QNX® PPS

QNX Persistent Publish/Subscribe Developer's Guide

For QNX® Neutrino® 6.4.x

Published under license by:

QNX Software Systems International Corporation
175 Terence Matthews Crescent
Kanata, Ontario
K2M 1W8
Canada
Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: `info@qnx.com`
Web: `http://www.qnx.com/`

Electronic edition published December 01, 2009.

Contents

About This Guide

The *QNX PPS Developer's Guide* includes:

- QNX PPS Service — an introduction to the QNX Persistent Publish/Subscribe service, and how to run it
- Objects and their Attributes — a description of the PPS service's objects and their attributes
- Persistence — How PPS manages persistence
- Publishing — how to publish to PPS
- Subscribing — how to subscribe to PPS
- Options and Qualifiers — pathname open options, and object and attribute qualifiers
- *ppsparse()* — a PPS API function that parses an object read from PPS

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**
Environment variables	**PATH**
File and pathnames	`/dev/null`
Function names	*exit()*
Keyboard chords	Ctrl-Alt-Delete
Keyboard input	**something you type**
Keyboard keys	Enter
Program output	**login:**
Programming constants	NULL
Programming data types	**unsigned short**
Programming literals	**0xFF**, **"message string"**
Variable names	*stdin*

continued...

Reference	Example
User-interface components	**Cancel**

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.

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

WARNING: 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 use a forward slash (`/`) as a delimiter in *all* pathnames, including those pointing to Windows files.

We also generally follow POSIX/UNIX filesystem conventions.

Technical support options

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

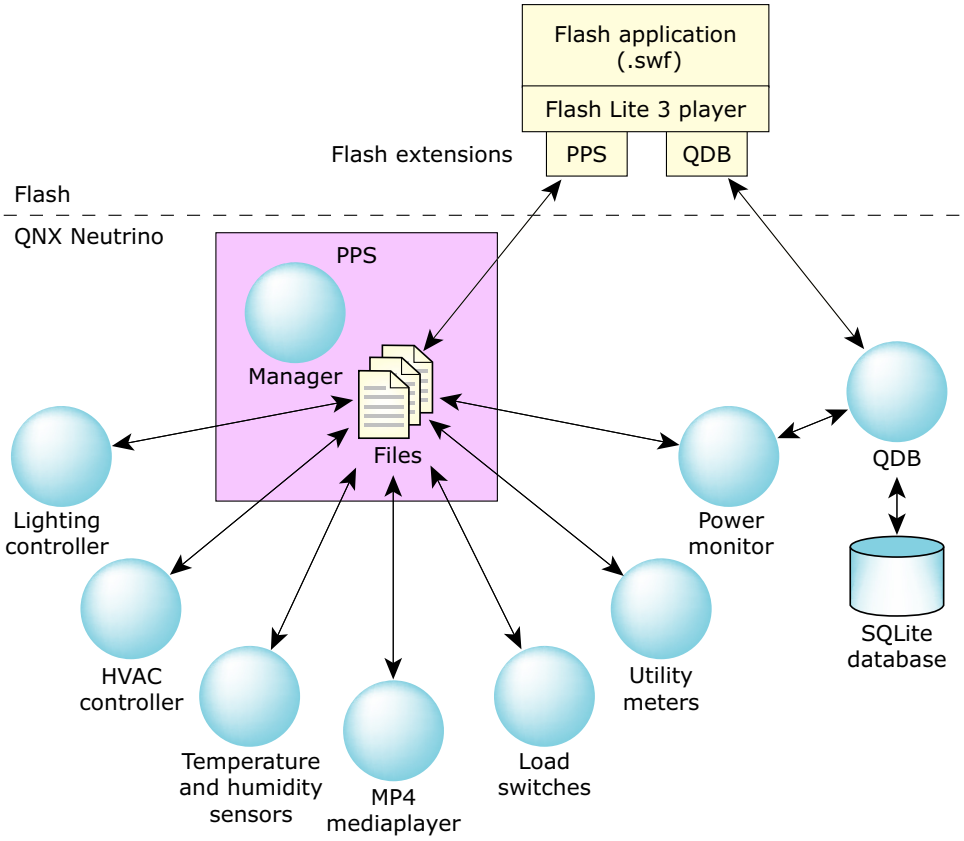
Chapter 1

QNX PPS Service

In this chapter. . .

The QNX Persistent Publish/Subscribe (PPS) service is a small, extensible publish/subscribe service that offers persistence across reboots. It is designed to provide a simple and easy to use solution for both publish/subscribe and persistence in embedded systems, answering a need for building loosely connected systems using asynchronous publications and notifications.

With PPS, publishing is asynchronous: the subscriber need not be waiting for the publisher. In fact, the publisher and subscriber rarely know each other; their only connection is an object which has a meaning and purpose for both publisher and subscriber.



The QNX Smart Energy Reference uses PPS..

"Push" and "pull" publishing

In its default implementation, PPS functions as a *push* publishing system. However, PPS supports a **`pull`** option, which enables a subscriber to *pull* data from a publisher, so that publishing is on-demand.

- While PPS can be used for inter-process communication, it should *not* be used to replace QNX native messages for high speed directed communication. (PPS is, of course, built on top of QNX native messages.)
- PPS is a resource manager and follows the priority inheritance rules common to all resource managers: it runs at the priority of the sending threads.
- Since PPS is completely asynchronous, there is no priority inheritance between PPS clients.

Running PPS

The PPS service can be run with the options listed below.

Syntax:

```
pps [options]
```

Options:

`-b`	Do *not* run in the background. Useful for debugging.
`-l`	Load all objects on startup. Default is to load objects on demand.
`-m` *mount*	Specify the mountpath for PPS. Default is **`/fs/pps/`**
`-p` *path*	Set the path for backing up the persistent storage.
`-v`	Enable verbose mode. Increase the number of “v”s to increase verbosity.

