-filter bit-mask settings.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl gets the ENCM device packet-filter bit-mask settings and populates the 16-bit unsigned integer whose address is passed-in with them.

For more details see Section 6.2.30 of *Universal Serial Bus Class Definitions for Communication Devices version 1.1*, available from *www.usb.org*.

See also: `OSPLUS_IOCTL_USB_ENCM_SET_PKT_FILTER`

OSPLUS_IOCTL_USB_ENCM_GET_STATS

Get the ENCM device statistics

Arguments: The address of a `usb_encm_statistic_t` to hold the statistic details.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl gets the ENCM device statistics and populates the `usb_encm_statistic_t` structure whose address is passed-in with them.

An `usb_encm_statistic_t` structure has the following format:

```
typedef struct usb_encm_statistic_s
{
      unsigned char statistic;
      unsigned int value;
} usb_encm_statistic_t;
```

The fields are explained in *Table 97*.

Table 97. usb_encm_statistic_t fields

Field name	Description
`statistic`	Statistic code of statistic to get (see *Table 36: ENCM statistics code defines in ioctls/usb/encm.h on page 53* for details)
`value`	Value of specified statistic

For more details see Section 6.2.31 of *Universal Serial Bus Class Definitions for Communication Devices version 1.1*, available from *www.usb.org*.

OSPLUS_IOCTL_USB_ENCM_SET_MAC

Set the ENCM device ethernet MAC address

Arguments: The address of a `usb_encm_mac_t` to containing the MAC address.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl sets the ENCM device MAC address to that specified in the `usb_encm_mac_t` structure whose address is passed-in.

An `usb_encm_mac_t` structure has the following format:

```
typedef struct usb_encm_mac_s
{
      unsigned char address[OSPLUS_ENCM_MAC_ADDRESS_LENGTH];
} usb_encm_mac_t;
```

The fields are explained in *Table 98*.

Table 98. usb_encm_mac_t fields

Field name	Description
`address`	MAC address, as a 12 digit ASCII string.

Note: For ENCM devices, the MAC address can be set either using this ioctl or the ioctl `OSPLUS_IOCTL_ETH_SET_MAC`*. See the OSPlus User Manual (ADCS 7702033)* for details of `OSPLUS_IOCTL_ETH_SET_MAC`*.*

See also: `OSPLUS_IOCTL_USB_ENCM_GET_MAC`

OSPLUS_IOCTL_USB_ENCM_GET_MAC

Get the ENCM device ethernet MAC address

Arguments: The address of a `usb_encm_mac_t` to hold the MAC address.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl gets the ENCM device MAC address and populates the `usb_encm_mac_t` structure whose address is passed-in with it. The MAC address is a 12 digit ASCII string.

Note: For ENCM devices, the MAC address can be obtained either using this ioctl or the ioctl `OSPLUS_IOCTL_ETH_GET_MAC`*. See the OSPlus User Manual (ADCS 7702033)* for details of `OSPLUS_IOCTL_ETH_GET_MAC`.

See also: `OSPLUS_IOCTL_USB_ENCM_SET_MAC`

3.6 USB host-to-host ioctl definitions

These USB host-to-host device specific ioctl definitions relate to the `device_ioctl()` call. Refer to the *OSPlus User Manual (ADCS 7702033)* for more details.

OSPLUS_IOCTL_USB_H2H_RESET_IN

Reset the host-to-host input pipe

Arguments: None.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl resets the host-to-host device input pipe. On success this function returns `0`, otherwise `-1` is returned.

See also: `OSPLUS_IOCTL_USB_H2H_RESET_OUT`

OSPLUS_IOCTL_USB_H2H_RESET_OUT

Reset the host-to-host output pipe

Arguments: None.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl resets the host-to-host device output pipe. On success this function returns `0`, otherwise `-1` is returned.

