Home

USB Host SDK for OS-9®

Version 2.0

RadiSys
THE POWER OF WE
www.radisys.com
Revision A • July 2006

Copyright and publication information

Disclaimer

Reproduction notice

Table of Contents

Chapter 1: Getting Started with USB Host for OS-9® 7

8 System Overview
9 System Requirements
9 Windows Development Platform Hardware Requirements
9 Windows Development Platform Software Requirements
10 OS-9 Target System/USB Host Hardware Requirements
11 Installing the Software
11 Installing to the Windows Development Platform
13 Installing to the OS-9 Target System/USB Host
13 USB Host Module List
16 Loading and Starting the USB Host Software
21 Example Commands
21 Checking for USB Devices
21 Getting Device Information
22 Checking for Data Transmission
24 Mouse Through MAUI®

Chapter 2: Using USB Host for OS-9 27

28 Overview
30 Hardware Controller Driver
30 Bus Methods Structure
30 Bus Methods Structure Fields
31 Pipe Methods Structure
31 Bus Methods Structure Fields
33 USB Management Driver
33 Bus Explore
35 Plug and Play

35 Match
36 Attach
36 Detach
37 Registering with usbman
39 Logical Device Drivers
39 LDD Initialization
39 LDD De-Initialization
40 Suggested OS-9 Interface
40 Setstats
40 Getstats
40 Plug-n-play
41 Standard OS-9 LDD Drivers
41 USB Mouse
42 Data Format
42 Use With MAUI
43 Testing the USB Mouse
44 USB Keyboard
44 Data Format
45 Use With MAUI
46 Testing the USB Keyboard
47 USB Printer
48 Testing the USB Printer
49 USB Mass Storage
50 Testing USB Mass Storage Devices
51 Generic USB Driver
51 Plug-n-Play
51 Accessing Endpoints with spugen
52 Testing spugen
55 Reference API
68 User-State Daemon Process

Chapter 3: USB Host API Reference 69

70 Pipe Functions List

71 Transfer Functions List
72 Interface Functions List
73 Device Functions List
74 Alphabetical Listing

Chapter 4: USB Host for OS-9 Utilities 131

Appendix A: Porting to the USB Host Stack 137

138 Writing the Logical Device Driver (LDD)
138 Creating a Directory Structure
140 Implementing your LDD
142 Additional File Information
143 Writing a Hardware Control Driver
143 Overview
143 Transfer Types
145 Bus Methods Structure
145 Calling usbman
146 Existing Drivers
148 Implementing the Driver
149 Testing the Driver

Chapter 1: Getting Started with USB Host for OS-9®

This chapter describes how to install and configure the USB Host SDK for OS-9® software on your Windows development platform and on your OS-9 target system. It includes the following sections:

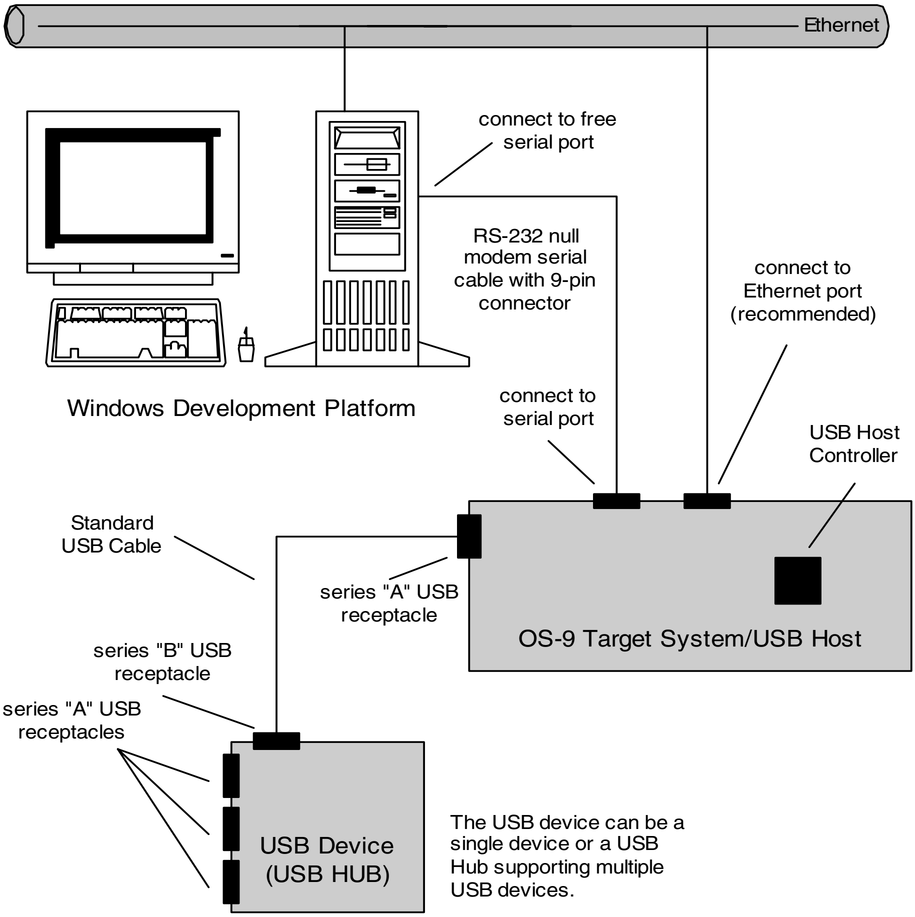
- **System Overview**
- **System Requirements**
- **Installing the Software**
- **Example Commands**

RadiSys.
MICROWARE SOFTWARE

System Overview

Figure 1-1 shows a typical development environment for using USB Host SDK for OS-9. It is recommended that you assemble and configure your development environment before software installation.

Figure 1-1 USB Host Development Environment



System Requirements

Windows Development Platform Hardware Requirements

Your Windows development platform must have the following minimum hardware characteristics:

- 250MB of free hard disk space
- the recommended amount of RAM for your particular operating system
- a CD-ROM drive
- a free serial port
- an Ethernet network card (optional but recommended)
- access to an Ethernet network (optional but recommended)

Windows Development Platform Software Requirements

The Windows development platform must have the following software installed:

- Microware OS-9 for Embedded Systems (aka OEM Package)
- USB Host SDK for OS-9 add-on
- Windows ME, 2000, NT 4.0, or XP
- terminal emulation program

Note

The terminal emulation program, Hyperterminal, ships with all Windows operating systems.

OS-9 Target System/USB Host Hardware Requirements

Your OS-9 target system/USB Host reference board requires the following hardware:

- a free serial port
- an RS-232 null modem serial cable with 9-pin connectors
- one or more USB ports
- a standard USB cable
- a free Ethernet port (optional but recommended)
- access to an Ethernet network (optional but recommended)

Note

Some USB Host Controllers require a non-cached memory shade.

Note

To use the USB Host system, you will also need standard USB devices such as a mice, keyboards, printers, or mass storage devices and the appropriate cables.

Installing the Software

Installing to the Windows Development Platform

The **USB Host SDK for OS-9** software package is an add-on to OS-9. OS-9 must be installed on your Windows development platform before the USB Host software is installed.

To install OS-9, insert the CD-ROM into your Windows development platform CD-ROM drive and follow the on-screen instructions. After OS-9 is installed, you will be able to choose **USB Host SDK for OS-9** from the Add-Ons menu.

For More Information

For detailed installation instructions, refer to the ***Getting Started with Microware Products*** manual. This manual is accessible via Acrobat Reader from the Microware OS-9 CD.

NOTE: Portions of the source code for the USB Host SDK have this copyright/license.

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. All advertising materials mentioning features or use of this software must display the following acknowledgement:

> This product includes software developed by the NetBSD Foundation, Inc. and its contributors.

4. Neither the name of The NetBSD Foundation nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE NETBSD FOUNDATION, INC. AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Installing to the OS-9 Target System/USB Host

Before installing the USB Host software onto your OS-9 target system/USB Host, you must complete the following steps:

Step 1. Assemble and configure your USB Host development environment hardware.

Step 2. Install OS-9 and the **USB Host SDK for OS-9** software onto your Windows development platform.

Step 3. Create an OS-9 ROM Image and load it onto your OS-9 target system /USB Host.

Step 4. Boot your OS-9 Target System/USB Host to an OS-9 prompt. The OS-9 prompt must be accessible via your terminal emulation program.

For More Information

Creating an OS-9 ROM Image, loading the image onto the target system, and booting to an OS-9 prompt is described in your target system's board guide. The board guides are accessible via Acrobat Reader from the Microware OS-9 CD.

USB Host Module List

After installing the **USB Host SDK for OS-9** add-on package onto your Windows development platform, the following USB Host modules will be present on your system:

- USB Controller Drivers

```
C:\MWOS\OS9000\<PROCESSOR>\PORTS\<BOARD_PORT>\CMDS\BOOTOBJS\usbhcd
C:\MWOS\OS9000\<PROCESSOR>\PORTS\<BOARD_PORT>\CMDS\BOOTOBJS\usbhcde
```

- USB Controller Driver Descriptors

```
C:\MWOS\OS9000\<PROCESSOR>\PORTS\<BOARD_PORT>\CMDS\BOOTOBJS\NULLFM\usbhc
C:\MWOS\OS9000\<PROCESSOR>\PORTS\<BOARD_PORT>\CMDS\BOOTOBJS\NULLFM\usbhc2
C:\MWOS\OS9000\<PROCESSOR>\PORTS\<BOARD_PORT>\CMDS\BOOTOBJS\NULLFM\usbhc3
C:\MWOS\OS9000\<PROCESSOR>\PORTS\<BOARD_PORT>\CMDS\BOOTOBJS\NULLFM\usbhc4
```

- USB Keyboard MAUI Protocol Module

```
C:\MWOS\OS9000\<PROCESSOR>\PORTS\<BOARD_PORT>\CMDS\BOOTOBJS\MAUI\mp_usbkbd
```

- USB Mouse and Keyboard MAUI CDB Module

```
C:\MWOS\OS9000\<PROCESSOR>\PORTS\<BOARD_PORT>\CMDS\BOOTOBJS\MAUI\cdb_usb*
```

- USB Host Manager Driver

```
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\USBH\usbman
```

- USB Host Manager Driver Descriptor

```
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\USBH\usb
```

- USB Mouse Driver

```
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\USBH\ums
```

- USB Mouse descriptors

```
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\USBH\um0
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\USBH\um1
```

- USB Keyboard Driver

```
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\USBH\ukbd
```

- USB Keyboard Descriptor

```
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\USBH\ukbd0
```

- USB Generic Driver

```
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\USBH\spugen
```

- USB Generic Driver descriptors

```
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\USBH\ugen0
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\USBH\ugen1
```

- USB Printer Driver

```
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\USBH\ulpt
```

- USB Printer Descriptor

```
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\USBH\ulp0
```

- USB Mass Storage Device Driver

```
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\USBH\udiskd
```

- USB Mass Storage Device Descriptors

```
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\USBH\DESC\muh*
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\USBH\DESC\uh*
```

- NullFM file manager

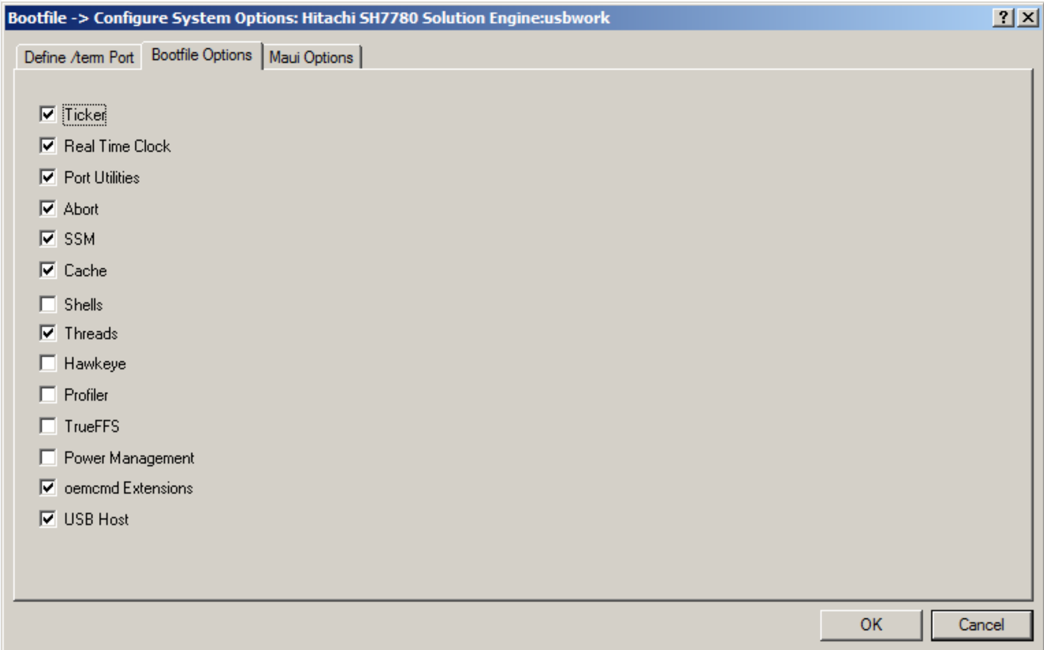
```
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\nullfm
```

Note

The drive letter, <processor> directory, and <board_port> directory will vary depending on your particular installation.

Loading and Starting the USB Host Software

The objective of this procedure is to move the USB Host modules, which are drivers and descriptors, from the Windows development platform onto the OS-9 target system/USB Host. Some port directories include support in the Wizard for USB Host. If your port directly supports USB host, there will be a USB Host check-box on the bootfile options tab.



Click this checkbox to enable USB Host support. You will also want to select MAUI, keyboard, and mouse from the master builder window to include the appropriate software.

Note

The USB Host software works best when the system tick rate (ticks per second) is 1000 or higher. This allows the USB Host software to accurately implements delays and time-outs. This value can be set on the “Init Options” tab of the Disk Options dialog.

If your port does not directly support USB host, you will need to manually load the software onto your target. There are several ways this can be accomplished and the following procedure describes only one method of accomplishing this task.

Step 1. On the Windows development platform, open a text editor, such as Notepad, and create a text file list that includes the USB Host modules. Be sure there is only one module per line and that you include the full path.

Your final text file should look something like the following:

```
C:\MWOS\OS9000\<PROCESSOR>\PORTS\<BOARD_PORT>\CMDS\BOOTOBJS\NULLFM\usbhcd
C:\MWOS\OS9000\<PROCESSOR>\PORTS\<BOARD_PORT>\CMDS\BOOTOBJS\NULLFM\usbhc
C:\MWOS\OS9000\<PROCESSOR>\PORTS\<BOARD_PORT>\CMDS\BOOTOBJS\NULLFM\usbhcde
C:\MWOS\OS9000\<PROCESSOR>\PORTS\<BOARD_PORT>\CMDS\BOOTOBJS\NULLFM\usbhc2
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\USBH\usbman
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\USBH\usb
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\USBH\ums
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\USBH\um0
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\USBH\um1
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\USBH\ukbd
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\USBH\ukbd0
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\USBH\spugen
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\USBH\ugen0
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\USBH\ugen1
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\USBH\ulpt
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\USBH\ulp0
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\USBH\udiskd
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\USBH\DESC\muh01
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\USBH\DESC\muh11
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\USBH\DESC\muh21
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\USBH\DESC\muh31
C:\MWOS\OS9000\<PROCESSOR>\CMDS\BOOTOBJS\nullfm
C:\MWOS\OS9000\<PROCESSOR>\CMDS\usbd
C:\MWOS\OS9000\<PROCESSOR>\CMDS\usbdevs
C:\MWOS\OS9000\<PROCESSOR>\CMDS\ugenstat
```

Note

The drive letter, <processor> directory, and <board_port> directory will vary depending on your particular installation.

