QNX Persistent Publish/Subscribe 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: March 06, 2020

Contents

About This Guide

The QNX Neutrino *PPS Developer's Guide* describes how to use Persistent Publish Subscribe to pass data between processes in a flexible manner that preserves information when the system is rebooted. The following links may help you find what you need:

To find out about:	See:
An introduction to the Persistent Publish/Subscribe service, and how to run it	*QNX Neutrino PPS Service*
A description of the PPS service's objects and their attributes	*Objects and Their Attributes*
How PPS manages persistence	*Persistence*
How to publish to PPS	*Publishing*
How to subscribe to PPS	*Subscribing*
How to use server PPS objects	*Working with server objects*
Pathname open options, and object and attribute qualifiers	*Options and Qualifiers*
A description of the publicly visible PPS encoding and decoding API functions and data types	*PPS Encoding and Decoding API*
Sample publishers and subscribers	*Examples*

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.

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

DANGER: 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
QNX Neutrino PPS Service

The QNX Neutrino Persistent Publish/Subscribe (PPS) service is a small, extensible publish and subscribe service that offers persistence across reboots. It's 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 that has a meaning and purpose for both publisher and subscriber.

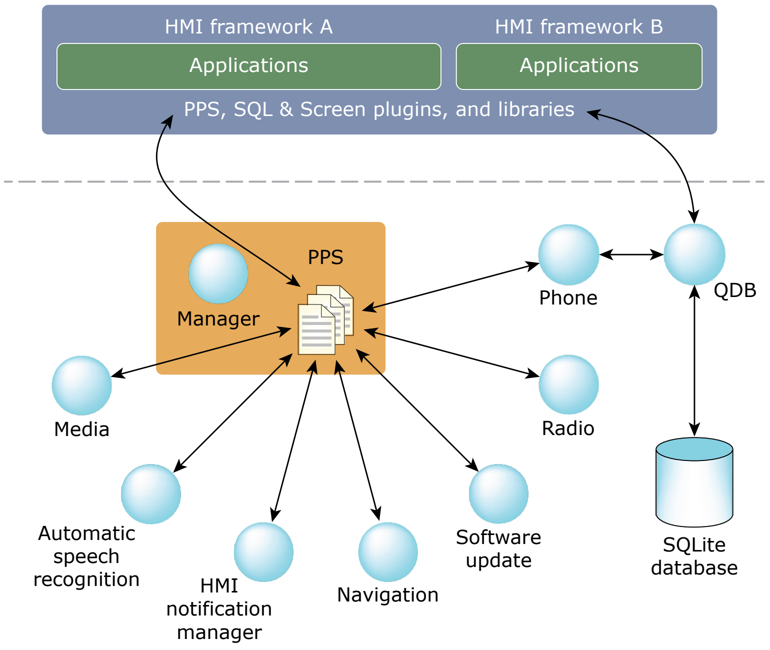

Figure 1: A complex application that uses PPS to share information between components.

QNX Neutrino also supports the D-Bus message-passing system. See "D-Bus" in the *Utilities Reference*).

Running PPS

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

Syntax:

`pps [`*options*`]`

Options:

`-A` ***path***

(QNX Neutrino 6.6 or later) Set the path to an Access Control List (ACL) configuration file. You can use more than one instance of this option. In the event of contradictory permissions, the permissions in the last configuration file listed take precedence. See "*Access Control List configuration file*."

`-a` ***num***

(QNX Neutrino 7.0 or later) The maximum number of open file handles allowed for the **.all** objects. The default and minimum value is 32.

`-b`

Don't run in the background. Useful for debugging.

`-C`

(QNX Neutrino 6.6 or later) Convert between **root** and non-**root** persistence formats, to correspond to the `-U` option.

`-D` ***dir***

(QNX Neutrino 7.0 or later) Specify the directory to put core files in. The default is none.

`-d` ***backlog***

Specify the default delta backlog, in kilobytes. The default is 256 bytes.

`-g`

(QNX Neutrino 7.0 or later) Enable debugging output. Additional `-g` options increase the level of the output.

`-l` ***argument***

("el") Set the object load behavior, as follows:

- 0 — load directory names and objects on demand. Default.
- 1 — load all directory and object names on startup, but don't load the object contents. Load the object contents on demand.
- 2 — load directories, objects, and object contents on startup.

`-m` ***mount***

Specify the mountpath for PPS. The default is **/pps/**.

`-P` ***priority***

(QNX Neutrino 6.6 or later) Specify the priority of the persistence thread. The default is `10`.

`-p` ***path***

Set the path for backing up the persistent storage. The default is **/var/pps**.

`-T` ***tolerance***

(QNX Neutrino 6.6 or later) The periodic persistence flush interval tolerance, in milliseconds. The default is off.

`-t` ***period***

Specify the periodicity of the forced persistence, in milliseconds. For example, `-t 5000` forces the PPS service to write to persistent storage every five seconds. The default is no forced persistence.

`-U` ***uid***`[`**:*gid***`[`**,*sup_gid***`]*]`

(QNX Neutrino 6.6 or later) Once running, run as the specified user (and optionally groups), so that the program doesn't need to run as **root**.

`-v`

Enable verbose mode. Increase the number of "v"s to increase verbosity.

Chapter 2
Objects and Their Attributes

The QNX Neutrino PPS service is a system with objects whose properties a publisher can modify.

Clients that subscribe to an object receive updates when that object changes, that is, when the publisher has modified it.

With PPS, your application can:

- publish changes to objects
- subscribe to objects to receive notifications of changes
- both publish and subscribe

Object files

PPS objects are implemented as files in a special PPS filesystem. By default, PPS objects appear under **/pps**, but this path depends on the `-m` option used when you start `pps`. There's never more than one instance of a PPS object, so changes to that object are immediately visible to subscribers.

Objects can contain attributes. Each attribute is represented by a line of text in the object's file. So, for example, you might publish an object called `Time` that represents the time of day and has integer attributes representing the current hour, minute, and second as follows:

```
@Time
hour::17
minute::04
second::38
```

In this case, the filename is **Time**, and each attribute is a text string in that file.

Because PPS objects are represented by files, you can:

- Create directories and populate them with PPS objects by creating files in those directories.

  In order to create an object, you need to have write permission in the PPS directory or the appropriate subdirectory.

- Use the *open()*, then the *read()* and *write()* functions to query and change PPS objects.
- Use standard utilities as simple debugging tools.

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 Software Systems, whose Internet web domain name is “qnx.com” should use **/pps/qnx**, while an organization with the domain name “example.net” should use **/pps/example**.

CAUTION: PPS objects are accessed through the filesystem and look like normal POSIX files. However, they aren't standard POSIX files, and some PPS behaviors differ from standard POSIX behaviors. For instance, if the allocated read buffer is too small for the data being read in, the read doesn't return a partial result; it fails.

Attribute order

PPS doesn't guarantee that attributes will be read in the same order that they were written to the object. That is, a publisher may write:

```
@Time
hour::17
minute::04
second::38
```

This may be read in by a subscriber as, for instance:

```
@Time
second::38
hour::17
minute::04
```

Related Links

Subscribing (p. 29)
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.

Options and Qualifiers (p. 39)

Special objects

PPS directories can include special objects that you can open to facilitate subscription behavior. The table below lists these special objects:

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, see "*Subscribing to multiple objects*" in the Subscribing chapter.

Object and directory sizes

Since PPS holds its objects in memory, they are small. Each object is allocated 32 kilobytes. This doesn't mean that each object uses 32 kilobytes of memory at run time; it uses only the amount of memory needed to internally represent its current attributes.

The number of PPS directories and objects is limited only by available memory. The depth of PPS directories is limited by the fact that the full pathnames of objects are stored in the persistent directory as files; the size of these pathnames is limited by the maximum filename size supported by the persistent filesystem used.

- PPS objects shouldn't be used as a dumping ground for large amounts of data. The size of most PPS objects should be measured in hundreds of *bytes*, *not* in kilobytes.
- PPS has been tested with the Power-Safe filesystem (**fs-qnx6.so**), using the default PPS options. This configuration supports a total pathname length of 517 bytes, with no individual pathname element longer than 508 bytes. That is, a nesting depth of 50 is possible on **fs-qnx6.so**, provided that the total length of the path is less than 517 bytes.

Change notification

PPS informs publishers and subscribers when it creates, deletes, or truncates an object.

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:

Prefix	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 nonpersistent attributes have been deleted. For more information, see *Options and Qualifiers*.

In addition, when an object is deleted, PPS sends a single `-@objectname` to any application that has that object open. Typical behavior for an application receiving this notification is to close the open file descriptor, since the file is no longer visible in the filesystem (POSIX behavior).

Attribute options always *precede* both the special character and the object or attribute name. See "*Object and attribute qualifiers*" in the Options and Qualifiers chapter.

Object syntax

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. For example, the PPS object "**PlayCurrent**" in the directory **/pps/media** will appear in a file listing as simply **/pps/media/PlayCurrent**.

In the results of a read of a PPS file, the first line identifies 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.

Suppose the PPS object "**PlayCurrent**" in the example above contains attributes describing the metadata for the currently playing song in a multimedia application. Let's assume that the attributes have the following format:

```
@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 *read()* call on this file returns 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.

Attribute syntax

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

Attribute names may consist of alphanumeric characters, underscores, and periods, but must begin with a letter or underscore. Attribute lines in a PPS object file 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 doesn't 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**
`::`	Also referred to as null encoding. Simple text terminated by a linefeed.
`: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"
`:b:`	Boolean
`:n:`	Numeric
`:b64:`	Base64 encoding
`:json:`	JavaScript Object Notation 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`)

PPS includes some useful routines to help with encoding; see the *PPS Encoding and Decoding API* and *PPS API reference* chapters.

Chapter 3 Persistence

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

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

- PPS may be used to create objects that are rarely (or never) published or subscribed to, but that require persistence.
- “Shutdown” means the orderly exit of the PPS server process. An orderly exit can be triggered by SIGTERM, SIGINT, or SIGQUIT. Since other signals (such as SIGKILL, SIGSEGV, and SIGABRT) don't result in an orderly exit, they don't constitute a “shutdown” for the purposes of persistence.

Persistent storage

PPS supports persistent storage across reboots. This storage requires a reliable filesystem.

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

- disk — Power-Safe 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 doesn't support a filesystem), you can create your own client that subscribes to the PPS object to be saved. On each object change, PPS will notify your client, allowing the client to update the NVRAM in real time.

Persistence and filesystem limitation

The persistence directory where PPS objects are stored uses exactly the same directory hierarchy as the PPS root directory. Object persistence is, therefore, limited by the path and filename lengths as well as the directory nesting limits of the underlying filesystem.

For example, the QNX Neutrino NFS server supports a maximum nesting depth of 15 levels. This limit also applies to PPS using this service.

Saving objects

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

When PPS saves its data in a persistent filesystem, it saves each object in its own file, in a directory hierarchy that reproduces the hierarchy of the PPS object tree. For example, with a default configuration, the PPS object **/pps/example/object1** is stored at **/var/pps/example/object1**.

The default location for the PPS directory is **/var/pps**. You can use the PPS `-p` option to change this location.

- It's possible to edit the persistent version of the PPS files. If security is important to your system, you should have PPS save its data in an encrypted filesystem.
- You can set object and attribute qualifiers to have PPS *not* save specific objects or attributes.

Changing the directory for persistent storage

The root PPS object tree (**/pps** by default) may look something like this:

```
# pwd
/pps
# ls -1F
accounts/
applications/
qnx/
qnxcar/
servicedata/
services/
system/
#
```

PPS populates its root object tree from the *persistence* tree (**/var/pps** by default), where the objects and attributes that you want to persist are stored.

To specify a different directory for persistent storage:

1. Create your own persistence directory (e.g., `mkdir /myobjects`).
2. Start the PPS service from a different mountpoint (e.g., **/fs/pps**) and specify your new persistence directory:

   ```
   pps -m /fs/pps -p /myobjects
   ```

You may want to run PPS with the `-t` option, which configures PPS to write to persistent storage at the interval you specify. Without the `-t`, you won't see any changes in your persistence directory until PPS exits. For more information, see *Running PPS*.

Loading objects

When PPS starts up, it immediately builds the directory hierarchy from the encoded filenames on the persistent filesystem.

In its default configuration, PPS defers loading the objects in the directories until the 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.

You can change the configuration by providing the `-l` (“el”) option on startup to have PPS:

- load directory and object names (but not the object contents), or
- load directories, objects, and object contents

For more information, see *Running PPS*.

Chapter 4
Publishing

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 nonblocking.

For a simple example, see "*Publishers*" in the Examples appendix.

When you write an attribute to the file, do so in a single operation. A single *write()* to an object is required to guarantee that simultaneous writes from multiple publishers can be handled correctly. For example, instead of this:

```
write( fd, "state::", 7);
if ( state == 0 )
   write( fd, "off", 3);
else
   write( fd, "on", 2);
```

do something like this:

```
snprintf( buf, sizeof(buf), "state::%s", state ? "on" : "off");
write( fd, buf, strlen(buf) );
```

Creating, modifying, and deleting objects and attributes

You can create, modify, and delete objects and attributes, as shown in the following table:

If you want to:	**Do this:**
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.
Delete an object	Delete the object file.
Create a new attribute	Write the attribute to the object file.
Modify an attribute	Write the new attribute value to the object file. Note that there's no need to *lseek()* to the attribute's position in the file first; seeking has no meaning in PPS.
Delete all existing attributes	Open the object with O_TRUNC.
Delete one attribute	Prefix its name with a minus sign, then call *write()*. For example: `// Delete the "url" attribute` `sprintf( ppsobj, "-url\n" );` `write( ppsobj-fd, ppsobj, strlen( ppsobj ) );`

CAUTION: Note the following about deleting attributes:

- Calling *ftruncate()* on an object file deletes all the object's attributes, whatever the value of the *length* argument.
- A simple Bourne shell redirection instruction (such as `echo attr::hello > /pps/object`) entered from the command line opens an object with O_TRUNC and deletes all attributes.

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 that applies to different attributes for the same object.

It's safe for more than one publisher to write to the objects because the `pps` manager guarantees that each PPS *write()* is atomic.

For example, in a multimedia system with a `PlayCurrent` object, **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.

In the example above, suppose the `PlayCurrent` object has the following attribute values:

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

If **io-media** updated the `time` attribute of the `PlayCurrent` object as follows:

```
// Update the "time" attribute
sprintf( ppsobj, "time::2.32\n" );
write( ppsobj-fd, ppsobj, strlen( ppsobj ) );
```

and then the HMI updated the `duration` attribute as follows:

```
// Update the "duration" attribute
sprintf( ppsobj, "duration::4.02\n" );
write( ppsobj-fd, ppsobj, strlen( ppsobj ) );
```

The result would be:

```
@PlayCurrent
author::Beatles
album::Abbey Road
title::Come Together
duration::4.02
time::2.32
```

For another example, see "*Publishers*" in the Examples appendix.

Chapter 5
Subscribing

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.

- PPS has a limit of 200 open file descriptors per PPS object.
- The behavior of a PPS read differs from standard POSIX behavior. With PPS, if the allocated read buffer is too small for the data being read in, the read doesn't return a partial result; it fails.

It's safe for more than one subscriber to read from the objects because the `pps` manager guarantees that each PPS *read()* is atomic.

Attribute order

PPS doesn't guarantee that attributes will be read in the same order that they were written to the object. That is, a publisher may write:

```
@Time
hour::17
minute::04
second::38
```

This may be read in by a subscriber as, for instance:

```
@Time
second::38
hour::17
minute::04
```

For information about routines that will help you parse attributes, see the *PPS Encoding and Decoding API* and *PPS API reference* chapters.

Waiting for data on a file descriptor

As is often the case, polling for PPS data is a bad idea. It's much better to wait for PPS data on a file descriptor, using one of the following mechanisms:

- a blocking *read()*
- the *ionotify()* mechanism to receive an event that you specify
- the *select()* function

A blocking *read()* is the simplest mechanism. Generally, you use *ionotify()* if you want to combine input from file descriptors with QNX Neutrino messaging in the same process. Use *select()* when you're handling multiple file descriptors from sockets, pipes, serial ports, and so on. These mechanisms are described below; for examples, see “*Subscribers*” in the Examples appendix.

CAUTION: By default, reads to PPS objects are nonblocking; that is, PPS defaults a normal *open()* to O_NONBLOCK, so that reads made by the client that opened the object don't block.

This behavior is atypical for most filesystems. It's done this way so that standard utilities won't 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, `tar` would never make it past the first file opened and read.

Using blocking reads

A blocking *read()* waits until the object or its attributes change, and 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 nonblocking reads, use the pathname: "`/pps/media/PlayList`"
- for blocking reads, use the pathname plus the option: "`/pps/media/PlayList?wait`"

For more information on the `?wait` option, see “*Pathname open options*.”

A typical loop in a subscriber would live in its own thread. For a subscriber that 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);
}
```

If you opened an object without the `?wait` option and want to change to blocking reads, you can clear the O_NONBLOCK bit by using *fcntl()*:

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

or *ioctl()*:

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

Using *ionotify()*

The PPS service implements *ionotify()* functionality, allowing subscribers to request notification via a pulse, signal, semaphore, etc. On notification of a change, you must issue a *read()* to the object file to get the contents of the object. For example:

```
/* Process events while there are some */
while ( ( flags = ionotify( fd, _NOTIFY_ACTION_POLLARM,
                            _NOTIFY_COND_INPUT, event ) != -1 )
          && (flags & _NOTIFY_COND_INPUT) )
{
    nbytes = read(fd, buf, sizeof(buf));
    if ( nbytes > 0 )
        process(buf);
}
/* If flags != -1, the event will be triggered in the future to get
   our attention */
```

For more information, see the entry for *ionotify()* in the QNX Neutrino *C Library Reference*.

Using *select()*

The *select()* function examines a set of file descriptors to see if they're ready for reading or writing. To use *select()* with PPS, set up an `fd_set` that includes the file descriptor for the PPS object. You can optionally set a time limit. For example:

```
FD_ZERO( &readfds );
FD_SET( fd, &readfds );

switch ( ret = select( fd + 1, &readfds, NULL, NULL, &timeout ) )
{
   case -1:
      /* An error occurred. */
      break;
   case  0:
      /* select() timed out. */
      break;
   default:
      if( FD_ISSET( fd, &readfds ) )
      {
         num_bytes = read( fd, buf, sizeof(buf) );
         if (num_bytes > 0)
```

```
                process(buf);
            }
        }
```

For more information, see the entry for *select()* in the *C Library Reference*.

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 *Options and Qualifiers* chapter.

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's 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.

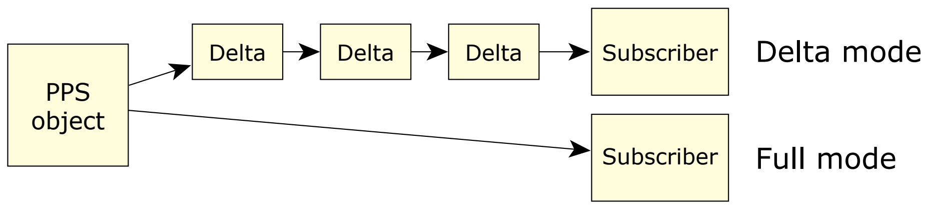

Figure 2: Comparison of PPS full and delta subscription modes.

In all cases, PPS maintains persistent objects with states—there's always an object. The mode used to open an object doesn't change the object; it determines only 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 doesn't 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 a number of special objects that facilitate subscribing to multiple objects:

Open this special object:	To receive notification of changes to:
.all	Any object in the directory
.notify	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. You can open multiple objects by calling *open()* and then *select()* on their file descriptors. More easily, you can open the special **.all** object, which merges all objects in its directory.

For example, assume the following object file structure under **/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 filesystem.
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 with 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 that you must remove before using the string.

Related Links

Options and Qualifiers (p. 39)

Using notification groups

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

Whenever data is 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 mounted at **/pps**, you could write something like the following:

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

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

sprintf(buf, "/pps/fish?notify=%s:water", noid);
fd1 = open(buf, O_RDONLY);
sprintf(buf, "/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's 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 that 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

PPS lets you use various pathname options when opening objects. PPS uses these pathname options to apply open options on the file descriptor used to open an object.

PPS also lets you use qualifiers to specify actions to take with an object or attribute (e.g., make an object nonpersistent 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 commas. For example:

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

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:

`backlog=`*num_kb*

The maximum delta size, in KB, to keep before flushing this OCB (Open Control Block). If you don't specify this option, or you specify a value of 0, then the default backlog value of 256 KB is used; you can override this by specifying the `-d` option when you start `pps`. The `flow` and `backlog` options are mutually exclusive.

If the total accumulated delta is more than *num_kb*, then a purge notification is generated, in the form:

`|@`*objname*

`cred`

Output the credentials for this object. You can use this option only in conjunction with the `flow` or `server` option. When a new client connects to a server or flow object, the connect notification also contains the client credentials, in the form:

`+@`*objname*`.`*client_id*`.n`*node_id*`.p`*process_id*`.u`*user_id*`.g`*group_id*`.g...`

`critical`

Designate the publisher as critical to the object. See "*The critical option*."

`crypt=`*domain*

Set the crypto domain for this object. In order to use this option, you must start the `pps` service with a `-p` option that specifies a location of persistent storage on a mounted Power-Safe (**fs-qnx6.so**) partition that contains encryption domains.

Assigning a PPS object to a domain means that it will be put into the specified encryption domain when it's saved in persistent storage. For example, this code:

```
open("/pps/test_obj?crypt=10", O_CREAT, 0666)
```

assigns the `test_obj` PPS object to encryption domain 10 when it's saved in persistent storage.

If you open multiple file descriptors for the same PPS object with different encryption domains, the last one is used. In the following example, the domain used is 11:

```
fd1 = open("/pps/test_obj", O_RDWR|O_CREAT, 0666);
fd2 = open("/pps/test_obj?crypt=10", O_RDWR, 0666);
fd3 = open("/pps/test_obj?crypt=11", O_RDWR, 0666);
```

`delta`

Open the object in delta mode. See "*Subscription modes*."

`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 and to monitor filesystem changes without having to also monitor attribute changes.

See "*Subscribing to multiple objects*."

`f=`*filter* ...

Place a filter on notifications based on changes to the listed attributes. See "*Filtering notifications*."

`flow`[=*num_kb*]

Treat the object as a server object, with purge and overflow notifications. The optional *num_kb* is the maximum backlog size in kilobytes. If you don't specify *num_kb*, or you specify a value of 0, the default backlog size of 256 KB is used; you can override this by specifying the `-d` option when you start `pps`. The format of the overflow notification is:

`^@`*objname*

The `flow` and `backlog` options are mutually exclusive. You can use the `hiwater` option with `flow` to generate overflow notifications.

See "*Server Objects*."

`hiwater=`*hw_percent*

Specify the flow high water mark as a percentage of the client backlog. If the backlog exceeds this percentage, overflow notifications are generated. Overflow notifications are generated for each delta buffered while the total accumulated delta size is above *hw_percent* of the specified backlog. The format of the overflow notification is:

```
^@objname
```

You can specify a value in the range from 1 to 99 percent; if you don't specify this option, the default of 100 percent is used. You can use the `hiwater` option only in conjunction with the `flow` option.

See "*Server Objects*."

`nopersist`

Make the object nonpersistent. When the system restarts, the object won't exist. The default setting is for all objects to be persistent and reloaded on restart. See "*Object and attribute qualifiers*."

`notify=`*id:value*

Associate the opened file descriptor with the notification group specified by *id*. This *id* is returned on the first read from an *open()* on the **.notify** file in the root of the PPS mountpoint. The *value* is an arbitrary string that you want to associate with the group.

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

See "*Subscribing to multiple objects*."

`opens`

Update an *_opens::rd,wr* attribute when the open count changes. This option lets you receive notifications about the number of file descriptors opened for an object for writing and reading. A notification is generated in following format:

```
%@objname.read_count,write_count
```

Notifications are sent each time that the number of readers or writers changes. These numbers increase each time that *open()* is called, and decrease when *close()* is called. O_RDONLY increases the number of readers, O_WRONLY the number of writers, and O_RDWR the number of readers and writers. For example, if you use the following code to create a PPS object:

```
fd = open("/pps/test_obj?opens", O_CREAT|O_RDWR, 0666);
```

then after reading from this file descriptor, the following notification is received:

```
%@test_obj.1,1
```

If you open the same PPS object with O_WRONLY, then the following notification is generated:

```
%@test_obj.1,2
```

`reflect`

Reflect attribute changes made on this object back to it.

`server`

Designate the publisher as a "server" for the object. See "*Server Objects*" for more information.

`verbose`[=*level*]

Set the verbosity level for this object. If you specify this option without an argument, the level is 9 (the highest).

`wait`

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

The `critical` option

You can use the `critical` option as a mechanism to clean up attributes on the abnormal termination of a publisher.

If you use this option when opening a file descriptor for a write, then when the file descriptor is closed, PPS deletes all nonpersistent 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()*, *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) isn't 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.

Filtering notifications

You can filter PPS notifications based on the names of attributes, the values of attributes, or a combination of the two.

To filter notifications, use the following syntax:

```
f=attrspec { +attrspec } ...
```

where *attrspec* is an attribute specification consisting of either an attribute's name or an expression specifying an attribute's value. If you want to be notified when the attribute is deleted, add a minus sign (−) before its name:

```
f=-attr
```

The syntax for specifying an attribute's value is:

attroperatorvalue

where *attr* is the attribute's name, *operator* is the operator used to determine the threshold for triggering notifications, and *value* is the value to compare to.