Chapter 2

Objects and their Attributes

In this chapter. . .

The QNX PPS service uses an object-based system; that is, a system with objects whose properties a publisher can modify. Clients that subscribe to an object receive updates when that object changes — when the publisher has modified it.

Thus, users can:

- publish to modify objects
- subscribe to receive notifications when objects have changed
- publish and subscribe to both modify objects, and receive notifications when objects have changed

For more information about publishing and subscribing, see the chapters Publishing and Subscribing.

The QNX PPS design is in many ways similar to many process control systems where the objects are control values updated by hardware or software. Subscribers can be alarm handling code, displays, and so on.

Object files

PPS objects are implemented as files in a special PPS filesystem. By default, PPS objects appear under **`/fs/pps`**. These objects contain attributes.

There is never more than one instance of a PPS object, so persistence is a natural property that can be applied to PPS objects.

You can:

- Create directories and populate them with PPS objects by creating files in the directories.
- Use the *open()*, then the *read()* and *write()* functions to query and change PPS objects.
- Use standard utilities as simple debugging tools.

For example:

Display all objects in the system:

```
ls -lR /fs/pps/
```

Display the current state of an object:

```
cat /fs/pps/media/PlayCurrent
```

```
cat /fs/pps/flash/apps/youtube
```

Monitor all changes to attributes in objects in the media directory, by opening the **`.all`** special object:

```
cat /fs/pps/media/.all?wait
```

For information about **.all**, see "Special objects" below; for information about **?wait**, see the chapter Options and Qualifiers.

Change the attribute of an object:

```
echo "attr::value" >>/fs/pps/objectfilename
```

In order to avoid possible confusion or conflict in the event that applications from different organizations are integrated to use the same PPS filesystem, we recommend that you use your organization's web domain name to create your directory inside the PPS directory. Thus, QNX, whose internet web domain name is "qnx.com" should use **/fs/pps/qnx**, while an organization with the domain name "example.net" should use **/fs/pps/example**.

Object and directory sizes

The number and depth of PPS directories, and the number of PPS objects in a directory is limited only by available memory.

Note, however, that PPS holds its objects in memory, and that these objects are small: the maximum size for a PPS object is 16 kilobytes. PPS objects should not be used as a dumping ground for large amounts of data. The size of most PPS objects should be measured in hundreds of *bytes*, and *not* in kilobytes.

Special objects

PPS directories can include special objects that you can open to facilitate subscription behavior. The table below lists these special objects, and indicates where to get more information about using them:

Object	Use
.all	Open to receive notification of changes to any object in this directory.
.notify	Open a notification file descriptor in the PPS filesystem root.

For more information about these objects, see "Subscribing to multiple objects" the chapter Subscribing.

Object syntax

In PPS, the first line of a file names the object. This line is prefixed with an "@" character to identify it as the object name. The lines that follow define the object's attributes. These lines have no special prefix.

For example, in the PPS filesystem under the directory **/fs/pps/media/**, the PPS object "**PlayCurrent**" contains the metadata for the currently playing song in a multimedia application. Let us assume that the metadata has the following schema:

```
@PlayCurrent
author::[Presentation text for track author]
album::[Presentation text for album name]
title::[Presentation text for track title]
duration::[Track duration, floating point number of seconds]
time::[Track position, floating point number of seconds]
```

An *open()* call followed by a *read()* call on this file would return the name of the object (the filename, with an "@" prefix), followed by the object's attributes with their values:

```
@PlayCurrent
author::Beatles
album::Abbey Road
title::Come Together
duration::3.45
time::1.24
```

- Object names may *not* contain any of the following: "@" (at sign), "?" (question mark), "/" (forward slash), linefeed (ASCII LF), or ASCII NUL.
- Every line in the PPS object is terminated with a linefeed ("\n" in C, or hexadecimal **0A**), so you must encode this character in a manner agreed upon by cooperating client applications. That is, any values containing ASCII LF or NUL characters must be encoded. The encoding field can be used to assist cooperating applications in determining what encoding is used in the value field.

Objects in filesystem listings

In listings of the PPS filesystem, PPS objects have no special identifiers. That is, they will appear just like any other file in a listing. Thus, the PPS object "**PlayCurrent**" used in the example above will appear in a listing as simply **/fs/pps/media/PlayCurrent**.

Change notification

When PPS creates, deletes or truncates an object (a file or a directory), it places a notification string into the queue of any subscriber or publisher that has open either that object or the **.all** special object for the directory with the modified object.

The syntax for this notification string is a special character prefix, followed by the object identifier "@", then the object name, as follows:

Symbol	Example	Meaning
+	**+@objectname**	PPS created the object. To know if a created object is a file or a directory, call *stat()* or *fstat()*.
-	**-@objectname**	PPS deleted the object.
#	**#@objectname**	PPS truncated the object.
*	***@objectname**	The object has lost a critical publisher. All non-persistent attributes have been deleted. For more information, see the chapter Pathname Open Options.
&	**&@objectname**	A subscriber has opened the object with the **pull** option. All publishers reading that object after having opened it with O_RDWR receive an ampersand prefixed to the file descriptor. For more information, see the chapter Pathname Open Options.

Responding to an object deletion

A deleted object is no longer visible in the filesystem (POSIX behavior), and only those processes with open file descriptors can continue accessing it. Therefore, typical behavior for an application receiving notification that an object has been deleted would be to close the file.

Attributes

PPS objects have user-defined attributes. Attributes are listed in a PPS object after the object name.

Attribue names may be composed of any alpha-numeric character, an underscore and a period; that is, any character from the set **[A-Za-z][A-Za-z0-9_.]***. Attribute lines in a PPS object are of the form *attrname:encoding:value\n*, where *attrname* is the attribute name, and *encoding* defines the encoding type for *value*. The end of the attribute name and the end of the encoding are marked by colons (":"). Subsequent colons are ignored.

