QNX® PPS

QNX Persistent Publish/Subscribe Developer's Guide

For QNX® Neutrino® 6.4.x

Published under license by:

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

Electronic edition published July 31, 2009.

Contents

About This Guide

The *QNX PPS Developer's Guide* includes:

- QNX PPS Service — a description of the QNX Persistent Publish/Subscribe service, and how to use it
- *psparse()* — a PPS API function that parses an object read from PPS

Typographical conventions

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

Reference	Example
Code examples	`if( stream == NULL )`
Command options	`-lR`
Commands	**`make`**
Environment variables	**PATH**
File and pathnames	`/dev/null`
Function names	*exit()*
Keyboard chords	Ctrl-Alt-Delete
Keyboard input	**`something you type`**
Keyboard keys	Enter
Program output	`login:`
Programming constants	NULL
Programming data types	**`unsigned short`**
Programming literals	`0xFF`, **`"message string"`**
Variable names	*stdin*
User-interface components	**Cancel**

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

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

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

Notes point out something important or useful.

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

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

Note to Windows users

In our documentation, we use a forward slash (`/`) as a delimiter in *all* pathnames, including those pointing to Windows files.

We also generally follow POSIX/UNIX filesystem conventions.

Technical support options

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

Chapter 1

QNX PPS Service

In this chapter. . .

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

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

PPS objects

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

The QNX PPS design is in many ways similar to many process control systems where the objects are control values updated by hardware or software. Subscribers can be alarm handling code, displays, and so on. Since there is a single instance of an object, persistence is a natural property that can be applied to it.

PPS object files

PPS objects exist as files in a special PPS filesystem. By default, PPS objects appear under **/fs/pps**.

You can create directories and populate them with objects by creating files in the directories, use *read()* and *write()* functions to query and change an object, and use standard utilities as simple debugging tools. For example:

Display all objects in the system:

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

Display the current state of an object:

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

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

Monitor all changes to attributes in objects in the media directory:

```
cat /fs/pps/media?dw
```

Change the attribute of an object:

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

PPS objects should not be used as a dumping ground for large amounts of data. It is expected that most objects will be measured in hundreds of *bytes* and not in hundreds of kilobytes.

Similarly, while PPS can be used for inter-process communication, it should *not* be used to replace QNX native messages for high speed directed communication. (PPS is, of course, built on top of QNX native messages.)

Object names

PPS always returns the object name (@name) on a *read()*. On a *write()*, the object name is optional, and it is expected that most publishers will omit it.

In the PPS filesystem listing, object names do not start with an "@". However, on a *read()* or *write()* an "@" is prefixed to object names to make them easily distinguishable from attributes.

Object rules

A PPS object is implemented as a file containing attributes. The first line of the file names the object and starts with an "@" character. The lines that follow define attributes.

For example, assume that we have an object called "**`PlayCurrent`**" in the PPS filesystem at **`/fs/pps/media/PlayCurrent`**. This object contains the metadata for the currently playing song. An *open()* call followed by a *read()* call on this file would return the name of the object (the filename), and the attribute pairs contained in the object:

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

Every line is terminated with a linefeed ("\n" in C, or hexadecimal **`0A`**).

Attribute names and rules

Attribute names (*attrname*) are composed from the set **`[a-z_][a-z0-9_]*`**. Attributes lines in a PPS object file and are of the form *attrname:encoding:value\n*, where *encoding* defines the encoding type for *value*.

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

- **`::`** — 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"
- **`:b64:`** — base 64 encoding

An attribute's *value* can be any sequence of characters, *except*: a null ("\0" in C, or hexadecimal **`0x00`**) a linefeed character ("\n" in C, or hexadecimal **`0x0A`**).

Object creation and deletion notification

When PPS creates or deletes an object it places a notification string into the queue of any subscriber or publisher that has the parent directory open (in either full or delta mode). The syntax of this string is simple: either a plus sign ("+") or a minus sign ("-") followed by the name of the created or deleted object. For example:

- **`+@objectname`** — the object was created
- **`-@objectname`** — the object was deleted

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 would be to close the file, since it is no longer visible in the filesystem (POSIX behavior), and only those processes with open file descriptors are able to continue to access it.