Step 2. Save this file as usb_mods.ml on your Windows system in a location of your choice.

Step 3. On the Windows development platform, open a DOS shell. Using DOS commands, navigate to the directory where `usb_mods.ml` is located, and type the following DOS command:

```
% os9merge -z=usb_mods.ml>usb_mods
```

This creates a merged file called `usb_mods`. `usb_mods` will be located in the same directory that `usb_mods.ml` is located.

Step 4. Load the `usb_mods` file onto the OS-9 target system/USB Host system's RAM.

From the Windows desktop, start Hawk™ by selecting `Start -> RadiSys -> Microware OS-9 for <product> -> Microware Hawk IDE`.

From the **Target Menu**, select `Load`. Enter the IP address of your OS-9 target. In the Module dialog, push the navigation button and navigate to the location of the `usb_mods` file and select `usb_mods`. Press the `Load` button.

Note

This procedure requires that the Hawk debugger daemons be loaded and running on the OS-9 target system. You can make this selection while building the OS-9 ROM Image.

Step 5. Start the USB Host software by typing the following command from the terminal emulation window on the Windows development platform:

```
$ usbd &
```

The USB Host modules are now loaded and running on your OS-9 target system/USB Host.

Note

This procedure assumes that you have access to an Ethernet network for loading the USB Host software from the Windows development system to the target system. If you do not have access to a network, you can load the USB Host software via FTP across the serial connection using OS-9 commands and your terminal emulation program.

Example Commands

Checking for USB Devices

Once the USB Host software is loaded onto your OS-9 target system/USB Host, you can check the system for existing USB devices. To see what devices are plugged into the USB, type the following command in the terminal emulation program window.

```
$ usbdevs
```

Following is an example response from the command:

```
Bus #0, Root Hub, Address 1,
[1] <empty>
[2] Address 2,  NOVATEK: ORTEK USB Keyboard

Bus #1, Root Hub, Address 1,
[1] <empty>
[2] <empty>

Bus #2, Root Hub, Address 1,
[1] Address 2,  SanDisk Corporation: U3 Cruzer Micro: 0000051015079136
[2] <empty>
[3] <empty>
[4] <empty>
```

The above example shows the three root hubs (two USB v1.1 controllers and one USB v2.0 controller). A USB keyboard is plugged into a USB v1.1 controller and a USB mass storage (Flash disk) device is plugged into the USB v2.0 controller. The keyboard and disk has been both been assigned address 2, but on different busses.

Getting Device Information

You can view information about USB devices on the system. For example, to learn more about the USB keyboard device in the example above, type the following command in the terminal emulation program window. Note that the bus number must be specified so that the device address is not ambiguous:

```
$ usbdevs -a=2 -b=0
```

Following is an example response from the command:

```
Address 2,  NOVATEK: ORTEK USB Keyboard (vendor 1444, product 38705)
    Device Descriptor:  max_packet 8, protocol 0, release 0.1, configurations 1
    Config. Descriptor (1):  interfaces 2, value 1, iconfig 0
        attributes 0xa0, max power 100 mA
    Interface Descriptor 1:  NOVATEK
        alt. setting 0, num eps 1,
        class 3, subclass 1, protocol 1, iInterface 4
    Interface Descriptor 2:  NOVATEK
        alt. setting 0, num eps 1,
        class 3, subclass 1, protocol 2, iInterface 4
```

```
$ usbdevs -a=2 -b=2
```

Following is an example response from the command:

```
Address 2,  SanDisk Corporation: U3 Cruzer Micro: 0000051015079136 (vendor 1921,
 product 21506)
    Device Descriptor:  max_packet 64, protocol 0, release 0.2, configurations 1
    Config. Descriptor (1):  interfaces 1, value 1, iconfig 0
        attributes 0x80, max power 200 mA
    Interface Descriptor 1:  alt. setting 0, num eps 2,
        class 8, subclass 6, protocol 80, iInterface 0
```

Checking for Data Transmission

You can determine if a USB device is sending data over the USB. For example, to determine if the USB keyboard device in the example above is sending keyboard data over the USB, type the following commands in the terminal emulation program window:

```
$ tmode nopause
$ dump /ukbd0
```

After typing the commands, type on the USB keyboard. Following is an example response from the command:

```
  Addr     0 1  2 3  4 5  6 7  8 9  A B  C D  E F 0 2 4 6 8 A C E
--------  ---- ---- ---- ---- ---- ---- ---- ---- ----------------
00000000  0000 0f00 0000 0000 0000 0000 0000 0000 ................
00000010  0000 0f00 0000 0000 0000 0d00 0000 0000 ................
         ***      2. duplicate lines ***
00000040  0000 0000 0000 0000 0000 0f00 0000 0000 ................
00000050  0000 0d00 0000 0000 0000 0f00 0000 0000 ................
00000060  0000 0d00 0000 0000 0000 0000 0000 0000 ................
00000070  0000 1c0c 0000 0000 0000 1c00 0000 0000 ................
00000080  0000 0000 0000 0000 0000 5100 0000 0000 ..........Q.....
```

You can press Ctrl-C to exit `dump`.

Mouse Through MAUI®

To use a USB Mouse as a MAUI® input device complete the following steps:

Step 1. Load the following special modules on the OS-9 target machine:

- Standard MAUI PS/2 Mouse protocol module.

```
C:\MWOS\OS9000\<PROCESSOR>\PORTS\<BOARD_PORT>\CMDS\BOOTOBJS\MAUI\mp_bsptr
```

- CDB Module that defines a USB Mouse for MAUI

```
C:\MWOS\OS9000\<PROCESSOR>\PORTS\<BOARD_PORT>\CMDS\BOOTOBJS\MAUI\cdb_usb
```

Step 2. Load the following MAUI modules on the OS-9 target. These modules are included with OS-9.

```
OS9000/<PROCESSOR>/CMDS/maui
OS9000/<PROCESSOR>/CMDS/BOOTOBJS/maui_inp
OS9000/<PROCESSOR>/CMDS/BOOTOBJS/maui_win
OS9000/<PROCESSOR>/CMDS/BOOTOBJS/mfm
OS9000/<PROCESSOR>/CMDS/BOOTOBJS/mauidev
OS9000/<PROCESSOR>/CMDS/BOOTOBJS/mauidrvr
OS9000/<PROCESSOR>/CMDS/MAUIDEMO/inp
```

Step 3. Type the following commands in the terminal emulation program window:

```
$ maui_inp &
$ tmode nopause
$ inp -i=/um0/mp_bsptr
```

Following is an example response from the command:

```
Opening device '/um0/mp_bsptr'
Send signal to 'inp' to end test
Expected device id 0x3fa8018
```

Step 4. Move the mouse, or click buttons on the mouse.

Following is an example response from the command:

```
+-------------------------------------------------------+
Device type:  +++ Pointer +++  Device ID:   0x3fa8018
| Sub-type: 0x1
|                  INP_PTR_DOWN
| Button changed: 1
| Button status 1 (0x1)
| New position (0,0)
| Simulating keysym: INP_KEY_NULL (0x0)
+-------------------------------------------------------+
Device type:  +++ Pointer +++  Device ID:   0x3fa8018
| Sub-type: 0x2
|                  INP_PTR_UP
| Button changed: 1
| Button status 0 (0x0)
| New position (0,0)
| Simulating keysym: INP_KEY_NULL (0x0)
+-------------------------------------------------------+
```

Step 5. You can press Ctrl-C to exit `inp`

Chapter 2: Using USB Host for OS-9

This chapter provides a description of the OS-9 implementation for USB host. It includes the following sections:

- **Overview**
- **Hardware Controller Driver**
- **USB Management Driver**
- **Logical Device Drivers**
- **Standard OS-9 LDD Drivers**
- **User-State Daemon Process**

RadiSys.
MICROWARE SOFTWARE

Overview

The stack for the OS-9 implementation of USB Host consists of the following three main components:

- Hardware Controller Drivers
- USB Management Driver
- Logical Device Drivers

Dividing the USB Host responsibilities between these components provides maximum modularity and flexibility, enables easy maintenance, and ensures performance. Each component is described in the following sections of this chapter.

Figure 2-1 provides a visual overview of the USB Host stack. **Figure 2-2** shows the overall USB Host architecture as it relates to an OS-9 system.

Figure 2-1 USB Host Stack

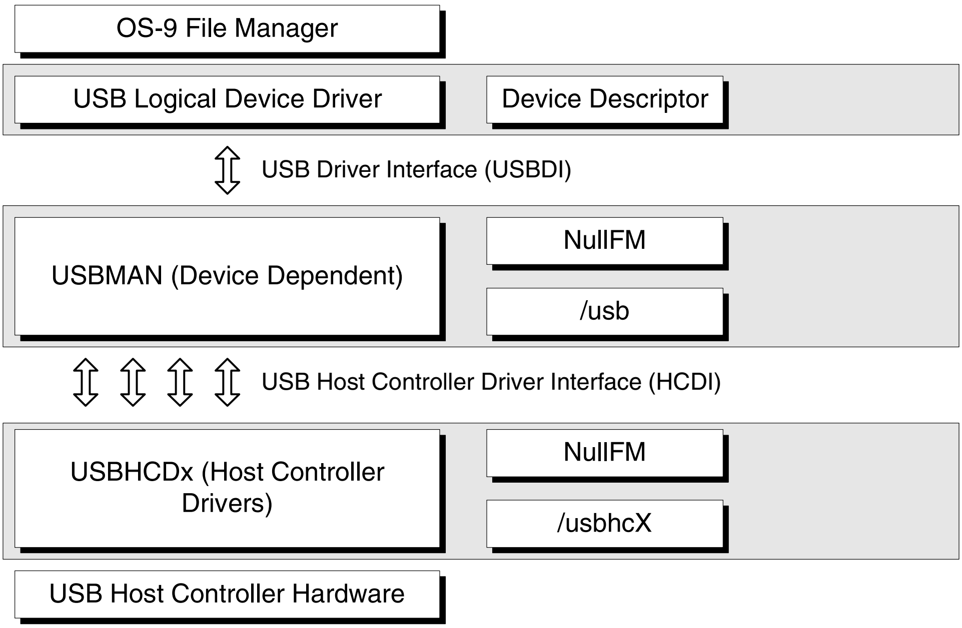
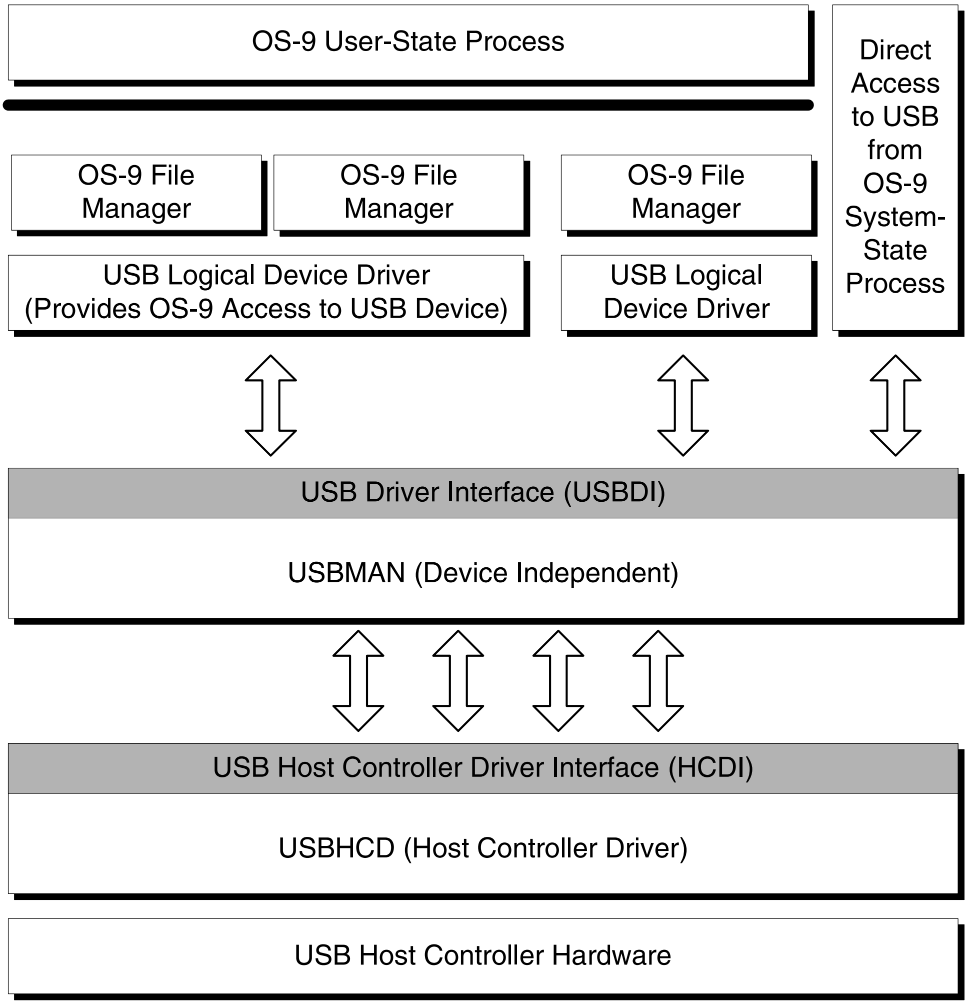


Figure 2-2 USB Host Architecture



Hardware Controller Driver

The hardware controller driver is responsible for initializing the USB hardware, scheduling transfers on the USB, managing the root hub, and notifying logical device drivers when a transfer has completed. This driver is given tasks to perform by `usbman` through the HCDI interface.

The HCDI interface is a series of function pointers into the Hardware Controller Driver to open and close pipes, allocate DMA memory, and perform transfers on the USB. There are two classifications of function pointers—bus methods and pipe methods.

Bus Methods Structure

Following is the bus methods structure:

```
struct usbd_bus_methods {
usbd_status               (*open_pipe)(struct usbd_pipe *pipe);
    void                  (*soft_intr)(void *);
    void                  (*do_poll)(struct usbd_bus *bus);
    usbd_status           (*allocm)(struct usbd_bus *bus, usb_dma_t *dma,
                           u_int32_t bufsize);
    void                  (*freem)(struct usbd_bus *bus, usb_dma_t *dma);
    struct usbd_xfer *    (*allocx)(struct usbd_bus *bus);
    void                  (*freex)(struct usbd_bus *bus, struct usbd_xfer *x);
};
```

Bus Methods Structure Fields

`open_pipe` Notifies the Hardware Controller Driver of a new transfer pipe to a device on the USB. This call modifies the `methods` and `methods_gp` fields of the given pipe structure.

`soft_intr` Notifies the Hardware Controller Driver there are potentially aborted transfers to be cleaned up.