PPS does not interpret the encoding; it simply passes the encoding through from publisher to subscriber. Thus, publishers and subscribers are free to define their own encodings to meet their needs. The table below describes possible encoding types:

Symbol	Encoding
::	Simple text terminated by a linefeed

continued...

Symbol	Encoding
`:c:`	C language escape sequences, such as "\t" and "\n". Note that "\n" or "\t" in this encoding is a "\" character followed by an "n" or "t"; in a C string this would be "\\n\\t"
`:b64:`	Base 64 encoding.

An attribute's *value* can be any sequence of characters, *except*:

- a null ("\0" in C, or hexadecimal `0x00`)
- a linefeed character ("\n" in C, or hexadecimal `0x0A`).

Chapter 3

Persistence

In this chapter. . .

PPS maintains its objects in memory while it is running. It will, as required:

- save its objects to persistent storage, either on demand while it is running, or at shutdown
- restore its objects on startup, either immediately, or on first access (deferred loading)

PPS may be used to create objects which are rarely (or never) published or subscribed to, but for which persistence is required.

Persistent storage

The underlying persistent storage used by PPS relies on a reliable filesystem, such as:

- disk — QNX 6 filesystem
- NAND Flash — ETFS filesystem
- Nor Flash — FFS3 filesystem
- other — customer generated filesystem

If you need to persist an object to specialized hardware, such as a small NVRAM, which does not support a file system, you can create your own client which subscribes to the a PPS object to be saved. On each object change, PPS will notify your client, allowing the client to update the NVRAM in realtime.

Saving objects

On shutdown, PPS always saves any modified objects to a persistent filesystem. You can also force PPS to save an object at any time by calling *fsync()* on the object.

When PPS saves to a persistent filesystem, it saves all objects to a single directory. The default location for this directory is **`/var/pps`**. You can use the PPS **`-p`** option to change this location.

- You can set object and attribute qualifiers to have PPS *not* save specific objects or attributes. For more information, see the chapter Options and Qualifiers.
- To ensure multiple language support, all strings should use UTF-8 to encode extended character sets.

Contents of saved files

PPS saves all objects in a directory to a single file; there will be one saved file per PPS directory. PPS encodes the directory path in the filename. For example, if PPS is mounted at **`/fs/pps`**, the following encodings are possible:

PPS directory	Filesystem filename
`/fs/pps`	`@`
`/fs/pps/media`	`@media`
`/fs/pps/apps`	`@apps`
`/fs/pps/apps/youtube`	`@apps@youtube`

To ensure data integrity, PPS performs CRCs on saved objects in the saved files.

Loading objects

When PPS starts up, it immediately builds the directory hierarchy from the encoded filenames on the persistent filesystem. It defers loading the objects in the directories until first access to one of the files. This access could be an *open()* call on a PPS object, or a *readdir()* call on the PPS directory.

Chapter 4

Publishing

In this chapter. . .

To publish to a PPS object, a publisher simply calls *open()* for the object file with O_WRONLY to publish only, or O_RDWR to publish and subscribe. The publisher can then call *write()* to modify the object's attributes. This operation is non-blocking.

Creating, modifying and deleting

You can create, modify, and delete objects and attributes:

To create a new object:

> Create a file with the name of the object. The new object will come into existence with no attributes. You can then write attributes to the object, as required.

To delete an object:

> Delete the object file.

To create a new attribute:

> Write the attribute to the object file.

To modify an attribute:

> Write the new attribute value to the object file.

To delete all existing attributes:

> Open the object with O_TRUNC.

To delete one attribute:

> Prefix its name with a minus sign, then call *write()*. For example:

```
fprintf( ppsobj, "-url\n" ); // Delete the "url" attribute
write( ppsobj-fd, ppsobj, strlen( ppsobj ) );
```

Multiple publishers

PPS supports multiple publishers that publish to the same PPS object. This capability is required because different publishers may have access to data which applies to different attributes for the same object.

In a multimedia system, for instance, **`io-media`** may be the source of a **`time::`***value* attribute, while the HMI may be the source of a **`duration::`***value* attribute. A publisher that changes only the **`time`** attribute will update only that attribute when it writes to the object. It will leave the other attributes unchanged. For example:

```
write()
  PlayCurrent::1.24
```

Chapter 5

Subscribing

In this chapter. . .

PPS clients can subscribe to multiple objects, and PPS objects can have multiple subscribers. When a publisher changes an object, all clients subscribed to that object are informed of the change.

To subscribe to an object, a client simply calls *open()* for the object with O_RDONLY to subscribe only, or O_RDWR to publish and subscribe. The subscriber can then query the object with a *read()* call.

A read returns the length of the data read, in bytes. If the allocated read buffer is too small for the data being read in, the read fails.

Blocking and non-blocking reads

By default, reads to PPS objects are non-blocking; that is, PPS defaults a normal *open()* to O_NONBLOCK, and reads made by the client that opened the object do not block. This behavior is atypical for most filesystems. It is implemented so that standard utilities will not hang waiting for a change when they make a *read()* call on a file.

For example, with the default behavior, you could tar up the entire state of PPS using the standard tar utility. Without this default behavior, however, tar would never make it past the first file opened and read.

Setting PPS to block

Though the PPS default is to open objects for non-blocking reads, the preferred method for querying PPS objects is to use blocking reads. With this method, a read waits until the object or its attributes change, then returns data.

To have reads block, you need to open the object with the **`?wait`** pathname open option, appended as a suffix to the pathname for the object. For example:

- to open the **`PlayList`** object for the default non-blocking reads, use the pathname:**`"/fs/pps/media/PlayList"`**
- to open the **`PlayList`** for blocking reads, use the pathname plus the option: **`"/fs/pps/media/PlayList?wait"`**

For information about **`?wait`** and other pathname open options, see the chapter Options and Qualifiers.

