# Device Publishers Developer's Guide

QNX Software Systems Limited
1001 Farrar Road
Ottawa, Ontario
K2K 0B3
Canada

Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: info@qnx.com
Web: http://www.qnx.com/



**Electronic edition published:** Wednesday, June 25, 2014

# Table of Contents

**About This Guide ..... 5**
Typographical conventions ..... 6
Technical support ..... 8

**Chapter 1: Device Publishers ..... 9**
Running a device publisher ..... 10
Role of device drivers and mcd ..... 11
PPS object types ..... 13
Plugins ..... 14
Plugin ratings ..... 15

**Chapter 2: The usblauncher Service ..... 17**
Support for USB On-The-Go (OTG) ..... 21
Device object ..... 22
Device control object ..... 26
Driver object ..... 28
Mount object ..... 29
Command line for usblauncher ..... 31
Using usblauncher to mount filesystems ..... 34
Configuration files ..... 36
Starting USB stack in host and device modes ..... 36
USB matching rules ..... 38
USB descriptors ..... 43
Supported third-party applications and protocols ..... 49

**Chapter 3: The mmcsdpub Publisher ..... 55**
Device object ..... 57
Driver object ..... 58
Mount object ..... 59
Command line for mmcsdpub ..... 61

**Chapter 4: The cdpub Publisher ..... 65**
Device object ..... 67
Device control object ..... 69
Driver object ..... 70
Mount object ..... 71
Command line for cdpub ..... 72

# About This Guide

The *Device Publishers Developer's Guide* is aimed at developers who write applications that read device information through the Persistent Publish/Subscribe (PPS) service. This guide describes the contents of all PPS objects created and used by device publishers and also lists the command options you can set when restarting the publishers.

This table may help you find what you need in this guide:

| To find out about: | Go to: |
|---|---|
| Device publisher responsibilities and the publishers that we ship | *Device Publishers* (p. 9) |
| Setting up PPS before running device publishers | *Running a device publisher* (p. 10) |
| The types of PPS objects written by the publishers and which directories store these objects | *PPS object types* (p. 13) |
| The `usblauncher` service, which enumerates USB devices, mounts their filesystems, and publishes their information | *The usblauncher Service* (p. 17) |
| The `mmcsdpub` publisher, which publishes information about cards inserted into SD slots | *The mmcsdpub Publisher* (p. 55) |
| The `cdpub` publisher, which publishes information about CD devices | *The cdpub Publisher* (p. 65) |

## Typographical conventions

Throughout this manual, we use certain typographical conventions to distinguish technical terms. In general, the conventions we use conform to those found in IEEE POSIX publications.

The following table summarizes our conventions:

| Reference | Example |
|---|---|
| Code examples | `if( stream == NULL)` |
| Command options | `-lR` |
| Commands | `make` |
| Constants | `NULL` |
| Data types | `unsigned short` |
| Environment variables | ***PATH*** |
| File and pathnames | `/dev/null` |
| Function names | *exit()* |
| Keyboard chords | **Ctrl–Alt–Delete** |
| Keyboard input | `Username` |
| Keyboard keys | **Enter** |
| Program output | `login:` |
| Variable names | *stdin* |
| Parameters | *parm1* |
| User-interface components | **Navigator** |
| Window title | **Options** |

We use an arrow in directions for accessing menu items, like this:

You'll find the Other... menu item under **Perspective** **Show View**.

We use notes, cautions, and warnings to highlight important messages:

Notes point out something important or useful.

Cautions tell you about commands or procedures that may have unwanted or undesirable side effects.

Warnings tell you about commands or procedures that could be dangerous to your files, your hardware, or even yourself.

**Note to Windows users**

In our documentation, we typically use a forward slash (/) as a delimiter in pathnames, including those pointing to Windows files. We also generally follow POSIX/UNIX filesystem conventions.

## Technical support

Technical assistance is available for all supported products.

To obtain technical support for any QNX product, visit the Support area on our website (*www.qnx.com*). You'll find a wide range of support options, including community forums.

# Chapter 1
# Device Publishers

Device publishers are hardware-support components that publish information on attached devices through the Persistent Publish/Subscribe (PPS) service. Applications use this information to access the content on devices so they can display media information in browsers or invoke multimedia services to read metadata from those devices.

Applications could send commands to device drivers to read device information but this design requires developers to learn hardware interfaces, which is inconvenient. Device publishers offer a reliable source of device information by allowing applications to subscribe to PPS objects to receive device updates. Refer to the *Persistent Publish/Subscribe Developer's Guide* for details on implementing a subscriber.

The information published for a device includes:

- its raw device name
- its mountpoint
- the names and indexes of its partitions
- connection information such as the bus number, device number, and slot number

Device publishers are designed for specific hardware types. We ship the following device publishers:

- `usblauncher`, for USB storage devices
- `mmcsdpub`, for MMC and SD cards
- `cdpub`, for CDs

The `usblauncher` service replaces `usbpub` and the other USB driver-related services with a single service for monitoring USB devices and publishing their information.

Device publishers don't monitor the physical connection of devices. Other system services monitor hardware events and respond when users attach or detach storage devices (e.g., USB sticks) or when they insert or remove mediastores (e.g., SD cards) from physical slots. For instance, the `io-usb` server monitors the hardware and reacts to each USB device attachment or detachment by notifying `usblauncher`.

The publishers run as persistent background processes. Typically, they're launched during bootup. You can restart publishers if they unexpectedly terminate.

## Running a device publisher

The PPS service must be running if you want to use the device publishers. All publishers write to objects in the same path; you must create the root directory in this path before starting any publisher.

*To set up PPS and run a device publisher:*

1. In a QNX Neutrino terminal, check whether PPS is running by viewing the list of active processes with `pidin` or `ps`.

   Depending on your system configuration, PPS may be launched at bootup.

2. If necessary, start PPS by entering `pps` in the same terminal.

   PPS creates a root directory (`/pps` by default) that provides a general path for storing all objects used by various platform services.

3. Enter `mkdir -p /pps/qnx` to create the PPS path required by the publishers.

   For the locations of the PPS objects written by the publishers and an overview of the object contents, see "*PPS objects types* (p. 13)".

4. Enter the command line for starting the device publisher you want to use.

   Suppose you want to run `usblauncher` to get information about USB devices. You would then enter a command similar to:

   ```
   usblauncher -c /etc/usblauncher/new_rules.lua -p 2 -vvv
   ```

The PPS and device publisher services are running. Your applications can now get information about connected devices by reading the objects under `/pps/qnx`.

## Role of device drivers and mcd

Device drivers enable communication with attached devices by supporting system calls such as *open()*, *read()*, and *write()*. Publishers use these same system calls to obtain information from drivers on the devices that they manage. Another system service, `mcd`, uses the drivers to mount device filesystems.

**How drivers are started**

For devices that are always physically present, such as SD card readers, their drivers are launched during system startup and run continuously.

For devices connected through standard hardware interfaces, notably USB storage devices, their drivers are started and stopped by system services when the devices are added and removed. For instance, the `usblauncher` publisher launches drivers for USB devices when they're attached.

**How publishers communicate with drivers**

Drivers create entries in `/dev` for attached devices. For example, a USB driver creates a file named `/dev/umass0` (or something similar) when the user inserts a USB stick. The same driver will delete that object when the user removes the USB stick.

Drivers don't send data to publishers. Instead, the publishers monitor specific device paths and when they detect state changes to the devices represented by those paths, the publishers invoke the operating system to learn which drivers manage those paths. Then, the publishers communicate with those drivers to retrieve new or updated device details, which they publish to PPS objects.

In the command lines that start the publishers, you must supply the list of device paths to monitor. Depending on the publisher type, these paths must refer to either individual `/dev` entries or to directories containing device objects of a given hardware type. For more details, see the usage pages for individual publishers.

The exception to this design is the `usblauncher` utility. Because it launches drivers in addition to publishing device information, `usblauncher` doesn't need to find new driver processes to retrieve device information. For an explanation of the alternative design that this utility uses, see "*The usblauncher Service* (p. 17)".

**How device filesystems are mounted**

If you're using `usblauncher` for device publishing, you can configure that utility to mount the filesystems of USB devices; this way, you don't need to use a separate service to mount filesystems (unless you also need to mount non-USB devices).

To mount non-USB devices, you must use the Media Content Detector (`mcd`) service, which is included with the product. The `mcd` service monitors the `/dev` directory. When an object is added, `mcd` mounts the filesystem of the newly attached device

based on the rules in its mountpoint file (`/etc/mcd.mnt` by default). See the `mcd` entry in the *Utilities Reference* for information on the mount rules.

To mount a device's filesystem, `mcd` sends a mount request to the appropriate device driver. The driver manages the paths of both the mountpoint (e.g., `/fs/usb0`) and the device object (e.g., `/dev/umass0`).

## PPS object types

Device publishers write device information to multiple PPS objects in fixed locations. Applications can monitor these objects to learn of device updates and to retrieve information that enables access to device contents.

All publishers output the following PPS objects for each device:

| Type | Description | Location |
|---|---|---|
| Device object | Contains physical and logical connectivity information specific to the device's hardware type. This information might include:<br>• the device's vendor ID<br>• the mediastore type<br>• the product serial number<br>• details on temperature and drive mechanism state<br>• slot information | `/pps/qnx/device/` |
| Device control object | Contains commands specifying device actions to perform, such as toggling the power on a port of a USB hub, ejecting a disc, or enabling or disabling disc removal.<br>Can also contain command outcomes, depending on the publisher type. | `/pps/qnx/device/<type>_ctrl`, where `<type>` is either `usb` or `cd0` |
| Driver object | Contains driver information, such as the process name and ID (*pid*), and a reference to the device object associated with this driver. | `/pps/qnx/driver/` |
| Mount object | Contains information on the device's filesystem, including:<br>• the mountpoint<br>• the filesystem type<br>• the partition label<br>• other fields specific to the hardware type | `/pps/qnx/mount/` |

## Plugins

Plugins are external components that help device publishers provide extra media-related information.

The plugins are shipped with the QNX SDK for Apps and Media, so you must install this product to use them. However, the plugins are useful only in systems that support media playback and external devices. If your system has no plugins installed, this doesn't cause an error; instead, the device publishers just write fewer data fields to the mount objects.

The plugin manager isn't packaged as a separate service or library but instead is integrated into all device publishers. At startup, publishers initialize the plugin manager by providing their type (e.g., `publisher-usb`) and the path of the plugin directory. The plugin manager loads, from the specified directory, all installed plugins that support the device publisher type and then builds a list of available plugins, sorted by rating. This last action allows the plugin manager to forward information requests to plugins with higher ratings first. Plugins with lower ratings are invoked only when the higher-rated plugins can't provide the requested information.

**Included plugins**

The QNX SDK for Apps and Media includes the following plugins:

**`generic`**

Reads volume names and generates volume IDs from the name and other fields such as the serial number. Supports all device types and reads `WMPInfo.xml` (if it's available on the filesystem).

**`ipod`**

Obtains the names and IDs of iPods.

**`mediafs`**

Obtains the names and IDs of Media FS devices.

**`audiocd`**

Extracts Table of Contents (TOC) and CD-Text information from audio CDs to fill in fields such as artist and album name.

## Plugin ratings

Plugins rate themselves on their ability to obtain information from various device types and report their ratings to the plugin manager so that it can choose the best plugin for filling in data fields.

Each plugin provides two ratings of itself:

**Support for a device publisher**

The plugin indicates whether or not it supports the specified device publisher.

**Information extraction ability**

The plugin provides a numeric score indicating its ability to read device information. The plugin manager uses these scores to rank plugins against each other.

The following table shows which plugins work with each of the device publishers, with the plugins listed from highest rated to lowest rated:

| Device publisher | Supporting plugins |
|---|---|
| `mmcsdpub` | `generic` |
| `cdpub` | `audiocd`, `generic` |
| `usblauncher` | `ipod`, `mediafs`, `generic` |

**Plugin selection**

Although the plugin manager ranks the plugins based on their ability to extract information, the particular plugin used for filling in data fields also depends on the device type.

Suppose you connect a Media FS device through a USB port. If the `usblauncher` publisher requested the device name and ID, the plugin manager would first invoke the `ipod` plugin but this plugin would fail to fill in the fields because the device isn't an iPod. Next, the plugin manager would invoke `mediafs`, which would try to obtain the requested information. If that second plugin failed for whatever reason, the plugin manager would invoke `generic`. So, the extra device information can come from any plugin that supports the publisher.

# Chapter 2
# The usblauncher Service

The `usblauncher` service enumerates USB devices, launches drivers for communicating with USB devices, and publishes device information through PPS.