`allocm`	Allocates memory suitable for DMA. This modifies the `dma` parameter. This will return `USBD_NORMAL_COMPLETION` on success, or `USBD_NOMEM` if no memory available.
`freem`	Frees memory allocated by `allocm`.
`allocx`	Allocates a transfer handle, and returns it.
`freex`	Frees a transfer handle allocated by `allocx.`

Pipe Methods Structure

The pipe methods structure below is initialized after calling `open_pipe` in the bus methods structure.

```
struct usbd_pipe_methods {
    usbd_status      (*transfer)(usbd_xfer_handle xfer);
    usbd_status      (*start)(usbd_xfer_handle xfer);
    void             (*abort)(usbd_xfer_handle xfer);
    void             (*close)(usbd_pipe_handle pipe);
    void             (*cleartoggle)(usbd_pipe_handle pipe);
    void             (*done)(usbd_xfer_handle xfer);
};
```

Bus Methods Structure Fields

`transfer`	Performs a transfer on the USB.
`start`	Starts the next transfer to the device.
`abort`	Aborts a transfer on the USB.
`close`	Closes a transfer. The transfer must not be active to call this (i.e. use `abort` first).
`cleartoggle`	Clears the data toggle back to 0.
`done`	Called after successfully completing a transfer.

The common names for the Hardware Controller Drivers begin with `usbhcd.` The device descriptors for these drivers are in the form `usbhc?` where `?` is either no character, ‘`2`’, ‘`3`’, or ‘`4`’. `usbman` will attempt to open usbhc, usbhc2, usbhc3, and usbhc4 (in that order) when initializing the USB stack. Typically, the drivers are organized such that low or full speed drivers are opened first and any high speed driver is opened last. This driver can be initialized with the iniz command (for example iniz /usbhc). Upon doing so, the Hardware Controller Driver will initialize the hardware and begin generating Start Of Frame packets every 1ms on the USB. The recommended method for intializing the USB stack, however, is to start the usbd daemon process.

The hardware controller driver is the only board specific module required for the OS-9 USB Stack. Consequently, it is found in the CMDS\BOOTOBJS directory of the board PORT directory.

USB Management Driver

The USB Management Driver, `usbman`, is a `nullfm` driver that implements the management layer of the USB Host software. It has the following responsibilities:

- Maintains bus topology
- Implements USBDI interface for LDDs
- Performs USB explore
- Implements hub driver
- Manages plug-n-play

`usbman` communicates directly to the Hardware Controller Driver through the HCDI interface and other setstat/getstat calls.

`usbman` is located in the following directory:

```
OS9000/<PROCESSOR>/CMDS/BOOTOBJS/USBH/usman
```

The `usbman` descriptor is located in the following directory:

```
OS9000/<PROCESSOR>/CMDS/BOOTOBJS/USBH/usb
```

Bus Explore

Most of the responsibilities of `usbman` revolve around a bus explore. The process is started by plugging in or removing a device from the USB. Below is a short description of the sequence of events in a bus explore:

Step 1. The hub driver, as a part of `usbman`, receives notification that its pipe has transferred data. The interrupt service routine for the hub driver sends a signal to the USBD daemon process indicating that a USB explore is required.

Note

The explore of the USB may take several seconds, thus necessitating the use of a process context for the explore.

Step 2. `usbd` wakes up and performs an explore setstat into `usbman`. The explore code in `usbman` performs a depth first search on the USB starting with the root hub.

Step 3. The explore code inspects each port on every hub, one at a time, to determine if any change is present. A change may be either something plugged in or removed, or an overcurrent condition.

- Device Removed

  If a device was removed, the `detach` LDD routine is called for the driver that is assigned to the device. The control pipe is then closed, and any memory is removed. If the device was a hub, then each downstream device will have its `detach` LDD routine called, followed by closing the control pipe and memory reclamation.

- Device Inserted

  If a device was inserted, `usbman` opens a control pipe and gathers basic information about the device. `usbman` then attempts to match an available LDD to this device using the `match` routine. If a driver matches, then the `attach` LDD routine is called, and this LDD is no longer considered available.

- Overcurrent

  If a port on a hub is overcurrent, it is treated as if the device was removed. However, the port may not be used again unless the entire hub is removed from the USB and re-inserted.

USB Addresses are assigned by `usbman`. There is no rule that a particular device will always be assigned a particular address. In addition, there is no order for matching a driver to a new device.

Plug and Play

Plug and play is accomplished in the OS-9 USB Host stack through a usbman callout mechanism. usbman makes this call to one of three possible functions: match, attach, or detach. Essentially, the call is initiated when there is a device modification; usbman recognizes the modification and calls the appropriate function to notify the LDD.

For example, suppose a device were removed from USB. At this point, usbman would call the detach function, which would then tell the available LDD to change the device information.

More information on the match, attach, and detach functions is provided in the following sections.

Match

match is called by usbman when there is an attempt to assign a device on the USB to an available LDD. This function is called after a device is plugged into the bus or after a driver registers with usbman.

In addition, match is passed as a device structure and an interface structure. Both of these represent the current state of the device on the USB. The match routine should look at these two parameters (device structure and interface structure) to determine whether or not the driver can communicate with the device.

Below is a sample prototype of the match function:

```
int os9_match(struct usbd_device *dev,
usbd_interface_handle iface)
```

The match function may perform transfers on the USB over the control pipe, since that has already been established by usbman. Such transfers would likely be to retrieve endpoint or vendor-specific descriptors. The configuration and interface for the device should not be set at this time. usbman will loop on each interface in a configuration (for each configuration).

The `match` function should return 0 if the device does not match what the driver expects. If the driver can communicate with the USB device, the match function should return any non-zero value.

Attach

The `attach` function is called by `usbman` after a successful call to the `match` function. Attach should open any relevant transfer pipes and perform any other setup required to initialize the device. This function will return a value from the `usb_status` enumerated type (located in `usb.h`). Below is a sample prototype for the `attach` function.

```
usbd_status os9_attach(usbd_device_handle dev,
usbd_interface_handle iface)
```

Note

If the `attach` function returns an error, the detach function will not be called. This means that the `attach` function must properly deallocate resources allocated prior to the error condition.

Detach

The `detach` function is called by `usbman` if the device is removed from the USB, or if the driver is de-registering itself with `usbman`. This function is responsible for deallocating any resources acquired in the `attach` routine. Normally, this means closing pipes and freeing memory. Below is an example detach function prototype:

```
usbd_status os9_detach(usbd_device_handle dev)
```

Registering with usbman

The following routines perform plug and play on the USB for OS-9. These routines are provide a way for usbman to call back into the LDD. Each LDD registers its functions with usbman when it initializes. Below is the plug and play structure followed by a brief description of each field.

```
typedef struct {
  usbd_status  (*detach)(struct usbd_device *dev);
  usbd_status  (*attach)(struct usbd_device *dev, usbd_interface_handle iface);
  int           (*match) (struct usbd_device *dev, usbd_interface_handle iface);
  void *gp;        /* ldd global pointer */
  void *dev_data;  /* (ldd) device specific data */
} usbd_ldd_t;
```

`detach` — This is called when a device is removed from the USB. This routine should close any interrupt, bulk, or isochronous pipes and any other resources allocated in the `attach` routine. `usbman` will close the control pipe.

`attach` — This is called after a successful return from `match`. This routine should open any pipes required for this device to function. It may also perform transfers over the control pipe.

`match` — This routine will determine if the given device and interface are appropriate for this LDD. If no match is possible, then return `UMATCH_NONE`. Otherwise, return `UMATCH_IFACECLASS`. This routine may also perform transfers over the control pipe. However, do not attempt to change the interface. If the given interface, `iface`, is not suitable, return `UMATCH_NONE`. The `usbman` explore routine will iterate over all interfaces.

`gp` — This is the LDD global pointer, and should be set properly by the LDD before registering the `attach`/`match`/`detach` routines with `usbman`.

`dev_data`	This is for specific use by the LDD. In some circumstances, it is useful to place information here in the `attach` routine.

Logical Device Drivers

A Logical Device Driver (LDD) implements code to support a particular USB device like a mouse, keyboard, or printer. It is intended that each LDD support the standard OS-9 interface as much as possible. These drivers interface to `usbman` using the USBDI interface. LDDs may use any OS-9 file manager, including `nullfm`.

LDD Initialization

Each LDD must perform the following steps once during initialization:

Step 1. Open /usb. This opens a path to `usbman` so that this LDD may use the USBDI interface.

Step 2. Perform a `GS_USB_USBMAN_IFACE` getstat to retrieve function pointers that implement the USBDI interface.

Step 3. Perform a `SS_USB_LDD_METHODS` setstat to register attach, match, detach routines with `usbman`.

LDD De-Initialization

Perform the following steps to de-initialize an LDD:

Step 1. Perform `SS_USB_LDD_METHODS` setstat (with enable field set to 0) to remove registration with `usbman`.

Step 2. Close path to `/usb`.

Suggested OS-9 Interface

It is suggested that each LDD support the standard OS-9 interface. Below is a list of setstats/getstats that each LDD should implement, if possible.

Setstats

SS_SENDSIG	Send signal on data registration.
SS_RELEASE	Remove SS_SENDSIG registration.

Getstats

SS_READY	Return number of bytes ready for read.

Plug-n-play

Each LDD must register an attach, match, and detach routine with usbman. These routines facilitate plug-n-play under OS-9. Following is a code snippet showing how to register these routines.

```
usbd_ldd_t mouse_ldd = {os9_detach_mouse,
               os9_attach_mouse,
               os9_match_mouse,
               NULL,   /* gp */
               NULL};  /* dev_data */


/* register attach/match/detach with usbman */
methods_pb.enable = 1;
mouse_ldd.gp = get_static();
methods_pb.ldd = &mouse_ldd;
err = _os_setstat(usb_path, SS_USB_LDD_METHODS, &methods_pb);
```

Removing registration with usbman should only occur in the term part of the driver. Following is an example:

```
/* un-register with usbman */
methods_pb.enable = 0;
methods_pb.ldd = &mouse_ldd;
(void) _os_setstat(usb_path, SS_USB_LDD_METHODS, &methods_pb);
```

Standard OS-9 LDD Drivers

The OS-9 USB Host Stack ships with the following Logical Device Drivers:

- **USB Mouse**
- **USB Keyboard**
- **USB Printer**
- **USB Mass Storage**
- **USB Mass Storage**

USB Mouse

The USB Host Mouse driver is implemented as a `nullfm` Driver. It supports the standard OS-9 interface for read, `SS_RELEASE`, `SS_SENDSIG`, and `SS_READY`. The standard OS-9 utilities, such as `dump`, can be used with this driver. This driver attaches to any device that declares itself to be a HID Mouse device with an x and y report. The driver and its descriptors are found in the following locations:

- Source Directory:

  `SRC/DPIO/NULLFM/DRVR/USBH/UMS`

- Driver Location:

  `OS9000/<PROCESSOR>/CMDS/BOOTOBJS/USBH/ums`

- Descriptor Location:

  `OS9000/<PROCESSOR>/CMDS/BOOTOBJS/USBH/um0`
  `OS9000/<PROCESSOR>/CMDS/BOOTOBJS/USBH/um1`

Data Format

The USB Host mouse driver generates PS/2 style data. Each mouse movement and/or button press is represented by 3 bytes. PS/2 only allows for 3 buttons and 8 bits of movement per data sample. Following is the data format:

```
Byte 0:  oy ox sy sx 1 b3 b2 b1
Byte 1:  x7 .. .. .. . .. .. x0  - signed x data
Byte 2:  y7 .. .. .. . .. .. y0  - signed y data
```

B1	button 1 down
B2	button 2 down
B3	button 3 down
Oy	overflow in y direction
Ox	overflow in x direction
Sy	sign bit in y direction
Sx	sign bit in x direction

Use With MAUI

To use the USB Mouse with MAUI, the correct protocol module `cdb` is required. The USB Mouse uses the `mp_bsptr` protocol module. This is the standard PS/2 mouse protocol module. Since the USB Mouse driver generates PS/2 data, `mp_bsptr` is very functional.

For applications to be aware of the USB Mouse, a `cdb` entry must be added. Following is a code snippet that shows a USB Mouse entry in a `cdb.a` file. This file is found in the following location:

`OS9000/<PROCESSOR>/PORTS/<BOARD>/MAUI/CDB`

```
psect cdb,(5<<8)+1,$8000,212,0,entry

org 0

entry:
   (Other entries here)
  dc.b "5:/um0/mp_bsptr:TY=\"ptr\":",13          * USB Mouse

  ends
```

Testing the USB Mouse

The USB Mouse can be tested in two ways. The first, and simplest method, is using the OS-9 `dump` utility. Following is an example of using OS-9 `dump`:

```
$ tmode nopause
$ dump /um0
(Move the mouse and press buttons)
  Addr     0 1  2 3  4 5  6 7  8 9  A B  C D  E F 0 2 4 6 8 A C E
-------- ---- ---- ---- ---- ---- ---- ---- ---- ----------------
00000000 0900 000b 0000 0900 000b 0000 0a00 0008 ................
00000010 0000 0803 0308 0403 0806 0408 0603 0806 ................
00000020 0308 0403 0803 0308 0002 08ff 0208 fe01 ..............~.
(Ctrl-C to exit)
Error #000:177
```

The second method for testing the mouse is to use the MAUI `inp` demo software. Following is an example of using `inp`:

```
$ maui_inp &
$ tmode nopause
$ inp -i=/um0/mp_bsptr
Opening device '/um0/mp_bsptr'
Send signal to 'inp' to end test
Expected device id 0x3fa8018
+-------------------------------------------------------+
Device type:  +++ Pointer +++  Device ID:   0x3fa8018
| Sub-type: 0x4
|                   INP_PTR_MOVE
| Button changed: 0
| Button status 0 (0x0)
| New position (-64,117)
| Simulating keysym: INP_KEY_NULL (0x0)
+-------------------------------------------------------+
Device type:  +++ Pointer +++  Device ID:   0x3fa8018
| Sub-type: 0x1
|                   INP_PTR_DOWN
| Button changed: 2
| Button status 2 (0x2)
| New position (-64,117)
| Simulating keysym: INP_KEY_NULL (0x0)
+-------------------------------------------------------+
Device type:  +++ Pointer +++  Device ID:   0x3fa8018
| Sub-type: 0x2
|                   INP_PTR_UP
| Button changed: 2
| Button status 0 (0x0)
| New position (-64,117)
| Simulating keysym: INP_KEY_NULL (0x0)
+-------------------------------------------------------+
(Ctrl-C to exit)
```

USB Keyboard

The USB Host Keyboard driver is implemented as a `nullfm` Driver. It supports the standard OS-9 interface for read, `SS_RELEASE`, `SS_SENDSIG`, and `SS_READY`. The standard OS-9 utilities, such as `dump`, can be used with this driver. This driver attaches to any device that declares itself to be a HID Keyboard that uses the BOOT Protocol. The driver and its descriptors are found in the following locations:

- Source Directory:

  `SRC/DPIO/NULLFM/DRVR/USBH/UKBD`

- Driver Location:

  `OS9000/<PROCESSOR>/CMDS/BOOTOBJS/USBH/ukbd`

- Descriptor Location:

  `OS9000/<PROCESSOR>/CMDS/BOOTOBJS/USBH/ukbd0`
  `OS9000/<PROCESSOR>/CMDS/BOOTOBJS/USBH/ukbd1`

Data Format

The USB Keyboard uses an 8-byte data format. Below is a C-style structure describing the format. A special protocol module, named `mp_usbkbd` was created to handle this exact format.

```
#define KEYSLOTS 6
typedef struct {
  u_int8   modifiers;
#define MOD_CONTROL_L0x01
#define MOD_CONTROL_R0x10
#define MOD_SHIFT_L0x02
#define MOD_SHIFT_R0x20
#define MOD_ALT_L0x04
#define MOD_ALT_R0x40
#define MOD_META_L0x08
#define MOD_META_R0x80
  u_int8   reserved;
  u_int8   keycode[KEYSLOTS];
} UKBD_DATA;
```

The USB Keyboard can handle up to 6 characters pressed at a time. The keycode array represents “down” keys. "Up" keys must be deduced from consecutive packets. That is to say, if a particular key is "down", and then is not present in the keycode array on the next packet, then the key is declared "up".