See also: `OSPLUS_IOCTL_USB_H2H_RESET_IN`

OSPLUS_IOCTL_USB_H2H_GET_STATUS

Return the host-to-host device status

Arguments: The address of an unsigned integer to store the status.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl returns the host-to-host status. To determine the status, the unsigned integer can be ANDed with the defines in *Table 39: Defines in ioctls/usb/h2h.h on page 55*. On success this function returns `0`, otherwise `-1` is returned.

OSPLUS_IOCTL_USB_H2H_WAIT_PEER_E

Wait for a specified timeout for a peer to connect

Arguments: A pointer to an `osclock_t` in memory containing the timeout value.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl waits for the specified timeout for a peer to connect to the host-to-host cable. If a peer is already connected it returns immediately. Otherwise a blocking wait is performed with the specified timeout given by the `osclock_t` passed-in. On success this function returns `0`. Otherwise `-1` if no peer is connected before the timeout expires.

See also: `OSPLUS_IOCTL_USB_H2H_WAIT_TX_RDY`

OSPLUS_IOCTL_USB_H2H_WAIT_TX_RDY

Wait for a specified timeout for a peer to become ready to transmit

Arguments: A pointer to an `osclock_t` in memory containing the timeout value.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl waits for the specified timeout for a connected peer to become ready to transmit. If a peer is not connected it immediately returns an error. Otherwise, a blocking wait is performed with the specified timeout given by the `osclock_t` passed-in. On success, `0` is returned. Otherwise `-1` if the peer is disconnected or the timeout expires.

See also: `OSPLUS_IOCTL_USB_H2H_WAIT_PEER_E`

3.7 USB host controller ioctl definitions

These USB host controller device specific ioctl definitions relate to the `device_ioctl()` call. Refer to the *OSPlus User Manual (ADCS 7702033)* for more details.

OSPLUS_IOCTL_USB_HOST_GET_PORTS

Return the number of ports on the controller's root hub

Arguments: The address of an integer to receive the number of root hub ports.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl returns the number of ports on the root hub for the host controller.

See also: `OSPLUS_IOCTL_USB_HOST_TEST_MODE`

OSPLUS_IOCTL_USB_HOST_TEST_MODE

Places a port on a host controller's root hub in to a test mode

Arguments: The address of a `usb_host_test_mode_params_t` structure.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl puts a port on the host controller's root hub in to the test mode that is required. The format of a `usb_host_test_mode_parms_t` structure is given in *Table 99*.

Table 99. usb_host_test_mode_params_t fields

Field name	Description
`port`	Port number to put in to test mode
`test`	Test mode to select

Ports are numbered from `0..`n`-1`, where n is the number of ports as returned from the `OSPLUS_IOCTL_USB_HOST_GET_PORTS` ioctl. The possible USB test modes are enumerated by the type `usb_host_test_mode_t`, described in *OSPLUS_IOCTL_USB_TEST_PORT on page 86*.

See also: `OSPLUS_IOCTL_USB_HOST_GET_PORTS`

OSPLUS_IOCTL_USB_HOST_RESUME_PORT

Takes the given host controller port out of suspend mode

Arguments: The port number to modify.

Returns: 0 for success, -1 for failure.

Description: This ioctl takes the given port out of suspend mode, and back into its normal operational mode. This wakes up any device attached to this port, taking it from its low power suspended state into its normal operational mode. The port number given must be between 0 and one less than the total number of ports (as returned by the `OSPLUS_IOCTL_USB_HOST_GET_PORTS` ioctl).

See also: `OSPLUS_IOCTL_USB_HOST_GET_PORTS`, `OSPLUS_IOCTL_USB_HOST_SUSPEND_PORT`

OSPLUS_IOCTL_USB_HOST_SUSPEND_PORT

Places the given host controller port into suspend mode

Arguments: The port number to modify.

Returns: 0 for success, -1 for failure.

Description: This ioctl places the given port into suspend mode. This places any device attached to this port into its low power suspended mode. The port number given must be between 0 and one less than the total number of ports (as returned by the `OSPLUS_IOCTL_USB_HOST_GET_PORTS` ioctl).