Supported operators are:

- <, <=, >, >=, =, ==, and != for integers. Integer values must be in the range of a `long long`; otherwise they're treated as strings.
- =, ==, and != for strings. Note that = and == are synonymous. String values can include the + character, but only by escaping it with \.

If you specify only an attribute's name, PPS notifies you of any updates where an attribute with that name is set. If you specify a name, operator, and value, PPS notifies you of any updates where the named attribute is set to a value that matches the given operator and value expression.

In both full and delta modes, the file descriptor will get notifications if any of the attribute specifications match:

- In full mode, the entire object is returned.
- In delta mode, only the specified attributes are returned. Changes to other attributes are filtered out.

In the following examples, the name of the object being opened is "objname":

- `/pps/objname?delta,f=attr1+attr2` — return change notifications for only the attributes named "attr1" and "attr2"
- `/pps/objname?delta,f=attr1<37` — return change notifications for only the attribute named "attr1" when its value is less than 37
- `/pps/objname?f=attr2<0+attr2>100` — return change notifications for the entire object when the attribute "attr2" has an integer value less than 0 or greater than 100
- `/pps/objname?delta,f=attr1=a\+b` — return change notifications only for the attribute "attr1" when it has a string value of "a+b"
- `/pps/objname?delta,f=attr1+attr2<10` — return change notifications only for the attribute "attr1" (for any change) and attribute "attr2" when it has an integer value less than 10.

Object and attribute qualifiers

PPS supports qualifiers to objects and their attributes.

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

n

Nonpersistence. If you set it on an object, the object becomes nonpersistent. If you set it on an attribute, the attribute becomes nonpersistent if the parent object is persistent; otherwise the qualifier is ignored. For more information, see "*Nonpersistence qualifier*."

i

Item. Specifies an item for a set attribute. For more information, see "*Item qualifier*."

- By default, qualifiers aren't set.
- On a *read()* call, you will see a preceding qualifier list "[*option letters*]" only for options that have been set.
- Attribute options always *precede* both the special character and the object or attribute name.

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 isn't specified, that qualifier isn't changed. For example:

To:	Specify:
Set the nonpersistence qualifier on an attribute	`[n]url::www.qnx.com`
Clear the nonpersistence qualifier on an attribute	`[-n]url::www.qnx.com`
Leave the current nonpersistence qualifier unchanged	`url::www.qnx.com`
Add `hammer` to a set	`[i]toolbox::hammer,`
Remove `screwdriver` from a set	`[-i]toolbox::screwdriver,`

Nonpersistence qualifier

You can use the nonpersistence (`n`) qualifier for objects and attributes. It's very useful for attributes that may not be valid across a system restart and don't need to be saved.

The table below describes the effects of the nonpersistence qualifier on PPS objects and attributes:

Syntax	Action	Object	Attribute
`n`	Set	Make the object and its attributes nonpersistent; ignore any persistence qualifiers set on this object's attributes.	Make the attribute nonpersistent.

Syntax	Action	Object	Attribute
`-n`	Clear	Make the object persistent; persistence of the object's attributes is determined by each attribute's qualifiers.	Make the attribute persistent, if the attribute's object is also persistent.

Setting the nonpersistence qualifier on an object overrides any nonpersistence qualifiers set on the object's attributes and is therefore convenient if you need to make a temporary object in which nothing persists.

Related Links

Persistence (p. 21)

Item qualifier

You can use the item (`i`) qualifier only for attributes. It causes PPS to treat the value following the qualifier as a set of items.

You must choose a character, such as a comma, to separate the items in a set. 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

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

```
[i]toolbox::hammer,
[i]toolbox::screwdriver,
```

To delete an item from a set, specify a minus sign:

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

The following examples show incorrect item syntax and aren't permitted:

```
[i]toolbox::hammer,screwdriver,
[-i]toolbox::hammer,screwdriver,
```

If you try to add an item more than once, PPS ignores the duplicate attempt. For example, if you write the following lines:

```
[i]toolbox::hammer,
[i]toolbox::hammer,
[i]toolbox::screwdriver,
```

a subscriber would read:

```
toolbox::hammer,screwdriver,
```

You can add null items to a set, like this:

```
[i]toolbox::,
```

A subscriber would read:

```
toolbox::hammer,screwdriver,,
```

Reading and writing from the command line

You can use standard command-line utilities to view the status of objects or to change their attributes.

To read objects, use the `cat` command. To write to objects, use `echo`. Here are some examples.

Using `cat` to read

View the current contents of the Bluetooth status object:

```
cat /pps/services/bluetooth/status
```

Monitor the changes to the **mpaudio** status object as they occur:

```
cat /pps/services/mm-control/mpaudio/status?wait,delta
```

Using `echo` to write

Set the *pause* attribute to `1` in the gears control object:

```
echo "pause:n:1" >> /pps/services/gears/control
```

Set the *demo_enabled* attribute to `false`, overwriting all other existing attributes in the **mytest** control object:

```
echo "demo_enabled:b:false" > /pps/mytest/control
```

Access Control List configuration file

The `-A` option for the `pps` command specifies the path to an Access Control List (ACL) configuration file, which can be used to set access permissions. Using the ACL configuration file eliminates reads to establish PPS object access permissions at startup, and can thus be used to reduce startup times.

For information about ACLs, see “Access Control Lists” in the QNX Neutrino *User's Guide* and the Working with ACLs chapter of the QNX Neutrino *Programmer's Guide*.

If you don't use an ACL configuration file, then you may need to use multiple `setfacl` commands (see the *Utilities Reference*) at startup to set access permissions for PPS objects. Since every `setfacl` command sends messages, this method of setting object access permissions increases messaging overhead. Using one or more ACL configuration files eliminates this messaging overhead and reduces the PPS startup time.

You can use more than one instance of the `-A` option to specify multiple ACL configuration files. In the event that the access permissions in the different files don't agree, the permissions in the last configuration file listed take precedence.

A PPS mountpoint can be associated with no more than one configuration.

You should place ACL configuration files in a secure storage location (not in the same location as the PPS objects).

ACL configuration file format

The ACL configuration file format is intended to facilitate both generation and parsing.

Descriptors

An ACL configuration file consists of zero or more text descriptors. A descriptor specifies properties of a PPS object path. In particular, it specifies access permissions (owner, mode, and ACL). A descriptor also records other important properties of the object, including whether it's a server object; whether it's persistent, and whether it should be created if it's missing on startup.

Descriptor format

A descriptor consists of two or more nonblank lines of text followed by a blank line (or end of file). The two mandatory lines of text define the:

- file or directory path
- file or directory details

These two mandatory lines may optionally be followed by an ACL, in either short or long text form.

The permissions described by the ACL (if one is present) take precedence over those specified in the details line. An ACL must be of a form usable by the *acl_from_text()* function (i.e., either short or long text form). The ACL must also be complete and valid according to *acl_valid()*. Specifically, an extended ACL must include an explicit ACL_MASK entry. No mask is computed if one is missing.

Leading and trailing whitespace are stripped from lines before processing.

Comments are introduced by the "#" character, and run to the end of the line; they are syntactically equivalent to whitespace.

Paths

Paths must be specified relative to the PPS mountpoint. They may *not* contain:

- extraneous path separators or relative components such as "." or ".."
- leading or trailing whitespaces
- the "#" character

Paths for directories *must* end with a single separator character.

Details

The details line must not contain extraneous whitespace, and must be of the form:

```
user:group:mode[:property[,property...]]
```

where:

- *user* is the file or directory owner
- *group* is the file or directory group
- *mode* is a bit map of file permissions: read, write, and execute for user, group, and other (as well as the setuid, setgid, and sticky bits), stored as an octal number

The properties are optional and consist of zero or more of the following:

Property	Description
O_CREAT	The object should be created if it's missing.
nopersist	Disable persistence for this object and its attributes.
server	Treat the object as a server object.

Sample ACL configuration file

The following example shows ACL configurations for a directory with an ACL in short text form, and for a file:

```
a/directory/
nobody:nobody:2711:O_CREAT # comment
user::rwx
group::x
other::x
mask::x                    # comment
group:nto:x
```

```
a/directory/file
nobody:nobody:640
```

Chapter 7
Server Objects

PPS supports a special type of object called a *server object*. When a client writes to a server object, only the application (called the *server*) that created the object with the `?server` option gets the message. Other clients can't see that message.

At write time, PPS appends a unique identifier to the object name so that the server knows which client connection is sending the message. This allows the connection to have stateful information. For example:

```
@foo.1234
```

indicates object `foo` with client identifier `1234`. When a client connects, the server reads a new object that is prefixed with a + symbol (for example, `+@foo.1234`). Disconnects are sent to the server, and the + prefix is changed to a - prefix.

When a server replies, it must write the object name with the unique identifier appended so that the response is sent only to the client that is marked by the unique identifier. If a server doesn't append the unique identifier to the object name, the message is broadcast to all clients that are connected to the object.

For more information about sending and receiving messages from server objects, see "*Working with server objects*."

An application that opens an object with the `?server` option automatically becomes a critical publisher of that object. It also automatically receives notifications in delta mode.

CAUTION: Don't duplicate file descriptors of server objects. Behavior with duplicate server file descriptors is undetermined.

The following figure shows a PPS transaction using the `?server` option:

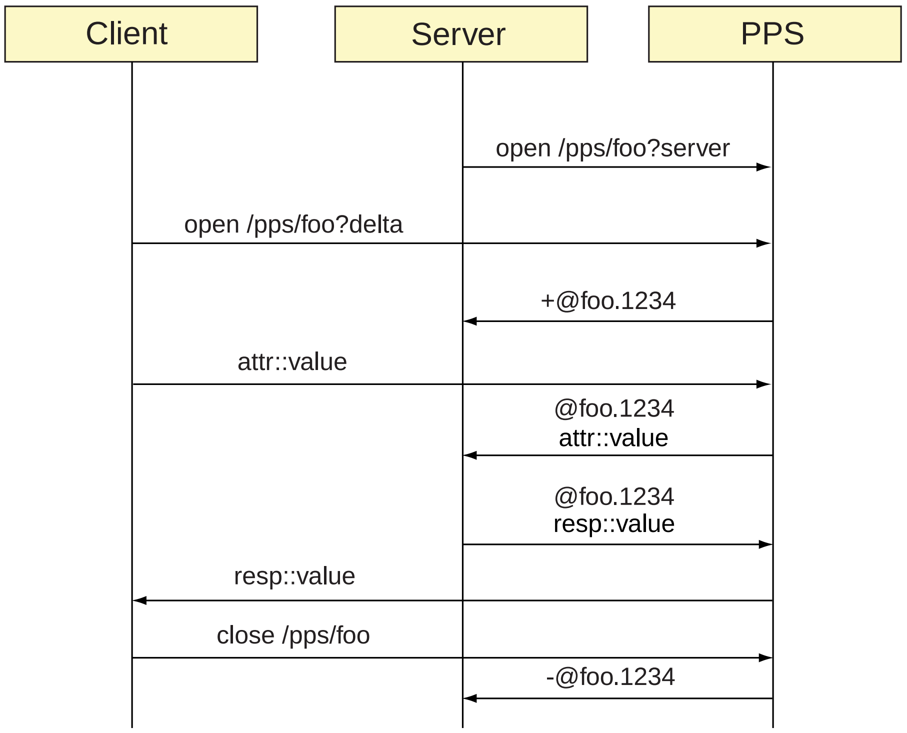

Figure 3: Using the `?server` option.

Working with server objects

PPS server mode provides point-to-point communication between a single server and one or more clients. Messages are formatted as for standard PPS usage, but there is no data persistence and there is no concept of a current object state. A client opening a PPS server object won't receive anything until the server explicitly sends a message.

Implementing a server or client in C/C++ is much easier if you use the PPS encoding and decoding API, since attributes are frequently not simple strings. See the *PPS Encoding and Decoding API* chapter for more information.

Sending messages through PPS server objects

A client and a server communicating through a PPS server object can use whatever messaging format is required.

In principle any PPS attributes can be sent to a server object. However, to aid interoperability between clients and servers, we recommend the protocol described in the following sections.

This protocol defines how messages are sent from one side to the other and how replies to messages are identified. The protocol is symmetric; the method used to send messages is independent of whether an application is a client or server.

Sending a message

A message that isn't sent as a response to another message can contain the following attributes:

`msg`

The type of message, typically the name of the function or command to execute. This should normally be short with no embedded spaces.

`id`

A string that identifies this instance of the message, which can be anything the client selects. The server always reflects it back. By convention, if the `id` is omitted, the server shouldn't provide a response. Therefore, the presence or absence of the `id` can be used to as a mechanism to request a response. Typically, the `id` is simply a sequence number represented as a string, although it might be anything. There is no requirement that the id be unique.

`dat`

Other data (for example, parameters) associated with the message. This attribute is optional. The `dat` attribute is frequently JSON encoded because there is often a need to encode more than a simple string.