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; then calls *write()* to modify the object's attributes. This operation is non-blocking.

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

- To create a new object, create a file with the name of the object. The new object will come into existence with no attributes. You can then write attributes to the object, as required.
- To add a new attribute, write it to the object file.
- To modify an attribute, write the new attribute value to the object file.
- To delete all existing attributes, open the object file with O_TRUNC.
- To delete an object, simply delete the object file.

Multiple publishers

PPS supports multiple publishers that publish to the same PPS object, since different publishers may have access to data which applies to different attributes that need to be updated. For example, **io-media** may be the source of a **time::***value* attribute, while the HMI may be the source of a **duration::***value* attribute.

A publisher that changes only the **time** attribute will update only that attribute when it writes to the object. It will leave the other attributes unchanged. For example:

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

Subscribing

When a publisher changes an object, all subscribers that have subscribed to an object are informed of the change.

To subscribe to an object, a subscriber simply calls *open()* for the object file with O_RDONLY to publish only, or O_RDWR to publish and subscribe; then queries the object file with a *read()* call.

State changes are tracked per open file descriptor. When you first open a file, its state is "changed", and, thus, the first read of the file will never block. This behavior is implemented on the assumption that when an application opens an object file, for that application the state of the object is unknown. This behavior also allows you to concatenate (**cat**) the file and see its current contents.

A successful read will clear for the subscriber that performed the read the changed state for the object file; the next read from this subscriber will block until the object state changes again. If the object state changes before the subscriber's next read, then that read will of course not block.

A typical loop in a subscriber would live in its own thread, and do the following:

```
for(;;) {  /* No error checking in this example */
    read(fd, buf, sizeof(buf);
    process(buf);
}
```

Blocking and non-blocking reads

A subscriber *read()* call on an object file will by default block and not return until the object changes. Therefore, if you want to query the object without any chance of blocking, set O_NOBLOCK on the file descriptor. The table below describes behavior on a *read()* call, where the object has not changed since the last read.

O_NOBLOCK	Behavior
Set	Read returns immediately with no data (0 bytes).
Clear	Read waits until the object changes, then returns data.

PPS default open behavior

PPS defaults a normal open to O_NONBLOCK, which is atypical for most filesystems.

This behavior is implemented so that standard utilities will not hang waiting for a change when they make a *read()* call on a file. For example, with the default behavior, you could tar up the entire state of PPS using the standard tar utility. Without this default behavior, however, tar would never make it past the first file opened and read.

Clearing O_NONBLOCK

Clearing the O_NONBLOCK bit is very useful for clients that wish to dedicate a thread that waits for changes to an object or its attributes.

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

```
flags = fcntl(fd, F_GETFD);
flags &= ~O_NONBLOCK;
fcntl(fd, F_SETFD, flags);
```

You can then issue a read that waits until the object changes. For example:

```
// Assume fd has cleared O_NONBLOCK on the fd before this loop
for(;;) { // No error checking in this example
read(fd, buf, sizeof(buf); // Read waits till object changes
process(buf);
}
```

See “Pathname open query options” below for information about how to change the open behavior to wait (clear O_NONBLOCK) so that you do not have to issue the code shown above.