This utility depends on the following services:

**io-usb**

The USB host stack, which monitors the USB hardware and notifies `usblauncher` of device attachments and detachments. The `usblauncher` service runs `io-usb` to act as the USB host, which means it controls communication on the bus.

**Drivers for USB host mode**

When running the USB host stack (`io-usb`), `usblauncher` can start different drivers to communicate with different types of devices. The simplest use case is when a user plugs in a mass-storage device, in which case `usblauncher` starts a `devb-umass` driver to communicate with it; if a device has multiple partitions, one such driver manages all of the device's filesystems. Other drivers that may be used in host mode include `mm-ipod`, `io-pkt` with a USB Ethernet driver, `devc-serusb`, and others.

**io-usb-dcd**

The USB device stack, which notifies `usblauncher` of state changes such as a device configuration update or a cable removal. The `usblauncher` service runs `io-usb-dcd` to act as the USB device, which means it services requests from the USB host.

**Drivers for USB device mode**

After starting the USB device stack (`io-usb-dcd`), `usblauncher` immediately starts the appropriate driver based on the device properties provided by the stack. For instance, the `devc-serusb_dcd` driver enables serial data transfer (see "*USB descriptors* (p. 43)", which explains the USB-to-serial communication configured in `usbser.lua`). The `devu-umass_client-block` driver supports mass storage devices when the stack runs in device mode. Other drivers that may be used in device mode include `mm-ipod`, `io-pkt` with `usbdnet`, and others.

### Benefits of usblauncher

The multipurpose `usblauncher` utility allows for a simple system setup and provides these benefits:

#### Drivers can be started immediately

Because `usblauncher` launches the USB drivers, it knows immediately whether a newly attached device is supported (i.e., if the system has a suitable driver for the device's type). Thus, when creating a PPS device object, `usblauncher` can publish the number of drivers it will start for the device; this lets applications subscribed to this PPS object know right away whether the device is supported.

#### Automatic filesystem mounting

The `usblauncher` utility can mount the filesystems of USB devices based on the rules in the `mcd` mountpoint file (`/etc/mcd.mnt` by default). Also, `usblauncher` can publish the statuses of mount operations to inform applications whether individual filesystems were successfully mounted.

If your system setup requires it, you can still use `mcd` to mount USB filesystems while running `usblauncher`. For information on how to do this, see "*Disabling usblauncher auto-mounter* (p. 35)".

#### Support for MirrorLink

A smartphone must have a Network Configuration Management (NCM) interface to make its applications accessible through a car infotainment system with MirrorLink. The `usblauncher` utility can request that a smartphone re-enumerate itself as an NCM device. This way, the user doesn't have to manually configure their smartphone for MirrorLink mode.

#### Support for Apple CarPlay

The `usblauncher` utility can request a device to become the USB host, to support Apple CarPlay (formerly known as iOS in the Car). When this role-swapping request succeeds, `usblauncher` can stop the USB host stack and start the USB device stack (and do its own role-swapping). The `usblauncher` utility can also probe a device to test whether it supports iAP2, allowing you to choose a driver to launch based on the test's outcome.

### Architecture

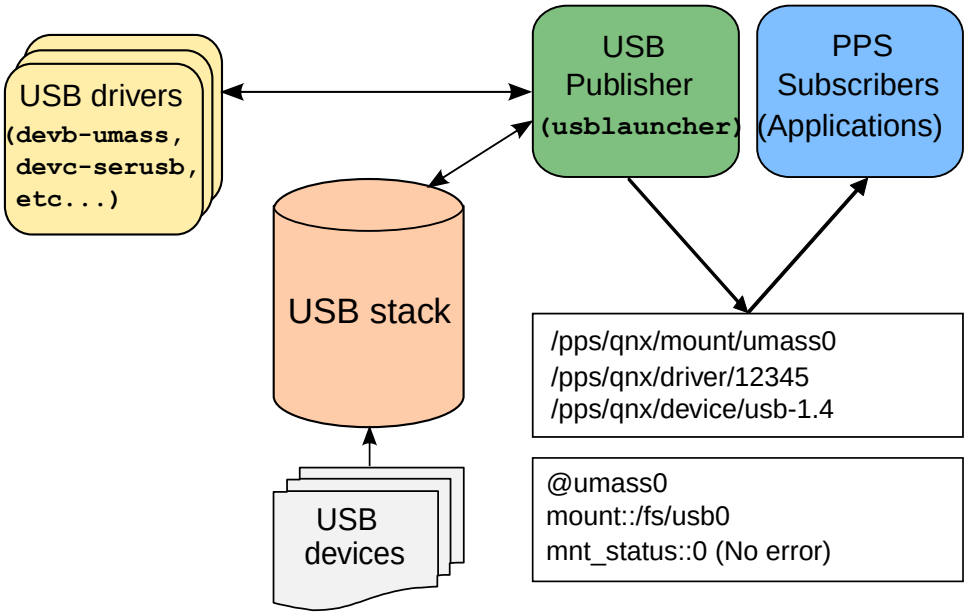


**Figure 1: Architecture of usblauncher**

The arrows between components in *Figure 1: Architecture of usblauncher* (p. 19) show information flow. When USB devices are attached or detached, they trigger hardware interrupts that controllers within the USB stack (either `io-usb` or `io-usb-dcd`) detect. These controllers then react to the device event by notifying `usblauncher`.

---

By default, `usblauncher` starts the USB stack in host mode, assuming you've defined the *host and device stack rules* (p. 36) in the configuration file. You can prevent the host stack from starting by using the *`-h` command option* (p. 34), which is useful when your setup allows you to run the first stack instance in device mode.

---

The `usblauncher` service queries the stack to retrieve the descriptors (i.e., properties) of any newly attached devices and then publishes this information to device objects. Next, `usblauncher` compares these descriptors against the USB device-matching rules in its configuration file. When it finds a match, `usblauncher` launches the appropriate driver (e.g., `devb-umass`, `devc-serusb`) and publishes the driver's information to a driver object.

To mount filesystems, `usblauncher` sends mount requests to the drivers that manage the paths in `/dev` for USB devices. The mountpoints requested by `usblauncher` are based on the rules in the specified mountpoint file (for details, see the *`-M` option for `usblauncher`* (p. 34)).

Finally, `usblauncher` communicates with the drivers to obtain mount information, which it writes into mount objects. Subscribed applications receive updates through PPS when device information changes.

## Support for USB On-The-Go (OTG)

The `usblauncher` service supports USB OTG by allowing role-swapping of the USB stack. This way, the target system can act sometimes as the host and sometimes as the slave device in a USB link.

Our `usblauncher` implementation does *not* support any OTG communication protocols designed to detect device attachments (ADP), manage power on a USB link (SRP), or negotiate the roles of two devices sharing a USB link (HNP). Instead, `usblauncher` offers software support for OTG by allowing client applications running on the target system to swap roles with devices at the other end of a USB link (i.e., a one-to-one connection). To do this, applications can instruct `usblauncher` to restart the USB stack in a different mode (i.e., host or device).

A client application must monitor the USB bus and other hardware to determine which mode the target system should enter into. When the application knows this, it can issue the `start_stack` command through PPS to the *device control object* (p. 26). When requesting that the stack run in device mode, the application can specify a USB personality to expose to the USB host at the other end of the connection. The `usblauncher` service will then launch the appropriate drivers for the specified personality.

Consider the following scenario, in which an application running on a computer detects hardware events and responds by changing the role of the USB stack:

1. **The user inserts a cable into a port on the target system**

   The application detects the cable attachment and tells `usblauncher` to start the USB stack in device mode by sending the `start_stack::device,`*n* command to the device control object. Here, *n* is a number indicating the set of USB descriptors that `usblauncher` should expose to the USB host at the other end of the connection.

2. **The user removes the cable**

   The application detects the cable removal and instructs `usblauncher` to slay the USB stack by issuing the `start_stack::none` command to the same PPS object.

3. **The user plugs a USB stick into the target**

   The application detects the USB stick and tells `usblauncher` to start the USB stack in host mode by issuing the `start_stack::host` command to the device control object. This way, the computer can manage the USB link and download data from the USB stick as needed.

## Device object

USB device objects have names in the form: `usb-`*`bus_number.device_number`*. If `usblauncher` is called with the `-S` option, the object name also includes the stack number in front of the bus number. For instance, for a USB device with a stack number of `0`, a bus number of `1`, and a device number of `3`, the device control object would be named `usb-0.1.3`.

We recommend learning about the PPS objects used by `usblauncher` by examining their contents, not their filenames because these filenames could change in the future.

Each device object that `usblauncher` writes to PPS contains the following fields:

| Name | Description | Type | Example |
|---|---|---|---|
| `bus` | Bus type | String | `USB` |
| `busno` | Bus number | Integer | `0x00` |
| `configuration` | Selected configuration | Integer | `1` |
| `configurations` | Number of configurations | Integer | `1` |
| `device_class` | Device class ID | Integer | `0xff` |
| `device_protocol` | Device protocol (present only if `device_class` is nonzero) | Integer | `0xff` |
| `device_subclass` | Device subclass ID (present only if `device_class` is nonzero) | Integer | `0x00` |
| `devno` | Device number | Integer | `0x04` |
| `drivers_matched` | Number of drivers matching the device based on the configuration file rules. A value of `0` means the device is unsupported. | Integer | `1` |
| `drivers_running` | Number of drivers launched for the device. This value can be less than `drivers_matched` if some drivers haven't been started yet or if some have terminated (possibly in error). | Integer | `0` |
| `iap` | Outcome of probe for the device's iAP2 support. This field is present only if the USB stack is running in host mode and | Integer with one of the following values: `-1` (when the device | `2` |

| Name | Description | Type | Example |
|---|---|---|---|
| | after the device was probed for iAP2 support. | doesn't support iAP2 or the probe failed) or 2 (when the device supports iAP2) | |
| `manufacturer` | Manufacturer name | String | `Kingston` |
| `max_packet_size0` | Maximum packet size | Integer | `64` |
| `product` | Product name | String | `DataTraveler G3` |
| `product_id` | OEM product ID | Integer | `0x1624` |
| `role_swap` | Reason for latest role swap of USB stack. This field is present only after switching the stack from host to device mode to support a third-party application. | String with one of the following values: `AppleDevice` (for iAP2 in client mode) or `DigitaliPodOut` (for CarPlay) | `AppleDevice` |
| `serial_number` | Product serial number | String | `0019E0014A16A931953F004E` |
| `stack_no` | USB stack number specified with `-S` command-line option for `usblauncher`.<br>This field differentiates devices when they have the same bus number (`busno`) and device number (`devno`) but are managed by different `io-usb` stacks. | Integer | `0` |
| `status` | Device status. Normally, this field is present only when the device is being reset; the field is deleted when the reset completes. | Integer followed by string, in the format: *`status (message)`* | When the device is being reset, this field contains: `-1 (Device reset)`. |
| `topology` | Duplicates upstream device and port numbers (`devno`, `upstream_port`) and provides upstream information for hub chain. | String containing integers in ordered pairs | `(2,1),(0,2)`<br>This value indicates a USB device is |

| Name | Description | Type | Example |
|---|---|---|---|
| | | | attached to hub device 2, port 1, and this hub is connected to the root device, port 2. |
| `upstream_device_address` | USB address where the device is connected. When the device is connected to a host controller, this field is `0`. When the device is connected to a USB hub, this field contains the hub's device address. | Integer | `1` |
| `upstream_host_controller` | Host controller number. This field is `0` unless you have multiple USB controllers, in which case it contains the number of the controller that detected the device. | Integer | `0` |
| `upstream_port` | Port number. When the device is connected to a hub, this field contains the port number on the hub. When the device is connected to the host controller, this field is `0`. | Integer | `4` |
| `upstream_port_speed` | Port speed. | String with one of the following values: `High`, `Low`, or `Full` | `High` |
| `vendor_id` | Manufacturer ID | Integer | `0x0951` |

#### Overcurrent condition

When an overcurrent is detected, `usblauncher` publishes a special type of device object that represents the USB hub that's reporting the overcurrent condition. The object doesn't provide information on the device causing the condition; in fact, this device is taken off the USB bus and its device, driver, and mount PPS objects are deleted.

The filename of the object is in this format: `usb-overcurrent-`*`stackno`*`.`*`busno`*`.`*`devno`*`.`*`portno`*. The stack number (*stackno*) is present only if you started `usblauncher` with the `-S` option.

The complete object looks like this:

```
status::-1 (Overcurrent)
```

```
bus::USB
stackno::0
busno::0x00
devno::0x01
portno::3
```

Once the overcurrent condition is cleared, the overcurrent object is deleted.