For More Information

For more information about the keyboard data packet, please refer to the Device Class Definition for Human Interface Devices (HID) at `www.usb.org`.

Use With MAUI

To use the USB Keyboard with MAUI, the correct protocol module and an updated `cdb` module will be required. The USB Keyboard uses the `mp_usbkbd` protocol module. This is found in the following location:

```
SRC/MAUI/MP/MP_USBKBD
```

For applications to be aware of the USB Keyboard, a `cdb` entry must be added. Below is a code snippet that shows a USB Keyboard entry in a `cdb.a` file. This file is found in the following location:

```
OS9000/<PROCESSOR>/PORTS/<BOARD>/MAUI/CDB
```

```
psect cdb,(5<<8)+1,$8000,212,0,entry

org 0

entry:
   (Other entries here)
  dc.b "5:/ukbd0/mp_usbkbd:TY=\"ptr\":",13          * USB Keyboard

  ends
```

The `mp_usbkbd` protocol module turns separate LEDs when the Caps Lock, Num Lock, or Scroll Lock key is pressed.

The key repeat functionality (keys that repeat while holding down a particular key) is not implemented. According to the USB HID specification, auto-repeating keys while they are down is a function of the USB Software, not the keyboard. Currently, this feature does not exist in the OS-9 Keyboard driver.

Testing the USB Keyboard

The USB Keyboard can be tested in two ways. The first method uses the standard OS-9 `dump` utility. Following is an example of using OS-9 `dump`:

```
$ dump /ukbd0

  Addr     0 1  2 3  4 5  6 7  8 9  A B  C D  E F 0 2 4 6 8 A C E
--------  ---- ---- ---- ---- ---- ---- ---- ---- ----------------
00000000  0000 0400 0000 0000 0000 0000 0000 0000 ................
00000010  0000 0500 0000 0000 0000 0000 0000 0000 ................
00000020  0000 0600 0000 0000 0000 0000 0000 0000 ................
00000030  0000 0700 0000 0000 0000 0000 0000 0000 ................
00000040  0000 0800 0000 0000 0000 0000 0000 0000 ................
00000050  0200 0000 0000 0000 0200 0400 0000 0000 ................
00000060  0200 0416 0000 0000 0200 0416 0700 0000 ................
00000070  0200 0407 0000 0000 0200 0700 0000 0000 ................
00000080  0200 0000 0000 0000 0000 0000 0000 0000 ................
00000090  0000 2c00 0000 0000 0000 0000 0000 0000 ..,.............
```

The second method for testing the USB Keyboard is to use the MAUI `inp` demo software. Following is an example of using `inp`:

```
$ maui_inp &
$ tmode nopause
$ inp -i=/ukbd0/mp_usbkbd
Opening device '/ukbd0/mp_usbkbd'
Send signal to 'inp' to end test
Expected device id 0x3fa8018
+-------------------------------------------------------+
Device type: +++ Key +++ Device ID:   0x3fa8018
| Sub-type: 0x4
|                  INP_KEYMOD_DOWN
| Keysym received: INP_KEY_NULL (0x0)
| Key modifiers: 0x1
|   Shft CapL Ctrl Alt  Meta Num  Scrl
|   L R  L R  L R  L R  L R  Lock Lock
|   x
+-------------------------------------------------------+
Device type: +++ Key +++ Device ID:   0x3fa8018
| Sub-type: 0x8
|                  INP_KEYMOD_UP
| Keysym received: INP_KEY_NULL (0x0)
| Key modifiers: 0x0
```

```
|   Shft CapL Ctrl Alt  Meta Num  Scrl
|   L R  L R  L R  L R  L R  Lock Lock
|
+--------------------------------------------------------+
Device type: +++ Key +++ Device ID:   0x3fa8018
| Sub-type: 0x4
|                 INP_KEYMOD_DOWN
| Keysym received: INP_KEY_NULL (0x0)
| Key modifiers: 0x4
|   Shft CapL Ctrl Alt  Meta Num  Scrl
|   L R  L R  L R  L R  L R  Lock Lock
|        x
+--------------------------------------------------------+
(Ctrl-C to exit)
```

USB Printer

The USB Host Printer driver is implemented as a `nullfm` Driver. It supports the standard OS-9 interface for write, `SS_RELEASE`, `SS_SENDSIG`, and `SS_READY`. The standard OS-9 utilities, such as `merge`, can be used with this driver. The driver and its descriptor are found in the following locations:

- Source Directory

  `SRC/DPIO/NULLFM/DRVR/USBH/ULPT`

- Driver Location

  `OS9000/<PROCESSOR>/CMDS/BOOTOBJS/USBH/ulpt`

- Descriptor Location

  `OS9000/<PROCESSOR>/CMDS/BOOTOBJS/USBH/ulp0`

The printer driver attaches to any device advertising itself as a uni-directional or bi-directional printer. The OS-9 USB Printer Driver does not modify the data sent to the printer. That is to say, the data the application writes to the printer must be understood by the printer. The `ulpt` driver does not massage the data.

For More Information

Information about USB printers is located at `www.usb.org`.

Testing the USB Printer

Following is an example of how to test a printer using the OS-9 `merge` utility. A sample text file can be found in the following location:

`SRC/DPIO/NULLFM/DRVR/USBH/ULPT/sample.txt`

```
$ merge sample.txt>/ulp0
```

Note

Many USB Printers that accept ASCII text require a <CR><LF> at the end of each line, and a Ctrl-L as a Form Feed character. A sample text file (`sample.txt`) exists in the source directory for the printer driver.

There is also a `usbprint` utility that can be used to print a file. Following is an example command for `usbprint`:

```
$ usbprint sample.txt
```

Note

The default print device is /ulp0.

USB Mass Storage

The USB Mass Storage driver is implemented as a `nullfm` Driver. It supports the standard OS-9 interface for a disk device (either Windows FAT format or OS-9's native RBF format). The driver and its descriptors are found in the following locations:

- Source Directory

```
OS9000/SRC/IO/RBF/DRVR/USBDISK
```

- Driver Location

```
OS9000/<PROCESSOR>/CMDS/BOOTOBJS/USBH/udiskd
```

- Descriptors Location

```
OS9000/<PROCESSOR>/CMDS/BOOTOBJS/USBH/DESC/muh* (for PCF)
OS9000/<PROCESSOR>/CMDS/BOOTOBJS/USBH/DESC/uh* (for RBF)
```

There are a large number of device descriptors for various uses.

- Since USB Mass Storage devices can appear and disappear dynamically, the device descriptors refer to disk 0 as the first disk device located on the busses, disk 1 as the second disk located on the busses, and so forth, up to 3 for the fourth disk located.
- The disks have partitions - partition 1 is the first partition and partition 2 is the second.
- USB Mass Storage devices can be formatted for use with PCF (Windows/MS-DOS format) or RBF (OS-9's native disk format). Descriptor names that begin with m (for MS-DOS) are for use with the PCF file manager.
- Descriptors with a single digit before any extension refer to the entire device, including the partition table itself.
- Some files contains simplified descriptor names historically used to refer to hard disks (e.g. h0 or h1).
- Some descriptors are format enabled - allowing the format command to rewrite the file structure of the device.

The following are some examples that illustrate the general format for the descriptor file names.

muh0 - PCF format descriptor for the entire first available disk

uh11fmt - RBF format descriptor for the first partition of the second available disk with formatting enabled

muh22.h2 - PCF format descriptor for the second partition of the third available disk with a module name of h2.

Testing USB Mass Storage Devices

Following is an example of how to test USB Mass Storage devices. This example uses a 1GB Flash disk and a 250GB external hard disk, both pre-formatted for Windows.

```
$ iniz muh01
$ iniz muh11
$ dir /muh01 /muh11

                     Directory of /muh01 01:30:30
Documents       LaunchU3.exe    System
                     Directory of /muh11 01:30:30
21-Apr-06       24-Apr-06       25-Apr-06       26-Apr-06       27-Apr-06
28-Apr-06       CMDS            Recycled        System Volume Information
~vspcache.dir
$ chd /muh01
$ makdir CMDS
$ chd CMDS
$ copy /muh11/CMDS/procs
$ dir
                     Directory of . 16:49:38
procs
$
```

Generic USB Driver

The Generic USB Driver (`spugen`) enables applications to configure and transfer data directly to a device on the USB. Only bulk and interrupt pipes are supported, and there is no intention of supporting isochronous pipes. `spugen` is a SoftStax® (SPF) driver, and requires edition 269 or greater of the SPF file manager. The driver and its descriptors are found in the following locations:

- Source Directory:

  `SRC/DPIO/SPF/DRVR/USBGEN`

- Driver Location:

  `OS9000/<PROCESSOR>/CMDS/BOOTOBJS/USBH/spugen`

- Descriptor Location:

  `OS9000/<PROCESSOR>/CMDS/BOOTOBJS/USBH/ugen0`
  `OS9000/<PROCESSOR>/CMDS/BOOTOBJS/USBH/ugen1`

Plug-n-Play

With respect to plug-n-play, `spugen` registers its attach, match, and detach routines like any other Logical Device Descriptor. However, `spugen` matches to any device. In addition, `usbman` will only attempt to match the generic driver with a device after all other drivers have been given an opportunity to match. Therefore, the desired configuration is to initialize (`iniz`) all non-generic devices as well as `spugen`. In this way, any device plugged into the USB will first try to attach to regular LDDs and will then try to attach to `spugen`.

Accessing Endpoints with `spugen`

`spugen` is a special LDD because it allows a direct connection to the control pipe, and also allows a direct connection to a specific endpoint on the USB device. For example, opening `/ugen0` will open the control pipe on the device. An application can then request configuration information or make requests to the device.

To open a specific endpoint on a USB device, append a # character followed by the endpoint number after the device name. For example, /ugen0#1 will open endpoint 1 on the USB device attached to /ugen0. /ugen1#2 will open endpoint 2 on the USB device attached to /ugen1.

The application can request information about the device by making various setstat calls into the spugen driver using the control pipe. In this way, the application can determine how many endpoints a device has, and the type of device, for example a printer mouse, or camera.)

Testing spugen

Following is a list of steps for testing spugen with a USB mouse. Before you start, make sure that the following SPF modules are on your OS-9 target. This can be determined by running the mdir utility on the USB Host machine.

```
OS9000/<PROCESSOR>/CMDS/BOOTOBJS/SPF/spf
OS9000/<PROCESSOR>/CMDS/mbinstall
```

Step 1. Type the following commands at the OS-9 prompt:

```
$ usbd &
$ usbdevs
```

Following is an example response from the command:

```
$ usbdevs
Bus #0, Root Hub, Address 1,
[1] <empty>
[2] Address 2,  Fellowes Inc.: Fellowes 5 Button

Bus #1, Root Hub, Address 1,
[1] <empty>
[2] <empty>

Bus #2, Root Hub, Address 1,
[1] <empty>
[2] <empty>
[3] <empty>
[4] <empty>
```

This response shows that a mouse is present on USB bus #0 (low and full-speed bus) at address 2.

Step 2. Type the following commands at the OS-9 prompt:

```
$ iniz /ugen0
$ ugenstat
```

Following is an example response from the command:

```
Device Descriptor:  12011001 00000008 25251389 22500102 0001
Fellowes Inc.  Fellowes 5 Button
Number of Configurations:  1
Config Descriptor 1:  09022200 010100a0 32
    Full Descriptor:  09022200 010100a0 32
                      09040000 01030102 00
                      09210001 00012248 00
                      07058103 08000a

    Number of interfaces:  1
    Interface Descriptor 0:  09040000 01030102 00

      Number of endpoints:  1
      Endpoint Descriptor 0:  07058103 08000a
```

This response shows that the UGEN driver is attached to the mouse. By decoding the configuration and endpoint data, this mouse has only one endpoint, numbered 1.

Step 3. Type the following commands at the OS-9 prompt:

```
$ dump "/ugen0#1"
```

Following is an example response from the command if the mouse is then moved:

```
  Addr     0 1  2 3  4 5  6 7  8 9  A B  C D  E F 0 2 4 6 8 A C E
--------  ---- ---- ---- ---- ---- ---- ---- ---- ----------------
00000000  0000 ff00 00ff 0100 00fb 0000 00fd 0300 ........{...}..
00000010  00fe 0500 0000 0100 00f6 0f00 00f5 1600 .~.......v...u..
00000020  00f3 1800 00f6 1800 00f8 1200 00f6 0a00 .s...v...x...v..
00000030  00f5 0b00 00f7 0900 00f8 0800 00fa 0400 .u...w...x...z..
00000040  00fa 0400 00fd 0200 00fd 0100 00fe 0100 .z...}...}...~..
00000050  00ff 0000 0001 0000 0000 ff00 0004 ff00 ................
00000060  0004 ff00 0006 fe00 0007 fe00 0008 ff00 ......~...~.....
00000070  0008 0000 000b ff00 000d 0000 000e 0000 ................
(Ctrl-C to exit)
```

For More Information

For information regarding the data format of the device, configuration, interface, and endpoint descriptors, please refer to the USB specifications, which can be found at `www.usb.org`.

Reference API

An Application may make many getstat/setstat calls into `spugen` to either query information about the device, or to set the device configuration. Below is a list of these, and their purposes. Since `spugen` is a SoftStax (SPF) driver, the standard SPF getstat/setstat parameter block is used. Structures used for ugen getstat/setstat values can be found in the following location:

`SRC/IO/USBH/DEFS/usb.h` and `SRC/DEFS/HW/usb_host.h.`

```
/* generic getstat/setstat parameter block */
struct spf_ss_pb {
    u_int32 code;/* setstat module code*/
    u_int32 size;/* size of mod_param*/
    void*   param;  /* module parameter block*/
    u_int8  updir;  /* gs/ss going up the stak flag */
        #define SPB_GOINGUP1/* Param blk is going up stack*/
        #define SPB_GOINGDWN 0/* Param blk going down stack*/
    u_int8  rsv[3];/* RESERVED FOR FUTURE USE!*/
};
```

GS_USB_GET_CONFIG

Get current Device Configuration Value

Syntax

```
int config;
err = _os_getstat(path, GS_USB_GET_CONFIG, &config);
```

Description

Get current device configuration value.