A typical loop in a subscriber would live in its own thread. For a subscriber that used the opened the object with the **`?wait`** option, this loop might do the following:

```
/* Assume that the object was opened with the ?wait option
   No error checking in this example. */
for(;;) {
    read(fd, buf, sizeof(buf)); // Read waits until the object changes.
    process(buf);
}
```

Clearing O_NONBLOCK

If you have opened an object without the **?wait** option, and want to change to blocking reads, you can clear the O_NONBLOCK bit, so that the subscriber waits for changes to an object or its attributes.

To clear the bit you can use the *fcntl()* function. For example:

```
flags = fcntl(fd, F_GETFL);
flags &= ~O_NONBLOCK;
fcntl(fd, F_SETFL, flags);
```

Or you can use the *ioctl()* function:

```
int i=0;
ioctl(fd,FIONBIO,&i);
```

After clearing the O_NONBLOCK bit, you can issue a read that waits until the object changes.

io_notify() functionality

The PPS service implements *io_notify()* functionality, allowing subscribers to request notification via a PULSE, SIGNAL, SEMAPHORE, etc. On notification of a change, a subscriber must issue a *read()* to the object file to get the contents of the object. For example:

```
/* Process events while there are some */
while(ionotify(fd, _NOTIFY_ACTION_POLLARM, _NOTIFY_COND_INPUT,
    &event) & _NOTIFY_CONT_INPUT) {
if(read(fd, buf, sizeof(buf)) > 0) // Best to read with O_NONBLOCK
    process(buf);
    }
/* The event will be triggered in the future to get our attention */
```

Getting notifications of data on a file descriptor

You can use either one of two simple mechanisms to receive notifications that data is available on a file descriptor:

- You can issue a blocking *read()* by either opening the object with the **?wait** syntax on the *open()* call, or by clearing the O_NONBLOCK flag using the *fnctl()* function after the *open()* call.
- You can use the QNX *io_notify()* mechanisms to receive a user-specified event; you can also use the *select()* function, which uses *io_notify()* under the covers. See "*io_notify()* functionality" above.

Subscription Modes

A subscriber can open an object in full mode, in delta mode, or in full and delta modes at the same time. The default is full mode. To open an object in delta mode, you need to open the object with the **`?delta`** pathname open option, appended as a suffix to the pathname for the object.

For information about **`?delta`** and other pathname open options, see the chapter Options and Qualifiers.

Full mode

In full mode (the default), the subscriber always receives a single, consistent version of the entire object as it exists at the moment when it is requested.

If a publisher changes an object several times before a subscriber asks for it, the subscriber receives the state of the object at the time of asking *only*. If the object changes again, the subscriber is notified again of the change. Thus, in full mode, the subscriber may miss multiple changes to an object — changes to the object that occur before the subscriber asks for it.

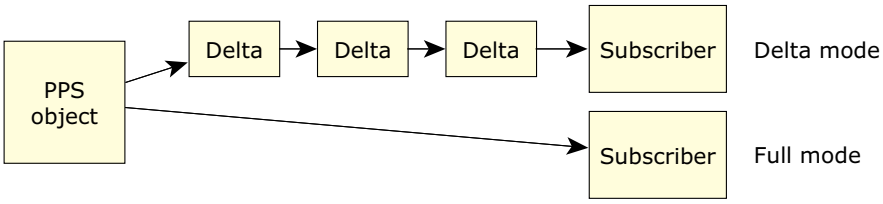
Delta mode

In delta mode, a subscriber receives only the changes (but all the changes) to an object's attributes.

On the first read, since a subscriber knows nothing about the state of an object, PPS assumes everything has changed. Therefore, a subscriber's first read in delta mode returns all attributes for an object, while subsequent reads return only the changes since that subscriber's previous read.

Thus, in delta mode, the subscriber always receives all changes to an object.

The figure below illustrates the different information sent to subscribers who open a PPS object in full mode and in delta mode.



Comparison of PPS full and delta subscription modes.

In all cases we have persistent objects with states — there is always an object. The mode used to open an object does not change the object; it only determines the subscriber's view of changes to the object.

Delta mode queues

When a subscriber opens an object in delta mode, the PPS service creates a new queue of object changes. That is, if multiple subscribers open an object in delta mode, each subscriber has its own queue of changes to the object, and the PPS service sends each subscriber its own copy of the changes. If no subscriber has an object open in delta mode, the PPS service does not maintain any queues of changes to that object.

On shutdown, the PPS service saves its objects, but objects' delta queues are lost.

Changes to multiple attributes

If a publisher changes multiple attributes with a single *write()* call, then PPS keeps the deltas together and returns them in the same group on a subscriber's *read()* call. In other words, PPS deltas maintain both time and atomicity of changes. For example:

```
write()                    write()
  time::1.23                 time::1.24
  duration::4.2              write()
                             duration::4.2

read()                   read()
  @objname                 @objname
  time::1.23               time:1.24
  duration::4.2            @objname
                           duration::4.2
```

Subscribing to multiple objects

PPS supports two special objects which facilitate subscribing to multiple objects:

- **`.all`** — open to receive notification of changes to any object in this directory.
- **`.notify`** — open to receive notification of changes to any object associated with a notification group.

Subscribe to all objects in a directory

PPS uses directories as a natural grouping mechanism to simplify and make more efficient the task of subscribing to multiple objects. Subscribers can open multiple objects, either by calling *open()* then *select()* on their file descriptors. More easily, they can open the special **`.all`** object, which merges all objects in its directory.