### Bad devices

When `io-usb` can't assign a device number to a USB device, `usblauncher` publishes a device object with fewer attributes than normal to describe the "bad" device. Also, the object's filename is in a different format: `usb-overcurrent-`*`stackno`*`.`*`busno`*`.`*`upstream_device_address`*`.`*`upstream_port`*. The stack number (*stackno*) is present only if you started `usblauncher` with the `-S` option.

The complete object looks like this:

```
status::48 (Not supported)
bus::USB
stackno::0
upstream_device_address::3
upstream_host_controller::0
upstream_port::7
upstream_port_speed::High
```

## Device control object

Applications can perform actions on USB hardware by writing commands to device control objects. Each `usblauncher` process creates one of these server PPS objects. By default, the object's path is `/pps/qnx/device/usb_ctrl` but you can change this path through `usblauncher` command options.

The stack number is appended to the object name if you run `usblauncher` with the `-S` option. For instance, if you issue the command `usblauncher -S 1`, the service creates a PPS object named `usb_ctrl1`. You must use this command option to distinguish between control objects if you want to run multiple `usblauncher` instances and use the same PPS directory for their device control objects.

When `usblauncher` receives updates (i.e., data from the object), it executes the specified commands. Commands must be written in the following format:

`<name>::<parameter>`

For details on publishing data to PPS objects, refer to the *Persistent Publish/Subscribe Developer's Guide*. Applications can publish the following to the control object:

| Command | Description | Parameter type and purpose |
|---|---|---|
| `port_power` | Set the power state on a port of a USB hub. This command is useful to repower the port if the upstream hub has disabled it (e.g., as a result of an overcurrent condition and you wish to see if the overcurrent condition still applies). | A set of integers identifying the device and the desired power state, in the following format: `busno,devno,power_state[,portno]`<br>If the optional `portno` is omitted, then all ports on that hub are controlled by the power level. The `power_state` can be either `0` (“off”) or `1` (“on”). |
| `start_stack` | Start a specific version of the USB stack. There are two versions: the host stack (`io-usb`) and the device stack (`io-usb-dcd`). Any current USB stack instance is stopped (if necessary) before the other stack version is started. You can specify `none` to slay the stack without restarting either version of it. | A string identifying the version of the USB stack to start. Can be one of: “`host`”, “`device,`*n*”, or “`none`”, where *n* is a one-based index into the `descriptors` list in the main configuration file. |
| `toggle_port_power` | Turn a USB port's power off and back on after a fixed delay. This command works only for ports on a USB hub capable of power switching on individual ports. | A set of integers identifying the device, in the following format: `busno,devno,port` |

The `port_power` and `toggle_port_power` commands aren't meant to be used in cases when the `start_stack` command is being issued.

When `usblauncher` executes commands, it publishes device information and command outcomes to the device control object. The `usblauncher` service can output the following attributes:

| Attribute | Description | Type | Example |
|---|---|---|---|
| `cmd_status` | Outcome of the command | An error string in the following format: *`errno (str_error(errno))`*.<br>If the command wasn't recognized, `usblauncher` doesn't publish a status to this object but instead writes an error message to *stdout* or `sloginfo`. | When the command was successful, this field contains: `0 (No error)`<br>If you tried to toggle the port power on a USB device that isn't a hub, this field contains: `48 (Not supported)` |
| `port_power` | Latest power setting for the port whose power is being toggled | Integer with one of the following values: `1` (for “on”), `0` (for “off”), or `-1` (for “unknown”, which means `usblauncher` couldn't read the power setting). | `1` |

## Driver object

USB driver objects have names that match the driver process IDs. This strategy ensures the distinctiveness of object names for different USB devices because process IDs are unique throughout the system.

We recommend learning about the PPS objects used by `usblauncher` by examining their contents, not their filenames because these filenames could change in the future.

Each driver object that `usblauncher` writes to PPS contains the following fields:

| Name | Description | Type | Example |
|---|---|---|---|
| `arguments` | Command-line arguments passed to driver for the device | String | `cam quiet blk cache=1m,vnode=384, auto=none,delwri=2:2,rmvto=none, noatime disk name=umass cdrom name=umasscd umass path=/dev/otg/io-usb, priority=21,vid=0xfca,did=0x8004, busno=0,devno=0x2,iface=0x1, ign_remove` |
| `interface` | USB interface number | Integer | `1` |
| `interface_class` | USB class ID | Integer | `0x08` |
| `interface_name` | USB interface name. This field is present only if the device defines it. | String | `RIM Mass Storage Device` |
| `interface_protocol` | USB interface protocol | Integer | `0x50` |
| `interface_subclass` | USB subclass ID | Integer | `0x06` |
| `name` | Driver process name | String | `devb-umass` |
| `pid` | Driver process ID | Integer | `593947` |
| `PPS_DEVICE_ID` | Device object path | String | `/pps/qnx/device/usb-0.2` |

## Mount object

USB mount objects have names in the form *rawdevice*[.*partition#*]. For example, if you attach a USB stick with a DOS partition and the raw device created is `/dev/umass0`, the mount object is named `/pps/qnx/mount/umass0.0`.

We recommend learning about the PPS objects used by `usblauncher` by examining their contents, not their filenames because these filenames could change in the future.

Each mount object that `usblauncher` writes to PPS contains the following fields:

| Name | Description | Type | Example |
|---|---|---|---|
| `blocks_size` | Size of each block (in bytes) | Integer | `512` |
| `blocks_total` | Total number of blocks | Integer | `1622502` |
| `fs_type` | Filesystem type. This field is present only if the filesystem is mounted. | String with one of the following values: `dos (fat32)`, `qnx4`, `qnx6`, `nt`, or `unknown` | `dos (fat32)` |
| `label` | Filesystem label. This field is present only if the filesystem is mounted. | String | `Home movies 1` |
| `mnt_status` | Outcome of the mount operation | Integer followed by string, in the format: *`errno`* (*`str_error(errno)`*) | `48 (Not supported)` |
| `mount` | Mountpoint | String | `/fs/usb0_0` |
| `partition` | Partition name. This field is present only if the mount object represents a partition. | String | `/dev/umass/umass0t6` |
| `partition_count` | Total number of partitions. This field is present only if the mount object represents an entire device, not a partition. | Integer | `1` |
| `partition_order` | Partition index. This field is present only if the mount object represents a partition. | Integer | `0` |

| Name | Description | Type | Example |
|---|---|---|---|
| `plugin_name` | Name of plugin supplying extra mount information, followed by mount fields returned by plugin | String in the following format: *pname* `[`*attr_name::value*`]*`<br>Here, *pname* is one of the following plugin names: `ipod`, `mediafs`, or `generic`.<br>The set of attributes written into this field depends on the plugin. | `plugin_name::generic name:: id::5d05f0df-68f4-4cfe -b06e-ad664aad3ec8` |
| `PPS_DRIVER_ID` | Driver object path | String | `/pps/qnx/driver/265837` |
| `PPS_RAWMOUNT_ID` | Mount object path | String | `/pps/qnx/mount/umass0` |
| `raw` | Name of raw device | String | `/dev/umass/umass0` |
| `read_only` | Read-only status of device. This field is present only if the filesystem is mounted. | Boolean: `0` if writable or `1` if read-only | `0` (writable) |
| `status` | Error string. This field is present only if an error occurred when accessing the device's mounted filesystem. | Integer followed by string, in the format: *`errno (str_error(errno))`* | `302 (Corrupted file system detected)` |

## Command line for usblauncher

*Start usblauncher device enumerator and publisher*

**Synopsis:**

```
usblauncher [-b] [-C] [-c config_file] [-e] [-h] [-l]
    [-M mnt_rules] [-m pps_path] [-n stack_name] [-P]
    [-p seconds] [-S stack_number] [-s dll_path] [-t] [-v]
```

**Options:**

**-b**

Run `usblauncher` in the foreground. This option is handy for debugging because you can press **Ctrl**–**C** to terminate the publisher.

By default, `usblauncher` runs in the background.

**-C**

Always set the device configuration when attaching a device. This option should be used alongside the `io-usb -C` option, which tells the USB stack never to set the device configuration at enumeration time. Note that this stack option doesn't affect hubs—their configurations are always set by the stack.

We recommend using the `-C` option for both `usblauncher` and `io-usb` when supporting MirrorLink devices, which will make `usblauncher` set the configuration after sending the NCM request.

By default, `usblauncher` selects a configuration only for devices with multiple configurations, based on the first driver that matches the device.

**-c *config_file***

The configuration file. At startup, `usblauncher` reads this file to learn the USB device-matching rules for different device types. These rules indicate which driver to launch for communicating with the device as well as which command options to provide to the driver.

If you don't specify a configuration file, `usblauncher` looks for the default file (`/etc/usblauncher/rules.lua`). In this case, the default file must exist or `usblauncher` won't run.

The configuration file named with this command option can include other Lua files by using the `dofile()` command. For example, if you use separate

files to define various descriptors to use when running in USB device mode, you can include these descriptor files in the main configuration file.

**-e**

Enable detection of extra events; specifically, bad device attachments and detachments as well as device resets.

By default, `usblauncher` doesn't detect and publish information about these extra events.

**-h**

Prevent the USB stack from starting in host mode, which is the default behaviour. Use this option when you want to support OTG but your client application will decide which stack version to start and then issue the `start_stack` command through PPS to tell `usblauncher` when to start a stack.

**-l**

Log messages to `sloginfo` instead of standard out.

By default, `usblauncher` logs messages to standard out.

**-M** ***mnt_rules***

The mountpoint file, which tells `usblauncher` where to mount the filesystems of devices represented by particular `/dev` entries. Any file named with this option must be in the same format as the default `mcd` mountpoint file (`/etc/mcd.mnt`). In this file, you can name not only mountpoints but also the mount options provided to the filesystem library, such as codepage mapping or long-filename handling in `fs-dos`.

Use the `-M` option if you want to assign nondefault mountpoints to devices. You can also name an empty file to prevent `usblauncher` from mounting USB devices (see "*Disabling `usblauncher` auto-mounter* (p. 35)" for details).

**-m** ***pps_path***

The PPS directory path. The subdirectories for storing the device, device control, driver, and mount objects are located in this directory. The default is `/pps/qnx/`.

**-n** ***stack_name***

The server name of the USB stack. The default is `/dev/io-usb/io-usb`. Use this option only if you don't support OTG and always run the USB host

stack (`io-usb`), never the USB device stack (`io-usb-dcd`), and if your USB host stack is stored at a nondefault location.

If you want to support OTG and use nondefault USB stack paths, you must define these paths in the `Host_Stack` and `Device_Stack` rules in the configuration file.

**`-P`**

Probe the media to get the partition count only if the device is ready.

By default, `usblauncher` doesn't wait until the device is ready to try to obtain the partition count.

**`-p`** ***seconds***

The polling interval (in seconds), which is how often `usblauncher` checks for changes to the mountpoints associated with the active drivers.

The default interval is three seconds.

**`-S`** ***stack_number***

The USB stack number. The `usblauncher` service stores this value in the information for each USB device it monitors, to differentiate the device from others that have the same bus number and device number but are managed by other `io-usb` servers.

**`-s`** ***dll_path***

The plugin path. At startup, `usblauncher` looks in this path for plugins it can load and then use to provide more detailed device information (see "*Plugins* (p. 14)").

If this option isn't specified, no plugins are loaded.

**`-t`**

Terminate launched drivers when exiting.

By default, `usblauncher` doesn't terminate driver processes while deleting device and driver objects during shutdown.

**`-v`**

Increase output verbosity. The `-v` option is cumulative, so you can use several `v`'s to increase verbosity. Setting one `v` logs USB device attachments and detachments. Setting two `v`'s adds the logging of PPS object creation and deletion. Setting three or four `v`'s logs more detailed events as well as errors that are less severe.

Output verbosity is handy when you're trying to understand the operation of `usblauncher`. However, when lots of `-v` arguments are used, the logging becomes quite significant. The verbosity setting is good for systems under development but probably shouldn't be used in production systems or during performance testing.

**Description:**

You should start `usblauncher` with an explicit command only if the process terminates unexpectedly. Before trying to start `usblauncher` manually, always confirm that the process isn't already running by checking the list of active processes with *pidin* or *ps*. If you want to restart only the USB stack, send the `start_stack` command to the *device control object* (p. 26).

The `usblauncher` command starts a multipurpose service that enumerates USB devices, launches drivers for communicating with those devices, and publishes their information through PPS.

On the command line, you can override the default configuration file to provide `usblauncher` with a set of custom rules for configuring different device types. You can also name your own mountpoint file to mount filesystems to nondefault locations, set how often `usblauncher` checks for mountpoint updates, and configure how it logs error and event information.