Return Value

`EIO`	I/O error retrieving configuration information from device.
`SUCCESS`	Retrieved configuration value.

GS_USB_GET_ALTINTERFACE

Get Alternate Interface Value

Syntax

```
struct usb_alt_interface ai;
err = _os_getstat(path, GS_USB_GET_ALTINTERFACE, &ai);
```

Description

Get alternate interface value.

Return Value

`ENIVAL`	No interface selected for this device.
`EIO`	Error retrieving alternate interface.
`SUCCESS`	Retrieved alternate interface value, and assigned to `ai->alt_no`.

GS_USB_GET_NO_ALT

Get Number of Alternate Interfaces

Syntax

```
struct usb_alt_interface ai;
err = _os_getstat(path, GS_USB_GET_NO_ALT, &ai);
```

Description

Get number of alternate interfaces.

Return Value

`ENIVAL`	No interface selected for this device.
`EIO`	Error retrieving alternate interface.
`SUCCESS`	Retrieved number of alternate interfaces, and assigned to `ai->alt_no`.

GS_USB_GET_DEVICE_DESC

Get Device Descriptor

Syntax

```
usb_device_descriptor_t dev_desc;
err = _os_getstat(path, GS_USB_GET_DEVICE_DESC, &dev_desc);
```

Description

Get device descriptor.

Return Value

`EINVAL`	No device descriptor available.
`SUCCESS`	Returns device descriptor.

GS_USB_GET_CONFIG_DESC

Get Current Configuration Descriptor

Syntax

```
struct usb_config_desc config_desc;
err = _os_getstat(path, GS_USB_GET_CONFIG_DESC, &config_desc);
```

Description

Get current configuration descriptor.

Return Value

`EINVAL`	No configuration descriptor available.
`SUCCESS`	Returns device descriptor.

GS_USB_GET_INTERFACE_DESC

Get Interface Descriptor on Device

Syntax

```
struct usb_interface_desc iface_desc;
err = _os_getstat(path,GS_USB_GET_INTERFACE_DESC,&iface_desc);
```

Description

Get interface descriptor on device.

- `iface_desc.config_index`

  Configuration index to use, or -1 for the current configuration.

- `iface_desc.interface_index`

  Interface index to use, or -1 for the current interface.

- `iface_desc.alt_index`

  Alternate index to use, or -1 for current alternate interface.

Return Value

`EINVAL`	No configuration or interface descriptor.
`SUCCESS`	Returns interface descriptor in `iface_desc.desc`.

GS_USB_GET_ENDPOINT_DESC

Get Endpoint Descriptor on Device

Syntax

```
struct usb_endpoint_desc ep_desc;
err = _os_getstat(path,GS_USB_GET_ENDPOINT_DESC,&iface_desc);
```

Description

Get endpoint descriptor on device.

- `ep_desc.config_index`

  Configuration index to use, or -1 for the current configuration.

- `ep_desc.interface_index`

  Interface index to use, or -1 for the current interface.

- `ep_desc.alt_index`

  Alternate index to use, or -1 for current alternate interface.

- `ep_desc.endpoint_index`

  Endpoint index to use.

Return Value

`EINVAL`	Could not get information on configuration descriptor.
`SUCCESS`	Endpoint descriptor copied to `ep_desc.desc`.

GS_USB_GET_STRING_DESC

get string descriptor from USB device

Syntax

```
struct usb_string_desc string_desc;
err = _os_getstat(path,GS_USB_GET_STRING_DESC,&string_desc);
```

Description

Get string descriptor from USB device.

- `string_desc.string_index`

  String index from device, configuration, or interface descriptor.

- `string_desc.language_id`

  Language to use (0 if ASCII).

Return Value

`EINVAL`	I/O error retrieving string descriptor.
`SUCCESS`	String descriptor copied to `string_desc.desc`.

SS_USB_SET_CONFIG

Set the Configuration Index

Syntax

```
int config_index=1;
err = _os_setstat(path,SS_USB_SET_CONFIG,&config_index);
```

Description

Set the configuration index.

Note

This must be done before any paths are opened to a specific endpoint, for example `/ugen0#1`.

Return Value

`EPERM`	No write permission on opened path.
`EIO`	Error setting configuration.
`SUCCESS`	Configuration set to given index.

SS_USB_SET_ALTINTERFACE

Sets the Alternate Interface

Syntax

```
struct usb_alt_interface alt_iface;
alt_iface.alt_no = 0;
err = _os_setstat(path,SS_USB_SET_ALTINTERFACE,&alt_iface);
```

Description

Sets the alternate interface.

Note

This must be done before any paths are opened to a specific endpoint, for example `/ugen0#1`.

Return Value

`EPERM`	No write permission on opened path.
`EIO`	Error setting configuration.
`SUCCESS`	Configuration set to given index.

SS_USB_DO_REQUEST

Performs a Device Specific Request Over the Control Pipe

Syntax

```
struct usb_ctl_request req;
err = _os_setstat(path,SS_USB_DO_REQUEST,&req);
```

Description

Performs a device specific request over the control pipe.

`req.addr`	Device address.
`req.request`	Standard 8 byte device request structure initialized.
`req.data`	Pointer to memory where data returned from the device will be stored.
`req.flags`	0, or `USBD_SHORT_XFER_OK`.

Return Value

`EPERM`	No write permission on opened path
`EINVAL`	Returned if a `SET ADDRESS`, `SET CONFIGURATION`, or `SET INTERFACE` request is attempted.
`EIO`	Error setting configuration.
`SUCCESS`	Request performed.

Any data returned will be stored in the `data` field. The actual number of bytes returned will be stored in `req.actlen`.

SS_USB_SET_SHORT_XFER

Allows Short Transfers

Syntax

```
err = _os_setstat(path,SS_USB_SET_SHORT_XFER,NULL);
```

Description

Allows short transfers (less than the maximum endpoint length) when reading data from the USB device. This is not for the control pipe, but for other endpoints, such as `/ugen0#1`.

Return Value

`EINVAL`	Attempt to set for control pipe (`/ugen0`), or no interrupt or bulk pipe open for read.
`SUCCESS`	Allow short reads on this pipe.

User-State Daemon Process

The user-state daemon process, `usbd`, serves the following purposes:

- Initializes the USB Host stack
- Activated to perform bus explore code in `usbman` when a device is plugged in or removed from the USB.
- Initiates an asynchronous "clear endpoint stall". This can occur if a driver determines an error condition in the interrupt service routine.

To initialize the USB stack, use the following command at the OS-9 prompt:

```
$ usbd &
```

`usbd` will respond to a signal 2 or 3, to shut down the stack. Also, all drivers must be de-initialized for the USB Host Stack to properly terminate.

Chapter 3: USB Host API Reference

This chapter provides a library function reference for USB Host for OS-9. It documents the USBDI interface.

The USBDI interface is the API that implements access to the USBMAN driver. Any USB logical device driver or system-state application accesses the USB through the USBDI API.

The function references are sorted into the following categories:

- **Pipe Functions List**
- **Transfer Functions List**
- **Interface Functions List**
- **Device Functions List**
- **Alphabetical Listing**

Pipe Functions List

Table 3-1 Pipe Functions

Function Name	Description
usbman_abort_pipe()	Abort a Pipe Operation
usbman_clear_endpoint_stall()	Clear STALLED Condition
usbman_clear_endpoint_stall_async()	Clear STALLED Condition
usbman_clear_endpoint_toggle()	Reset Endpoint Toggle
usbman_close_pipe()	Close Pipe
usbman_do_request()	Perform Transfer Over Control Pipe
usbman_do_request_flags()	Perform Transfer
usbman_open_pipe()	Create Bulk Transfer Pipe
usbman_open_pipe_intr()	Create Interrupt Pipe
usbman_pipe2device_handle()	Return Device Handle

Transfer Functions List

Table 3-2 Transfer Functions

Function Name	Description
usbman_alloc_buffer()	Allocate a DMA Buffer
usbman_alloc_xfer()	Allocate a Transfer Structure
usbman_bulk_transfer()	Perform Bulk Transfer
usbman_free_buffer()	Free DMA Buffer
usbman_free_xfer()	Free Transfer
usbman_get_buffer()	Return Current DMA Buffer Pointer
usbman_get_xfer_status()	Get Transfer Status
usbman_setup_default_xfer()	Initialize Transfer Handle
usbman_setup_isoc_xfer()	Initialize ISOC Transfer
usbman_setup_xfer()	Assign Fields in Transfer
usbman_sync_transfer()	Perform Asynchronous Transfer
usbman_transfer()	Initialize Bulk Transfer

Interface Functions List

Table 3-3 Interface Functions

Function Name	Description
`usbman_endpoint_count()`	Return Number of Endpoints
`usbman_free_report_desc()`	Deallocate Memory
`usbman_get_config()`	Request Configuration Descriptor
`usbman_get_hid_descriptor()`	Request HID Descriptor
`usbman_get_report()`	Request HID Report Descriptor
`usbman_get_report_descriptor()`	Request HID Report Descriptor
`usbman_interface2device_handle()`	Return Device Handle
`usbman_interface2endpoint_descriptor()`	Return Endpoint Descriptor
`usbman_read_report_desc()`	Allocate and read the report descriptor
`usbman_set_idle()`	Silence Report on the Interrupt In Pipe
`usbman_set_interface()`	Request Interface Change
`usbman_set_protocol()`	Switch Between Boot and Report Protocol
`usbman_set_report()`	Perform Set Report Request

Device Functions List

Table 3-4 Device Functions

Function Name	Description
`usbman_device2interface_handle()`	Return Interface Handle
`usbman_get_config_desc()`	Get Configuration Descriptor
`usbman_get_config_desc_full()`	Request Configuration Descriptor
`usbman_get_device_desc()`	Request Device Descriptor
`usbman_get_string_desc()`	Request String Descriptor
`usbman_interface_count()`	Return Number of Interfaces
`usbman_set_config_index()`	Set Configuration Index
`usbman_set_config_no()`	Set Configuration

Alphabetical Listing

Table 3-5 Alphabetical Listing of Functions

Function Name	Description
`usbman_abort_pipe()`	Abort a Pipe Operation
`usbman_alloc_buffer()`	Allocate a DMA Buffer
`usbman_read_report_desc()`	Allocate and read the report descriptor
`usbman_alloc_xfer()`	Allocate a Transfer Structure
`usbman_bulk_transfer()`	Perform Bulk Transfer
`usbman_clear_endpoint_stall()`	Clear STALLED Condition
`usbman_clear_endpoint_stall_async()`	Clear STALLED Condition
`usbman_clear_endpoint_toggle()`	Reset Endpoint Toggle
`usbman_close_pipe()`	Close Pipe
`usbman_device2interface_handle()`	Return Interface Handle
`usbman_do_request()`	Perform Transfer Over Control Pipe
`usbman_do_request_flags()`	Perform Transfer
`usbman_endpoint_count()`	Return Number of Endpoints
`usbman_find_edesc()`	Return Endpoint Descriptor
`usbman_find_idesc()`	Return Interface Descriptor
`usbman_free_buffer()`	Free DMA Buffer
`usbman_free_report_desc()`	Deallocate Memory

Table 3-5 Alphabetical Listing of Functions

Function Name	Description
`usbman_free_xfer()`	Free Transfer
`usbman_get_buffer()`	Return Current DMA Buffer Pointer
`usbman_get_config()`	Request Configuration Descriptor
`usbman_get_config_desc()`	Get Configuration Descriptor
`usbman_get_config_desc_full()`	Request Configuration Descriptor
`usbman_get_device_desc()`	Request Device Descriptor
`usbman_get_device_descriptor()`	Return Device Descriptor
`usbman_get_hid_descriptor()`	Request HID Descriptor
`usbman_get_no_alts()`	Get Number of Alternate Interfaces
`usbman_get_report()`	Request HID Report Descriptor
`usbman_get_report_descriptor()`	Request HID Report Descriptor
`usbman_get_string_desc()`	Request String Descriptor
`usbman_get_xfer_status()`	Get Transfer Status
`usbman_interface_count()`	Return Number of Interfaces
`usbman_interface2device_handle()`	Return Device Handle
`usbman_interface2endpoint_descriptor()`	Return Endpoint Descriptor
`usbman_open_pipe()`	Create Bulk Transfer Pipe
`usbman_open_pipe_intr()`	Create Interrupt Pipe
`usbman_pipe2device_handle()`	Return Device Handle

Table 3-5 Alphabetical Listing of Functions

Function Name	Description
usbman_set_config_index()	Set Configuration Index
usbman_set_config_no()	Set Configuration
usbman_set_idle()	Silence Report on the Interrupt In Pipe
usbman_set_interface()	Request Interface Change
usbman_set_protocol()	Switch Between Boot and Report Protocol
usbman_set_report()	Perform Set Report Request
usbman_setup_default_xfer()	Initialize Transfer Handle
usbman_setup_isoc_xfer()	Initialize ISOC Transfer
usbman_setup_xfer()	Assign Fields in Transfer
usbman_sync_transfer()	Perform Asynchronous Transfer
usbman_transfer()	Initialize Bulk Transfer

usbman_abort_pipe()

Abort a Pipe Operation

Syntax

```
usbd_status usbman_abort_pipe(usbd_pipe_handle pipe);
```

Description

Aborts a pipe operation. This function returns USBD_NORMAL_COMPLETION if the operation is successful.

Parameters

pipe	A valid open pipe.

See Also

usbman_close_pipe()
usbman_open_pipe()

usbman_alloc_buffer()

Allocate a DMA Buffer

Syntax

```
void *usbman_alloc_buffer(
     usbd_xfer_handle  xfer,
     u_int32_t         size);
```

Description

Allocates a DMA buffer for the given transfer handle `xfer`. Returns NULL if the allocation fails; otherwise returns the pointer to the allocated memory.

Parameters

`xfer`	Must be a valid transfer handle; returned from `usbman_alloc_xfer()`.
`size`	Number of bytes to allocate.

Modifies

`xfer -> dmabuf`	Updated to store reference to allocated memory.
`xfer -> rqflags`	`URQ_DEV_DMABUF` flag set.

See Also

`usbman_free_buffer()`
`usbman_get_buffer()`

usbman_alloc_xfer()

Allocate a Transfer Structure

Syntax

```
usbd_xfer_handle usbman_alloc_xfer(usbd_device_handle);
```

Description

Allocates a `usbd_xfer` structure, and returns it to the calling function.

Parameters

`dev`	A valid `usbd_device_handle`.

See Also

`usbman_free_xfer()`

usbman_bulk_transfer()

Perform Bulk Transfer

Syntax

```
usbd_status usbman_bulk_transfer(
     usbd_xfer_handle  xfer,
     usbd_pipe_handle  pipe,
     u_int16_t         flags,
     u_int32_t         timeout,
     void              *buf,
     u_int32_t         *size,
     char              *lbl);
```

Description

Performs a bulk transfer to or from a device. This call will not return until the transfer is successful, or has timed out. This call returns `USBD_NORMAL_COMPLETION` if the transfer was successful; `USBD_INTERRUPTED` if the transfer was interrupted by a deadly IO signal; `USBD_IOERROR` if a transfer failed; and `USBD_TIMEOUT` if the transfer timed out.