Replying to a message

The attributes that can be sent in reply to a message are as follows:

`res`

A string that's identical to the value of the `msg` attribute of the message this is a response to.

`id`

A string that is identical to the value of the `id` attribute of the message this is a response to.

`dat`

Other data that is associated with the response. This attribute is optional. The `dat` attribute is frequently JSON-encoded because there is often a need to encode more than a simple string.

`err`

If present, this indicates that the request failed. It should be the number of the error as defined in **errno.h**. For example, for EBUSY you would put `16` in this field.

`errstr`

An optional string that could contain further explanation of the error or debugging information. For example, a more verbose explanation might be required for an error resulting from a request that includes a SQL statement, since errors might occur for diverse reasons.

Supporting cancellation of requests

If a server supports a request that can be cancelled after being made, we recommend that the cancellation take the form of a message with a value of "cancel" for the `msg` attribute and with the same `id` attribute as the original request. It isn't necessary to support the cancellation of requests, but if your application does support it, this standard message should be used.

Chapter 8
PPS Encoding and Decoding API

The PPS encoding and decoding API makes it easy to encode and decode complex data structures such as arrays and objects in PPS attributes and to ensure attributes are properly typed.

It's usually much easier to use these functions for all but the simplest of tasks and is often much safer. Although you can use the standard **libc** functions to encode and decode PPS data, experience has shown it's sometimes harder to do this correctly than one might think.

PPS attribute encodings

The PPS encoding and decoding API supports the attribute encodings described in *Attribute syntax*.

To allow interoperability between PPS clients of varying sophistication, the simplest encoding is always used by the encoder functions. For example, consider a string that can be represented using the null encoding, C encoding, and JSON. If the string being encoded doesn't include new line characters, the null encoding is used; if it does, the encoding switches to C.

The existence of these encodings should be important only if you're going to write your own PPS parsers or if you need to deal with PPS data from shell scripts.

Encoding PPS data

The encoding functions store PPS data in the data structure `pps_encoder_t`. In many cases, you can reuse the same `pps_encoder_t` structure multiple times. To begin encoding, you call either *pps_encoder_initialize()* if you're starting with a `pps_encoder_t` structure that hasn't been initialized, or *pps_encoder_reset()* to start again with one that you've previously used.

Following this, you make calls to functions such as *pps_encoder_add_string()* or *pps_encoder_add_int()* to add data elements. Once everything has been added, you can obtain a pointer to the encoded data by calling *pps_encoder_buffer()*, and you can find out the length of the encoded data by calling *pps_encoder_length()*.

The PPS encoder functions can encode both simple attribute types such as strings and numbers, as well as complex types, including objects and arrays. To create objects, call the function *pps_encoder_start_object()* to start the object, and *pps_encoder_end_object()* to end it. To create arrays, you use *pps_encoder_start_array()* and *pps_encoder_end_array()* instead. Objects and arrays must be properly nested.

While all the functions for adding data return a status, it's typically not necessary to check it after each call. Once any function fails, all subsequent calls will fail until the encoder is reset. Thus a reasonable strategy is to make all calls assuming they will succeed and only test the return value of *pps_encoder_buffer()*.

To take a simple example, suppose we want to encode PPS data to represent GPS information. In this case, the PPS data has a `speed` attribute representing the current speed, a `city` attribute with the name of the current city, and a `position` attribute that contains the latitude and longitude of the current position. You might use the following code to encode this data:

```
pps_encoder_t encoder;

pps_encoder_initialize(&encoder, false);
pps_encoder_start_object(&encoder, "@gps");
pps_encoder_add_double(&encoder, "speed", speed);
pps_encoder_add_string(&encoder, "city", city);
pps_encoder_start_object(&encoder, "position");
pps_encoder_add_double(&encoder, "longitude", lon);
pps_encoder_add_double(&encoder, "latitude", lat);
pps_encoder_end_object(&encoder);
pps_encoder_end_object(&encoder);

if ( pps_encoder_buffer(&encoder) != NULL ) {
    write( fd,
           pps_encoder_buffer(&encoder),
           pps_encoder_length(&encoder) );
}
pps_encoder_cleanup(&encoder);
```

The purpose of each call is as follows:

pps_encoder_initialize()

Initializes the `pps_encoder_t` structure from an unknown state. The second parameter is `false`, indicating that we're encoding as PPS data. To have the data encoded as JSON, pass `true` instead.

pps_encoder_start_object()

Begins a new object with a name `@gps`. For writing ordinary PPS data, you don't need to start and end an object. If this were part of a server using PPS server mode, this step would be necessary to address a message to a single client.

pps_encoder_add_double(), ***pps_encoder_add_string()***

Adds double and string attributes to the object. There are also functions to add integers, Booleans, and nulls. In this case, an attribute name is being passed in these calls since we're encoding into an object; if we were in the middle of an array, we would just pass NULL for the names.

pps_encoder_start_object(), ***pps_encoder_end_object()***

Because we want both the latitude and longitude to be contained within a single position attribute, we have to call *pps_encoder_start_object()* again. Having done this, we then call *pps_encoder_add_double()* to add the latitude and longitude and then *pps_encoder_end_object()* to return back to the PPS level.

pps_encoder_end_object()

Ends the PPS object we're encoding.

pps_encoder_buffer()

Returns a pointer to the encoder buffer. We're using it here in two ways. The encoder returns a non-NULL pointer only if there have been no errors and all objects and arrays have been ended. So the first call here is testing that we have valid data to send; the second call is providing the buffer in a call to write the data. The string in the buffer has a zero byte to terminate it.

pps_encoder_length()

Returns the length of the data.

The resulting PPS object should look like this:

```
@gps
speed:n:65.412
city::Ottawa
position:json:{"latitude":45.6512,"longitude":-75.9041}
```

Decoding PPS data

For the purposes of the decoder functions of this API, an object is a container that holds zero or more attributes of any type, each of which can be referenced by name. An array is a container that holds zero or more values that are referenced by position. Unlike arrays in C, an array here can use a different data type for each element. When the JSON encoding is used, individual PPS attributes can themselves be objects or arrays, which can be nested.

The PPS decoder functions allow both PPS- and JSON-encoded data to be parsed. Once a data string has been parsed, the `pps_decoder_t` structure maintains a representation of this data as a tree. Immediately upon parsing the data, you are positioned at the root of the tree. Using the PPS decoder functions you can:

- extract simple types such as numbers or strings that are located at the current level
- move into more deeply nested objects or arrays by calling *pps_decoder_push()*
- return to an outer level by calling *pps_decoder_pop()*

The decoder is always positioned within some object or array, either at one of its elements or off the end at a special element of type PPS_TYPE_NONE. Many of the decoder functions take a name argument that indicates the PPS attribute or object property name to look for. If the name is NULL, the current element is used. When extracting data from arrays, the name must always be NULL because array elements don't have names. When you successfully extract a data element, for example by calling *pps_decoder_get_int()* or after you call *pps_decoder_pop()*, the position automatically moves to the next element. So, for example, you could extract all elements of an array of numbers using code like this:

```
while ( pps_decoder_type(decoder, NULL) != PPS_TYPE_NONE ) {
   pps_decoder_get_double(decoder, NULL, &values[i++]);
}
```

Let's look at a complete example, using the following PPS data:

```
@gps
city::Ottawa
speed:n:65.412
position:json:{"latitude":45.6512,"longitude":-75.9041}
```

To extract this data, you might use code like this:

```
const char *city;
double lat, lon, speed;
pps_decoder_t decoder;

pps_decoder_initialize(&decoder, NULL);
pps_decoder_parse_pps_str(&decoder, buffer);
pps_decoder_push(&decoder, NULL);
pps_decoder_get_double(&decoder, "speed", &speed);
pps_decoder_get_string(&decoder, "city", &city);
```

```
pps_decoder_push(&decoder, "position");
pps_decoder_get_double(&decoder, "latitude", &lat);
pps_decoder_get_double(&decoder, "longitude", &lon);
pps_decoder_pop(&decoder);

pps_decoder_pop(&decoder);

if ( pps_decoder_status(&decoder, false) == PPS_DECODER_OK ) {
    . . .
}
pps_decoder_cleanup(&decoder);
```

Let's take a look at each of these function calls:

pps_decoder_initialize()

Initializes the `pps_decoder_t` structure from an unknown state. You can skip the subsequent call to *pps_decoder_parse_pps_str()* and just pass the data to be parsed in this call, but typically you'll be parsing multiple buffers of PPS data, so you'll need both steps.

pps_decoder_parse_pps_str()

Parses the PPS data into the decoder's internal data structures. This process modifies the buffer that's passed in, and the internal structure uses pointers into this buffer, so the contents of the buffer must remain valid and unchanged until the parsing results are no longer needed.

pps_decoder_push()

Causes the decoder to descend into the `gps` object to allow the values of the attributes to be extracted. In this particularly simple situation, this step might seem unnecessary. But if, for example, the data came from reading the **.all** special object, which returns multiple objects, it's important to indicate which particular object you're extracting data from.

pps_decoder_get_double()*, *pps_decoder_get_string()

Extract data from the PPS object, in this case the `speed` and `city` attributes. Note that the order of these calls doesn't have to match the order of the attributes in the original PPS data.

pps_decoder_push()*, *pps_decoder_get_double()*, *pps_decoder_pop()

In this case, the `latitude` and `longitude` are both contained in a single JSON-encoded PPS attribute. Therefore, before extracting `latitude` and `longitude`, we have to descend into the object they're contained in by calling *pps_decoder_push()*. Having pushed into the object, we can extract the two values just as if they had been PPS attributes. After the position data has been extracted, we then call *pps_decoder_pop()* to go back a level in case we need to extract more PPS attributes.

pps_decoder_pop()

Performs the reverse action of the initial *pps_decoder_push()* and pops out of an object (in this case the **gps** object) to its parent. Calling *pps_decoder_pop()* in this case isn't really required, but if we had read **.all** and there were multiple objects contained in the data, we would need this call to advance to the next object.

pps_decoder_status()

Returns the status of the decoder object. If this is PPS_DECODER_OK, then all parsing and data extraction occurred successfully. The preceding calls have the effect that if they fail, they will update the decoder status, so that you can perform a series of operations and check the status at the end, rather than checking at each stage. In this case, the second parameter to *pps_decoder_status()* is `false`, meaning that the status shouldn't be reset to PPS_DECODER_OK when the function returns. Parsing more data using *pps_decoder_parse_pps_str()* also has the effect of resetting the status (unless it fails).

pps_decoder_cleanup()

Once you've finished with a decoder object, *pps_decoder_cleanup()* frees any memory that was allocated. You need to call this only when you no longer require the decoder; if you have more data to parse, you can simply call *pps_decoder_parse_pps_str()* again.

Handling unknown data

In some cases you'll need to deal with data that is of an unknown type or that varies.

The decoder provides the functions *pps_decoder_type()* and *pps_decoder_name()*, which provide the type and attribute name associated with the current element. These functions allow your application to iterate over an object in the same manner as an array, advancing from element to element and extracting the type, name, and value of each element.