By default, `usblauncher` starts the USB stack in host mode, assuming you've defined the appropriate rules in the configuration file. You can prevent the launching of this stack by using the `-h` option, if you plan to use your system as the USB device.

The `usblauncher` service runs as a self-contained process that doesn't require any user input. It has no client utility for performing device-publishing tasks on request or for adjusting any of its settings. You can restart the USB stack in a different role by issuing the `start_stack` command through PPS, but to reconfigure `usblauncher` in any other way, you must change the options in its command line and restart the service. We recommend putting the `usblauncher` command line in a startup script (e.g., `startup.sh`) to automatically launch the service during bootup.

## Using usblauncher to mount filesystems

The `usblauncher` service can mount the filesystems of attached USB devices based on the rules in a mountpoint file.

The `usblauncher` service reads either the default `mcd` mountpoint file (`/etc/mcd.mnt`) or another file specified with the `-M` option in the command line that started the service. Basically, `usblauncher` performs the function of the `MOUNT_FSYS` rule in the `mcd` configuration file while using the standard mechanism of *mount* to do the actual mounting.

### Preventing mcd from mounting USB devices

To use the usblauncher auto-mounter feature when your system runs mcd, you must do one of the following two actions:

- In the mcd configuration file (/etc/mcd.conf), disable any MOUNT_FSYS rule that references a mountpoint file with USB device listings (e.g., /dev/umass*t[146]).
- Delete the USB device listings in any mountpoint file referenced by a MOUNT_FSYS rule.

These actions ensure that usblauncher and mcd don't try to mount the same filesystems, which will happen if both processes are tasked with assigning mountpoints for the same USB-related /dev entries. For more information on mcd and its configuration file, refer to the mcd entry in the *Utilities Reference*.

### Disabling usblauncher auto-mounter

If your system setup requires you to use mcd for mounting all devices, you can disable the usblauncher auto-mounter feature by specifying an empty mountpoint file with the -M option. With no matching mountpoint entries for any attached devices, usblauncher won't try to mount any filesystems.

This strategy is handy if you need to mount the filesystems of both USB and non-USB devices and you prefer to use a common service (mcd) for all mounting operations. Unfortunately, usblauncher will no longer be able to log the mount attempts and publish accurate mnt_status values through PPS.

## Configuration files

The `usblauncher` configuration files contain instructions for launching drivers for particular devices as well as descriptors to expose to USB hosts when running the stack in device mode.

Configuration files for `usblauncher` are interpreted as Lua scripts (see *the Lua project homepage* for information on the Lua language). By default, `usblauncher` reads the file at `/etc/usblauncher/rules.lua`, but you can specify a nondefault configuration file by using the `-c` command-line option. You can name only one configuration file, but this file can include other Lua files by using the `dofile` command. For ease of maintenance, the platform image stores the USB device mode descriptors in separate Lua files, which are included in the main file so that `usblauncher` can override the default descriptors after launching the USB device stack.

### Starting USB stack in host and device modes

Your configuration file must contain commands for launching the USB stack in each of the host and device modes if you want to support USB On-The-Go (OTG). Switching stack modes dynamically allows for creating devices capable of acting as either a USB host or a USB device, without requiring separate USB controllers and ports.

The `rules.lua` configuration file shipped with the platform contains sample rules for launching the USB stack in both the host and device modes:

```
Host_Stack = {
    cmd  = 'io-usb -c -d ehci-mx28 ioport=0x02184100,irq=75,\z
           phy=0x020c9000';
    path = '/dev/otg/io-usb';
}

dofile '/etc/usblauncher/iap2.lua'
dofile '/etc/usblauncher/iap2ncm.lua'
dofile '/etc/usblauncher/umass.lua'

Device_Stack = {
    cmd  = 'io-usb-dcd -d iap2-mx6sabrelite-ci \z
           ioport=0x02184000,irq=75,vbus_enable';
    path = '/dev/otg/io-usb-dcd';
    descriptors = { iap2, iap2ncm, umass };
}
```

These rules contain the following fields:

**cmd**

Stores the command line for starting the stack in a particular mode. The command line consists of the process name followed by hardware-specific options for configuring the stack. This excerpt shows the options for the i.MX6Q SABRE Lite platform; your system might use different options

depending on the board you're running on. You can find information on all options supported by the host and device stacks in the `io-usb` and `io-usb-dcd` references.

**`path`**

Provides the full path of the USB stack service. In this release, there are separate services for the host and device modes.

The `usblauncher` service appends the stack path to the string in `cmd` by using the `-n` option; it's not necessary to provide this option in `cmd` when using a nondefault path.

**`descriptors`**

*Optional and used only with device mode.*

Lists *USB descriptors* (p. 43) that your system can expose to a USB host. Each descriptor specifies a set of hardware and connection properties based on a particular device function (e.g., mass storage, serial data transfer). These descriptors override the default descriptors compiled into the USB device stack.

The `dofile` statements that include the configuration files containing the descriptors must be placed above the `Device_Stack` rule to make them visible in the `descriptors` list.

Based on the device function you want your system to support, you can select a descriptor by sending the `start_stack::device,`*n* command to the device control object. Here, *n* is a one-based index in the `descriptors` list (i.e., `1` refers to the first list entry). If you don't define the `descriptors` list or you provide an out-of-bounds index or no index in the `start_stack` command, the compiled-in descriptors (but no overridden descriptors) are used.

You can use the `RoleSwap_AppleDevice` and `RoleSwap_DigitaliPodOut` flags to enable client-mode iAP2 and the CarPlay application. Both these flags cause `usblauncher` to restart the stack in device mode and expose a different set of descriptors. For `RoleSwap_AppleDevice`, `usblauncher` uses the first item in the list; for `RoleSwap_DigitaliPodOut`, it uses the second item. For more information on these flags, see "*Supported third-party applications and protocols* (p. 49)".

## USB matching rules

The USB matching rules in the main configuration file (`rules.lua`) provide `usblauncher` with commands for launching the appropriate driver with the appropriate configuration when a device is attached through a USB port.

The matching rules specify device properties (e.g., vendor ID, device ID, USB class) that `usblauncher` compares against the descriptors of newly attached devices; these descriptors are provided by the USB stack. When `usblauncher` finds a matching rule with properties matching those of a newly attached USB device, it executes the driver command specified in that rule.

### Matching rule levels

Matching rules are expressed in a hierarchy that defines different levels at which to match device descriptors. Each level of depth is more specific than the last, meaning that its rules match a more restricted set of devices. The five levels in device specification rules are:

1. `device`
2. `configuration`
3. `interface`
4. `class`
5. `ms_desc`

When an outer (more general) rule matches a newly attached device, `usblauncher` compares the inner (more specific) rules in a depth-first search order. You can omit some or all of the inner rules if you don't care about matching any descriptors at that level.

For rules at the same level, `usblauncher` applies the rule listed the earliest in the configuration file. You can confirm the exact search order by calling *show_config()* at the end of your configuration file.

The device specification that follows provides three levels of matching rules. The driver command line is enclosed in double quotes and prefixed with the `driver` keyword. The names of macro variables in the driver command line are the same as the attribute names in PPS device objects.

```
device(<vid>, <did>, <rev_low>, <rev_high>) {
    configuration(1) {
        class(<class>, <subclass>, <protocol>) {
            driver"io-fs-media -dxxx,\z
                    device=$(busno):$(devno):$(interface)";
        };
    };
};
```

### Matching rule arguments

The matching rules take variable numbers of arguments so that they may be as specific as necessary. The following table outlines the descriptors matched at each device specification level:

| Level | Descriptors matched | Rule syntax |
|---|---|---|
| `device` | vendor ID, device ID, earliest revision, latest revision | `device(<vid>, <did>, <rev_low>, <rev_high>) -- most specific`<br>`device(<vid>, <did>, <rev_low>)`<br>`device(<vid>, <did>)`<br>`device() -- most general` |
| `product` (see *Footnote.*) | vendor ID, lowest device ID, highest device ID | `product(<vid>, <did_low>, <did_high>) -- most specific`<br>`product(<vid>, <did_low>)`<br>`product(<vid>)`<br>`product() -- most general` |
| `configuration` | configuration number | `configuration(<num>) -- specific`<br>`configuration() -- general` |
| `interface` | interface number or range indicated by lowest and highest numbers | `interface(<num_low>, <num_high>) -- most specific`<br>`interface(<num>)`<br>`interface() -- most general` |
| `class` | interface class, subclass, and protocol | `class(<class>, <subclass>, <protocol>) -- most specific`<br>`class(<class>, <subclass>)`<br>`class(<class>)`<br>`class() -- most general` |
| `ms_desc` | Microsoft descriptor | `ms_desc(<compatibleID>, <subcompatibleID>, <vendorID>) -- most specific`<br>`ms_desc(<compatibleID>, <subcompatibleID>)`<br>`ms_desc(<compatibleID>)`<br>`ms_desc() -- most general` |

[1] The `product` rule is the same as the `device` rule except that you can specify a range of device IDs.

### Variables

You can define variables at the top of the configuration file to hold configuration values used in the matching rules:

```
-- definitions of common USB properties
USB_CLASS_AUDIO        = 0x01
    USB_AUDIO_SUBCLASS_CONTROL        = 0x01
    USB_AUDIO_SUBCLASS_STREAMING      = 0x02
        USB_AUDIO_PROTOCOL_1_0 = 0x00
        USB_AUDIO_PROTOCOL_2_0 = 0x20
USB_CLASS_PHYSICAL     = 0x05
USB_CLASS_IMAGING = 0x06
    USB_IMAGING_SUBCLASS_STILL = 0x01
        USB_IMAGING_STILL_PROTOCOL_PTP = 0x01
USB_CLASS_PRINTER      = 0x07
USB_CLASS_MASS_STORAGE = 0x08
```

The matching rules that follow demonstrate the use of variables. The first specification matches any device that has a mass-storage class while the second specification matches a Sony Walkman device with an MTP Microsoft descriptor.

```
-- generic mass storage rule
device() {
    class(USB_CLASS_MASS_STORAGE) {

        DISK_OPTS  = "cam quiet blk cache=1m,vnode=384,\z
                        auto=none,delwri=2:2,rmvto=none,noatime \z
                        disk name=umass cdrom name=umasscd";
        UMASS_OPTS = "umass priority=21";

        driver"devb-umass $(DISK_OPTS) $(UMASS_OPTS),\z
                vid=$(vendor_id),did=$(product_id),\z
                busno=$(busno),devno=$(devno),\z
                interface=$(interface),ign_remove";
    };
};


-- Sony Walkman
device(0x054c, 0x03fd) {

        ms_desc("MTP", "", 1) {
                driver"io-fs-media -dpfs,device=$(busno):\z
                        $(devno):$(interface)";
        };
};
```

Notice that variables can be defined in the scope of a rule and used in the driver command line. When searching for variables, `usblauncher` searches the definitions from the innermost to the outermost scope.

### Flags

You can also place flags in a rule to change the range of devices matched as well as the actions performed after matching a device.

At the `device` level, the `Ignore` flag instructs `usblauncher` to ignore the device by not attaching to it and reading its descriptors. In this case, no PPS objects are published and no driver is launched. Consider the following example:

```
device(0x0e0f, 0x0003) {
    Ignore;  -- don't even attach to this device
}
```

Because it performs the `Ignore` check before attaching to the device, `usblauncher` doesn't have any device descriptors to match against. This means that it can look at only the vendor ID and device ID supplied by the stack when the device was physically attached to the system. Therefore, the `device` rule can't be made more specific about ignoring certain devices.

For rules at inner levels, the `Ignore` check is done after retrieving device descriptors, which means that you can make these rules more specific. At any inner level, the `Ignore` flag stops `usblauncher` from launching a driver and from matching the device with any subsequent rules.

```
product(0x05AC) {
    class(USB_CLASS_AUDIO, USB_AUDIO_SUBCLASS_CONTROL) {
        driver"io-audio -dipod busno=$(busno),devno=$(devno),\z
                cap_name=ipod-$(busno)-$(devno)";
    };
    class(USB_CLASS_HID) {
        driver"io-fs-media -dipod,$(IPOD_OPTS)";
    };
    class(USB_CLASS_MASS_STORAGE) {
        Ignore;
    };
};
```

You can include the `Default` flag in a `configuration` rule to limit driver matching to the device's default configuration, as demonstrated in the following example:

```
configuration(0x48E0) {
    Default;
    driver"io-fs-media -dipod,$(IPOD_OPTS)";
}
```

The `Never` flag means no device is matched. This flag lets you temporarily disable a rule. The device can match another rule specified later in the configuration file and have an alternative driver launched.