For example, assume the following object file structure under **`/fs/pps`**:

```
rear/left/PlayCurrent
rear/left/Time
rear/left/PlayError
```

If you open **rear/left/.all** you will receive a notification when any object in the **rear/left** directory changes. A read in full mode will return at most one object per read.

```
read()
@Time
  position::18
  duration::300

read()
@PlayCurrent
  artist::The Beatles
  genre::Pop
  ... the full set of attributes for the object
```

If you open a **.all** object in delta mode, however, you will receive a queue of every attribute that changes in any object in the directory. In this case, a single *read()* call may include multiple objects.

```
read()
@Time
  position::18
@Time
  position::19
@PlayCurrent
  artist::The Beatles
  genre::Pop
```

Notification groups

PPS provides a mechanism to associate a set of file descriptors with a notification group. This mechanism allows you to read only the PPS special notification object to receive notification of changes to any of the objects associated with that notification group.

Creating notification groups

To create a notification group:

1. Open the **.notify** object in the root of the PPS file system.
2. Read the **.notify** object; the first read of this file returns a short string (less than 16 characters) with the name of the group to which other file descriptors should associate themselves.

To associate a file descriptor to a group, on an open, specify the pathname open option **?notify=***group:value*, where:

- *group* is the string returned by the first read from the **.notify** file
- *value* is any arbitrary string; a subscriber will use this string to determine which objects bound to the notification group have data available for reading

The returned notification group string has a trailing linefeed character, which you must remove before using the string.

For information about **?notify** and other pathname open options, see the chapter Options and Qualifiers.

Using notification groups

Once you have created a notification group and associated file descriptors to it, you can use this group to learn about changes to any of the objects associated with it.

Whenever there is data available for reading on any of the group's file descriptors, reads to the notification object's file descriptor return the string passed in the **?notify=***group:value* pathname option.

For example, with PPS is mounted at **/fs/pps**, you could write something like the following:

```
char noid[16], buf[128];
int notify_fd, fd1, fd2;

notify_fd = open("/fs/pps/.notify", O_RDONLY);
read(notify_fd, &noid[0], sizeof(noid));

sprintf(buf, "/fs/pps/fish?notify=%s:water", noid);
fd1 = open(buf, O_RDONLY);
sprintf(buf, "/fs/pps/dir/birds?notify=%s:air", noid);
fd2 = open(buf, O_RDONLY);

while(read(notify_fd, &buf, sizeof(buf) > 0) {
    printf("Notify %s\n", buf);
}
```

The data printed from the "while" loop in the example above would look something like the following:

```
Notify 243:water
Notify 243:water
Notify 243:air
Notify 243:water
Notify 243:air
```

When reading from an object that is bound to a notification group, a subscriber should do multiple reads for each change indicated. There may be more than one change on an item, but there is no guarantee that every change will be indicated on the notification group's file descriptor.

Notification of closed file descriptors for objects

If a file descriptor for an object which is part of a notification group is closed, the string passed with the change notification is prefixed by a minus (“-”) sign. For example:

```
-243:air
```

Chapter 6

Options and Qualifiers

In this chapter. . .

PPS supports pathname open options, and objects and attribute qualifiers. PPS uses pathname open options to apply open options on the file descriptor used to open an object. Object and attribute qualifiers set specific actions to take with an object or attribute; for example, make an object non-persistent, or delete an attribute.

Pathname open options

PPS objects support an extended syntax on the pathnames used to open them. Open options are added as suffixes to the pathname, following a question mark ("?"). That is, the PPS service uses any data that follows a question mark in a pathname to apply open options on the file descriptor used to access the object. Multiple options are separated by question marks. For example:

- **`"/fs/pps/media/PlayList"`** — open the **`PlayList`** file with no options
- **`"/fs/pps/media/PlayList?wait"`** — open the **`PlayList`** file with the **`wait`** option
- **`"/fs/pps/media/Playlist?wait,delta"`** — open **`PlayList`** file with the **`wait`** and **`delta`** options
- **`"/fs/pps/media/.all?wait"`** — open the **`media`** directory with the **`wait`** option
- **`"/fs/pps/fish?notify=345:water"`** — open **`fish`** and associate it with **`.notify`** group 345
- **`"/fs/pps/squid?pull"`** — open **`squid`** with the **`pull`** option

The syntax used for specifying PPS pathname open query options will be easily recognizable to anyone familiar with the *getsubopt()* library routine.

Supported pathname open options include:

`critical` — Designate the publisher as critical to the object. See "Critical option" below.

`delta` — Open the object in delta mode.

See "Subscribing to multiple objects" in the chapter Subscribing.

`deltadir` — Return the names of all objects (files) in the directory — valid only on the special **`.all`** object in a directory.

If any objects in the directory are created or deleted, these changes are indicated by a "+" (created) or a "-" (deleted) sign prefixed to their names. This behavior allows you to effectively perform a *readdir()* within PPS, as well as monitor filesystem changes without having to also monitor attribute changes.

See “Subscribing to multiple objects” in the chapter Subscribing.

f=*attr+attr ...*
Place a filter on notifications based upon changes to the listed attribute names only. Multiple attributes are separated by “+” signs.

In full mode, the file descriptor will only get notifications if one of the listed attributes changes. When this occurs the entire object is returned.

In delta mode, only the listed attributes will be queued. Changes to non-listed attributes are filtered out.

n
Make the object non-persistent. When the system restarts, the object will not exist. The default setting is for all objects to be persistent and reloaded on restart. See “Object and attribute options” below.

notify=*id:value*
Associate the opened file descriptor with the notify group specified by *id*. This *id* is returned on the first read from an *open()* all on the “.notify” file in the root of the PPS mount point.

Reads of the “.notify” file will return the string: *id:value* whenever data is available on the file descriptor opened with the *notify=* query.

See “Subscribing to multiple objects” in the chapter Subscribing.

pull
Open the object so that reads inform the publisher to update the object. The reader blocks until a publisher updates the object. See “Pull option” option below.

wait
Open the file with the O_NONBLOCK flag clear so that *read()* calls wait until the object changes or a delta appears. See the chapter Subscribing.