See also: `OSPLUS_IOCTL_USB_HOST_GET_PORTS`, `OSPLUS_IOCTL_USB_HOST_RESUME_PORT`

3.8 USB hub ioctl definitions

These USB hub device specific ioctl definitions relate to the `device_ioctl()` call. Refer to the *OSPlus User Manual (ADCS 7702033)* for more details.

OSPLUS_IOCTL_USB_HUB_SET_FEAT

Set a USB hub or port feature

Arguments: The address of a structure containing the hub port to address and the feature to set.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl sets a hub or port feature. The address of a `usb_hub_feature_t` structure must be passed-in. This structure contains the hub port to address (or hub if port is zero) and the feature to set. On success `0` is returned, otherwise `-1` is returned. The `usb_hub_feature_t` structure is defined as:

```
typedef struct usb_hub_feature_s
{
      unsigned int port;
      unsigned int feature;
} usb_hub_feature_t;
```

The fields of a `usb_hub_feature_t` structure are described in *Table 100*

Table 100. usb_hub_feature_t fields

Field name	Description
`port`	Port to address (if zero request relates to hub only)
`feature`	Feature to set/clear

The USB hub/port features to set/clear are defined in *Table 43: USB hub feature defines in ioctls/usb/hub.h on page 56*, *Table 44: USB hub port feature defines in ioctls/usb/hub.h on page 57* and *Table 45: USB hub port test selector defines in ioctls/usb/hub.h on page 57*.

See also: `OSPLUS_IOCTL_USB_HUB_CLR_FEAT`

OSPLUS_IOCTL_USB_HUB_CLR_FEAT

Clear a USB hub or port feature

Arguments: The address of a structure containing the hub port to address and the feature to clear.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl clears a hub or port feature. The address of a `usb_hub_feature_t` structure must be passed-in. This structure contains the hub port to address (or hub if port is zero) and the feature to clear. On success, `0` is returned, otherwise, `-1` is returned.

The USB hub features to clear are defined in *Table 43: USB hub feature defines in ioctls/usb/hub.h on page 56*, *Table 44: USB hub port feature defines in ioctls/usb/hub.h on page 57* and *Table 45: USB hub port test selector defines in ioctls/usb/hub.h on page 57*.

See also: `OSPLUS_IOCTL_USB_HUB_SET_FEAT`

OSPLUS_IOCTL_USB_HUB_GET_STATUS

Set a USB hub or port feature

Arguments: The address of a structure containing the hub port to address and an unsigned integer to store the status.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl gets the status of a hub or port. The address of a `usb_hub_status_t` structure must be passed-in. This structure contains the hub port to address (or hub if the port is zero) and an unsigned integer to store the status. On success, `0` is returned, otherwise, `-1` is returned.

The `usb_hub_status_t` structure is defined as:

```
typedef struct usb_hub_status_s
{
      unsigned int port;
      unsigned int status;
} usb_hub_status_t;
```

The fields of a `usb_hub_status_t` structure are described in *Table 101*.

Table 101. usb_hub_status_t fields

Field name	Description
`port`	Port to address (if zero request relates to hub only)
`status`	Status of port/hub

The status can be determined by ANDing the status returned with the defines as shown in *Table 46: USB hub port status defines in ioctls/usb/hub.h on page 58*.

3.9 USB VOIP ioctl definitions

These USB VOIP class device specific ioctl definitions relate to the `device_ioctl()` call. For further information, refer to the *OSPlus User Manual (ADCS 7702033)*.

OSPLUS_IOCTL_USB_VOIP_GET_INFO

Return basic information about the phone and its configuration

Arguments: A pointer to an `usb_voip_info_t` structure in memory to receive the phone information.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl returns basic information about the phone, such as:

– the model of the phone

– the type of display

– the features it supports

This ioctl also returns the USB audio class device names given to the phone microphone and speakers. Pass the address of an `usb_voip_info_t` structure to be filled in by this call.

The `usb_voip_info_t` structure has the following format:

```
typedef struct usb_voip_info_s
{
      usb_voip_model_t model_type;
      usb_voip_lcd_t   lcd_type;
      char             ring_tone;
      char             dial_tone;
      unsigned int     version;
      char             capture_device[32];
      char             playback_device[32];
} usb_voip_info_t;
```

The fields are explained in *Table 102*.