The `Always` flag means any device is matched (if it hasn't been matched with another rule already). For example, although the `class` rule in the following device specification indicates an interface class of `8`, the rule actually matches devices of any class:

```
device() {
    class(8) {
        Always;
        start"echo this is busno=$(busno) devno=$(devno) inserted";
     };
};
```

These last two flags provide a helpful way to test a rule during development.

### Multiple interface classes

You can specify multiple interface classes for one device or product rule, as in the sample iPod device specification that follows.

```
-- iPod
product(0x05AC, 0x1200, 0x12FF) {

    class(USB_CLASS_AUDIO, USB_AUDIO_SUBCLASS_CONTROL) {

        driver"io-audio -dipod busno=$(busno),devno=$(devno),\z
               cap_name=ipod-$(busno)-$(devno)";
    };

    class(USB_CLASS_HID) {
        driver"io-fs-media -dipod,transport=usb:busno=$(busno):\z
               devno=$(devno):\z
               audio=/dev/snd/ipod-$(busno)-$(devno),\z
               darates=+8000:11025:12000:16000:22050:24000,\z
               playback,acp=i2c:addr=0x11:path=/dev/i2c99,\z
               fnames=short,config=/etc/mm/ipod.cfg";
    };

    class(USB_CLASS_MASS_STORAGE) {
        Never;
    };
};
```

### Foreach rule

With the `foreach` rule, you can launch a particular driver for any device in a given list of devices. This rule saves you from having to specify many device (or product) rules that differ only by their device or vendor IDs, as seen in the example that follows:

```
char_devices = {
    device(0x0557, 0x2008);  -- ATEN_232A/GUC_232A
    device(0x2478, 0x2008);  -- TRIPP LITE U2009-000-R
    device(0x9710, 0x7720);  -- MOSCHIP 7720
    device(0x0403, 0x6001);  -- FTDI 8U232AM
    device(0x1199, 0x0120);  -- Sierra AirCard U595
    device(0x0681, 0x0040);  -- Siemens HC25
    device(0x1bc7, 0x1003);  -- Telit UC864E
    device(0x067b, 0x2303);  -- Prolific
}

foreach (char_devices) {
    driver" devc-serusb -d vid=$(vendor_id),did=$(product_id),\z
            busno=$(busno),devno=$(devno)";
}
```

The `foreach` rule also works with device-specification rules of inner scopes such as `configuration` or `interface`.

### Driver and start functions

You can specify a driver with either of the `driver` or `start` functions. The difference is that `driver` is used to launch programs that are expected to run as long as the

device is attached. These programs are usually daemons. Consider the following rule for launching a daemon:

```
device() {
    class(USB_CLASS_MASS_STORAGE) {
        driver"devb-umass $(DISK_OPTS) $(UMASS_OPTS),\z
                vid=$(vendor_id),did=$(product_id),busno=$(busno),\z
                devno=$(devno),interface=$(interface)";
    };
};
```

The `start` function is used for launching short-lived commands, such as the `echo` command shown in the following example:

```
device() {
    class(USB_CLASS_MASS_STORAGE) {
        start"echo Device busno=$(busno) devno=$(devno) inserted";
    };
};
```

## USB descriptors

When your system is acting as the USB device in a link with another system acting as the USB host, you can specify which USB descriptors you want to expose to the host. These USB descriptors indicate your system's device function (e.g., mass storage, serial data transfer).

Defining your own USB descriptors allows you to override the default descriptors any time you use `usblauncher` to restart the USB stack in device mode. The descriptor overriding is done by issuing the `start_stack::device,`*n* command to the *device control object* (p. 26). Here, *n* is an index into the one-based list of descriptors (i.e., `1` refers to the first entry). In the QNX SDP image, this list is found in the main configuration file (`rules.lua`) but the individual descriptors are stored in separate Lua files (e.g., `umass.lua`, `usbser.lua`).

You can change the USB descriptors exposed to a USB host by issuing a `start_stack` command after the USB stack is started but before the link with the host is enabled (i.e., before the host can detect the attachment and enumerate the device). Because `usblauncher` doesn't monitor hardware for device or cable attachments, it's up to client applications to detect when a USB host is trying to set up a link and to then decide on the appropriate USB device role for the system and expose the device descriptors based on this role (see "*Support for USB On-The-Go (OTG)* (p. 21)" for a typical device role-swapping scenario).

### USB descriptor levels

USB descriptors are expressed in a hierarchy that defines different levels for storing the communication and data properties of a device. Each level of depth is more specific than the last, meaning it describes a more specialized set of USB properties. The four levels in USB descriptors are:

**`device`**

Represents the entire device. A device can have only one `device` descriptor.

**`configuration`**

Specifies details on device power usage and stores `interface` descriptors. Currently, `usblauncher` supports only one configuration per device.

**`interface`**

Groups `endpoint` descriptors to define a single device feature.

**`endpoint`**

Acts as a single channel for USB data, similar to a socket in a program. The USB host uses `endpoint` information to determine bandwidth requirements.

You can find detailed information on these four descriptor levels, including lists of USB fields applicable to each level, on the *USB Descriptors* page of the *USB in a NutShell* online blog.

### Variables

You can store common USB descriptor values in variables and later refer to these variables when defining USB fields. Here are the variables defined at the top of `usbser.lua`:

```
--  Sample USB descriptors for a USB Serial device

USB_CONFIG_SELFPOWERED          = 0xC0
USB_CONFIG_REMOTEWAKEUP         = 0x20
USB_MAX_CURRENT                 = 0
```

### Class-specific descriptors

Some interfaces require *class-specific descriptors*, which provide information specific to the communication or data class supported by the interface. Class-specific descriptors are expressed as the concatenation of all *functional descriptors* for the class. Functional descriptors provide information such as call management capabilities, supported network control notifications, and so on.

In `usbser.lua`, we define class-specific descriptors for a communication device. Because there are many class-specific descriptor formats, which are defined outside the core USB specification, we don't include any helper templates for them. Therefore, you need to define these descriptors as an array of bytes, in the same order they would be sent over the wire. You can refer to this array later when defining the descriptor for an interface.

There is one exception to the strict byte-by-byte copying and data transfering done by `usblauncher`: when it finds a string in your array of bytes, `usblauncher` converts the string to double-byte format, stores it in a table of strings, and then replaces the raw bytes in the array with the corresponding index from the strings table. For all other bytes, you must hand-code them:

```
comm_descriptors = {
-- Header Functional Descriptor
    0x05,                   -- bFunctionLength
    0x24,                   -- bDescriptorType
    0x00,                   -- bDescriptorSubType
    0x10,                   -- bcdCDC   (LSB)
    0x01,                   -- bcdCDC   (MSB)

-- Call Managment Functional Descriptor
    0x05,                   -- bFunctionLength
    0x24,                   -- bDescriptorType
    0x01,                   -- bDescriptorSubType
    0x00,                   -- bmCapabilities - ENOSUP
    0x01,                   -- bDataInterface

-- Abstract Control Model Function Descriptor
    0x04,                   -- bFunctionLength
    0x24,                   -- bDescriptorType
    0x02,                   -- bDescriptorSubType
    0x02,                   -- bmCapabilities

-- Abstract Control Model Union Descriptor
    0x05,                   -- bFunctionLength
    0x24,                   -- bDescriptorType
    0x06,                   -- bDescriptorSubType
    0x00,                   -- bControlInterface
    0x01,                   -- bSubordinate interface
}
```

### Descriptor templates

Before you can fill in the templates that correspond to the four USB descriptor levels, you must first create an empty table:

```
usbser = {}
```

You can now define the USB device properties that you want to expose to a USB host. The `Device{}` construct lets you specify device descriptor fields, which list manufacturer information and the device's supported USB classes and protocols. The sample `rules.lua` file provides some *predefined variables* (p. 40) for USB class and subclass types. To improve readability, you can refer to these variables instead of putting in literal integer values. In all descriptor templates, the `bLength` and `bDescriptorType` fields are filled in for you. Depending on the descriptor type, other fields may also be filled in.

When you assign strings to the table entries for the `iManufacturer`, `iProduct`, and `iSerialNumber` device descriptor fields, `usblauncher` stores a double-byte version of each string in a special strings table and replaces the literal values of those

entries with the associated indexes in the strings table. Most likely, you'll want to define strings for these previous three fields along with values for these other fields:

- `bDeviceClass`
- `bDeviceSubClass`
- `idVendor`
- `idProduct`

The *USB Descriptors* page explains the meaning of all *device descriptor fields*.

You must assign the filled-in `Device{}` template to the `device` entry in the table that you created:

```
usbser.device = Device{
    bDeviceClass = USB_CLASS_COMMS,
    bDeviceSubClass = USB_COMMS_SUBCLASS_MODEM,
    bDeviceProtocol = 0x0,
    bMaxPacketSize = 64,
    idVendor = 0x1234,
    idProduct = 0x4,
    bcdDevice = 0x0100,
    manufacturer = 'Acme Corporation',
    product = 'CDC Serial Peripheral',
    serial = 'xxxx-xxxx-xxxx',
    bNumConfigurations = 1,
}
```

This release supports only one configuration per device, so `bNumConfigurations` must be set to `1`. You must define separate configuration descriptors for Full Speed and for High Speed connections, but each USB device will use only one configuration in an active link.

Next, specify the remaining descriptor types as part of one configuration, using the `Config{}` construct. Fields like `bmAttributes` and `bMaxPower` take values described in the *configuration descriptor fields* section of the *USB Descriptors* page. You can define a string for the `iDescription` field, in which case `usblauncher` stores a double-byte version of this string in the strings table and writes the appropriate table index in place of the string bytes when sending data over the wire. The `wTotalLength` and `bNumInterfaces` fields are also filled in by `usblauncher`.

To specify the full-speed configuration, you must assign the filled-in `Config{}` template to the `fs_config` entry in the main table:

```
usbser.fs_config = Config{  -- full speed
    bConfigurationValue = 1,
    bmAttributes = USB_CONFIG_SELFPOWERED,
    bMaxPower = USB_MAX_CURRENT,
    description = 'Default Configuration',
    interfaces = {
        Association{
```

The high-speed configuration is specified in a similar manner; you must complete a `Config{}` template but assign it to the `hs_config` entry:

```
usbser.hs_config = Config{  -- high speed
    bConfigurationValue = 1,
    bmAttributes = USB_CONFIG_SELFPOWERED,
    bMaxPower = USB_MAX_CURRENT,
    description = 'Default Configuration',
    interfaces = {
        Association{
```

Next, define a nested table in the `interfaces` entry, in each of the full-speed and high-speed configurations. You can group multiple interfaces together by defining an Interface Association Descriptor (IAD), using the `Association{}` construct. An IAD has a class, subclass, protocol, and optionally a string to describe it, followed by a list of interfaces, each of which is defined by a separate `Iface{}` template:

```
        Association{
            bInterfaceClass = USB_CLASS_COMMS,
            bInterfaceSubClass = USB_COMMS_SUBCLASS_MODEM,
            bInterfaceProtocol = 0x0,
            description = 'Serial Port Interface',
            interfaces = {
                Iface{
```

When you assign a string to the IAD `description` entry, `usblauncher` writes the table index for this string into the `iInterface` descriptor field for each interface defined within the IAD.

An `Iface{}` defines an interface's USB class, subclass, and protocol as well as a `bAlternateSetting` entry, which determines the autogenerated value written into the `bInterfaceNumber` descriptor field. The interface number increases by `1` for each new interface whose `bAlternateSetting` value is `0`. When this value is `1`, the interface is an alternate interface and therefore has the same interface number as the previous interface.

In both the full-speed and high-speed configurations, you must define a Communication Class interface to handle device management and possibly call management. You can define any number of Data Class interfaces to support the transfer of data with a certain structure and usage. Each such interface is specified in its own `Iface{}` tag.

For the Communication Class interfaces, we assign *class-specific descriptors* (p. 44) by setting the `class_specific` entry to the array of bytes containing those descriptors (which we defined earlier):

```
                Iface{
                    bInterfaceClass = USB_CLASS_COMMS,
                    bInterfaceSubClass = USB_COMMS_SUBCLASS_MODEM,
                    bInterfaceProtocol = 0x0,
                    bAlternateSetting = 0,
                    description =
                      'Serial Port Communication Class Interface',
                    class_specific = comm_descriptors,
                    endpoints = {
```

In our example, the Data Class interfaces don't use class-specific descriptors, so the `class_specific` entry isn't set for those interfaces.

Finally, you must provide a list of endpoints in each interface. For the Communication Class interfaces, you can use the `InterruptIn{}` construct to set the `wMaxPacketSize` and `bInterval` endpoint descriptor fields (as defined in the USB specification):

```
endpoints = {
    InterruptIn{wMaxPacketSize = 8,
                           bInterval = 8}
}
```

In this case, `usblauncher` fills in the `bEndpointAddress` and `bmAttributes` fields, which are sent along with the other endpoint information to the USB host.