Critical option

The critical option can be used as an attribute cleanup mechanism on the abnormal termination of a publisher.

If this option is used when opening a file descriptor for a write, when the file descriptor is closed PPS deletes all non-persistent attributes and prefixes an asterisk (“*”) to the object name in the notification string it sends to all subscribers. PPS does *not* provide a list of the deleted attributes.

Duplicate *critical* file descriptors

You should never have more than one critical file descriptor for any one PPS object.

File descriptors can be duplicated, either explicitly by *dup()*, *dup2()* or *fcntl()*, etc.; or implicitly by *fork()*, *spawn()*, etc. Duplicated descriptors in effect increment a reference count on the underlying critical descriptor. The behavior required of critical objects (the notification and deletion of volatile attributes) is not triggered until the reference count for a file descriptor drops to zero, indicating that the original and all duplicates are closed.

However, if you open a PPS object more than once in critical mode, each file descriptor behaves as a critical descriptor: if the reference count of any one file descriptor drops to zero, the notification and deletion behavior is triggered for the object — even if other descriptors remain open.

Pull option

The **`pull`** option allows clients to pull data from a publisher, so that publishing is on-demand.

In its default implementation, PPS acts as *push* publishing system; that is, publishers push data to objects, and subscribers read data upon notification or at their leisure. However, some data, such as packet counts on an interface, changes far too fast to be efficiently published through PPS using its default push publishing.

When a subscriber that opened an object with the **`pull`** option issues a *read()* call, all publishers that opened that object receive a notification to write current data to the object. The subscriber's read blocks until the object's data is updated, and returns with the new data.

To notify publishers that a subscriber wants new data for an object, PPS enqueues the object's name, prefixed with an ampersand ("&"), on the object's file descriptor. For example, if a client specifies the **`pull`** option when opening **`squid`**: **`"/fs/pps/squid?pull"`**, publishers that opened **`squid`** will receive **`&@squid`**.

This mechanism creates a *pull* publishing system where the subscriber retrieves data from the publisher at whatever rate it requires — in effect, on-demand publishing.

Object and attribute qualifiers

Object and attribute qualifiers are contained in square brackets ("[*qualifier*]"), and prefixed to lines containing an object or an attribute name.

You can set qualifiers to *read()* and *write()* calls by starting a line containing an object or attribute name with an open square bracket, followed by a list of single-letter or single-numeral qualifiers and terminated by a close square bracket.

Qualifiers supported for objects and attributes are:

`n` No-persistence; see "No-persistence qualifier" below.

Qualifiers supported for attributes *only* are:

i	Item; see "Item qualifier" below.
0 to **7**	Quality; see "Quality qualifier" below.

- Qualifier defaults are always "clear".
- On a *read()* call you will only see a preceding qualifier list "[*option letters*]" for options which have been set.
- Attribute options always *precede* both the special character and the object or attribute name.

Setting qualifiers

If nothing precedes a qualifier, that qualifier is set. It the qualifier is preceded by a minus sign ("-"), that qualifier is cleared. If a qualifier is not specified, that qualifier is not changed. For example:

- **[n]url::www.qnx.com** — set the no-persist qualifier on this attribute
- **[-n]url::www.qnx.com** — clear the no-persist qualifier on this attribute
- **url::www.qnx.com** — do not change the current no-persist qualifier on this attribute
- **[4]author::Beatles** — change **author** if the currently quality is less than or equal to 4
- **[i]items::hammer,** — add **hammer** to the set
- **[-i]items::screw driver,** — remove **screw driver** from the set

No-persistence qualifier

The **no-persistence** qualifier can be used for objects and attributes. It is very useful on attributes that may not be valid across a system restart and do not need to be saved.

The table below describes the effects of the **no-persistence** qualifier on PPS objects and attributes:

Syntax	Action	Object	Attribute
n	Set	Make the object and its attributes non-persistent; ignore any persistence qualifiers set on this object's attributes.	Make the attribute non-persistent.
-n	Clear	Make the object persistent; persistence of the object's attributes is determined by the individual attribute's qualifiers.	Make the attribute persistent, *if* the attribute's object is also persistent.

Setting the **no-persistence** qualifier on an object overrides any **no-persistence** qualifiers set on the object's attributes, and is therefore convenient if you need to make a temporary object in which nothing persists.

For more information about persistence, see the chapter Persistence.

Item qualifier

The **item** qualifier can be used for attributes *only*. It causes PPS to treat the value following the qualifier as a *set of items*. Items in a set are separated by a user-defined *separator*, such as a comma.

The item separator:

- is required
- must be the last character in a value that uses the item qualifier
- can be any character *not* used in the items

Adding and deleting set items

You may add or delete only one set item at a time. For example, to add items to a set:

```
[i]items::hammer,
[i]items::screw driver,
```

Or, to delete an item from a set:

```
[-i]items::hammer,
```

Incorrect item syntax

The following are not permitted:

```
[i]items::hammer,screw driver,
```

Or:

```
[-i]items::hammer,screw driver,
```

Examples

In the examples below, the separator is a comma: "**,**".

Example 1: Duplicate items

This example shows that PPS ignores a duplicate attempt to add the same item to a set. The following lines written:

```
[i]items::hammer,
[i]items::hammer,
[i]items::screw driver,
```

would result in the following being read by a subscriber:

```
items::hammer,screw driver,
```

Example 2: Null items

This example shows how PPS supports a null item in a set. The following line written to the set created in the previous example:

```
[i]items::,
```

would result in the following being read by a subscriber:

```
items::hammer,screw driver,,
```

Example 3: Delete an item

This example shows how to delete an item from a set. The following line written to the set created and updated in the previous examples:

```
[-i]items::hammer,
```

would result in the following being read by a subscriber:

```
items::screw driver,,
```

Quality qualifier

The **`quality`** qualifier can be used for attributes *only*. It sets the quality of the data associated with an attribute. It can have any value from 0 to 7. If this qualifier is not specified, PPS assumes the default value 0 (zero).