At all times, the decoder is positioned on an element of an array or object. So, how do you know if you're within an object or array? The API provides the special attribute name ".", which refers to the current container much like "." refers to the current directory in a filesystem. Your application can determine the type of the current container by calling the function *pps_decoder_type()* with the "." attribute. For example:

```
if ( pps_decoder_type(&decoder, ".") == PPS_TYPE_ARRAY ) {
      . . .
}
```

A good example of how to handle unknown data is the *pps_decoder_dump_tree()* function, which takes an arbitrary object or array and dumps it to a file. The complete source for this function is as follows:

```
void
pps_decoder_dump_tree(pps_decoder_t *decoder, FILE *fp)
{
 pps_decoder_state_t state;

 pps_decoder_get_state(decoder, &state);
 pps_decoder_dump_collection(decoder, fp, 0);
 fprintf(fp, "\n");
 pps_decoder_set_state(decoder, &state);
}

static void
pps_decoder_dump_collection(pps_decoder_t *decoder, FILE *fp, int prefix)
{
 int len = pps_decoder_length(decoder);
 int i;
 int flags = pps_decoder_flags(decoder, ".");

 if ( flags ) {
  fprintf(fp,"(%s)", flags & PPS_DELETED ? "deleted" :
    flags & PPS_CREATED ? "created" :
    flags & PPS_TRUNCATED ? "truncated" :
    flags & PPS_OVERFLOWED ? "overflowed" :
    flags & PPS_PURGED ? "purged" : "other");
  return;
 }

 (void)pps_decoder_goto_index(decoder, 0);
 if ( pps_decoder_type(decoder,".") == PPS_TYPE_OBJECT ) {
  fprintf(fp, "{\n");
  for ( i = 0; i < len; ++i ) {
   fprintf(fp,"%*s%s (%s) ", prefix+2, "", pps_decoder_name(decoder),
     pps_datatypes[pps_decoder_type(decoder,NULL)]);
   pps_decoder_dump_value(decoder, fp, prefix + 2);
  }
  fprintf(fp,"%*s}",prefix, "");
 }
 else {
  fprintf(fp, "[\n");
  for ( i = 0; i < len; ++i ) {
   fprintf(fp,"%*s%d (%s) ", prefix+2, "", i,
     pps_datatypes[pps_decoder_type(decoder,NULL)]);
   pps_decoder_dump_value(decoder, fp, prefix + 2);
  }
  fprintf(fp,"%*s]", prefix, "");
 }
}

static void
pps_decoder_dump_value(pps_decoder_t *decoder, FILE *fp, int prefix)
{
 double dvalue;
 int64_t ivalue;
 bool bvalue;
 const char *svalue;

 switch ( pps_decoder_type(decoder,NULL) ) {
 case PPS_TYPE_NUMBER:
  if ( pps_decoder_is_integer(decoder, NULL) ) {
   (void)pps_decoder_get_int64(decoder, NULL, &ivalue);
   fprintf(fp, "%" PRId64, ivalue);
  }
  else {
   (void)pps_decoder_get_double(decoder, NULL, &dvalue);
   fprintf(fp, "%lf", dvalue);
  }
  break;
 case PPS_TYPE_BOOL:
  (void)pps_decoder_get_bool(decoder, NULL, &bvalue);
  fprintf(fp, "%s", bvalue ? "true" : "false");
  break;
 case PPS_TYPE_STRING:
  (void)pps_decoder_get_string(decoder, NULL, &svalue);
  fprintf(fp, "%s", svalue);
  break;
 case PPS_TYPE_ARRAY:
 case PPS_TYPE_OBJECT:
  (void)pps_decoder_push(decoder, NULL);
  pps_decoder_dump_collection(decoder, fp, prefix);
  (void)pps_decoder_pop(decoder);
  break;
 case PPS_TYPE_NULL:
 default:
  (void)pps_decoder_next(decoder);
 }
 fprintf(fp,"\n");
}
```

Dealing with errors

There are a number of ways to deal with errors when using the decoder functions.

Many of the functions return a status. For example, *pps_decoder_get_string()* returns PPS_DECODER_OK if it succeeds, and another value otherwise. While you can take the traditional approach by checking the return value of each function and then taking the appropriate action, this isn't the preferred method because it tends to be verbose and no more effective than the alternatives.

One way of handling errors is the "all or nothing" approach. In some cases, a PPS message (or JSON string), is expected to contain a particular set of attributes; if any are missing, it's considered an error. In this case, you can attempt to extract all the attributes expected and just check the final status of the `pps_decoder_t` structure. For example, if you need a message to contain the three attributes `name`, `size`, and `serial_no`, you could do this:

```
pps_decoder_get_string(decoder, "name", &name);
pps_decoder_get_int(decoder, "size", &size);
pps_decoder_get_string(decoder, "serial_no", &serial_no);

if ( pps_decoder_status(decoder, true) != PPS_DECODER_OK ) {
   printf("Bad message\n");
   . . .
}
```

In this case, rather than individually check whether each attribute was fetched successfully, we just check at the end that everything was OK.

The above method of error handling works fine in many cases, but sometimes there's no fixed set of attributes that must always be included. So another way of handling errors is to pre-initialize all variables, extract values from the PPS data, and just use the results. If an attribute was missing or of the wrong type, the corresponding variable is left with the value it was initialized with. For example:

```
char *name = NULL;
int size = -1;
char *serial_no = NULL;

pps_decoder_get_string(decoder, "name", &name);
pps_decoder_get_int(decoder, "size", &size);
pps_decoder_get_string(decoder, "serial_no", &serial_no);

if ( name != NULL ) {
. . .
}
```

You can, of course, use some hybrid of the two approaches. For example, you might fetch all mandatory attributes first, checking that the final status is OK, and then fetch all optional attributes, relying on having initialized your variables with appropriate default values.

Other features of PPS

Besides conveying the values of attributes, PPS can also signal when attributes and objects are deleted and when objects are created or truncated.

You can obtain this extra data by calling the *pps_decoder_flags()* function, which returns the flags associated with an object or attribute. These flags consist of any combination of the values in the enumeration `pps_attrib_flags_t`. For example, if it's possible for the `gps` object to be deleted, then before calling *pps_decoder_push()*, you might first do the following:

```
if ( pps_decoder_flags(&decoder, NULL) & PPS_DELETED ) {
   . . .
}
```

These flags are important when writing a program that acts as a PPS server. For example, to handle client connections and disconnections, you could use code such as:

```
char *clientid;

// The data read from pps isn't null-terminated, so we need to
// terminate it now.
buffer[len] = '\0';
pps_decoder_parse_pps_str(&decoder, buffer);

// Get the ID of this client
clientid = pps_decoder_name(&decoder);

if ( pps_decoder_flags(&decoder, NULL) & PPS_CREATED ) {
   // This is a new client
}
else if ( pps_decoder_flags(&decoder, NULL) & PPS_DELETED ) {
   // The client has just disconnected
}
else {
   // Regular message from client
}
```

Chapter 9
PPS API reference

This chapter describes the publicly visible PPS API functions and data types.

The PPS header file **pps.h** is found in the **$QNX_TARGET/usr/include** directory.

pps_attrib_flags_t

The states for PPS objects and attributes

The values enumerated 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 nonpersistent attributes have been deleted.
- PPS_OVERFLOWED — queued data overflowed because it wasn't read fast enough.

pps_attrib_t

```
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;
```

Description:

The `pps_attrib_t` data structure carries information about PPS objects and attributes obtained by running the *ppsparse()* function. It includes these members:

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's set to -1 if the index isn't 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 of PPS data that *ppsparse()* just parsed. It's set to NULL if no attribute name was found.
attr_index	`int`	The index for *attr_name* in the *attrnames* array. It's set to -1 if the index isn't found or *attrnames* is NULL.
encoding	`char *`	A pointer to a string that indicates the encoding used for the PPS attribute. This value is relevant only if the *ppsparse()* return value is PPS_ATTRIBUTE.
value	`char *`	A pointer to the value of a PPS attribute. This value is relevant only if the *ppsparse()* return value is PPS_ATTRIBUTE.
flags	`int`	Flags indicating that parsing has found a PPS special character prefixed to a line or that the line is incomplete.

Member	Type	Description
options	`int`	Indicates which nonnegated options are prefixed in square brackets to a line.
option_mask	`int`	A mask of the options (both negated and nonnegated) prefixed to a line.
quality	`int`	Not used.
line	`char *`	Pointer to the beginning of the line parsed by *ppsparse()*, for use in case of a parsing error.
reserved	`int array`	For internal use.

pps_decoder_cleanup()

Clean up a pps_decoder_t structure

Synopsis:

```
#include <pps.h>

void pps_decoder_cleanup(pps_decoder_t *decoder);
```

Arguments:

decoder

A pointer to the PPS decoder structure.

Library:

libpps

Description:

The function *pps_decoder_cleanup()* cleans up a `pps_decoder_t` structure, freeing any allocated memory. Call this function only when the decoder structure is no longer needed. You don't need to call *pps_decoder_cleanup()* between calls to *pps_decoder_parse_pps_str()* or *pps_decoder_parse_json_str()*.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_decoder_dump_tree()

Write a decoded PPS data structure in human-readable format to a file

Synopsis:

```
#include <pps.h>

void pps_decoder_dump_tree(pps_decoder_t *decoder, FILE *fp);
```

Arguments:

decoder

A pointer to the PPS decoder structure.

fp

A pointer to the file to write to.

Library:

libpps

Description:

The function *pps_decoder_dump_tree()* writes the contents of a decoded PPS data structure in human-readable format to the specified file.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_decoder_error_t

pps_decoder_error_t

The values enumerated by `pps_decoder_error_t` define the errors that can be returned by the PPS decoder functions. These values include:

- PPS_DECODER_OK — no error occurred.
- PPS_DECODER_NO_MEM — an error occurred while allocating memory during the parsing operation.
- PPS_DECODER_BAD_TYPE — there was a request for the wrong type of data.
- PPS_DECODER_NOT_FOUND — the requested item wasn't found.
- PPS_DECODER_PARSE_ERROR — an error occurred while parsing JSON-encoded data.
- PPS_DECODER_DELETED — the requested item was deleted.
- PPS_DECODER_CONVERSION_FAILED — a numeric conversion was out of range or would result in a loss of precision.
- PPS_DECODER_POP_AT_ROOT — a call was made to *pps_decoder_pop()* while at the root of the tree (that is, there is nothing to pop to).

pps_decoder_flags()

Return the flags associated with the current or named attribute

Synopsis:

```
#include <pps.h>

int pps_decoder_flags(pps_decoder_t *decoder,
                      const char *name);
```

Arguments:

decoder

A pointer to the PPS decoder structure.

name

The name of the attribute to provide the flags for. If NULL, the flags of the current node are returned.

Library:

libpps

Description:

The function *pps_decoder_flags()* returns the flags associated with either the current node or the node of the specified name.

Returns:

The flags of the specified attribute. See *pps_attrib_flags_t*.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_decoder_get_bool()

Extract a Boolean value from the current or named node

Synopsis:

```
#include <pps.h>

pps_decoder_error_t pps_decoder_get_bool(
                                    pps_decoder_t *decoder,
                                    const char *name,
                                    bool *value);
```

Arguments:

decoder

A pointer to the PPS decoder structure.

name

The name of the property to extract the value from. Specify NULL to extract the data from the current node.

value

A pointer to a Boolean to take the result.

Library:

libpps

Description:

The function *pps_decoder_get_bool()* extracts a Boolean value from the current node or the node of the specified name. Following successful extraction, the decoder advances to the next node.

Returns:

PPS_DECODER_OK

Success.

PPS_DECODER_BAD_TYPE

There was a type mismatch.

PPS_DECODER_DELETED

The attribute was deleted.

PPS_DECODER_NOT_FOUND

The specified node doesn't exist.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_decoder_get_double()

Extract a double value from the current or named node

Synopsis:

```
#include <pps.h>

pps_decoder_error_t pps_decoder_get_double(
                                    pps_decoder_t *decoder,
                                    const char *name,
                                    double *value);
```

Arguments:

decoder

A pointer to the PPS decoder structure.

name

The name of the property to extract the value from. Specify NULL to extract the data from the current node.

value

A pointer to a `double` to take the result.

Library:

libpps

Description:

The function *pps_decoder_get_double()* extracts a double value from the node having the specified name or, if the *name* argument is NULL, the current node. Following successful extraction, the decoder advances to the next node.

Returns:

PPS_DECODER_OK

Success.

PPS_DECODER_DELETED

The attribute was deleted.

PPS_DECODER_BAD_TYPE

There was a type mismatch.

PPS_DECODER_NOT_FOUND

The specified node doesn't exist.

PPS_DECODER_CONVERSION_FAILED

The value of the attribute is outside the range or would lose precision when converted to the specified type.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_decoder_get_int()

Extract an integer value from the current or named node

Synopsis:

```
#include <pps.h>

pps_decoder_error_t pps_decoder_get_int(
                                    pps_decoder_t *decoder,
                                    const char *name,
                                    int *value);
```

Arguments:

decoder

A pointer to the PPS decoder structure.

name

The name of the property to extract the value from. Specify NULL to extract the data from the current node.

value

A pointer to an integer to take the result.

Library:

libpps

Description:

The function *pps_decoder_get_int()* extracts an integer value from the node having the specified name, or if the *name* argument is NULL, the current node. Following successful extraction, the decoder advances to the next node.

Returns:

PPS_DECODER_OK

Success.

PPS_DECODER_BAD_TYPE

There was a type mismatch.

PPS_DECODER_DELETED

The attribute was deleted.

PPS_DECODER_NOT_FOUND

The specified node doesn't exist.

PPS_DECODER_CONVERSION_FAILED

The value of the attribute is outside the range or would lose precision when converted to the specified type.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_decoder_get_int64()

Extract a 64-bit integer value from the current or named node

Synopsis:

```
#include <pps.h>

pps_decoder_error_t pps_decoder_get_int64(
                                          pps_decoder_t *decoder,
                                          const char *name,
                                          int64_t *value);
```

Arguments:

decoder

A pointer to the PPS decoder structure.

name

The name of the property to extract the value from. Specify NULL to extract the data from the current node.

value

A pointer to a 64-bit integer to take the result.

Library:

libpps

Description:

The function *pps_decoder_get_int64()* extracts a 64-bit integer value from the current node or the node of the specified name. Following successful extraction, the decoder advances to the next node.

Returns:

PPS_DECODER_OK

Success.

PPS_DECODER_BAD_TYPE

There was a type mismatch.

PPS_DECODER_DELETED

The attribute was deleted.

PPS_DECODER_NOT_FOUND

The specified node doesn't exist.