Parameters

`xfer`	A transfer handle allocated with `usbman_alloc_xfer`.
`pipe`	An open pipe to the device.
`flags`	0 means no special flags. `USBD_NO_COPY`: do not copy data from `buf` to `DMA` buffer. `USBD_FORCE_SHORT_XFER`: force last short packet on write.
`timeout`	Number of milliseconds to wait for device to respond to transfer. `USBD_NO_TIMEOUT`: wait forever.

*buf	Write: contains data transfer to device. Read: valid memory location to store data read from device.
*size	Write: number of bytes to transfer to device Read: number of bytes to read from device (size of buf)
*lbl	Unused.

Modifies

This call modifies various fields in `xfer`.

usbman_clear_endpoint_stall()

Clear STALLED Condition

Syntax

```
usbd_status usbman_clear_endpoint_stall(
     usbd_pipe_handle  pipe);
```

Description

Clears the STALLED condition of the device. This will also reset the data toggle to 0. Clearing the endpoint stall is usually not necessary, unless a device has returned `USBD_STALLED` in response to a data transfer. This call returns `USBD_NORMAL_COMPLETION` if successful and `USBD_IOERROR` if the device did not respond.

Parameters

`pipe`	An open pipe to the device.

Modifies

This call sets the toggle state of the pipe to 0.

See Also

`usbman_clear_endpoint_stall_async()`
`usbman_clear_endpoint_toggle()`

usbman_clear_endpoint_stall_async()

Clear STALLED Condition

Syntax

```
usbd_status usbman_clear_endpoint_stall_async(
     usbd_pipe_handle  pipe);
```

Description

Clears the STALLED condition of the device and resets the data toggle to 0. This is identical to `usbman_clear_endpoint_stall`, except that this operation is not performed until some time later. This call will return immediately. This version of the call is useful if the endpoint stall needs to be cleared in interrupt context. This call returns `USBD_NORMAL_COMPLETION` if successful and `USBD_IOERROR` if the device did not respond.

Parameters

`pipe`	An open pipe to the device.

Modifies

This function sets the toggle state of the pipe to 0.

See Also

`usbman_clear_endpoint_stall()`
`usbman_clear_endpoint_toggle()`

usbman_clear_endpoint_toggle()

Reset Endpoint Toggle

Syntax

```
void usbman_clear_endpoint_toggle(
     usbd_pipe_handle  pipe);
```

Description

Resets the endpoint toggle to 0. Resetting the endpoint toggle is only necessary if resetting the device, or if clearing the endpoint stall.

Parameters

`pipe`	An open pipe to the device.

Modifies

This function sets the toggle state of the pipe to 0.

See Also

`usbman_clear_endpoint_stall()`
`usbman_clear_endpoint_stall_async()`

usbman_close_pipe()

Close Pipe

Syntax

```
usbd_status usbman_close_pipe(usbd_pipe_handle pipe);
```

Description

Closes pipe and frees interrupt pipe transfer buffer. This function returns `USBD_NORMAL_COMPLETION` if the operation is successful and `USBD_PENDING_REQUESTS` in the middle of the operation.

Parameters

`pipe`	A valid open pipe.

See Also

`usbman_abort_pipe()`
`usbman_open_pipe()`

usbman_device2interface_handle()

Return Interface Handle

Syntax

```
usbd_status usbman_device2interface_handle(
     usbd_device_handle     dev,
     u_int8_t               ifaceno,
     usbd_interface_handle  *iface);
```

Description

Returns the specified interface handle for the given device. This function returns `USBD_NORMAL_COMPLETION` if the operation is successful; `USBD_NOT_CONFIGURED` if there is no configuration descriptor for this device; and `USBD_INVAL` if the `ifaceno` parameter is out of range.

Parameters

`dev`	A valid device handle.
`ifaceno`	Interface number. This is between 0 and n-1, where n is the number of interfaces.
`*iface`	If successful, the interface handle will be stored in `*iface`.

usbman_do_request()

Perform Transfer Over Control Pipe

Syntax

```
usbd_status usbman_do_request(
     usbd_device_handle    pipe,
     usb_device_request_t  *req,
     void                  *data);
```

Description

Performs a transfer over the control pipe to the specified device. The data transferred is a fixed 8-byte structure defined by the USB specification. If any data is returned from the device, it is copied into the data parameter. The data parameter must be large enough to hold such information.

This function returns `USBD_NORMAL_COMPLETION` if successful; `USBD_NOMEM` if no memory is available; `USBD_IOERROR` when there is a transfer error to the device; and `USBD_STALLED` if the transfer caused the device to STALL.

Parameters

`pipe`	A valid device handle.
`*req`	8-byte request structure that is properly defined.
`*data`	NULL if no return data; otherwise pointer to return data memory.

usbman_do_request_flags()

Perform Transfer

Syntax

```
usbd_status usbman_do_request_flags(
     usbd_device_handle    pipe,
     usb_device_request_t  *req,
     void                  *data,
     u_int16_t             flags,
     int                   *actlen,
     u_int32_t             timeout);
```

Description

Performs the same function as `usbman_do_request` with the addition of three parameters: `flags`, `actlen`, and `timeout`.

This functions returns `USBD_NORMAL_COMPLETION` if successful; `USBD_NOMEM` if no memory is available; `USBD_IOERROR` when there is a transfer error to the device; and `USBD_STALLED` if the transfer caused the device to STALL.

Parameters

`pipe`	A valid device handle.
`*req`	8-byte request structure that is properly defined.
`*data`	NULL if no return data; otherwise pointer to return data memory.
`flags`	Flags normally passed to create a transfer handle: `USBD_NO_COPY`, `USBD_SHORT_XFER_OK`, `USBD_FORCE_SHORT_XFER`.
`*actlen`	Receives the number of bytes of data transferred from the device.

`timeout`	Specifies the time in which to perform the request before aborting the request. Passing 0 specifies no timeout should be used. The timeout value is in terms of milliseconds.

usbman_endpoint_count()

Return Number of Endpoints

Syntax

```
usbd_status usbman_endpoint_count(
     usbd_interface_handle  iface,
     u_int8_t               *count);
```

Description

Returns the number of endpoints in the current interface. Upon completion, this function returns `USBD_NORMAL_COMPLETION`.

Parameters

`iface`	A valid interface handle that contains a valid interface descriptor.
`*count`	Receives the number of endpoints in this interface.

See Also

`usbman_interface_count()`

usbman_find_edesc()

Return Endpoint Descriptor

Syntax

```
usb_endpoint_descriptor_t *usbman_find_edesc(
     usb_config_descriptor_t  *cd,
     int                      ifaceidx,
     int                      altidx,
     int                      endptidx);
```

Description

Returns the specified endpoint descriptor for the current configuration descriptor. Upon completion, this function returns a pointer to the requested endpoint descriptor, or NULL if not found.

Parameters

`*cd`	A valid configuration descriptor.
`ifaceidx`	Interface number in the configuration.
`altidx`	Alternate index in the configuration (0 if none).
`endptidx`	Endpoint index in the interface.

See Also

`usbman_find_idesc()`

usbman_find_idesc()

Return Interface Descriptor

Syntax

```
usb_interface_descriptor_t *usbman_find_idesc(
     usb_config_descriptor_t  *cd,
     int                      ifaceidx,
     int                      altidx);
```

Description

Returns the specified interface descriptor given a configuration descriptor.

Parameters

`*cd`	A valid configuration descriptor.
`ifaceidx`	Interface number in the configuration.
`altidx`	Alternate index in the configuration (0 if none).

See Also

`usbman_find_edesc()`

usbman_free_buffer()

Free DMA Buffer

Syntax

```
void usbman_free_buffer(usbd_xfer_handle xfer);
```

Description

Frees a DMA buffer for the given transfer handle. This should only be called if `usbman_alloc_buffer()` was successfully called on the given transfer handle. No return value.

Parameters

`xfer`	Must be a valid transfer handle, returned from `usbman_alloc_xfer()`.

Modifies

`xfer->dmabuf`	Deallocates memory.
`xfer->rqflags`	Clears the `URQ_DEV_DMABUF` and `URQ_AUTO_DMABUF` flags.

See Also

`usbman_alloc_buffer()`
`usbman_get_buffer()`

usbman_free_report_desc()

Deallocate Memory

Syntax

```
void usbman_free_report_desc(
     void  *descp,
     int   mem);
```

Description

Deallocates memory for a HID report descriptor. `descp` must be a value returned from `usbd_read_report_desc`.

Parameters

`*descp`	A report descriptor pointer to `free`.
`mem`	Unused.

See Also

`usbman_read_report_desc()`

usbman_free_xfer()

Free Transfer

Syntax

```
usbd_status usbman_free_xfer(usbd_xfer_handle xfer);
```

Description

Frees `xfer`. Also will free the `DMA` buffer if present.

Parameters

`xfer`	A valid `usbd_xfer_handle` structure that was allocated by `usbd_alloc_xfer()`.

See Also

`usbman_alloc_xfer()`

usbman_get_buffer()

Return Current DMA Buffer Pointer

Syntax

```
void *usbman_get_buffer(usbd_xfer_handle xfer);
```

Description

Returns the current DMA buffer pointer for the given transfer handle. If no DMA buffer has been allocated, NULL is returned.

Parameters

`xfer`	Must be a valid transfer handle, returned from `usbman_alloc_xfer()`.

See Also

`usbman_alloc_buffer()`
`usbman_free_buffer()`

usbman_get_config()

Request Configuration Descriptor

Syntax

```
usbd_status usbman_get_config(
     usbd_device_handle  dev,
     u_int8_t            *conf);
```

Description

Requests the configuration descriptor from the given device. This call will perform a transfer using the control pipe over the USB. This function returns USBD_NORMAL_COMPLETION if successful; USBD_NOMEM if no memory is available; USBD_IOERROR when there is a transfer error to the device; and USBD_STALLED if the transfer caused the device to STALL.

Parameters

dev	A valid USB device handle.
*conf	Pointer to at least 9 bytes, the size of the standard configuration descriptor.

See Also

usbman_get_config_desc()
usbman_get_config_desc_full()

usbman_get_config_desc()

Get Configuration Descriptor

Syntax

```
usbd_status usbman_get_config_desc(
     usbd_device_handle        dev,
     int                       confidx,
     usb_config_descriptor_t   *d);
```

Description

Get configuration descriptor from device handle (for example `dev->cdesc`).

Parameters

`dev`	A valid `usbd_device_handle`.
`confidx`	Configuration index.
`*d`	Address of storage for the basic configuration description.

See Also

`usbman_get_config()`
`usbman_get_config_desc_full()`

usbman_get_config_desc_full()

Request Configuration Descriptor

Syntax

```
usbd_status usbman_get_config_desc_full(
     usbd_device_handle  dev,
     int                 conf,
     void                *d,
     int                 size);
```

Description

Requests the configuration descriptor from the given device. The configuration index and the amount of data to receive is specified by the conf and size parameters. This function returns `USBD_NORMAL_COMPLETION` if successful; `USBD_NOMEM` if no memory is available; `USBD_IOERROR` when there is a transfer error to the device; and `USBD_STALLED` if the transfer caused the device to STALL.

Parameters

`dev`	A valid USB device handle.
`conf`	Specifies configuration index for descriptor.
`*d`	Pointer to at least 9 bytes, the size of the standard configuration descriptor.
`size`	Number of bytes in configuration to request.

See Also

`usbman_get_config()`
`usbman_get_config_desc()`

usbman_get_device_desc()

Request Device Descriptor

Syntax

```
usbd_status usbman_get_device_desc(
     usbd_device_handle        dev,
     usb_device_descriptor_t   *d);
```

Description

Requests the device descriptor from the given device. This function returns `USBD_NORMAL_COMPLETION` if successful; `USBD_NOMEM` if no memory is available; `USBD_IOERROR` when there is a transfer error to the device; and `USBD_STALLED` if the transfer caused the device to STALL.

Parameters

`dev`	A valid USB device handle.
`*d`	Pointer to 18 bytes, size of the standard device descriptor.

See Also

`usbman_get_device_descriptor()`

usbman_get_device_descriptor()

Return Device Descriptor

Syntax

```
usb_device_descriptor_t *usbman_get_device_descriptor(
     usbd_device_handle  dev)
```

Description

Returns the device descriptor retrieved after the device was initially explored.

Parameters

`dev`	A valid USB device handle.

See Also

`usbman_get_device_desc()`

usbman_get_hid_descriptor()

Request HID Descriptor

Syntax

```
usb_hid_descriptor_t *usbman_get_hid_descriptor(
     usbd_interface_handle  ifc);
```

Description

Requests the HID descriptor for the given interface handle. The HID descriptor is normally retrieved with the configuration descriptor. This function returns a pointer to the HID descriptor. If no HID descriptor is found, NULL is returned.

Parameters

`ifc`	A valid interface handle.

See Also

`usbman_get_report()`
`usbman_get_report_descriptor()`

usbman_get_no_alts()

Get Number of Alternate Interfaces

Syntax

```
int usbman_get_no_alts(
     usb_config_descriptor_t  *cdesc,
     int                      ifaceno);
```

Description

Get the number of alternate interfaces in the given configuration descriptor and interface number. Upon completion, this function returns the number of alternate interfaces.

Parameters

`*cdesc`	A valid configuration descriptor.
`ifaceno`	Interface number.

usbman_get_report()

Request HID Report Descriptor

Syntax

```
usbd_status usbman_get_report(
     usbd_interface_handle  iface,
     int                    type,
     int                    id,
     void                   *data,
     int                    len);
```

Description

Requests the HID report descriptor for the given interface. This will cause a data transfer on the USB. This function returns USBD_NORMAL_COMPLETION if successful; USBD_NOMEM if no memory is available; USBD_IOERROR when there is a transfer error to the device; and USBD_STALLED if the transfer caused the device to STALL.

Parameters

iface	A valid interface handle.
type	UHID_INPUT_REPORT, UHID_OUTPUT_REPORT, UHID_FEATURE_REPORT.
id	HID id.
*data	Pointer to memory where report will be stored.
len	Number of bytes of data to retrieve of HID descriptor.

See Also

usbman_get_hid_descriptor()
usbman_get_report_descriptor()

usbman_get_report_descriptor()

Request HID Report Descriptor

Syntax

```
usbd_status usbman_get_report_descriptor(
     usbd_device_handle  dev,
     int                 ifcno,
     int                 size,
     void                *d);
```

Description

Requests a HID report descriptor for the given device. This will cause a data transfer on the USB. This function returns USBD_NORMAL_COMPLETION if successful; USBD_NOMEM if no memory is available; USBD_IOERROR when there is a transfer error to the device; and USBD_STALLED if the transfer caused the device to STALL.

Parameters

dev	A valid USB device.
ifcno	Interface number.
size	Number of bytes to request.
*d	Pointer to memory to store requested report descriptor.

See Also

usbman_get_hid_descriptor()
usbman_get_report()

usbman_get_string_desc()

Request String Descriptor

Syntax

```
usbd_status usbman_get_string_desc(
     usbd_device_handle       dev,
     int                      sindex,
     int                      langid,
     usb_string_descriptor_t  *sdesc,
     int                      *size);
```

Description

Requests the string descriptor for the given device. This will cause a data transfer on the USB. Upon successful completion, the string descriptor will be stored in `sdesc`. This function returns `USBD_NORMAL_COMPLETION` if successful; `USBD_NOMEM` if no memory is available; `USBD_IOERROR` when there is a transfer error to the device; and `USBD_STALLED` if the transfer caused the device to STALL.