This qualifier is useful when multiple publishers may be able to provide data for an attribute, but with different levels of quality. It allows a publisher to update attribute data *only if* the quality of its data is equal to or greater than the quality of the existing data.

To ensure that attribute data quality remains the same or increases as different publishers report their values asynchronously, simply have each publisher set the **`quality`** qualifier to the appropriate level when it publishes an attribute.

ppsparse()

Parse an object read from PPS

Synopsis:

```
#include <ppsparse.h>

extern pps_status_t ppsparse( char **ppsdata,
                              const char * const *objnames,
                              const char * const *attrnames,
                              pps_attrib_t *info,
                              int parse_flags );
```

Arguments:

ppsdata	A pointer to a pointer to the current position in the buffer of PPS data. The function updates this pointer as it parses the options; see the "Description" below.
objnames	A pointer to a NULL-terminated array of object names. If this value is not NULL, *ppsparse()* looks up any object name it finds and provides its index in the **pps_attrib_t** structure.
attrnames	A pointer to a NULL-terminated array of attribute names. If this value is not NULL, *ppsparse()* looks up any attribute name it finds and provides its index in the **pps_attrib_t** structure.
info	A pointer to the data structure **pps_attrib_t**, which carries detailed about a line of PPS data.
parse_flags	Reserved for future use.

Library:

libc.

Description:

The function *ppsparse()* parses the next line of a buffer of PPS data. This buffer must be terminated by a null ("\0" in C, or hexadecimal **0x00**).

The first time you call this function after reading PPS data, you should set *ppsdata* to reference the start of the buffer with the data. As it parses each line of data, *ppsparse()*:

- places the information parsed from the buffer in the **pps_attrib_t** data structure
- updates the pointer to the next PPS line in the buffer

When it successfully completes parsing a line, *ppsparse()* returns the type of line parsed, or end of data; see **pps_status_t** below.

pps_attrib_t

```
typedef struct {
    char *obj_name;
    int  obj_index;
    char *attr_name;
    int  attr_index;
    char *encoding;
    char *value;
    int  flags;
    int  options;
    int  option_mask;
    int  quality;
    char *line;
    int  reserved[3];
} pps_attrib_t;
```

The **pps_attrib_t** data structure carries parsed PPS object and attribute information. It includes the members described in the table below:

Member	Type	Description
obj_name	**char**	A pointer to the name of the last PPS object encountered. *ppsparse()* sets this pointer only if it encounters a PPS object name. You should initialize this pointer before calling *ppsparse()*.
obj_index	**int**	The index for *obj_name* in the *objnames* array. It is set to -1 if the index is not found or *objnames* is NULL. You should initialize this value before calling *ppsparse()*.
attr_name	**char**	A pointer to the name of the attribute from the line PPS data that *ppsparse()* just parsed. It is set to NULL if no attribute name was found.
attr_index	**int**	The index for *attrj_name* in the *attrnames* array. It is set to -1 if the index is not found or *attrnames* is NULL.
encoding	**char**	A pointer to a string that indicates the encoding used for the PPS attribute. See “Attributes” in the chapter PPS Objects and Attributes. This value is only relevant if the *ppsparse()* return value is PPS_ATTRIBUTE. See **pps_status_t** below.
value	**char**	A pointer to the value of a PPS attribute. This value is only relevant if the *ppsparse()* return value is PPS_ATTRIBUTE. See **pps_status_t** below.
flags	**int**	Flags indicating that parsing has found a PPS special character prefixed to a line, or that the line is incomplete. See **pps_attrib_flags_t** below.

continued...

Member	Type	Description
options	`int`	Indicates which non-negated options are prefixed in square brackets to a line. See "Object and attribute options" in the chapter Options.
option_mask	`int`	A mask of the options (both negated and non-negated) prefixed to a line. See `pps_options_t` below.
quality	`int`	The quality of the attribute. See "Quality option" in the chapter Options.
line	`char`	Pointer to the beginning of the line parsed by *ppsparse()*, for use in case of a parsing error.
reserved[3]	`int`	For internal use.

pps_attrib_flags_t

The enumerated values PPS_* defined by `pps_attrib_flags_t` define the possible states for PPS objects and attributes. These states include:

- PPS_INCOMPLETE — the object or attribute line is incomplete
- PPS_DELETED — the object or attribute has been deleted
- PPS_CREATED — the object has been created
- PPS_TRUNCATED — the object or attribute has been truncated
- PPS_PURGED — a critical publisher has closed its connection and all non-persistent attributes have been deleted; see "Critical option" in the chapter Options.

See also "Change notification" in the chapter PPS Objects.

pps_options_t

The enumerated values PPS_NOPERSIST defined by `pps_options_t` define values for PPS options:

- PPS_NOPERSIST — no-persistence option
- PPS_ITEM — item option

pps_status_t

The enumerated values PPS_* defined by `pps_status_t` define the possible *ppsparse()* return values. These values include:

- PPS_ERROR — the line of PPS data is invalid
- PPS_END — end of data, or incomplete line
- PPS_OBJECT — data for the given object follows

- PPS_OBJECT_CREATED — an object has been created
- PPS_OBJECT_DELETED — an object has been deleted
- PPS_OBJECT_TRUNCATED — an object has been truncated (all attributes were removed)
- PPS_ATTRIBUTE — and attribute has been updated
- PPS_ATTRIBUTE_DELETED — an attribute has been deleted

Returns:

The *ppsparse()* function returns:

`≥0` Success.

`-1` An error occured (*errno* is set).