Table 102. usb_voip_info_t values

Field name	Description
`model_type`	Enumeration describing the model type. For phone model type see *Table 96*
`lcd_type`	Enumeration describing the LCD type. For LCD type see *Table 97*
`ring_tone`	Value indicating if ring tone is supported (`1`) or not (`0`)
`dial_tone`	Value indicating if dial tone is supported (`1`) or not (`0`)
`version`	Version of phone

Table 102. usb_voip_info_t values (continued)

Field name	Description
`capture_device`	Name of audio device used for audio capture on phone (usually microphone)
`playback_device`	Name of audio device used for audio playback on phone (usually speakers)

OSPLUS_IOCTL_USB_VOIP_GET_KEYPRESS

Read a keypress from the phone keypad

Arguments: A pointer to an `usb_voip_keypress_t` structure in memory to receive the keypress data.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl returns a phone keypress. Pass the address of an `usb_voip_keypress_t` structure to be filled in by this call.

The `usb_voip_keypress_t` structure has the following format:

```
typedef struct usb_voip_keypress_s
{
      usb_voip_key_t key;
      unsigned char  raw;
} usb_voip_keypress_t;
```

The fields are explained in *Table 103* and the values for raw keypresses are given in *Table 104*.

Table 103. usb_voip_keypress_t fields

Field name	Description
`key`	Enumeration describing the decoded keypress. See *Table 104* for details.
`raw`	Raw keypress

Table 104. usb_voip_key_t values

Field name	Description
`usb_voip_key_none`	No keypress
`usb_voip_key_unknown`	Unknown keypress
`usb_voip_key_pickup`	Pickup key
`usb_voip_key_hangup`	Hangup key
`usb_voip_key_cancel`	Cancel key
`usb_voip_key_up`	Up arrow key
`usb_voip_key_down`	Down arrow key
`usb_voip_key_left`	Left arrow key
`usb_voip_key_right`	Right arrow key

Table 104. usb_voip_key_t values (continued)

Field name	Description
`usb_voip_key_in`	In key
`usb_voip_key_out`	Out key
`usb_voip_key_1`	1 key
`usb_voip_key_2`	2 key
`usb_voip_key_3`	3 key
`usb_voip_key_4`	4 key
`usb_voip_key_5`	5 key
`usb_voip_key_6`	6 key
`usb_voip_key_7`	7 key
`usb_voip_key_8`	8 key
`usb_voip_key_9`	9 key
`usb_voip_key_star`	* key
`usb_voip_key_0`	0 key
`usb_voip_key_pound`	# key
`usb_voip_key_contacts`	Contacts key

OSPLUS_IOCTL_USB_VOIP_LCD_CLEAR

Clear the phone LCD display

Arguments: None.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl clears the phone LCD display.

See also: `OSPLUS_IOCTL_USB_VOIP_GET_INFO`, `OSPLUS_IOCTL_USB_VOIP_LCD_SEG7_WRITE`, `OSPLUS_IOCTL_USB_VOIP_LCD_PIXEL_BLIT`

OSPLUS_IOCTL_USB_VOIP_LCD_PIXEL_BLIT

Blit part of an off-screen buffer to the phone LCD display

Arguments: A pointer to an `usb_voip_blit_t` structure in memory containing the buffer to blit.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl blits part of an off-screen graphics buffer to the phone display. Pass the address of an `usb_voip_blit_t` structure that has been initialized with a pointer to the bit map of the image to appear on the phone display. The structure also specifies the co-ordinates of the rectangular area to copy.

The `usb_voip_blit_t` structure has the following format:

```
typedef struct usb_voip_blit_s
{
      unsigned short    x1;
      unsigned short    y1;
      unsigned short    x2;
      unsigned short    y2;
      unsigned char    *data;
} usb_voip_blit_t;
```

The fields are explained in *Table 105*.