Parameters

`dev`	A valid device handle.
`sindex`	String index.
`langid`	Language ID.
`*sdesc`	Pointer to string descriptor structure.
`*size`	Receives the actual length of the string descriptor.

usbman_get_xfer_status()

Get Transfer Status

Syntax

```
void usbman_get_xfer_status(
     usbd_xfer_handle      xfer,
     usbd_private_handle   *priv,
     void                  **buffer,
     u_int32_t             *count,
     usbd_status           *status);
```

Description

Returns information regarding the given `xfer` transfer handle.

Parameters

`xfer`	A valid `xfer` handle.
`*priv`	Receiving the private data area for the transfer.
`**buffer`	Receiving the DMA buffer.
`*count`	Receiving the total number of bytes transferred.
`*status`	Returns the transfer status.

usbman_interface_count()

Return Number of Interfaces

Syntax

```
usbd_status usbman_interface_count(
     usbd_device_handle  dev,
     u_int8_t            *count);
```

Description

Returns the number of interfaces for the current configuration.

Parameters

`dev`	A valid device.
`*count`	Receiving the number of interfaces.

usbman_interface2device_handle()

Return Device Handle

Syntax

```
void usbman_interface2device_handle(
     usbd_interface_handle  iface,
     usbd_device_handle     *dev);
```

Description

Returns the device handle for a given interface handle. An interface cannot exist without an associated device handle.

Parameters

`iface`	A valid interface handle.
`*dev`	Receives the device handle associated with `iface`.

usbman_interface2endpoint_descriptor()

Return Endpoint Descriptor

Syntax

```
usb_endpoint_descriptor_t
*usbman_interface2endpoint_descriptor(
     usbd_interface_handle  iface,
     u_int8_t               address);
```

Description

Returns the endpoint descriptor given an interface handle. Upon completion, this function returns a pointer to an endpoint descriptor, or NULL if the index is out of range.

Parameters

`iface`	A valid interface handle.
`address`	Endpoint number.

usbman_open_pipe()

Create Bulk Transfer Pipe

Syntax

```
usbd_status usbman_open_pipe(
     usbd_interface_handle  iface,
     u_int8_t               address,
     u_int8_t               flags,
     usbd_pipe_handle       *pipe);
```

Description

Creates a bulk transfer pipe to the given endpoint. The address (endpoint) will be checked to see if it is valid. This function returns `USBD_NORMAL_COMPLETION` if the call is successful; `USBD_BAD_ADDRESS` if the endpoint is invalid; `USBD_IN_USE` if the pipe is already opened to endpoint, but the caller wanted an exclusive connection.

Parameters

`iface`	
`address`	Endpoint on USB bus.
`flags`	Passing `USBD_EXCLUSIVE_USE` will open the pipe exclusively for the caller.
`pipe`	A new pipe will be created and returned in this parameter.

See Also

`usbman_open_pipe_intr()`

usbman_open_pipe_intr()

Create Interrupt Pipe

Syntax

```
usbd_status usbman_open_pipe_intr(
     usbd_interface_handle  iface,
     u_int8_t               address,
     u_int8_t               flags,
     usbd_pipe_handle       *pipe,
     usbd_private_handle    priv,
     void                   *buffer,
     u_int32_t              length,
     usbd_callback          cb
     int                    interval);
```

Description

Creates an interrupt pipe to the given endpoint.

Parameters

`iface`	A valid interface.
`address`	Endpoint on USB bus.
`flags`	`USBD_EXCLUSIVE_USE`: open exclusive pipe.
`*pipe`	New pipe will be returned in this parameter.
`priv`	Parameter passed to interrupt service routine.
`*buffer`	Data buffer. Must be big enough according to class definition of device.
`length`	Bytes in data buffer.
`cb`	Interrupt service routine—called when data transfers, or transmission error.
`int`	Polling interval.

See Also

`usbman_open_pipe()`

usbman_pipe2device_handle()

Return Device Handle

Syntax

```
usbd_device_handle usbman_pipe2device_handle(
     usbd_pipe_handle  pipe);
```

Description

Returns the device handle associated with the given pipe. Upon completion, this function returns the device handle associated with this pipe.

Parameters

`pipe`	A valid pipe handle, created by `usbman_open_pipe`.

usbman_read_report_desc()

Allocate and read the report descriptor

Syntax

```
usbd_status usbman_read_report_desc(
     usbd_interface_handle  ifc,
     void                   **descp,
     int                    *sizep,
     usb_malloc_type        mem);
```

Description

Calculates the size of the descriptor and allocates memory for it. Returns the Report Descriptor for a HID device (for example a mouse or keyboard).

Parameters

`ifc`	A valid interface handle.
`**descp`	Memory for the report descriptor is allocated and stored in `*descp`.
`*sizep`	The size of the memory allocated is stored in `*sizep`.
`mem`	Ignored.

See Also

`usbman_free_report_desc()`

usbman_set_config_index()

Set Configuration Index

Syntax

```
usbd_status usbman_set_config_index(
     usbd_device_handle  dev,
     int                 index,
     int                 msg);
```

Description

Sets the configuration index for the given device. This will perform transfers over the USB. This call assumes that no interrupt, bulk, or isochronous pipes are open on `dev`.

This function returns `USBD_NORMAL_COMPLETION` if successful; `USBD_NOMEM` if no memory is available; `USBD_IOERROR` when there is a transfer error to the device; `USBD_STALLED` if the transfer caused the device to STALL; `USBD_INVAL` when a bad configuration descriptor is retrieved from the device; and `USBD_NO_POWER` when the device exceeds available power on the hub.

Parameters

`dev`	A valid device handle.
`index`	Configuration index to set.
`msg`	Unused.

See Also

`usbman_set_config_no()`

usbman_set_config_no()

Set Configuration

Syntax

```
usbd_status usbman_set_config_no(
     usbd_device_handle  dev,
     int                 no,
     int                 msg);
```

Description

Sets the configuration for the given device specified by `config_no`. This will perform transfers over the USB. This call assumes that no interrupt, bulk, or isochronous pipes are open on `dev`.

This function returns `USBD_NORMAL_COMPLETION` if successful; `USBD_NOMEM` if no memory is available; `USBD_IOERROR` when there is a transfer error to the device; `USBD_STALLED` if the transfer caused the device to STALL; `USBD_INVAL` when a bad configuration descriptor is retrieved from the device; and `USBD_NO_POWER` when the device exceeds available power on the hub.

Parameters

`dev`	A valid device handle.
`no`	Configuration index to set.
`msg`	Unused.

See Also

`usbman_set_config_index()`

usbman_set_idle()

Silence Report on the Interrupt In Pipe

Syntax

```
usbd_status usbman_set_idle(
     usbd_interface_handle  iface,
     int                    duration,
     int                    id);
```

Description

Silences a particular report on the interrupt In Pipe until a new event occurs or until the specified time passes. Valid for an HID device only.

Parameters

`iface`	A valid interface handle.
`duration`	Duration of the file.
`id`	Identification for `idle`.

See Also

`usbman_set_protocol()`

For More Information

For more information refer to the USB HID 1.1 Specification.

usbman_set_interface()

Request Interface Change

Syntax

```
usbd_status usbman_set_interface(
     usbd_interface_handle  iface,
     int                    altidx);
```

Description

Requests an interface change specified by `iface->index`. This will perform transfers on the USB. This function returns `USBD_NORMAL_COMPLETION` if successful; `USBD_NOMEM` if no memory is available; `USBD_IOERROR` when there is a transfer error to the device; and `USBD_STALLED` if the transfer caused the device to STALL.

Parameters

`iface`	A valid interface handle.
`altidx`	Alternate interface handle, 0 if none.

usbman_set_protocol()

Switch Between Boot and Report Protocol

Syntax

```
usbd_status usbman_set_protocol(
     usbd_interface_handle  iface,
     int                    report);
```

Description

Switches between the boot protocol and report protocol for an HID device.

Parameters

`iface`	Valid interface.
`report`	0: boot protocol.
	1: report protocol.

See Also

`usbman_set_idle()`

For More Information

For more information refer to the USB HID 1.1 Specification.

usbman_set_report()

Perform Set Report Request

Syntax

```
usbd_status usbman_set_report(
     usbd_interface_handle  iface,
     int                    type,
     int                    id,
     void                   *data,
     int                    len);
```

Description

Performs a set report request to the given interface. This function returns `USBD_NORMAL_COMPLETION` if successful; `USBD_NOMEM` if no memory is available; `USBD_IOERROR` when there is a transfer error to the device; and `USBD_STALLED` if the transfer caused the device to STALL.

Parameters

`iface`	A valid interface handle.
`type`	`UHID_INPUT_REPORT`, `UHID_OUTPUT_REPORT`, `UHID_FEATURE_REPORT`.
`id`	Report value id.
`*data`	Pointer to memory for request data.
`len`	Length of `data`.

See Also

`usbman_set_idle()`
`usbman_set_protocol()`

For More Information

For more information refer to the USB HID 1.1 Specification.

usbman_setup_default_xfer()

Initialize Transfer Handle

Syntax

```
void usbman_setup_default_xfer(
     usbd_xfer_handle        xfer,
     usbd_device_handle      dev,
     usbd_private_handle     priv,
     u_int32_t               timeout,
     usb_device_request_t    *req,
     void                    *buffer,
     u_int32_t               length,
     u_int16_t               flags,
     usbd_callback           cb);
```

Description

Initializes a transfer handle `xfer` with given parameter values. Upon completion, this function returns nothing.

Parameters

`xfer`	A valid transfer handle returned from `usbman_alloc_xfer`.
`dev`	A valid USB device associated with the transfer.
`priv`	Parameter passed to interrupt service routine.
`timeout`	Milli-seconds to wait before timing out, or `USBD_NO_TIMEOUT`.
`*req`	Device request if using control pipe, otherwise NULL.
`*buffer`	Memory to hold transfer.
`length`	Bytes in `buffer`.

flags	USBD_NO_COPY, USBD_SYNCHRONOUS, USBD_SHORT_XFER_OK, or USBD_FORCE_SHORT_XFER.
cb	Function to be called when transfer has completed.

See Also

usbman_setup_isoc_xfer()
usbman_setup_xfer()

usbman_setup_isoc_xfer()

Initialize ISOC Transfer

Syntax

```
void usbman_setup_isoc_xfer(
      usbd_xfer_handle     xfer,
      usbd_pipe_handle     pipe,
      usbd_private_handle  priv,
      u_int16_t            *frlengths,
      u_int32_t            nframes,
      u_int16_t            flags,
      usbd_callback        cb);
```

Description

Initializes a transfer handle `xfer` with given parameter values. Upon completion, this function returns nothing.

Parameters

`xfer`	A valid transfer handle.
`pipe`	A valid open pipe.
`priv`	Parameter passed to interrupt service routine.
`*frlengths`	Array of frame lengths.
`nframes`	Number of frames (elements in `frlengths`).
`flags`	`USBD_NO_COPY`, `USBD_SYNCHRONOUS`, `USBD_SHORT_XFER_OK`, or `USBD_FORCE_SHORT_XFER`.
`cb`	Function to be called when transfer has completed.

See Also

`usbman_setup_default_xfer()`

usbman_setup_xfer()

usbman_setup_xfer()

Assign Fields in Transfer

Syntax

```
void usbman_setup_xfer(
     usbd_xfer_handle     xfer,
     usbd_pipe_handle     pipe,
     usbd_private_handle  priv,
     void                 *buffer,
     u_int32_t            length,
     u_int16_t            flags,
     u_int32_t            timeout,
     usbd_callback        cb);
```

Description

Initializes a transfer handle `xfer` with given parameter values. Upon completion, this function returns nothing.

Parameters

`xfer`	A valid transfer handle.
`pipe`	A valid open pipe.
`priv`	Parameter passed to interrupt service routine.
`*buffer`	Receiving the DMA buffer.
`length`	Bytes in data buffer.
`flags`	`USBD_NO_COPY`, `USBD_SYNCHRONOUS`, `USBD_SHORT_XFER_OK`, or `USBD_FORCE_SHORT_XFER`.
`timeout`	Number of milliseconds to wait for device to respond to transfer.
`cb`	Function to be called when transfer has completed.

See Also

usbman_setup_default_xfer()
usbman_setup_isoc_xfer()

usbman_sync_transfer()

Perform Asynchronous Transfer

Syntax

```
usbd_status usbman_sync_transfer(
     usbd_xfer_handle  req);
```

Description

Performs a synchronous transfer on the USB. The transfer handle `req` specifies direction, data, timeout, and transfer type. This call will not return until the transfer has completed successfully, timed out, or a USB error occurs.

This function returns `USBD_NORMAL_COMPLETION` if successful; `USBD_NOMEM` if no memory is available; `USBD_IOERROR` when there is a transfer error to the device; `USBD_STALLED` if the transfer caused the device to STALL; and `USBD_TIMEOUT` when no transfer occurred because the time interval expired.

Parameters

`req`	A valid transfer handle.

See Also

`usbman_bulk_transfer()`
`usbman_transfer()`

usbman_transfer()

Initialize Bulk Transfer

Syntax

```
usbd_status usbman_transfer(usbd_xfer_handle req);
```

Description

Initiates a bulk data transfer, either incoming or outgoing. This function returns `USBD_NORMAL_COMPLETION` if the operation is successful; `USBD_NOMEM` if there is no memory to allocate `DMA` buffer; and `USBD_TIMEOUT` if the operation timed out.

Parameters

`req`	A valid usbd_xfer structure as allocated by `usbd_alloc_xfer()`.

Chapter 4: USB Host for OS-9 Utilities

This chapter provides a description of the USB Host for OS-9 utilities. **Table 4-1** summarizes the USB utilities.

Table 4-1 USB Host for OS-9 Utilities

Name	Description
`usbdevs`	Print Current Devices on the USB
`usbprint`	Print Source File
`ugenstat`	Display Descriptors for Given UGEN Descriptor

Syntax

```
usbdevs [options]
```

Source

```
SRC/IO/USBH/UTILS/USBDEVS
```

Options

`-a[=]<addr>`	Display device address `<addr>` information.
`-b[=]<bus>`	Specify the bus on which to access device at `<addr>`. The default bus number is 0.
`-e`	Display extended information.

Description

This utility prints out the current devices on the USB. This information includes the device descriptor, configuration descriptor, interface descriptor, and any string descriptors. The `-a` and `-b` options can be used to select a particular device by USB address/bus and display extended information for that device.

Example

The following example shows three root hubs, two with two ports and one with four ports. A USB v1.1 device is plugged into a USB v1.1 3-port hub. A USB v2.0 Flash disk is plugged into a port on the high-speed bus #2.

```
$ usbdevs
Bus #0, Root Hub, Address 1,
[1] <empty>
[2] Address 2, Hub (vendor 1228, product 4386)
    [1] Address 3,  Fellowes Inc.: Fellowes 5 Button
    [2] <empty>
    [3] <empty>

Bus #1, Root Hub, Address 1,
[1] <empty>
```

```
[2] <empty>

Bus #2, Root Hub, Address 1,
[1] <empty>
[2] Address 2,  SanDisk Corporation: U3 Cruzer Micro: 0000051015079136
[3] <empty>
[4] <empty>
$ usbdevs -a=2 -b=2
Address 2,  SanDisk Corporation: U3 Cruzer Micro: 0000051015079136 (vendor 1921,
 product 21506)
    Device Descriptor:  max_packet 64, protocol 0, release 0.2, configurations 1
    Config. Descriptor (1):  interfaces 1, value 1, iconfig 0
        attributes 0x80, max power 200 mA
    Interface Descriptor 1:  alt. setting 0, num eps 2,
        class 8, subclass 6, protocol 80, iInterface 0
```

Syntax

```
usbprint [options] <source-file> [<printer-device>]
```

Source

SRC/IO/USBH/UTILS/USBPRINT

Options

-m	Search for source file in module directory.