Examples:

The following test application shows how *ppsparse()* can be used:

```
#ifdef PPSPARSE_TEST
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>


// Test data to use if a file name is not provided as a command line argument.
// This data may be read-only in code space, so we copy it.
char *testdata =
    "@book\n"
    "title::Money money1\n"
    "author:c:Money money2\n"
    "[n]title::Nopersist option\n"
    "[-n]title::Negated option\n"
    "[-xnp]time:c:Unknown options\n"
    "badattr:Improperly formatted\n"
    "[n]@song\n"
    "title::Money money4\n"
    "Just some garbage\n"
    "[n]+newAttr:abc:New attribute\n" // New attribute (not currently used by pps)
    "+newAttr\n"                     // New attribute, missing encoding etc
    "-deleteAttr::\n"                // Deleted attribute with encoding etc
    "-deleteAttr\n"                  // Deleted attribute without encoding etc
    "-@unidentified\n"               // Deleted object
    "#@unidentified\n"               // Truncated object
    "[i]flags::flg1,\n"              // Item flag
    "[-i]flags::flg1,\n"             // Negated item flag
    "[7n]quality::qualityAttribute\n" // Attribute with quality setting
```

```
    "Incomplete line"
      ;

int
main(int argc, char *argv[])
{
    enum {
        ATTR_AUTHOR,
        ATTR_TITLE,
        ATTR_TIME,
        LAST_ATTR
    };
    static const char *const attrs[] ={
        [ATTR_AUTHOR] = "author",
        [ATTR_TITLE]  = "title",
        [ATTR_TIME]   = "time",
        [LAST_ATTR]   = NULL
    };
    enum {
        OBJECT_BOOK,
        OBJECT_FILM,
        OBJECT_SONG,
        LAST_OBJECT
    };
    static const char *const objs[] = {
        [OBJECT_BOOK] = "@book",
        [OBJECT_FILM] = "@film",
        [OBJECT_SONG] = "@song",
        [LAST_OBJECT] = NULL
    };

    char buffer[1024];
    char  *ppsdata = buffer;
    int fd = -1;
    pps_attrib_t info;
    pps_status_t rc;
    int lineno = 0;

    if ( argc > 1 ) {
        fd = open(argv[1], O_RDONLY);
        if ( fd < 0 ) {
            perror(argv[1]);
            exit(1);
        }
    }
    else {
        strcpy(buffer, testdata);
    }
    memset(&info, 0, sizeof(info));

    while ( ppsdata != NULL ) {
        if ( fd >= 0 ) {
```

```
    int n = read(fd, ppsdata, sizeof(buffer) - (ppsdata - buffer) - 1);
    if ( n <= 0 ) {
        exit(0);
    }
    ppsdata[n] = '\0';
    ppsdata = buffer;
}
while ( (rc = ppsparse(&ppsdata, objs, attrs, &info, 0)) ) {
    printf("%d --------------------------\n%s  ",++lineno,
           rc == PPS_ERROR ? "PPS_ERROR" :
           rc == PPS_END ? "PPS_END" :
           rc == PPS_OBJECT ? "PPS_OBJECT" :
           rc == PPS_OBJECT_CREATED ? "PPS_OBJECT_CREATED" :
           rc == PPS_OBJECT_DELETED ? "PPS_OBJECT_DELETED" :
           rc == PPS_OBJECT_TRUNCATED ? "PPS_OBJECT_TRUNCATED" :
           rc == PPS_ATTRIBUTE ? "PPS_ATTRIBUTE" :
           rc == PPS_ATTRIBUTE_DELETED ? "PPS_ATTRIBUTE_DELETED" : "?");
    if ( info.flags ) {
        printf("flags:%s%s%s%s%s  ",
               info.flags & PPS_INCOMPLETE ? "inc" : "",
               info.flags & PPS_CREATED ? "+" : "",
               info.flags & PPS_DELETED ? "-" : "",
               info.flags & PPS_TRUNCATED ? "#" : "",
               info.flags & PPS_PURGED ? "*" : "");
    }
    if ( info.options || info.option_mask ) {
        printf("options:%s%s  mask:%s%s  ",
               info.options & PPS_NOPERSIST ? "n" : "",
               info.options & PPS_ITEM ? "i" : "",
               info.option_mask & PPS_NOPERSIST ? "n" : "",
               info.option_mask & PPS_ITEM ? "i" : "");
    }
    printf("object:%s (%d)  ",info.obj_name ? info.obj_name : "NULL",
           info.obj_index);
    if ( rc == PPS_ATTRIBUTE || rc == PPS_OBJECT_DELETED ) {
        printf("attr:%s (%d)  ",
               info.attr_name ? info.attr_name : "NULL", info.attr_index);
        if ( rc == PPS_ATTRIBUTE  ) {
            printf("quality:%d  encoding:%s  value:%s",info.quality,
                   info.encoding, info.value);
        }
    }
    printf("\n");

    // now put everything back so we can print out the line.
    if ( info.encoding )
        info.encoding[-1] = ':';
    if ( info.value )
        info.value[-1] = ':';
    printf("%s\n",info.line);
}
if ( fd >= 0 ) {
```

```
            // Note: When reading directly from PPS, you don't need to deal
            // with the PPS_INCOMPLETE flag. It is needed only to allow
            // parsing of PPS data where the data is not provided in complete
            // lines.  In this case, the partial line is moved to the beginning
            // of the buffer, more data is read and the parsing is attempted again.
            if ( info.flags & PPS_INCOMPLETE ) {
                memmove(buffer, info.line, ppsdata - info.line);
                ppsdata = buffer + (ppsdata - info.line);
            }
            else {
                ppsdata = buffer;
            }
        }
        else {
            ppsdata = NULL;
        }
    }

    return 0;
}

#endif
```

Classification:

QNX

Safety	
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Caveats:

During parsing, separators (":" and "\n") in the input string may be changed to null characters.

See also:

The rest of the *QNX PPS Developer's Guide*.

Index

P

Q

R

S

T

U