Table 105. usb_voip_blit_t fields

Field name	Description
`x1`	X co-ordinate of area to blit
`y1`	Y co-ordinate of area to blit
`x2`	X co-ordinate of area to blit
`y2`	Y co-ordinate of area to blit
`data`	Pointer to off-screen buffer

See also: `OSPLUS_IOCTL_USB_VOIP_GET_INFO`, `OSPLUS_IOCTL_USB_VOIP_LCD_CLEAR`

OSPLUS_IOCTL_USB_VOIP_LCD_SEG7_WRITE

Write to the seven segment LCD phone display

Arguments: A pointer to an `usb_voip_seg7_t` structure in memory containing the text to display.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl displays text or symbols on a seven segment phone display. Pass the address of an `usb_voip_seg7_t` structure that has been initialized with the text to be displayed.

The structure defines the starting offset for the text (offsets for the start of each line on the display are specified in *Table 49: VOIP seven-segment display defines in ioctls/usb/voip.h on page 59*) and also the text to display. At those offsets where the seven segment display can only display a symbol, such as on the second line of the display, the symbol is displayed if the string contains a `'.'` character at the appropriate position. Any other character does not display the symbol. The `usb_voip_seg7_t` structure has the following format:

```
typedef struct usb_voip_seg7_s
{
      unsigned int offset;
      char         *text;
} usb_voip_seg7_t;
```

The fields are explained in *Table 106*.

Table 106. usb_voip_seg7_t fields

Field name	Description
`offset`	Offset on the LCD display to begin writing text
`text`	Pointer to a string containing the text to write

See also: `OSPLUS_IOCTL_USB_VOIP_GET_INFO`, `OSPLUS_IOCTL_USB_VOIP_LCD_CLEAR`

OSPLUS_IOCTL_USB_VOIP_SET_DIAL_TONE

Turn the phone dial tone on or off

Arguments: The value `1` or `0`, to turn the dial tone on or off.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl turns the phone dial tone on or off.

Note: Not all phones support the dial tone. The information returned by `OSPLUS_IOCTL_USB_VOIP_GET_INFO` *indicates whether the phone supports a dial tone or not.*

See also: `OSPLUS_IOCTL_USB_VOIP_GET_INFO`

OSPLUS_IOCTL_USB_VOIP_SET_LED

Turn the phone LED/backlight on or off

Arguments: The value `1` or `0`, to turn the LED/backlight on or off.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl turns the phone LED (or, if the phone have a graphical display, then display backlight) on or off according to the argument passed.

OSPLUS_IOCTL_USB_VOIP_SET_RING_STYLE

Set a new phone ringer style

Arguments: A pointer to an `usb_voip_ringtone_t` structure in memory containing the ring tone to set.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl allows an application to set a custom phone ringer style.

Note: Not all phones support the ringer, so the information returned by `OSPLUS_IOCTL_USB_VOIP_GET_INFO` *indicates whether the phone supports a ringer or not.*

A ring tone consists of a sequence of frequencies (up to a maximum of 32), each of which sounds for a specified amount of time. Pass the address of a `usb_voip_ringtone_t` structure that has been initialized with a sequence of frequency/duration pairs.

The `usb_voip_ringtone_t` structure has the following format:

```
typedef struct usb_voip_ringtone_s
{
          unsigned int     count;
          struct {
                    unsigned short frequency;
                    unsigned short duration;
          } notes[32];
} usb_voip_ringtone_t;
```

The fields are explained in *Table 107*.

Table 107. usb_voip_ringtone_t fields

Field name	Description
`count`	Number of notes defined for ring tone
`notes`	This is an array of structures that contains a 16-bit frequency and a 16-bit duration describing the note to play. The frequency is supplied as a negative value, so for frequency 1250Hz, specify a value of -1250. The duration to play the frequency is defined in hundredths of a second. Both the frequency and duration should be specified in big endian format.