Description

This utility prints the source file to the specified printer device. If no printer device is specified, it will default to /ulp0.

Example

- Printing using the standard USB printer driver.

```
$ usbprint sample.txt /ulp0
```

- Printing using the Generic USB driver.

```
$ usbprint sample.txt "/ugen0#2"
```

ugenstat

Display Descriptors for Given UGEN Descriptor

Syntax

```
ugenstat [device]
```

Source

```
SRC/IO/USBH/UTILS/UGENSTAT
```

Description

This utility displays the device, configuration, interface, endpoint, and string descriptors for the given UGEN device descriptor. If no descriptor is specified, the default will be /ugen0.

Example

The following example shows a mouse attached to `/ugen0`.

```
$ ugenstat /ugen0
Device Descriptor:  12010001 00000008 03067168 00010422 0001
NOVATEK         USB Mouse STD.
Number of Configurations:  1
Config Descriptor 1:  09022200 010100a0 32

    Number of interfaces:  1
    Interface Descriptor 0:  09040000 01030102 00

      Number of endpoints:  1
      Endpoint Descriptor 0:  07058103 08000a
```

For More Information

The data format printed for the descriptors is defined in the USB 1.1 documentation. This can be found at `www.usb.org`.

Appendix A: Porting to the USB Host Stack

This chapter details how to port to the USB Host stack. The following sections are included:

- **Writing the Logical Device Driver (LDD)**
- **Writing a Hardware Control Driver**

RadiSys.
MICROWARE SOFTWARE

Writing the Logical Device Driver (LDD)

This section will describe how to make a new Logical Device Driver for the USB Host Stack. Any file manager may be used for an LDD, but in this chapter, the driver will be under the NullFM File Manager.

Before you begin, you will need to decide the following information:

- the directory name for the LDD
- the driver name
- the descriptor name

The makefile and all of the source code files for the LDD will reside in the following directory:
`/mwos/SRC/DPIO/NULLFM/DRVR/USBH/<YOUR_LDD_DIRECTORY_NAME>`

Both the driver and descriptor modules will be located in the following directory:

`/mwos/OS9000/<PROCESSOR>/CMDS/BOOTOBJS/USBH`

Creating a Directory Structure

The first step in writing an LDD is to create a directory structure for your NullFM driver. This will be the directory in which you will copy and modify files from the sample driver directory (`SAMPLE_LDD`). Follow the procedure below to create this structure and associated files for your new LDD.

Step 1. Create a new folder in the `/mwos/SRC/DPIO/NULLFM/DRVR/USBH` directory. This folder will contain the source files and makefiles for your NullFM driver.

Step 2. Create a `DEFS` directory within the folder you just created. This directory will contain all header files specific to this driver and descriptor.

Step 3. Copy the following files from the `SAMPLE_LDD` directory (sample driver) into your driver directory:

- `drvr.mak`
- `init.c`
- `makefile`
- `rw.c`
- `desc.mak`
- `hw.c`
- `main.c`
- `os9_dev.c`
- `stat.c`

Step 4. Copy the following files from the `SAMPLE_LDD/DEFS` directory into the `DEFS` directory of your driver:

- `defsfile.h`
- `desc.h`
- `funcs.h`
- `usbh_desc.h`

Implementing your LDD

Below is a step-by-step guide of which code to modify in each file copied from the SAMPLE_LDD directory. This step-by-step guide details an example scenario using a camera driver and descriptor. (ucamera is the driver name, and ucamera0 is the descriptor name.)

Step 1. Modify the drvr.mak file to change the driver name and directory. To do this, change the TRGTS and DRVNAME macros to the name of your LDD driver. Then, change the LOCDRV macro to the source directory name of your LDD. Below is an example that shows the driver name as ucamera and the directory as UCAMERA.

```
TRGTS= ucamera

LOCDRV= USBH/UCAMERA

DRVNAME= ucamera
```

Step 2. Modify the descriptor name in the desc.mak file. To do this you will need to change the TRGTS macro. Below is an example that shows a descriptor name of ucamera0.

```
TRGTS= ucamera0
```

Note

The descriptor name and driver name must be different.

Step 3. Modify the `desc.h` file located in the `DEFS` directory. This file contains the basic descriptor information for your LDD NullFM driver. You will need to change the `DRIVERNAME` definition and the descriptor name pre-processing conditional. Below is an example:

```
#if defined(ucamera0)


#define DRIVERNAME        "ucamera"

#define FILEMANAGERNAME   "nullfm"

#define VECTOR 0

#define IRQLEVEL 5

#define PRIORITY 20

#define PORTADDR(void*)0x0

#define DEVICE_MODE       FAM_READ|FAM_WRITE


#endif
```

Step 4. Modify the `os9_dev.c` file to incorporate device specific changes to the `os9_match`, `os9_detach`, `os9_attach`, and the `os9_intr` routines.

For More Information

For more information on these routines, refer to the **Logical Device Drivers** section of Chapter 2 of this manual.

Step 5. If your driver should respond to either a read or write on an open path, modify the `rw.c` file. In addition, the `data_available` function should be modified to return the number of bytes available for read.

Additional File Information

Below is a list of files that may not require direct modification.

`makefile`	main makefile that builds the driver and descriptor
`init.c`	implements driver initialization and termination routines
`hw.c`	called during initialization and termination to open a path and register with `usbman`
`main.c`	main psect for this driver
`stat.c`	contains setstat and getstat routines for this driver.
`DEFS/defsfile.h`	main include file to include other header files
`DEFS/funcs.h`	contains all global function/type definitions for the driver
`DEFS/usbh_desc.h`	file that allows you to extend the driver static storage definition

Writing a Hardware Control Driver

This section describes the steps necessary to write a new hardware control driver for the USB Host stack for OS-9.

Note

Before reading this chapter, be certain you have perused **Chapter 2: Using USB Host for OS-9** of this manual.

Overview

A USB hardware driver is responsible for initializing the USB hardware, scheduling transfers, and servicing interrupts. The USB manager, `usbman`, is responsible for scheduling all transfers for the hardware controller driver. It is the responsibility of the hardware driver to perform these transfers and provide notification when the transfers are complete.

Transfer Types

The hardware controller driver must implement following six types of transfer:

- root hub control
- root hub interrupt
- device control
- device interrupt
- device bulk
- device isochronous transfers

Each transfer type has a function block associated with it. This function block allows usbman to call directly into the hardware control driver to start transfers, close a pipe, abort a pipe, and other such operations. Below is the definition of the transfer function block located in usbdivar.h:

```
struct usbd_pipe_methods {

usbd_status      (*transfer)(usbd_xfer_handle xfer);

usbd_status      (*start)(usbd_xfer_handle xfer);

void             (*abort)(usbd_xfer_handle xfer);

void             (*close)(usbd_pipe_handle pipe);

void             (*cleartoggle)(usbd_pipe_handle pipe);

void             (*done)(usbd_xfer_handle xfer);

};
```

Bus Methods Structure

The hardware control driver must also implement a bus methods structure; this is another way that usbman can call directly into the hardware control driver. This structure contains functions for opening a pipe, allocating and freeing memory, and allocating and freeing DMA memory.

Below is the structure definition located in usbdivar.h.

```
struct usbd_bus_methods {
usbd_status          (*open_pipe)(struct usbd_pipe *pipe);
void                 (*soft_intr)(struct usbd_bus *);
void                 (*do_poll)(struct usbd_bus *);
usbd_status           (*allocm)(struct usbd_bus *, usb_dma_t *,
u_int32_t bufsize);
void                  (*freem)(struct usbd_bus *, usb_dma_t *);
struct usbd_xfer *   (*allocx)(struct usbd_bus *);
void                 (*freex)(struct usbd_bus *,
                      struct usbd_xfer *);
};
```

The bus methods function block is returned by the hardware control driver in response to a GS_USB_BUS_METHODS getstat. Usbman performs this getstat while initializing the USB stack.

Calling usbman

The hardware controller driver may also call into usbman on two occasions: to insert a transfer into the list and to notify usbman when a transfer was completed. These methods are given to the hardware control driver from usbman by the SS_USB_MAN_METHODS setstat. This means the hardware control driver must acknowledge this setstat and store the methods and global pointer for usbman.

Existing Drivers

Because a sample driver does not currently exist, you must start from one of the three existing hardware controller drivers: EHCI, OHCI, PHCI, SL811HST, or UHCI. Below is a brief description of each driver.

EHCI	commonly used on desktop computers (`www.usb.org/developers/docs.html`) This driver creates a series of schedules for the hardware to act upon. The EHCI controller generates an interrupt as the various tasks are completed. This driver also relies on the fact that there will be low and/or full-speed driver support (companion OHCI or UHCI controllers). Like OHCI, this driver requires some type of shared memory betweent the processor and controller.
OHCI	commonly used on desktop computers (`www.usb.org/developers/docs.html`) This driver stores an elaborate list of items to transfer and only generates an interrupt after a successful transfer on the USB. The OHCI controller walks the transfer list and schedules USB time in hardware. In addition, this driver requires some type of shared memory between the processor and the controller.
PHCI	driver for the Philips ISP1161/2 embedded USB Host chip This hardware is more CPU intensive than the OHCI driver. Software must schedule transfers every millisecond, but more than one transfer may be scheduled. At the end of the frame (one millisecond), the software

	must determine which transfers were successful and schedule more transfers on the USB for the next frame.
SL811HS	driver for the ScanLogic 811HS USB Host chip This is the most CPU intensive hardware because the hardware driver must schedule every transfer on the USB. This results in many interrupts per frame (millisecond). SL811HS does not have an integrated root hub. Instead, the driver is notified of a voltage change on the bus, where it must then determine if something was inserted or removed from the root hub.
UHCI	commonly used on Intel-based desktop computers (http://www.intel.com/design/USB/UHCI11D.htm) This driver stores a simple list of items to transfer and generates an interrupt after a successful transfer on the USB. The UHCI controller walks the transfer list that the driver schedules for the USB. In addition, this driver requires some type of shared memory between the processor and the controller.

Implementing the Driver

To implement the driver, complete the following steps:

Step 1. Make a new directory in /mwos/SRC/DPIO/NULLFM/DRVR/USBH and a DEFS subdirectory and copy files from one of the existing drivers.

Step 2. Create a new directory and DEFS subdirectory in the board port to contain the makefiles and board definitions for this driver (/MWOS/OS9000/<PROCESSOR>/PORTS/<BOARD>/NULLFM/YOUR_DRIVER_NAME). Copy port files from an existing USB Host driver into this directory. These makefiles will require some modification in order to redefine any source or include paths.

Step 3. If your driver uses DMA, you will need to define the following symbol: USE_NONCACHED_MEM. This will include code in usb_mem.c to perform memory allocation for DMA memory. The malloc_dma function defined in this file performs an allocation out of a non-cached memory shade. This function will also ensure that the memory allocated is on the proper alignment boundary.

When using the USB_NONCACHED_MEM define, DMA memory allocations occur out of the M_USB_DMA memory shade. To reduce memory fragmentation, the MAUI memory APIs are used. Thus, a MAUI memory shade for M_USB_DMA must be created before using the malloc_dma function. (Refer to init.c in the OHCI, UHCI, or EHCI driver)

Step 4. Update the desc.h file located in /MWOS/OS9000/<PROCESSOR>/PORTS/<BOARD>/NULLFM/<YOUR_DRIVER>/DEFS. In particular, the VECTOR, IRQLEVEL, PRIORITY, and PORTADDR must be updated to reflect the proper values for the board.

Step 5. Update the USB hardware specific file in the driver. This file contains the hardware initialization, termination, interrupt service routine, and usbman entrypoints. Development of this driver will take time, but can be achieved if tested. The section below contains more information on testing the USB Host driver.

Testing the Driver

Testing a USB Host driver occurs in several phases starting with the most basic test: initializing and de-initializing the driver. Below is a sample command used to initialize your driver on your OS-9 target. You will need your driver, descriptor, and the NullFM File Manager on your OS-9 target.

```
$iniz /usbhc
```

After the above command is issued, the `init` entrypoint in the NullFM driver will be called. When this is complete, perform the following steps:

Step 1. Set a breakpoint on this function and step through the code to see if hardware initialization occurred properly.

Step 2. Turn on the start-of-frame (SOF) interrupt in the initialization code for your driver. SOF interrupts occur every 1 milli-second; you will know if one has occurred by setting a breakpoint on your interrupt service routine.

Step 3. Test termination of the driver by typing the following command:

```
$ deiniz /usbhc
```

After this command is issued, the `term` entrypoint is called. It is important to make sure that the hardware is turned off properly and that interrupts have been masked and memory deallocated. Repeated `iniz` and `deiniz` commands can be used to test memory leakage by using the `mfree` command.

Step 4. Determine if a root hub interrupt is being raised. To do this, set a breakpoint in the part of your interrupt service routine that handles the root hub interrupt.

Step 5. Iniz your driver and plug in a device like a hub or mouse into the USB port. The root hub interrupt should fire when the device is plugged in. If your hardware does not have an integrated root hub into the chip, refer to the SL811HS driver.

Step 6. Iniz `usbman`. Below is a sample of how to do this. You will need the driver, descriptor, the NullFM file manager, `usbman` driver, and `usbman` descriptor on your OS-9 target.

```
$ iniz /usb
```

Iniz-ing `/usb` will cause `usbman` to initialize and iniz your hardware driver. At this point, there will be an exchange of information between `usbman` and your driver via getstat/setstats. If this swap of information is successful, your driver and `usbman` have exchanged entrypoints.

Step 7. Start the `usbd` daemon. This opens a path to `usbman` and perform an explore on USB. Using the `-v` option will print out each occurrence of a USB explore. The `usbd` program performs an explore whenever a root hub interrupt occurs.

Note
It is important to plug in and out a device multiple times to ensure that the root hub interrupt is working properly.

The `-v` option command is shown below:

```
$ usbd -v
```

At this point, the `usbdevs` program can be used to print out information about devices on the USB.

Note
As soon as a device is plugged into the USB, an explore should occur.

When the explore is successful, the `usbdevs` program prints out the configuration information for the device. It is helpful to leave `usbd -v` running in the foreground on the console and use the `usbdevs` program on a second serial port (or telnet window).

You should be able to run `usbdevs` after plugging in or removing a device on the USB. `usbdevs` will display current topology. If it does not, you have a USB transfer problem.

Step 8. As a final test, perform the tests in **Chapter 1: Getting Started with USB Host for OS-9®** of this manual once more This will ensure that control and interrupt pipes are working properly. If you require a device with bulk or isochronous endpoints, you will need to write a separate application to perform the tests relating to those endpoints.