For the Data Class interfaces, you can use the `BulkIn{}` and `BulkOut{}` constructs to set limits on the packet sizes for outgoing and incoming bulk data. Each construct takes only the `wMaxPacketSize` field:

```
endpoints = {
    BulkOut{wMaxPacketSize = 64},
    BulkIn{wMaxPacketSize = 64},
}
```

When you define maximum packet sizes for bulk data transfer on an endpoint, `usblauncher` sets the `bInterval` descriptor field to `0` and calculates the `bEndpointAddress` and `bmAttributes` fields.

For all other endpoint properties, you can use either the `EndpointIn{}` or the `EndpointOut{}` construct and then set the `bmAttributes`, `wMaxPacketSize`, and `bInterval` fields according to the USB specification. In this case, `usblauncher` calculates the `bEndpointAddress` field.

If any expected field is left undefined, `usblauncher` logs a warning message and assigns `0` in place of the missing field.

### Bypassing helper templates to fully define USB descriptors

The helper templates don't cover every USB device scenario because `usblauncher` fills in many descriptor fields for you, at all hierarchy levels in the USB descriptors. For full control over what's stored in the descriptors, you can define their raw bytes, as demonstrated in the included `raw_desc_usbser.lua` file. If you want to fully define USB descriptors byte by byte, you must set certain keys in the descriptor table:

- `.device`
- `.fs_config`
- `.hs_config`

- `.strings`

Any descriptor table that you create must be referenced in the list of descriptor overrides (i.e., the `Device_Stack.descriptors` list in the main configuration file). For example, to use the `serial_raw_desc{}` table defined in `raw_desc_usbser.lua`, you need to put an entry for that table in the descriptors list:

```
descriptors = { iap2, iap2ncm, serial_raw_desc };
```

For two-byte fields such as `wMaxPacketSize` and `wTotalLength`, you can access their least-significant and most-significant bytes by using the `lsb()` and `msb()` functions, which is necessary when defining these fields byte by byte.

The `.device`, `.fs_config`, and `.hs_config` keys must be Lua strings (which aren't null-terminated, as are C strings). You can use Lua's `string.char()` function to convert an array of bytes into a string, the expected type for these keys. However, the `.strings` key must be a table of strings. We provide the `double_byte()` helper function so you don't have to hand-code the double-byte representations of these strings when expressing them as raw bytes.

## Supported third-party applications and protocols

The `usblauncher` service supports several third-party applications and protocols for accessing content on attached devices. To work with these applications and protocols, you can define configuration rules that swap the role of the USB stack, probe a device for its iAP2 protocol support, or request a device to switch to another mode.

### Enabling Apple CarPlay

To support Apple CarPlay, you must define both the `Host_Stack` and `Device_Stack` rules. Also, your system must be running the USB stack when the Apple device (e.g., an iPhone 5) is first connected, so `usblauncher` can learn of the connected device. If you're running the stack in host mode, you must restart it in device mode. To enable this role swapping, add the `RoleSwap_DigitaliPodOut` flag to the matching rule for Apple devices (which have a vendor ID of `0x05AC`):

```
product(0x05AC, 0x1200, 0x12FF) {

    RoleSwap_DigitaliPodOut;

    -- drivers below are matched when the role swap request fails
    class(USB_CLASS_AUDIO, USB_AUDIO_SUBCLASS_CONTROL) {
        driver"io-audio -dipod busno=$(busno),devno=$(devno),\z
                cap_name=ipod-$(busno)-$(devno)";
    };
    class(USB_CLASS_HID) {
        driver"io-fs-media -dipod,$(IPOD_OPTS)";
    };
};
```

To obtain the drivers needed to support iOS in automotive systems, contact your Project Manager or QNX customer support.

In the failure-handling section of this rule, the `HID` class must specify a driver or script so that `usblauncher` can execute the role-swap request specified above. If you don't want your system to support Apple devices when continuing to run as the USB host after a failed role-swap request, insert a "stub" command such as "`echo HID for iPod`" within the `HID` class rule.

If you provide descriptor overrides by defining the `Device_Stack.descriptors` list, `usblauncher` will use the descriptors from the *second item* in this list to override the device stack's internal settings when restarting the USB stack in device mode after successfully processing the `RoleSwap_DigitialiPodOut` flag.

### Enabling iAP2 in client mode

To support iPod Accessory Protocol 2 (iAP2) in client mode, you must define the `Host_Stack` and `Device_Stack` rules and launch the USB stack before connecting any devices that support iAP2 for playback control. To verify that a device supports iAP2 and request the necessary role-swapping of the USB stack, add the `RoleSwap_AppleDevice` flag to the matching rule for Apple devices:

```
product(0x05AC, 0x1200, 0x12FF) {

    RoleSwap_AppleDevice;

    -- drivers below are matched when the role swap request fails
    class(USB_CLASS_AUDIO, USB_AUDIO_SUBCLASS_CONTROL) {
        driver"io-audio -dipod busno=$(busno),devno=$(devno),\z
                cap_name=ipod-$(busno)-$(devno)";
    };
    class(USB_CLASS_HID) {
        driver"io-fs-media -dipod,$(IPOD_OPTS)";
    };
};
```

In the failure-handling section of this rule, the `HID` class must specify a driver or script so that `usblauncher` can execute the iAP2 probe and role-swap request specified above. If you don't want your system to support Apple devices when continuing to run as the USB host after a failed verification of iAP2 support or a failed role-swap request, insert a "stub" command such as "`echo HID for iPod`" within the `HID` class rule.

If you define the `Device_Stack.descriptors` list, `usblauncher` will use the descriptors from the *first item* in this list to override the device stack's internal settings when restarting the USB stack after successfully processing the `RoleSwap_AppleDevice` flag.

### Probing for iAP2 support

The `Probe_iAP2` flag doesn't trigger a role-swap request but allows you to probe a device for its iAP2 support. For example, you may want to start a different driver depending on the iAP version supported:

```
product(0x05AC, 0x1200, 0x12FF) {

    Probe_iAP2;

    -- drivers below are matched when iAP2 isn't supported
    -- in client mode on the target
    class(USB_CLASS_AUDIO, USB_AUDIO_SUBCLASS_CONTROL) {
        driver"io-audio -dipod busno=$(busno),devno=$(devno),\z
                cap_name=ipod-$(busno)-$(devno)";
    };
};
```

### Using multiple flags

You can use combinations of these role-swapping and device-probing flags. Their order has no significance, because `usblauncher` always processes the flags in this sequence:

1. `RoleSwap_DigitaliPodOut`
2. `Probe_iAP2`
3. `RoleSwap_AppleDevice`

The `RoleSwap_AppleDevice` flag triggers a probe for iAP2 support on the connected device, before the actual role-swapping is attempted, so the `Probe_iAP2` is ignored if it's specified when this role-swapping flag is also specified.

If an iAP2 probe is attempted but no role swap happens, either because it wasn't requested or because the device doesn't support iAP2, the *device object* (p. 22) will contain the `iap` attribute. This attribute will be set to `2` if iAP2 support was confirmed or `-1` if iAP2 isn't supported or the probe failed.

### Launching a driver in host mode for Apple devices

You can define a custom function to choose between launching iAP1 or iAP2 drivers in host mode, based on the result of an iAP2 probe. This function can read the device object's `iap` attribute (which is set when the device is probed) and then generate a driver command based on that attribute's value:

```
product(0x05AC, 0x1200,0x12FF)
{
    Probe_iAP2;

    class(USB_CLASS_AUDIO, USB_AUDIO_SUBCLASS_CONTROL)
    {
        driver"io-audio <full options>"
```

```
    };
    class(USB_CLASS_HID) {
        custom = function(obj)
            if obj.EXTRA and obj.EXTRA:find('iap::2') then
                return "mm-ipod <put the full options here>";
            else -- for iAP1
                return "io-fs-media <put the full options here>";
            end
        end;
        driver"$(custom)";
    };
}
```

The Lua interpreter evaluates the function when expanding the `driver"$(custom)"` string, and then passes the resulting string to `usblauncher`, which executes it to launch the driver.

### Enabling MirrorLink

The MirrorLink solution lets you access applications on devices such as smartphones through a car infotainment system. To enable MirrorLink on an attached USB device, `usblauncher` can send that device a request to switch to a Network Configuration Management (NCM) personality. To support this MirrorLink command, you must define a table in the configuration file as follows:

```
MirrorLink = {
    version = 0x0101,
    vendor_id = 0xABAB,
    blacklist = {
        { vid = 0x0781; did = 0x74e0 }; -- SanDisk Sansa Fuze+
        { vid = 0x0951; did = 0x1624 }; -- Kingston DataTraveler G2
        { vid = 0x05ac;              }; -- any Apple device
    };
    timeout = 2000,
    ignore  = false,
    stall_reset = false
}
```

This table provides the field values for the MirrorLink command to enable the NCM personality. The `version` and `vendor_id` fields are two-byte integers. Each item in the `blacklist` list can be simply a vendor ID (`vid`) or both the `vid` and device ID (`did`). You should add blacklist items for all devices not suitable for MirrorLink; these devices won't receive requests to switch to an NCM personality.

The `timeout` field specifies how long to wait, in milliseconds, for a response to the MirrorLink command to enable NCM on a device. It also specifies how long to wait for a device that accepts the MirrorLink command to leave the bus.

Normally, a device leaves the bus and comes back with an NCM personality; however, some noncompliant USB devices may respond successfully to the MirrorLink command to enable NCM but they may never leave the bus. In this latter case, the timeout determines how long `usblauncher` waits before continuing to match and launch any drivers for devices that haven't left the bus.

You can set the `ignore` field to `true` to prevent `usblauncher` from matching the NCM interface to a driver rule, which would select a configuration and launch a driver for that interface. The `ignore` setting is useful if you're running a resident driver outside of `usblauncher` that manages the NCM interface.

If you do use `usblauncher` to start a driver for the NCM interface, you don't need to restart the USB stack in device mode (unlike when supporting CarPlay). For MirrorLink, the car infotainment system remains the USB host.

Setting `stall_reset` to `true` enables resetting the device if it responded with the USB STALL packet. Some devices don't behave correctly unless they're reset in this case.

# Chapter 3
# The mmcsdpub Publisher

The `mmcsdpub` device publisher retrieves and publishes information about cards inserted into SD readers.

This publisher depends on the following services:

**Drivers for SD card readers**

Depending on your system, you might use either a `devb-sdmmc` or a `devb-mmcsd` driver. Regardless of its exact type or your hardware, the SD driver runs continuously and manages the SD device paths and communication with cards.

**mcd**

The `mcd` utility mounts the filesystems of inserted SD (and MMC) cards by making requests to the SD driver. The mountpoints chosen by `mcd` depend on the rules in its mountpoint file.

## Architecture

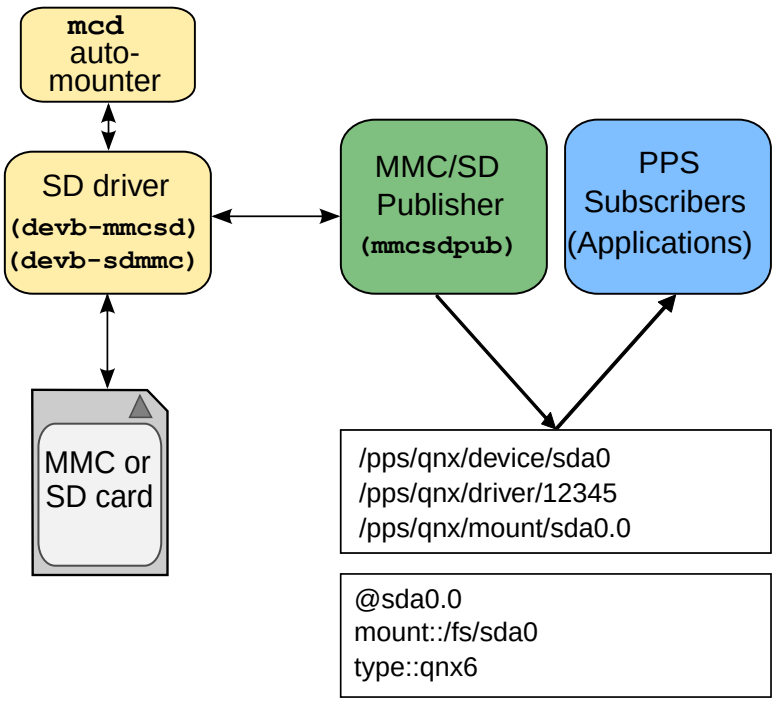


**Figure 2: MMC/SD publisher architecture**

The arrows between components in *Figure 2: MMC/SD publisher architecture* (p. 55) show information flow. For example, when SD cards are inserted into or removed from card reader slots, the SD driver updates the states of the SD device entries (e.g., `/dev/sda0` or `/dev/sdb0`). The `mcd` service monitors the states of the devices represented by those entries and when it notices that a card has been inserted, `mcd` sends mount requests to the SD driver process, which manages the device paths.