PPS_DECODER_CONVERSION_FAILED

The value of the attribute is outside the range or would lose precision when converted to the specified type.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_decoder_get_state()

Return the current state of the decoder

Synopsis:

```
#include <pps.h>

void pps_decoder_get_state(pps_decoder_t *decoder,
                           pps_decoder_state_t *state);
```

Arguments:

decoder

A pointer to the PPS decoder structure.

state

A pointer to a structure to hold the state.

Library:

libpps

Description:

The function *pps_decoder_get_state()* sets the argument *state* to point to the current node of the decoder structure. Obtaining the state allows you to return the decoder to a known state after a sequence of data extraction or navigation calls.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_decoder_get_string()

Extract a string value from a node

Synopsis:

```
#include <pps.h>

pps_decoder_error_t pps_decoder_get_string(
                                            pps_decoder_t *decoder,
                                            const char *name,
                                            const char **value);
```

Arguments:

decoder

A pointer to the PPS decoder structure.

name

The name of the property to extract the value from. Specify NULL to extract the data from the current node.

value

A pointer to a location where the function can store a pointer to the string. Note that the resulting string is a pointer into the original buffer.

Library:

libpps

Description:

The function *pps_decoder_get_string()* extracts a string value from the node having the specified name, or if the *name* argument is NULL, the current node. Following successful extraction, the decoder advances to the next node.

Returns:

PPS_DECODER_OK

Success.

PPS_DECODER_BAD_TYPE

There was a type mismatch.

PPS_DECODER_DELETED

The attribute was deleted.

PPS_DECODER_NOT_FOUND

The specified node doesn't exist.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_decoder_goto_index()

Advance to the indicated element or property of an array or object

Synopsis:

```
#include <pps.h>

pps_decoder_error_t pps_decoder_goto_index(
                                    pps_decoder_t *decoder,
                                    int index);
```

Arguments:

decoder

A pointer to the PPS decoder structure.

index

The index of the element to go to. The index of the first element is `0` (zero).

Library:

libpps

Description:

The function *pps_decoder_goto_index()* advances to the indicated element or property of an array or object.

Returns:

PPS_DECODER_OK

Success.

PPS_DECODER_NOT_FOUND

There is no next property/element.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No

Safety:	
Thread	Yes

pps_decoder_initialize()

Initialize the decoder structure

Synopsis:

```
#include <pps.h>

pps_decoder_error_t pps_decoder_initialize(
                                   pps_decoder_t *decoder,
                                   char *str);
```

Arguments:

decoder

A pointer to the decoder data type to initialize.

str

An initial NULL-terminated string to parse. If it's NULL, you must call *pps_decoder_parse_pps_str()* to parse the string.

Library:

libpps

Description:

The function *pps_decoder_initialize()* initializes the specified decoder structure from an unknown state.

Returns:

PPS_DECODER_OK

Success.

>=1

An error occurred. See *pps_decoder_error_t*.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No

Safety:	
Thread	Yes

pps_decoder_is_integer()

Test whether a node is an integer

Synopsis:

```
#include <pps.h>

bool pps_decoder_is_integer(pps_decoder_t *decoder,
                            const char *name);
```

Arguments:

decoder

A pointer to the decoder structure.

name

The name of the property or attribute to examine. If NULL, the type of the current node is used.

Library:

libpps

Description:

The function *pps_decoder_is_integer()* returns `true` if the current node or the node of the given name (if the decoder is currently within an object) is an integer.

Returns:

`true` if the current node or the node of the given name (if the decoder is currently within an object) is an integer.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_decoder_length()

Return the number of elements or properties in the current object or array

Synopsis:

```
#include <pps.h>

int pps_decoder_length(pps_decoder_t *decoder);
```

Arguments:

decoder

A pointer to the decoder data type.

Library:

libpps

Description:

The function *pps_decoder_length()* returns the number of elements or properties in the current object or array.

Returns:

The property or element count.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_decoder_name()

Return the name of the current node

Synopsis:

```
#include <pps.h>

const char *pps_decoder_name(pps_decoder_t *decoder);
```

Arguments:

decoder

A pointer to the PPS decoder structure.

Library:

libpps

Description:

The function *pps_decoder_name()* returns the name of the current node. This is applicable only if the current node represents an object.

Returns:

The name of the current node.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_decoder_next()

Advance to the next node

Synopsis:

```
#include <pps.h>

pps_decoder_error_t pps_decoder_next(pps_decoder_t *decoder);
```

Arguments:

decoder

A pointer to the PPS decoder structure.

Library:

libpps

Description:

The function *pps_decoder_next()* advances to the next node in the current object or array.

Returns:

PPS_DECODER_OK

Success.

PPS_DECODER_NOT_FOUND

There is no next property/element.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_decoder_parse_json_str()

Parse a string of JSON-formatted data

Synopsis:

```
#include <pps.h>

pps_decoder_error_t pps_decoder_parse_json_str(
                                    pps_decoder_t *decoder,
                                    char *str);
```

Arguments:

decoder

A pointer to the PPS decoder structure.

str

A pointer to a string containing PPS data.

Library:

libpps

Description:

The function *pps_decoder_parse_json_str()* parses a string of JSON-formatted data. Except for the format of the data passed in the *str* argument, the behavior of this function is the same as for *pps_decoder_parse_pps_str()*.

Returns:

PPS_DECODER_OK

Success.

>=1

An error occurred. See *pps_decoder_error_t*.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No

Safety:	
Thread	Yes

pps_decoder_parse_pps_str()

Parse a string of PPS-formatted data

Synopsis:

```
#include <pps.h>

pps_decoder_error_t pps_decoder_parse_pps_str(
                                    pps_decoder_t *decoder,
                                    char *str);
```

Arguments:

decoder

A pointer to the PPS decoder structure.

str

A pointer to a null-terminated string containing PPS data.

Library:

libpps

Description:

The function *pps_decoder_parse_pps_str()* parses a string of PPS-formatted data into the decoder's internal data structures.

This function modifies the buffer that is passed in the *str* argument; the internal decoder structure makes use of pointers into this buffer. Consequently, the contents of the buffer must remain valid and unchanged until the results of parsing are no longer needed.

This function supports a number of PPS attribute encodings including the null encoding, `c` for C strings, `b` for Booleans, `n` for numbers, and `json` for JSON. The *str* argument must contain a null-terminated string.

Returns:

PPS_DECODER_OK

Success.

>=1

An error occurred. See *pps_decoder_error_t*.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_decoder_pop()

Ascend to the parent of the current node

Synopsis:

```
#include <pps.h>

pps_decoder_error_t pps_decoder_pop(pps_decoder_t *decoder);
```

Arguments:

decoder

A pointer to the PPS decoder structure.

Library:

libpps

Description:

The function *pps_decoder_pop()* ascends to the parent of the current object or array. Following this call, the current node will be the node that follows the object or array popped out of (that is, the sibling of the node that was current at the time this function was called).

Returns:

PPS_DECODER_OK

Success.

PPS_DECODER_POP_AT_ROOT

Already positioned at the root node.

>=1

An error occurred. See *pps_decoder_error_t*.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_decoder_push()

Descend into a lower-level node

Synopsis:

```
#include <pps.h>

pps_decoder_error_t pps_decoder_push(pps_decoder_t *decoder,
                                     const char *name);
```

Arguments:

decoder

A pointer to the PPS decoder structure.

name

The name of the property or attribute to descend into.

Library:

libpps

Description:

The function *pps_decoder_push()* descends into an array or object. If successful, subsequent data will be returned for properties or elements of that object or array.

Returns:

PPS_DECODER_OK

Success.

PPS_DECODER_BAD_TYPE

There was a type mismatch.

PPS_DECODER_DELETED

The attribute was deleted.

PPS_DECODER_NOT_FOUND

The specified node doesn't exist.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_decoder_push_array()

Descend into a lower-level array

Synopsis:

```
#include <pps.h>

pps_decoder_error_t pps_decoder_push_array(
                                    pps_decoder_t *decoder,
                                    const char *name);
```

Arguments:

decoder

A pointer to the PPS decoder structure.

name

The name of the object to descend into.

Library:

libpps

Description:

The function *pps_decoder_push_array()* descends into an array. If successful, subsequent data will be returned for properties or elements of that array. This function is identical to *pps_decoder_push()*, except that it will fail if the specified element isn't an array.

Returns:

PPS_DECODER_OK

Success.

PPS_DECODER_BAD_TYPE

There was a type mismatch.

PPS_DECODER_DELETED

The attribute was deleted.

PPS_DECODER_NOT_FOUND

The specified node doesn't exist.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_decoder_push_object()

Descend into a lower-level object

Synopsis:

```
#include <pps.h>

pps_decoder_error_t pps_decoder_push_object(
                                    pps_decoder_t *decoder,
                                    const char *name);
```

Arguments:

decoder

A pointer to the PPS decoder structure.

name

The name of the object to descend into.

Library:

libpps

Description:

The function *pps_decoder_push_object()* descends into an object. If successful, subsequent data will be returned for properties or elements of that object. This function is identical to *pps_decoder_push()*, except that it fails if the specified element isn't an object.

Returns:

PPS_DECODER_OK

Success.

PPS_DECODER_BAD_TYPE

There was a type mismatch.

PPS_DECODER_DELETED

The attribute was deleted.

PPS_DECODER_NOT_FOUND

The specified node doesn't exist.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_decoder_reset()

Reset the pps_decoder_t structure

Synopsis:

```
#include <pps.h>

void pps_decoder_reset(pps_decoder_t *decoder);
```

Arguments:

decoder

A pointer to the PPS decoder structure.

Library:

libpps

Description:

The function *pps_decoder_reset()* resets the `pps_decoder_t` structure so that it's positioned at the root node, which is the state the structure is in immediately after parsing data.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_decoder_set_position()

Position the decoder at the specified node

Synopsis:

```
#include <pps.h>

pps_decoder_error_t pps_decoder_set_position(
                                             pps_decoder_t *decoder,
                                             const char *name);
```

Arguments:

decoder

A pointer to the PPS decoder structure.

name

The name of the property or attribute to set the position to.

Library:

libpps

Description:

The function *pps_decoder_set_position()* searches for the node of the specified name in the current object and positions the decoder at that node.

Returns:

PPS_DECODER_OK

Success.

PPS_DECODER_NOT_FOUND

Call failed.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No

Safety:	
Thread	Yes

pps_decoder_set_state()

Set the state of the decoder

Synopsis:

```
#include <pps.h>

void pps_decoder_set_state(pps_decoder_t *decoder,
                           pps_decoder_state_t *state);
```

Arguments:

decoder

A pointer to the PPS decoder structure.

state

A pointer to a structure containing the decoder's state.

Library:

libpps

Description:

The function *pps_decoder_set_state()* returns the decoder to a state obtained by a call to *pps_decoder_get_state()* by setting the current node of the decoder to the node indicated by the *state* argument.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_decoder_state_t

pps_decoder_state_t

```
typedef struct {
    pps_node_t *node;
} pps_decoder_state_t;
```

Description:

The `pps_decoder_state_t` data structure stores the current state of the decoder structure, *pps_decoder_t*. This data structure is used by the functions *pps_decoder_get_state()* and *pps_decoder_set_state()*.

Member	Type	Description
node	`pps_node_t*`	A pointer to a node of a PPS object.

pps_decoder_status()

Return the current error status of the decoder

Synopsis:

```
#include <pps.h>

pps_decoder_error_t pps_decoder_status(
                                    pps_decoder_t *decoder,
                                    bool clear);
```

Arguments:

decoder

A pointer to the PPS decoder structure.

clear

Flag to indicate whether the error status should be reset. To reset, set to `true`. To leave the error status intact, set to `false`.

Library:

libpps

Description:

The function *pps_decoder_status()* returns the current error status of the decoder. If an error occurs during an attempt to extract data or push into objects, the decoder is set to an error state. Rather than check return codes after every operation, you can perform a series and then check if the entire set completed successfully.

Returns:

The error status of the decoder. See *pps_decoder_error_t*.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_decoder_t

pps_decoder_t

Description:

The `pps_decoder_t` data structure is a data type that carries information about parsed PPS objects and attributes. It's used by the *pps_decoder_*()* functions.

pps_decoder_type()

Return the data type of the current or named node

Synopsis:

```
#include <pps.h>

pps_node_type_t pps_decoder_type(pps_decoder_t *decoder,
                                 const char *name);
```

Arguments:

decoder

A pointer to the PPS decoder structure.

name

The name of the property or attribute to provide the type for. If NULL, the type of the current node is returned.

Library:

libpps

Description:

The function *pps_decoder_type()* returns the data type of the current node or the node of the given name (if currently with an object). The name "." can be used for the current object or array as a means of determining whether the current node represents an object or an array.