See also: `OSPLUS_IOCTL_USB_VOIP_GET_INFO`,
`OSPLUS_IOCTL_USB_VOIP_SET_RING_TONE`,
`OSPLUS_IOCTL_USB_VOIP_SET_RING_VOLUME`

OSPLUS_IOCTL_USB_VOIP_SET_RING_TONE

Turn the phone ringer on or off

Arguments: The value `1` or `0`, to turn the phone ringer on or off.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl turns the phone ringer on or off.

Note: Not all phones support the ringer. The information returned by `OSPLUS_IOCTL_USB_VOIP_GET_INFO` *indicates if the phone supports a ringer.*

See also: `OSPLUS_IOCTL_USB_VOIP_GET_INFO`, `OSPLUS_IOCTL_USB_VOIP_SET_RING_STYLE`, `OSPLUS_IOCTL_USB_VOIP_SET_RING_VOLUME`

OSPLUS_IOCTL_USB_VOIP_SET_RING_VOLUME

Set the phone ringer volume

Arguments: An unsigned integer with a value between `0` and `255` to set as the ringer volume.

Returns: Returns `0` for success, `-1` for failure.

Description: This ioctl sets the phone ringer volume to the value pointed to by the argument.

See also: `OSPLUS_IOCTL_USB_VOIP_GET_INFO`, `OSPLUS_IOCTL_USB_VOIP_SET_RING_STYLE`, `OSPLUS_IOCTL_USB_VOIP_SET_RING_TONE`

3.10 USB driver function definitions

USBLINK_Initialize

Initialize the USB driver

Definition:

```
int USBLINK_Initialize (
  void *uncached,
  size_t size)
```

Arguments:

`uncached`	The pointer to the block of contiguous uncached memory.
`size`	The size of the block.

Returns: Returns `0` for success, or `-1` on failure.

Description: This call initializes the USB driver. Upon successful completion, all USB ports on the system will be active, enabling USB device insertion and extraction events to be detected, and USB I/O to take place.

4 Revision history

Table 108. Document revision history

Date	Revision	Changes
14-Feb-2011	G	Updates to *OSPLUS_IOCTL_USB_BULK_TRANSFER on page 80*.
30-Sep-2010	F	Throughout: amended type `str_t` to `osplus_str_t`. Updates to *Section 2.4: USB driver configuration on page 10*. Corrected an error in *Figure 1 on page 11*. Added *Limiting SCSI transfer size on page 20*. Updated *Table 65: Mass storage device supported OSPlus common ioctls on page 66*. Added ioctl *OSPLUS_IOCTL_USB_ISOC_INFO on page 77*. Updated ioctl *OSPLUS_IOCTL_USB_ISOC_TRANSFER on page 78*.
26-Apr-2010	E	Added *Section 2.4.1: Power management on page 12*.
4-Feb-2010	D	Revisions to *Section 2.7.1: USB audio class devices on page 14*. Added *USB communications class RNDIS wrapper* to *Section 2.7.2: USB communications class devices on page 16*. Revisions to *Section 2.7.4: USB HID devices on page 18*. Additions made to *Section 2.9: USB driver device usage model on page 22*. Added *USB communications class RNDIS devices* to *Section 2.9.2: USB communications class devices on page 28*. Added ioctl *OSPLUS_IOCTL_USB_TREE_NODE_INFO on page 89*.
28-Oct-2008	C	Added information relating to composite devices to *Section 2.9: USB driver device usage model on page 22*. Added information on transferring audio to and from devices in *Section 2.9.1: USB audio class devices on page 22*. Added new ioctl `OSPLUS_IOCTL_USB_COMPOSITE_INFO` to *Chapter 3: USB ioctl and driver definitions*. Revised definition of ioctl `OSPLUS_IOCTL_USB_AUDIO_TRANSFER_STATUS` in *Chapter 3: USB ioctl and driver definitions*.
27-Oct-2007	B	Updated to new corporate template. Added USB audio and VOIP documentation. Updated for new test ioctls. Update to ethernet documentation.
22-Sep-2006	A	Initial release. Information separated from OSPlus User Manual (ADCS 7702033)

List of IOCTLS and Functions

Index