Meanwhile, `mmcsdpub` also monitors those same `/dev` entries and when it notices a state change, `mmcsdpub` communicates with the driver to retrieve the device and mount information, which it then writes to the appropriate PPS objects. Subscribed applications receive PPS updates when the content of those objects change, which occurs whenever SD or MMC cards are inserted or removed.

The `mmcsdpub` publisher works with setups where the device entries exist as long as the driver is running (even when the slots are empty) and also with setups where the device entries get created and deleted when cards are inserted and removed.

## Device object

Each device object that `mmcsdpub` writes to PPS contains the following fields:

| Name | Description | Type | Example |
|---|---|---|---|
| `bus` | Bus type | String | `SD` |
| `card_type` | Card type | String with one of the following values: `SD`, `MMC`, or `unknown` | `SD` |
| `device_id` | OEM device ID | Integer | `24` |
| `ecc_count` | Number of times the driver has reported Error Correction Code (ECC) errors. This field is present only when its value is nonzero. | Integer | `1` |
| `locked` | Whether the card is locked | Boolean: `0` if not locked or `1` if locked | `0` (not locked) |
| `present_at_startup` | Whether the card was present at startup (and not inserted by the user) | Boolean: `0` if not present or `1` if present | `0` (not present) |
| `product_name` | OEM product name | String | `SD8GB` |
| `serial_number` | Product serial number | Integer | `1526735628` |
| `slot_name` | Slot name passed to driver | String | `slot_1` |
| `speed` | Driver clock rate (in Hertz) | Integer | `48000000` |
| `status` | Error string. For “bad devices” (i.e., devices that `mmcsdpub` couldn't read), only this field is present. | String | `device not recognized` |
| `vendor_id` | Manufacturer ID | Integer | `65` |
| `write_protection` | Status of write-protect switch | Boolean: `0` if disabled or `1` if enabled | `0` (disabled) |

## Driver object

Each driver object that `mmcsdpub` writes to PPS contains the following fields:

| Name | Description | Type | Example |
|---|---|---|---|
| `PPS_DEVICE_ID` | Device object path | String | `/pps/qnx/device/sda0` |

## Mount object

Each mount object that `mmcsdpub` writes to PPS contains the following fields:

| Name | Description | Type | Example |
|---|---|---|---|
| `blocks_size` | Size of each block (in bytes) | Integer | `512` |
| `blocks_total` | Total number of blocks | Integer | `3805121` |
| `fs_type` | Filesystem type. This field is present only if the filesystem is mounted. | String with one of the following values: `dos (fat32)`, `qnx4`, `qnx6`, `nt`, or `unknown` | `dos (fat32)` |
| `label` | Filesystem label. This field is present only if the filesystem is mounted. | String | `Cottage Weekend 1` |
| `mount` | Mountpoint | String | `/fs/sdb0` |
| `partition` | Partition name. This field is present only if the mount object represents a partition. | String | `/dev/sdb0t12` |
| `partition_count` | Total number of partitions. This field is present only if the mount object represents an entire device and not a partition. | Integer | `1` |
| `partition_order` | Partition index. This field is present only if the mount object represents a partition. | Integer | `0` |
| `plugin_name` | Name of plugin supplying extra mount information, followed by mount fields returned by plugin | String in the following format:<br>`generic [<attr_name>::<value>]*`<br>The `generic` plugin is the only plugin supported for `mmcsdpub`. | `plugin_name::generic name::UNTITLED 1 id::72a354f7-906e-424f-985e-ab7a71b7971a` |
| `PPS_DRIVER_ID` | Driver object path | String | `/pps/qnx/driver/74550` |
| `PPS_RAWMOUNT_ID` | Mount object path | String | `/pps/qnx/mount/sdb0` |
| `raw` | Name of raw device | String | `/dev/sdb0` |

| Name | Description | Type | Example |
|---|---|---|---|
| `read_only` | Read-only status of device. This field is present only if the filesystem is mounted. | Boolean: `0` if writable or `1` if read-only | `1` (read-only) |
| `status` | Error string. This field is present only if an error occurred when accessing the card's mounted filesystem. | Integer followed by string, in the format: *`errno`* (*`str_error(errno)`*) | `302 (Corrupted file system detected)` |

## Command line for mmcsdpub

*Start mmcsdpub device publisher*

**Synopsis:**

```
mmcsdpub [-b] [-D] [-d] [-e]
    -f device_path [-f device_path]* [-l] [-m pps_path]
    [-p insertion:removal] [-s dll_path] [-v]
```

**Options:**

**-b**

Run `mmcsdpub` in the foreground (not background). This option is handy for debugging because you can press **Ctrl**–**C** to terminate the publisher.

By default, `mmcsdpub` runs in the background.

**-D**

Inform `mmcsdpub` of one or more devices managed by a `devb-sdmmc` driver.

This option applies to the devices named in `-f` options that appear later in the command line.

**-d**

Inform `mmcsdpub` of one or more devices managed by a `devb-mmcsd` driver.

This option applies to the devices named in `-f` options that appear later in the command line. A `devb-mmcsd` driver is used by default, so you need this option only if you used `-D` earlier to specify the alternative `devb-sdmmc` driver type for other devices.

**-e**

Enable auto-enumeration and unmounting of device partitions, based on the state of inserted media (i.e., cards in SD slots). When this option is used, `mmcsdpub` enumerates partitions on SD cards when they're ready and then unmounts the partitions when the cards are removed. This activity applies to device paths named in subsequent `-f` options. For example, consider the command line:

```
mmcsdpub -f /dev/sda0 -e -f /dev/sdb0
```

This command forces `mmcsdpub` to enumerate any partitions on the device represented by `/dev/sdb0` (much like running `mount -e /dev/sdb0`)

and to unmount these partitions when appropriate but not to enumerate or unmount the partitions represented by `/dev/sda0`.

Typically, `-e` is used when the `mmcsd normv` option is used for `devb-mmcsd`. This driver option keeps the `/dev` entry whether or not a card is inserted in the slot.

By default, auto-enumeration and unmounting of SD and MMC device partitions is disabled. Also, the `-e` option is ignored for a device path if `-D` is also used in front of it. For instance, the auto-enumeration wouldn't apply to the device path named in this command line:

```
mmcsdpub -D -e -f /dev/sda0
```

**`-f` *device_path***

Monitor the specified device path (i.e., `/dev` entry) for state changes to a card reader. You can name multiple devices in separate `-f` options to make `mmcsdpub` monitor and communicate with a select group of devices to obtain information on inserted SD or MMC cards.

This flag and a device entry must be specified at least once on the command line.

**`-l`**

Log messages to `sloginfo` instead of standard out.

By default, `mmcsdpub` logs messages to standard out.

**`-m` *pps_path***

The PPS directory path. The subdirectories for storing the device, driver, and mount objects are located in this directory. The default is `/pps/qnx/`.

**`-p` *insertion:removal***

The insertion and removal polling intervals (in milliseconds), which are how often `mmcsdpub` checks for SD card insertions and removals. Often, you'll want the insertion interval to be shorter than the removal interval because detecting new SD cards and publishing their information through PPS has higher priority than cleaning up PPS objects for cards that have been removed.

The default insertion interval is `1000 ms`; the default removal interval is `2000 ms`.

**`-s` *dll_path***

The plugin path. At startup, `mmcsdpub` looks in this path for plugins it can load and then use to provide more detailed device information (see *Plugins* (p. 14)).

If this option isn't specified, no plugins are loaded.

**`-v`**

Increase output verbosity. The `-v` option is cumulative, so you can use several `v`'s to increase verbosity. Setting one `v` logs SD and MMC card insertions and removals. Setting two `v`'s adds the logging of PPS object creation and deletion. Setting three `v`'s logs more detailed events as well as errors that are less severe.

Output verbosity is handy when you're trying to understand the operation of `mmcsdpub`. However, when many `v`'s are used, the logging becomes quite significant. The verbosity setting is good for systems under development but probably shouldn't be used in production systems or during performance testing.

**Description:**

You should start `mmcsdpub` with an explicit command only if the process terminates unexpectedly. Before trying to start `mmcsdpub` manually, always confirm that the process isn't already running by checking the list of active processes with *pidin* or *ps*.

The `mmcsdpub` command starts the SD device publisher, which monitors SD entries in `/dev` and publishes up-to-date information on SD (or MMC) cards through PPS.

You must provide one or more *device paths* (through `-f` options) to tell `mmcsdpub` which device entries to monitor for detecting state changes to card readers. You can put `-D` or `-d` options in front of device paths named with `-f`, to indicate which driver manages certain device paths. Through other command options, you can set how often `mmcsdpub` checks for SD card insertions and removals and also adjust how `mmcsdpub` logs error and event information.

The `mmcsdpub` service runs as a self-contained process that doesn't require any user input or accept any commands. It has no client utility for performing device-publishing tasks on request or for adjusting any of its settings. To reconfigure `mmcsdpub`, you must change the options in its command line and restart the service. We recommend putting the `mmcsdpub` command line in a startup script (e.g., `startup.sh`) to launch the publisher automatically during bootup.

# Chapter 4
# The cdpub Publisher

The `cdpub` device publisher retrieves and publishes information about CDs.

This publisher depends on the following services:

**devb-eide**

The CD driver. You may use a different driver on your system (e.g., `devb-eide-mmx`), but regardless of the hardware, the CD driver process runs continuously and manages the CD device path and communication with all CD device types (e.g., DVDs, audio CDs, CD-ROMs).

**mcd**

The `mcd` utility mounts the filesystems of inserted CDs by making requests to the CD driver. The mountpoints chosen by `mcd` depend on the rules in its mountpoint file.

## Architecture

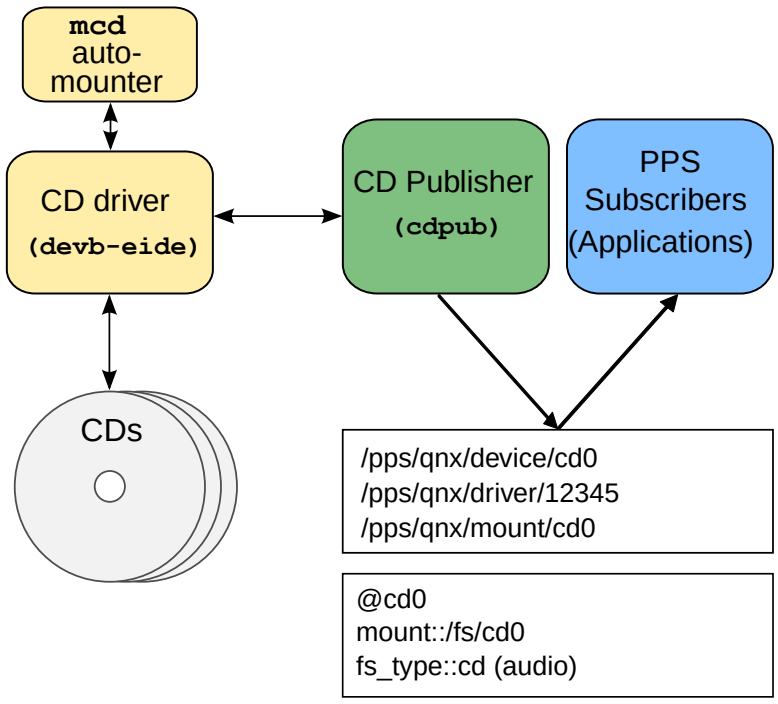


**Figure 3: CD publisher architecture**

The arrows between components in *Figure 3: CD publisher architecture* (p. 65) show information flow. For example, when discs are inserted into or removed from the CD drive, the driver updates the state of the CD device object (i.e., `/dev/cd0`). That filesystem entry exists as long as the driver is running, even when the CD drive is empty. Meanwhile, `mcd` monitors the device object for state changes and when it notices that a new CD has been inserted, `mcd` consults its mount rules (based on the device's type) and if warranted, sends a mount request to the `devb-eide` driver process, which manages the device path.

The `cdpub` publisher also monitors the `/dev` entry (i.e., device object) and when it notices a state change, `cdpub` communicates with the `devb-eide` process to obtain the latest device and mount information, which it then writes to the appropriate PPS objects. Subscribed applications receive PPS updates when CDs are inserted or removed.