Returns:

The data type of the referenced node. See *pps_node_type_t*.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_encoder_add_bool()

Add a Boolean to the current object or array

Synopsis:

```
#include <pps.h>

pps_encoder_error_t pps_encoder_add_bool(
                                    pps_encoder_t *encoder,
                                    const char *name,
                                    bool value);
```

Arguments:

encoder

A pointer to the PPS encoder structure.

name

If the current node is an object, the name of the new attribute. If within an array, this must be NULL.

value

The boolean value to add.

Library:

libpps

Description:

The function *pps_encoder_add_bool()* adds a boolean value to the current object or array.

Returns:

PPS_ENCODER_OK

Success.

>=1

An error occurred. See *pps_decoder_error_t*.

The status is sticky. If a call to encode something fails, all subsequent calls will show failure until the encoder is reset.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_encoder_add_double()

Add a double to the current object or array

Synopsis:

```
#include <pps.h>

pps_encoder_error_t pps_encoder_add_double(
                                    pps_encoder_t *encoder,
                                    const char *name,
                                    double value);
```

Arguments:

encoder

A pointer to the PPS encoder structure.

name

If the current node is an object, the name of the new attribute. If within an array, this must be NULL.

value

The double value to add.

Library:

libpps

Description:

The function *pps_encoder_add_double()* adds a double value to the current object or array.

Returns:

PPS_ENCODER_OK

Success.

>=1

An error occurred. See *pps_decoder_error_t*.

The status is sticky. If a call to encode something fails, all subsequent calls will show failure until the encoder is reset.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_encoder_add_from_decoder()

Add part or all of the contents of a decoder to an encoder

Synopsis:

```
#include <pps.h>

pps_encoder_error_t pps_encoder_add_from_decoder(
                                    pps_encoder_t *encoder,
                                    const char *name,
                                    pps_decoder_t *decoder,
                                    const char *propName);
```

Arguments:

encoder

A pointer to the PPS encoder structure.

name

If the current node is an object, the name of the new attribute. If within an array, this must be NULL.

decoder

A pointer to the PPS decoder structure.

propName

The name of the property to add from the decoder. If NULL, this function adds the node at the current position in the decoder.

Library:

libpps

Description:

The function *pps_encoder_add_from_decoder()* adds part or all of the contents of a PPS decoder to a PPS encoder. This allows data previously decoded to be encoded again. Upon completion of the call the decoder is left in the same state as at the time of the call.

Returns:

PPS_ENCODER_OK

Success.

>=1

An error occurred. See *pps_decoder_error_t*.

The status is sticky. If a call to encode something fails, all subsequent calls will show failure until the encoder is reset.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_encoder_add_int()

Add an integer to the current object or array

Synopsis:

```
#include <pps.h>

pps_encoder_error_t pps_encoder_add_int(
                                    pps_encoder_t *encoder,
                                    const char *name,
                                    int value);
```

Arguments:

encoder

A pointer to the PPS encoder structure.

name

If the current node is an object, the name of the new attribute. If within an array, this must be NULL.

value

The integer value to add.

Library:

libpps

Description:

The function *pps_encoder_add_int()* adds an integer value to the current object or array.

Returns:

PPS_ENCODER_OK

Success.

>=1

An error occurred. See *pps_decoder_error_t*.

The status is sticky. If a call to encode something fails, all subsequent calls will show failure until the encoder is reset.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_encoder_add_int64()

Add a 64-bit integer to the current object or array

Synopsis:

```
#include <pps.h>

pps_encoder_error_t pps_encoder_add_int64(
                                    pps_encoder_t *encoder,
                                    const char *name,
                                    int64_t value);
```

Arguments:

encoder

A pointer to the PPS encoder structure.

name

If the current node is an object, the name of the new attribute. If within an array, this must be NULL.

value

The 64-bit integer value to add.

Library:

libpps

Description:

The function *pps_encoder_add_int64()* adds a 64-bit integer value to the current object or array.

Returns:

PPS_ENCODER_OK

Success.

>=1

An error occurred. See *pps_decoder_error_t*.

The status is sticky. If a call to encode something fails, all subsequent calls will show failure until the encoder is reset.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_encoder_add_json()

Add a JSON-encoded string to the current object or array

Synopsis:

```
#include <pps.h>

pps_encoder_error_t pps_encoder_add_json(
                                    pps_encoder_t *encoder,
                                    const char *name,
                                    const char *value);
```

Arguments:

encoder

A pointer to the PPS encoder structure.

name

If the current node is an object, the name of the new attribute. If within an array, this must be NULL.

value

The JSON-encoded string to add.

Library:

libpps

Description:

The function *pps_encoder_add_json()* adds a JSON-encoded string to the current object or array.

Returns:

PPS_ENCODER_OK

Success.

>=1

An error occurred. See *pps_decoder_error_t*.

The status is sticky. If a call to encode something fails, all subsequent calls will show failure until the encoder is reset.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_encoder_add_null()

Add a null to the current object or array

Synopsis:

```
#include <pps.h>

pps_encoder_error_t pps_encoder_add_null(
                                        pps_encoder_t *encoder,
                                        const char *name);
```

Arguments:

encoder

A pointer to the PPS encoder structure.

name

If the current node is an object, the name of the new attribute. If within an array, this must be NULL.

Library:

libpps

Description:

The function *pps_encoder_add_null()* adds a null to the current object or array.

Returns:

PPS_ENCODER_OK

Success.

>=1

An error occurred. See *pps_decoder_error_t*.

The status is sticky. If a call to encode something fails, all subsequent calls will show failure until the encoder is reset.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_encoder_add_string()

Add a string to the current object or array

Synopsis:

```
#include <pps.h>

pps_encoder_error_t pps_encoder_add_string(
                                            pps_encoder_t *encoder,
                                            const char *name,
                                            const char *value);
```

Arguments:

encoder

A pointer to the PPS encoder structure.

name

If the current node is an object, the name of the new attribute. If within an array, this must be NULL.

value

The string to add.

Library:

libpps

Description:

The function *pps_encoder_add_string()* adds a string to the current object or array.

Returns:

PPS_ENCODER_OK

Success.

>=1

An error occurred. See *pps_decoder_error_t*.

The status is sticky. If a call to encode something fails, all subsequent calls will show failure until the encoder is reset.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_encoder_buffer()

Return a pointer to the buffer holding the encoded data

Synopsis:

```
#include <pps.h>

const char *pps_encoder_buffer(pps_encoder_t *encoder);
```

Arguments:

encoder

A pointer to the PPS encoder structure.

Library:

libpps

Description:

The function *pps_encoder_buffer()* returns a pointer to the buffer holding the encoded data. This function returns a valid pointer only if there have been no errors and all objects have been closed off. Therefore, a null pointer can indicate either an error situation or incomplete data.

Returns:

A pointer to the encoder buffer, or NULL if not available.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_encoder_cleanup()

Clean up an encoder structure, releasing any allocated memory

Synopsis:

```
#include <pps.h>

void pps_encoder_cleanup(pps_encoder_t *encoder);
```

Arguments:

encoder

A pointer to the PPS encoder structure.

Library:

libpps

Description:

The function *pps_encoder_cleanup()* cleans up an encoder structure, releasing any allocated memory.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_encoder_clear_option()

Disable an encoder option

Synopsis:

```
#include <pps.h>

void pps_encoder_clear_option( pps_encoder_t *encoder,
                               pps_encoder_option_t option );
```

Arguments:

encoder

A pointer to the PPS encoder structure.

option

The option that you want to disable; one of:

- PPS_ENCODER_CLEAR_MEMORY — clear any allocated memory before freeing it.
- PPS_ENCODER_ENCODE_JSON — encode entirely as JSON. The default behavior is to encode as PPS, although the encoding automatically switches to JSON for complex attributes.

Library:

libpps

Description:

The *pps_encoder_clear_option()* function disables the specified option. You should change option states only immediately after calling *pps_encoder_initialize()*, *pps_encoder_cleanup()*, or *pps_encoder_reset()*.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_encoder_delete_attribute()

Add an attribute and mark it as deleted

Synopsis:

```
#include <pps.h>

pps_encoder_error_t pps_encoder_delete_attribute(
                                        pps_encoder_t *encoder,
                                        const char *name);
```

Arguments:

encoder

A pointer to the PPS encoder structure.

name

The name of the deleted attribute.

Library:

libpps

Description:

The function *pps_encoder_delete_attribute()* deletes an attribute from the current object. This call is valid only if encoding PPS data and only when called at the highest level.

Returns:

PPS_ENCODER_OK

Success.

>=1

An error occurred. See *pps_decoder_error_t*.

The status is sticky. If a call to encode something fails, all subsequent calls will show failure until the encoder is reset.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_encoder_end_array()

End the current array

Synopsis:

```
#include <pps.h>

pps_encoder_error_t pps_encoder_end_array(
                                        pps_encoder_t *encoder);
```

Arguments:

encoder

A pointer to the PPS encoder structure.

Library:

libpps

Description:

The function *pps_encoder_end_array()* ends the current array, returning to the array's parent object or array.

Returns:

PPS_ENCODER_OK

Success.

>=1

An error occurred. See *pps_decoder_error_t*.

The status is sticky. If a call to encode something fails, all subsequent calls will show failure until the encoder is reset.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_encoder_end_object()

End the current object

Synopsis:

```
#include <pps.h>

pps_encoder_error_t pps_encoder_end_object(
                                    pps_encoder_t *encoder);
```

Arguments:

encoder

A pointer to the PPS encoder structure.

Library:

libpps

Description:

The function *pps_encoder_end_object()* ends the current object, returning to the object's parent object or array.

Returns:

PPS_ENCODER_OK

Success.

>=1

An error occurred. See *pps_decoder_error_t*.

The status is sticky. If a call to encode something fails, all subsequent calls will show failure until the encoder is reset.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_encoder_error_t

pps_encoder_error_t

The values enumerated by `pps_encoder_error_t` define the errors that can be returned by the PPS encoder functions. These values include:

- PPS_ENCODER_OK — no error occurred.
- PPS_ENCODER_NO_MEM — an error occurred while allocating memory during the encoding operation.
- PPS_ENCODER_BAD_NESTING — a call to a *pps_encoder_start_*()* function didn't have a matching call to a *pps_encoder_end_*()* function.
- PPS_ENCODER_INVALID_VALUE — there was an attempt to add an invalid value to an encoder.
- PPS_ENCODER_MISSING_ATTRIBUTE_NAME — there was an attempt to add a PPS attribute with no attribute name.
- PPS_ENCODER_NOT_FOUND — there was an attempt to add a decoder property that doesn't exist.

pps_encoder_initialize()

Initialize an encoder structure

Synopsis:

```
#include <pps.h>

void pps_encoder_initialize(pps_encoder_t *encoder,
                            bool encodeJSON);
```

Arguments:

encoder

A pointer to the PPS encoder structure.

encodeJSON

If true, data is encoded as JSON rather than PPS.

Library:

libpps

Description:

The function *pps_encoder_initialize()* initializes an encoder structure from an unknown state. It initializes the memory used by the encoder to a known value, and then sets the encoding level to either 0 or 1, depending on the value specified for *encodeJSON* (0 for PPS encoding ; 1 for JSON encoding).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_encoder_length()

Return the length of the encoded data

Synopsis:

```
#include <pps.h>

int pps_encoder_length(pps_encoder_t *encoder);
```

Arguments:

encoder

A pointer to the PPS encoder structure.

Library:

libpps

Description:

The function *pps_encoder_length()* returns the current length of the data encoded by the encoder.

Returns:

The length of the encoded data, in bytes.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_encoder_reset()

Reset an encoder prior to encoding new data

Synopsis:

```
#include <pps.h>

void pps_encoder_reset(pps_encoder_t *encoder);
```

Arguments:

encoder

A pointer to the PPS encoder structure.

Library:

libpps

Description:

The function *pps_encoder_reset()* resets an encoder prior to encoding new data. If you're going to reuse the encoder, it's typically better to call *pps_encoder_reset()* than *pps_encoder_cleanup()*; if you use *pps_encoder_reset()*, the encoder will eventually acquire a buffer large enough such that it will require no subsequent memory allocation.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_encoder_set_option()

Enable an encoder option

Synopsis:

```
#include <pps.h>

void pps_encoder_set_option( pps_encoder_t *encoder,
                             pps_encoder_option_t option );
```

Arguments:

encoder

A pointer to the PPS encoder structure.

option

The option that you want to enable; one of:

- PPS_ENCODER_CLEAR_MEMORY — clear any allocated memory before freeing it.
- PPS_ENCODER_ENCODE_JSON — encode entirely as JSON. The default behavior is to encode as PPS, although the encoding automatically switches to JSON for complex attributes.

Library:

libpps

Description:

The *pps_encoder_set_option()* function enables the specified option. You should change option states only immediately after calling *pps_encoder_initialize()*, *pps_encoder_cleanup()*, or *pps_encoder_reset()*.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_encoder_start_array()

Start a new array

Synopsis:

```
#include <pps.h>

pps_encoder_error_t pps_encoder_start_array(
                                    pps_encoder_t *encoder,
                                    const char *name);
```

Arguments:

encoder

A pointer to the PPS encoder structure.

name

The name of this array if it's embedded in an object. This must be NULL if contained within an array.