io_notify() functionality

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

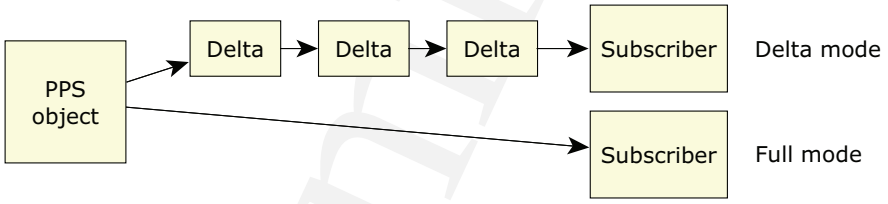
```
/* Process events while there are some */
while(ionotify(fd, _NOTIFY_ACTION_POLLARM, _NOTIFY_COND_INPUT,
    &event) & _NOTIFY_CONT_INPUT) {
if(read(fd, buf, sizeof(buf)) > 0) // Best to read with O_NONBLOCK
    process(buf);
    }
/* The event will be triggered in the future to get our attention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

Opening objects in “full” and “delta” modes

A subscriber can open an object in full mode, in delta mode, or in full and delta modes at the same time. The table below lists the flags set and the behavior for each mode:

Mode	Open flags	Description
Full	O_RDONLY	Read returns full object. Only the current state is returned, even if multiple changes have occurred since the last read. See “Full mode” below.
Delta	O_RDONLY | O_ASYNC	Read returns only changes (deltas) to the object. All changes are returned. Appending **`.delta`** to the filename of the object being opened will set O_ASYNC. This is a convenience to allow you to access delta mode using standard POSIX commands like **`cat`**. See “Delta mode” below.

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



PPS full and delta subscription modes.

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

Full mode

In full mode, the subscriber always receives a single, consistent version of the entire object as it exists at the moment when it is requested. If a publisher changes an object several times before a subscriber asks for it, the subscriber receive the state of the object at the time of asking *only*. If the object changes again, the subscriber is notified again of the change.

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.

Using delta mode

While full mode meets the needs of a large set of use cases, there are cases where delta mode is needed.

As an example, consider that the currently playing song is represented by the object **`PlayCurrent`**, as presented above under "Object rules". Any number of HMIs can subscribe to this object in order to display information about the currently playing song.

If the data changes, the PPS service notifies HMIs subscribed for full mode with a complete description of the object. However, if an HMI needs to know if an error has occurred during playback so that, for example, it can grey out the song title in its display of available titles, the case is not so straightforward. To keep track of unplayable tracks, we add the local database synchronizer as a subscriber to the track information, so that it can mark the track as unplayable in multimedia database.

We can add an attribute, such as **`error::bad`** attribute to **`PlayCurrent`**, and have the HMI and the database synchronizer look for this attribute. This strategy works only if the HMI and database synchronizer run *before* the error is cleared or overwritten, however. If the HMI or the database synchronizer run *after* the error is cleared or overwritten, the object will have changed twice (marked as bad, then cleared) without their being informed, and they will have missed one of the errors.

We can solve this problem by creating an object, **`PlayError`**, and having the HMI and the data synchronizer open it in delta mode. Having opened the object in delta mode, they will receive from the PPS service only attributes that have changed, but they will receive *all changes* to attributes in the object. No changes will be missed.

Delta mode queues

The PPS service creates a new queue of object changes when a subscriber opens an object in delta mode. In other words, 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. (PPS maintains reference counts to avoid making multiple copies of object attributes.)

If no subscriber has an object open in delta mode, the PPS service does not maintain any queues of changes to that object.

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

Changes to multiple attributes

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