## Device object

The device object that `cdpub` writes to PPS contains the following fields:

| Name | Description | Type | Example |
|---|---|---|---|
| `bus` | Bus type | String | `CD` |
| `device_type` | Device type | String with one of the following values: `unknown`, `cdrom`, or `changer`. Presently, support for `changer` is experimental. | `cdrom` |
| `media_type` | Media type | String with one of the following values: `unknown`, `blank`, `audiocd`, `dvd`, or `cd` | `audiocd` |
| `present_at_startup` | Indicates if media was present before driver handler started | Boolean: `0` means the media wasn't present when the driver started; `1` means it was present | `0` |
| `prevent_media_removal` | Indicates if media can be removed | Boolean: `0` means media can be removed; `1` means it can't be removed | `0` |
| `serial` | Device serial number reported by CD drive | String whose format is device-specific | `FUJITSU TEN DVD-ROM DV-05FT2` |
| `slot_state` | Active slot mechanical state | String with one of the following values: `unknown`, `empty`, `loading`, `ready`, `ejecting`, or `ejected` | `ready` |
| `slot_status` | Slot status based on sense code received from drive | String in the form: `key/asc/ascq`. These device-specific values are reported by the CD drive and collectively convey the slot status. | `0/0/0` |
| `temperature_simulation` | Indicates temperature value and state are simulated. | Integer set to `1` when temperature is simulated. Otherwise, this field isn't present. | `1` |
| `temperature_state` | Temperature state provided by CD drive. | String with one of the following values: `unknown`, `normal`, `overtemp`, `undertemp`, or `unsupported`. Which of these values can be reported depends on the device. | `normal` |

| Name | Description | Type | Example |
|---|---|---|---|
| `temperature_value` | Temperature provided by CD drive. Present only if drive can report temperature. The units depend on the driver's firmware. | Integer whose range is device-specific | 24 |

## Device control object

Applications can perform actions on a CD by writing commands to the `cdpub` device control object. The `cdpub` process creates this PPS server object. By default, the object's path is `/pps/qnx/device/cd0_ctrl` but you can change this path through `cdpub` command options. When `cdpub` receives updates (i.e., data from the object), it executes the specified commands.

Commands must be written in the following format:

```
<name>::<parameter>
```

For details on publishing data to PPS objects, refer to the *Persistent Publish/Subscribe Developer's Guide*. Applications can publish the following commands:

| Command | Description | Parameter type and purpose |
|---|---|---|
| `eject` | Eject the device | Integer indicating the slot number (see *Footnote.*) |
| `enable` | Request that the drive enable or disable media access operations | Integer set to one of the following values: `0` (to stop the disc), `1` (to start the disc and read the TOC), `2` (to stop the disc and eject it if possible), or `3` (to load the disc) |
| `load` | Load the device | Integer indicating the slot number (see *Footnote.*) |
| `power_condition` | Request that the drive be put into a specific power state | Integer between the values of `0` and `5`. The power states supported are vendor-specific. |
| `prevent_media_removal` | Adjust setting for preventing disc from being ejected | Integer set to one of the following values: `0` to allow disc removal or `1` to prevent it |
| `reload` | Reload the device | Integer indicating the slot number (see *Footnote.*) |
| `temperature_simulation` | Enable or disable simulation of temperature value and state | Integer set to one of the following values: `0` to enable simulation or `1` to disable it |
| `temperature_state` | Set temperature state (in simulation) | Integer set to one of the following values: `0` for `unknown`, `1` for `normal`, `2` for `overtemp`, `3` for `unsupported`, or `4` for `undertemp` |
| `temperature_value` | Set temperature value (in simulation) | Integer whose range is device-specific |
| `unload` | Unload the device | Integer indicating the slot number (see *Footnote.*) |

[1] For single-slot devices, this parameter isn't required but you must put the token seperator (`::`) after the command name for the command to be properly parsed.

## Driver object

The driver object that `cdpub` writes to PPS contains the following fields:

| Name | Description | Type | Example |
|---|---|---|---|
| `PPS_DEVICE_CTRL_ID` | Device control object path | String | `/pps/qnx/device/cd0_ctrl` |
| `PPS_DEVICE_ID` | Device object path | String | `/pps/qnx/device/cd0` |

## Mount object

The mount object that `cdpub` writes to PPS contains the following fields:

| Name | Description | Type | Example |
|---|---|---|---|
| `content_type` | Type of content on the disc. | String with one of the following values: `data`, `vcd`, `svcd`, `audiocd`, `dvdaudio`, or `dvdvideo` | `audiocd` |
| `fs_type` | Filesystem type. This field is present only if the filesystem is mounted. | String in the following format: `cd (<type>)`, where `<type>` could be `audio`, `joliet`, `iso9660`, or other values. | `cd (audio)` |
| `label` | Label of the partition. This field is present only if a label is found. | String | `The Doors Greatest Hits` |
| `mount` | Mountpoint. This field is present only if the filesystem is mounted. | String | `/fs/cd0` |
| `plugin_name` | Name of plugin supplying extra mount information, followed by mount fields returned by plugin | String in the following format:<br>`<pname> [<attr_name>::<value>]*`<br>Here, `<pname>` is one of the following plugin names: `audiocd`, or `generic`.<br>The set of attributes written into this field depends on the plugin. | `plugin_name::audiocd`<br>`album::Innervisions`<br>`artist::Stevie Wonder`<br>`dts::0`<br>`id::f510df13`<br>`toc::150 15001 27891 45959 61580 80504 97768 114709 134020 151840 170834 189820 210820 232075 244826 263058 286468 297515 310984 324080` |
| `PPS_DRIVER_ID` | Driver object path | String | `/pps/qnx/driver/64315` |
| `raw` | Name of raw device | String | `/dev/cd0` |
| `status` | Error string. This field is present only if an error occurred when accessing the CD's mounted filesystem. | Integer followed by string, in the format: *`errno (str_error(errno))`* | `302 (Corrupted file system detected)` |

## Command line for cdpub

*Start cdpub device publisher*

**Synopsis:**

```
cdpub [-b] -f raw_device[:cam] [-l] [-m pps_path]
      [-n iterations] [-p insertion:removal] [-r scope:init]
      [-s dll_path] [-v]
```

**Options:**

**-b**

Run `cdpub` in the foreground (not background). This option is handy for debugging because you can press **Ctrl**–**C** to terminate the publisher.

By default, `cdpub` runs in the background.

**-f** ***raw_device[:cam]***

Device path of the CD drive to monitor for state changes. You can name only one device, with or without its device name and unit number: `/dev/cd0` or `/dev/cd0:/dev/cam0/000`.

The CD driver creates the `/dev/cam0/000` entry, which gives `cdpub` another device path for issuing commands (e.g., eject, load) when the path of the `/dev/cd0` path is busy (perhaps due to a filesystem operation).

This flag and one of the two listed device paths must be specified once on the command line; `cdpub` ignores extra `-f` options.

**-l**

Log messages to `sloginfo` instead of standard out.

By default, `cdpub` logs messages to standard out.

**-m** ***pps_path***

The PPS directory path. The subdirectories for storing the device, device control, driver, and mount objects are located in this directory. The default is `/pps/qnx/`.

**-n** ***iterations***

Number of polling intervals (i.e., CD drive state readings) to skip before checking the device temperature. The time between successive polling

intervals can be one of two values: either the insertion polling interval or the removal polling interval, depending on the CD drive state. After querying the device for its temperature, `cdpub` updates the `temperature_state` and `temperature_value` attributes in the device object.

By default, `cdpub` skips `5` polling intervals between temperatures checks.

**`-p`** ***insertion:removal***

The insertion and removal polling intervals (in milliseconds), which are how often `cdpub` checks for CD insertions and removals. Often, you'll want the insertion interval to be shorter than the removal interval because detecting new CDs and publishing their information through PPS has higher priority than cleaning up PPS objects for CDs that have been removed.

The default insertion interval is `1000 ms`; the default removal interval is `2000 ms`.

**`-r`** ***scope:init***

Removal-prevention settings for CD devices. Both settings are Boolean, so they must be either `0` or `1`.

For *scope*, a value of `0` specifies a local scope for the removal-prevention setting, which means `cdpub` (and not the actual device) enforces the removal-prevention policy. A value of `1` makes `cdpub` query the device for its own removal-prevention setting before it attempts to update or report the setting when requested by an application.

For *init*, a value of `0` makes `cdpub` allow device removals when it starts, while a value of `1` makes it prevent device removals. Applications can change the removal-prevention setting by writing a command to the device control object (see *Device control object* (p. 69) for more information).

The default `cdpub` behavior is to use a local scope and to allow device removal.

**`-s`** ***dll_path***

The plugin path. At startup, `cdpub` looks in this path for plugins it can load and then use to provide more detailed device information (see *Plugins* (p. 14)).

If this option isn't specified, no plugins are loaded.

**`-v`**

Increase output verbosity. The `-v` option is cumulative, so you can use several `v`'s to increase verbosity. Setting one `v` logs CD insertions and removals. Setting two `v`'s adds the logging of PPS object creation and deletion. Setting three or four `v`'s logs more detailed events as well as errors that are less severe.

Output verbosity is handy when you're trying to understand the operation of `cdpub`. However, when many `v`'s are used, the logging becomes quite significant. The verbosity setting is good for systems under development but probably shouldn't be used in production systems or during performance testing.

**Description:**

You should start `cdpub` with an explicit command only if the process terminates unexpectedly. Before trying to start `cdpub` manually, always confirm that the process isn't already running by checking the list of active processes with *pidin* or *ps*.

The `cdpub` command starts the CD device publisher, which monitors the CD drive and publishes up-to-date information on CD devices through PPS.

You must name the raw device (i.e., `/dev/cd0`) to tell `cdpub` which device path to use for monitoring the CD drive state. Through other command options, you can set how often `cdpub` checks for CD insertions and removals and specify the default removal-prevention policy. You can also adjust how `cdpub` logs error and event information.

The `cdpub` service runs as a self-contained process that doesn't require any user input or accept any commands. It has no client utility for performing device-publishing tasks on request or for adjusting any of its settings. To reconfigure `cdpub`, you must change the options in its command line and restart the service. We recommend putting the `cdpub` command line in a startup script (e.g., `startup.sh`) to launch the publisher automatically during bootup.

# Index


## A

Apple CarPlay 49
    enabling 49

## C

cdpub 65, 67, 69, 70, 71, 72
    command line 72
    device control object 69
    device object 67
    driver object 70
    mount object 71
    overview 65

## D

device drivers 11
device publishers 9, 10, 11, 13, 14
    device object management 11
    mountpoint management 11
    overview 9
    plugins, See plugins
    PPS objects 13
    prerequisite services 10
Disabling MOUNT_FSYS rule in mcd configuration file 35
Disabling usblauncher mounting of filesystems 35

## I

iAP2 50, 51
    enabling in client mode 50
    launching a driver in host mode 51
    probing a device for its support 51
io-usb 17
io-usb-dcd 17

## L

Lua 36
    language 36

## M

mcd 11
MirrorLink 52
mmcsdpub 55, 57, 58, 59, 61
    command line 61
    device object 57
    driver object 58
    mount object 59
    overview 55
Mounting USB device filesystems 34

## O

OTG support 21

## P

plugins 14, 15
    manager 14
    ratings 15
    selection 15
    support for device publishers 15

## R

running a device publisher 10

## S

sending commands to a CD 69
sending commands to a USB port 26
starting a version of the USB stack 26
starting cdpub 72
starting mmcsdpub 61
starting usblauncher 31
system services required for cdpub 65
system services required for mmcsdpub 55
system services required for usblauncher 17

## T

Technical support 8
Typographical conventions 6

## U

USB descriptors 43, 44, 45, 48
    class-specific descriptors 44
    defining raw bytes 48
    descriptor templates 45
    levels 43
    variables 44
USB device mode 17
    drivers 17
    stack service, See io-usb-dcd
USB host mode 17, 32
    command-line option for disabling 32
    drivers 17
    stack service, See io-usb
USB matching rules 38, 39, 40, 42
    arguments 39
    driver and start functions 42
    flags 40
    foreach rule 42
    levels 38
    multiple interface classes 42

USB matching rules *(continued)*
variables 40
usblauncher 17, 21, 22, 26, 28, 29, 31, 34, 36
command line 31
device control object 26
device object 22
driver object 28
mount object 29
usblauncher *(continued)*
on-the-go (OTG) support 21
overview 17
stack modes, See USB device mode
using with mcd 34
usblauncher configuration files 36, 43
specifying device functions, See USB descriptors