Library:

libpps

Description:

The function *pps_encoder_start_array()* starts a new array. Subsequent elements are added to this array.

Returns:

PPS_ENCODER_OK

Success.

>=1

An error occurred. See *pps_decoder_error_t*.

The status is sticky. If a call to encode something fails, all subsequent calls will show failure until the encoder is reset.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_encoder_start_object()

Start a new object

Synopsis:

```
#include <pps.h>

pps_encoder_error_t pps_encoder_start_object(
                                    pps_encoder_t *encoder,
                                    const char *name);
```

Arguments:

encoder

A pointer to the PPS encoder structure.

name

If this object is within another object, *name* provides the object's property name. If the object is an element of an array, this must be NULL.

Library:

libpps

Description:

The function *pps_encoder_start_object()* starts a new object. Subsequent properties are added to this object.

Returns:

PPS_ENCODER_OK

Success.

>=1

An error occurred. See *pps_decoder_error_t*.

The status is sticky. If a call to encode something fails, all subsequent calls will show failure until the encoder is reset.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

pps_encoder_t

Encoder structure

Description:

The `pps_encoder_t` data structure carries information to be encoded as PPS objects and attributes. It's used by the *pps_encoder_*()* functions.

pps_node_type_t

pps_node_type_t

The values enumerated by `pps_node_type_t` define the possible types for PPS nodes during parsing. These values include:

- PPS_TYPE_NULL — the node is NULL.
- PPS_TYPE_BOOL — the node is an attribute with a Boolean data value.
- PPS_TYPE_NUMBER — the node is an attribute with a numeric data value.
- PPS_TYPE_STRING — the node is an attribute with a string data value.
- PPS_TYPE_ARRAY — the node is an array.
- PPS_TYPE_OBJECT — the node is an object.
- PPS_TYPE_NONE — the requested attribute doesn't exist.
- PPS_TYPE_UNKNOWN — the requested attribute exists but was invalid or not recognized.
- PPS_TYPE_DELETED — the requested attribute was deleted.

pps_options_t

PPS options

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

- PPS_NOPERSIST — nonpersistence option
- PPS_ITEM — item option

pps_status_t

ppsparse() return values

The values enumerated 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 — an attribute has been updated.
- PPS_ATTRIBUTE_DELETED — an attribute has been deleted.
- PPS_OBJECT_OVERFLOWED — there was an overflow in the data queued for an object.

ppsparse()

Parse an object read from PPS

Synopsis:

```
#include <pps.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.

objnames

A pointer to a NULL-terminated array of object names. If this value isn't 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 isn't 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 details about a line of PPS data.

parse_flags

Reserved for future use; specify 0 for this argument.

Library:

libpps

Description:

The function *ppsparse()* provides a lower-level alternative to the *pps_decoder_*()* functions (which themselves use *pps_parse()*). Except in special circumstances, it's better to use the *pps_decoder_*()* functions than to use *ppsparse()*.

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 in the `pps_status_t` data structure.

Returns:

>=0

Success.

-1

An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

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

Appendix A
Examples

This appendix includes some examples of PPS publishers and subscribers.

Publishers

Here are some examples of a simple publisher that loops forever, changing the value of an attribute. Each publisher also updates its own counter attribute, so we can be sure that the programs are actually working. They assume that the system is using the default PPS directory, **/pps**.

Although these examples check for the existence of **/pps** (to make sure that PPS is running), you need to create the **/pps/example** directory before running them.

One publisher

This publisher opens (perhaps creating) the `button` object, and then loops, toggling the button's state.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>

int main(int argc, char *argv[])
{
   int fd;
   int state = 0;
   char buf[256];
   struct stat stat_buf;
   int count = 0;
   ssize_t len, bytes_written;

   /* Is PPS running? */
   if (stat( "/pps", &stat_buf) != 0)
   {
      if (errno == ENOENT)
         printf ("The PPS server isn't running.\n");
      else
         perror ("stat (/pps)");
      return EXIT_FAILURE;
   }

   /* Create the "button" object (if it doesn't already exist). */
   fd = open( "/pps/example/button", O_RDWR | O_CREAT, S_IRWXU | S_IRWXG | S_IRWXO );
   if ( fd < 0 )
   {
      perror ("Couldn't open /pps/example/button");
      return EXIT_FAILURE;
   }

   /* Loop forever, toggling the state of the button. */
   while ( 1 )
   {
      usleep (500);
      count++;
      len = snprintf(buf, 256, "state::%s\npub1::%d", state ? "on" : "off", count);
      bytes_written = write( fd, buf, len );
      if (bytes_written == -1)
      {
         perror ("write()");
      }
      else if (bytes_written != len)
      {
         printf ("Bytes written: %d String length: %d\n", bytes_written, len);
      }

      if ( state == 0 )
         state = 1;
      else
         state = 0;
   }

   return EXIT_SUCCESS;
}
```

Note the following:

- You can use *stat()* on **/pps** to determine whether or not PPS is running. If you don't do this, and PPS isn't running, you'll get an error when you try to open the file for the object.
- In order to publish, you need to *open()* the file for (at least) writing.
- As noted earlier, when you write an attribute to the file, you must do so in a single operation, in order to guarantee that simultaneous writes from multiple publishers are handled correctly.

An additional publisher

This publisher updates the same object as the first publisher, but it toggles the button's color attribute.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>

int main(int argc, char *argv[])
{
   int fd;
   int color = 0;
   int count = 0;
   char buf[256];
   struct stat stat_buf;
   ssize_t len, bytes_written;

   /* Is PPS running? */
   if (stat( "/pps", &stat_buf) != 0)
   {
      if (errno == ENOENT)
         printf ("The PPS server isn't running.\n");
      else
```

```
         perror ("stat (/pps)");
      return EXIT_FAILURE;
   }

   /* Create the "button" object (if it doesn't already exist). */
   fd = open( "/pps/example/button", O_RDWR | O_CREAT, S_IRWXU | S_IRWXG | S_IRWXO );
   if ( fd < 0 )
   {
      perror ("Couldn't open /pps/example/button");
      return EXIT_FAILURE;
   }

   /* Loop forever, changing the color. */
   while ( 1 )
   {
      usleep (300);
      count++;
      len = snprintf(buf, 256, "color::%s\npub2::%d", color ? "red" : "green", count);
      bytes_written = write( fd, buf, len );
      if (bytes_written == -1)
      {
          perror ("write()");
      }
      else if (bytes_written != len)
      {
          printf ("Bytes written: %d String length: %d\n", bytes_written, len);
      }

      if ( color == 0 )
          color = 1;
      else
         color = 0;
   }

   return EXIT_SUCCESS;
}
```

This program is very similar to the first publisher. You can have as many publishers as you wish updating the same object; PPS makes sure that each update is an atomic operation—the publishers don't even have to know that each other exists.

Subscribers

Here are some examples of a subscriber that works with the publishers given above. They basically loop forever, reading updates to the `button` object, but each uses a different method of waiting for data.

Using a blocking *read()*

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>

int main(int argc, char *argv[])
{
   int fd;
   char buf[256];
   int num_bytes;

   fd = open( "/pps/example/button?wait,delta", O_RDONLY );
   if ( fd < 0 )
   {
      perror ("Couldn't open /pps/example/button");
      return EXIT_FAILURE;
   }

   /* Loop, echoing the attributes of the button. */
   while (1)
   {
      num_bytes = read( fd, buf, sizeof(buf) );
      if (num_bytes > 0)
      {
        write (STDOUT_FILENO, buf, num_bytes);
      }
   }

   return EXIT_SUCCESS;
}
```

Note the following:

- Because this program doesn't intend to publish data, it opens the file for the object in read-only mode.
- We specify the `wait` pathname open option so that the *read()* becomes a blocking operation, instead of the default nonblocking.

If both sample publishers are running, the output from this subscriber looks something like this:

```
@button
color::red
pub1::536
pub2::878
```

```
state::on

@button
color::green
pub2::879

@button
state::off
pub1::537

@button
color::red
pub2::880

@button
state::on
pub1::538
```

The output includes only the updates made by each publisher; the first time the subscriber reads the data, it gets all of the attributes, because their values are all new to this subscriber.

If we don't specify the `delta` option, the output includes all of the object's attributes whenever the subscriber reads the file, whether their values changed or not:

```
@button
color::green
pub1::654
pub2::1073
state::on

@button
color::red
pub1::654
pub2::1074
state::on

@button
color::green
pub1::654
pub2::1075
state::on

@button
color::green
pub1::655
pub2::1075
state::off
```

Using *ionotify()*

This subscriber uses the default nonblocking reads with *ionotify()*:

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
```

```
#include <fcntl.h>
#include <string.h>
#include <sys/neutrino.h>
#include <sys/netmgr.h>
#include <sys/siginfo.h>
#include <sys/iomgr.h>

int main(int argc, char *argv[])
{
   int fd;
   char buf[256];
   int num_bytes;
   int chid, coid, ret;
   struct sigevent my_event;
   struct _pulse pulse;

   fd = open( "/pps/example/button?delta", O_RDONLY );
   if ( fd < 0 )
   {
      perror ("Couldn't open /pps/example/button");
      return EXIT_FAILURE;
   }

   /* We need a channel to ourself. */
   chid = ChannelCreate (0);
   if ( chid < 0 )
   {
      perror ("Couldn't create a channel");
      return EXIT_FAILURE;
   }

   coid = ConnectAttach (0, 0, chid, _NTO_SIDE_CHANNEL, 0);
   if ( coid < 0 )
   {
      perror ("Couldn't create a connection");
      return EXIT_FAILURE;
   }

   /* Set up a sigevent to notify us of PPS data. */
   SIGEV_PULSE_INIT( &my_event, coid, SIGEV_PULSE_PRIO_INHERIT,
                     _PULSE_CODE_MINAVAIL, 0);

   /* Loop, echoing the attributes of the button. */
   while (1)
   {
      ret = ionotify( fd, _NOTIFY_ACTION_POLLARM, _NOTIFY_COND_INPUT, &my_event);
      while ((ret != -1) && (ret & _NOTIFY_COND_INPUT))
      {
         /* Data is available right now. Read the attributes and print
            them on stdout. */
         num_bytes = read( fd, buf, sizeof(buf) );
         if (num_bytes > 0)
         {
           write (STDOUT_FILENO, buf, num_bytes);
```

```
            }

            ret = ionotify( fd, _NOTIFY_ACTION_POLLARM, _NOTIFY_COND_INPUT, &my_event);
         }

         if (ret == -1)
         {
            perror ("ionotify() failed");
            return EXIT_FAILURE;
         }

         /* Block until PPS informs us that there's new data. */
         ret = MsgReceivePulse_r( chid, &pulse, sizeof(pulse), NULL );
         if ( ret < 0 )
         {
             printf ("MsgReceivePulse_r(): %s\n", strerror (ret));
             return EXIT_FAILURE;
         }
      }

    return EXIT_SUCCESS;
 }
```

Note the following:

- You can still use full or delta mode with this method.
- You need to create a channel and then attach to it. In this example, we're using a pulse as the notification method.
- The call to *ionotify()* requests a pulse whenever input is available on the file descriptor; if *ionotify()* returns _NOTIFY_COND_INPUT, the data was already available.
- Once you receive one pulse, you need to rearm with another call to *ionotify()*, so that's why this call is inside the `while` loop.
- If you don't use *MspReceivePulse()* or *MspReceivePulse_r()*, the program still works, but it uses a lot more CPU time because it never blocks, and hence effectively just polls, which is a bad thing to do.

Using *select()*

This subscriber uses nonblocking reads with *select()*, which requires less setup than the *ionotify()* version:

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/select.h>

int main(int argc, char *argv[])
{
   int fd;
   char buf[256];
   int num_bytes;
   int ret;
   fd_set readfds;
   struct timeval timeout;

   fd = open( "/pps/example/button?delta", O_RDONLY );
   if ( fd < 0 )
   {
      perror ("Couldn't open /pps/example/button");
      return EXIT_FAILURE;
   }

   timeout.tv_sec = 2;
   timeout.tv_usec = 0;

   /* Loop, echoing the attributes of the button. */
   while (1)
   {
      FD_ZERO( &readfds );
      FD_SET( fd, &readfds );

      switch ( ret = select( fd + 1, &readfds, NULL, NULL, &timeout ) )
      {
         case -1:
            perror( "select()" );
            return EXIT_FAILURE;
         case  0:
            printf( "select() timed out.\n" );
            break;
         default:
            if( FD_ISSET( fd, &readfds ) )
            {
               /* Read the attributes and print them on stdout. */
               num_bytes = read( fd, buf, sizeof(buf) );
               if (num_bytes > 0)
               {
                  write (STDOUT_FILENO, buf, num_bytes);
               }
            }
      }
   }

   return EXIT_SUCCESS;
}
```

Index