```
write()                          write()
```

```
  time::1.23
  duration::4.2
```

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

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

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

Subscribing to multiple objects

PPS uses directories as a natural grouping mechanism to simplify and make more efficient the task of subscribing to multiple objects. Subscribers can open multiple objects, either by calling *open()* then *select()* on the the objects, or, more easily, by reading from a PPS directory, which merges all objects immediately below it.

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

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

If you open **rear/left** you will receive a notification when any file below it changes; and a read in full mode will return at most one object file 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 directory in delta mode, however, you will receive a queue of every attribute that changes in any file in the directory. In this case a single *read()* call may include multiple object files.

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

Options

You can set options to *read()* and *write()* calls by starting a line containing an object or attribute name with an open square bracket, followed by a list of single-letter options and terminated by a close square bracket. If nothing precedes an option letter, that option is set. It the option letter is preceded by a minus sign, that option is cleared. If an option letter is not specified, that option is not changed. For example:

- **`[n]url::www.qnx.com`** — Set the no-persist option on this attribute.
- **`[-n]url::www.qnx.com`** — Clear the no-persist option on this attribute.
- **`url::www.qnx.com`** — Do not change the current no-persist option on this attribute..

Pathname open query options

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

- **`/fs/pps/media/PlayList`** — open the **`PlayList`** file with no options
- **`/fs/pps/media/PlayList?w`** — open the **`PlayList`** file with the wait option
- **`/fs/pps/media/Playlist?w,d`** — open **`PlayList`** file with the wait and delta options
- **`/fs/pps/media?w`** — open the **`media`** directory with the wait option

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

The following table lists the currently supported pathname open query options:

Option	Letter	Description
Delta	d	Open the object in delta mode. This option is equivalent to using O_ASYNC with the *open()* call, but it allows this behavior for a utility which can not specify O_ASYNC.

continued...

Option	Letter	Description
Filter	f=*attr* f=*attr*+*attr* ...	Place a filter on notifications based upon changes to the listed attribute names only. In full mode, the file descriptor will only get notifications if one of the listed attributes changes. When this occurs the entire object is returned. In delta mode, only the listed attributes will be queued. Changes to non-listed attributes are filtered out.
No-persistence	n	Make the object non-persistent. When the system restarts, the object will not exist. The default setting is for all objects to be persistent and reloaded on restart. See "No-persistence option" and "Persistence" below.
Wait	w	Open the file with the O_NONBLOCK flag clear so that *read()* calls wait until the object changes or a delta appears. See "Subscribing" above.

No-persistence option

The no-persistence option is very useful on attributes that may not be valid across a system restart, but it can alos be set for objects. The table below describes the effects of the **`n`** (no-persist) option on PPS objects and attributes:

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

- Option defaults are always "clear".
- On a *read()* call you will only see a preceding option list "[*option letters*]" for options which have been set.

For more information about persistence, see "Persistence" below.

Persistence

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

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

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

- disk — QNX 6 filesystem
- Nand Flash — ETFS or ETFS2 filesystem
- Nor Flash — FFS3 filesystem
- other — customer generated filesystem

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

Saving PPS objects to persistent storage

On shutdown, PPS always saves to a persistent filesystem any directories with modified objects. You can also force a directory save at any time by calling *fsync()* on an object in the PPS directory you want to save. On a load from a saved file, PPS recreates its saved objects in a PPS directory.

You can set options to have PPS *not* save specific objects or attributes. For more information, see "Pathname open query options" and "No-persistence option" above.

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

For more information about PPS startup options, see "Running PPS" below.

Contents of saved files

The minimum unit PPS saves to a persistent filesystem is a PPS directory. PPS writes all the objects in a PPS directory as an atomic blob into a single file on the persistent filesystem. Thus, your system will have one persistent file for every PPS directory of objects. Saving a directory as a single entity ensures consistence across all PPS objects in a directory that contains related or linked data.

PPS encodes in the filenames of the files it saves to a persistent filesystem, the paths of the saved objects in the PPS directory hierarchy. For example, if PPS is mounted at **`/fs/pps`**, the following encodings are possible:

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

- To ensure data integrity, PPS performs CRCs on saved objects in the saved files.
- To reduce the size of saved files, you may compressed the data with a very fast LZO compressor.
- PPS may be used to create objects which are rarely (or never) published or subscribed to, but for which persistence is required.

Object loading and restoration

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

When PPS restores its objects, it does *not* remove the file on the permanent filesystem from which it restored the objects. Since this file is not updated when objects in memory change, over time the data in this file becomes stale.

Assuming a proper commit or shutdown, however, this backup file does get updated with the latest consistent copy of the PPS objects.

Restoring objects after improper shutdowns

If the system fails to shutdown properly, on a reload, PPS will attempt to load stale objects. The question of whether a stale object is better than no object is an application decision.

This decision does not need to be global, however, and can be made for each object. PPS provides the capability to detect stale data, and an object property that determines what to do with the object when its data is stale. The table below describes possible course of action when PPS encounters stale objects:

Object Extension	Action on Restore
Stale valid	The object is restored from the data.

continued...

Object Extension	Action on Restore
Stale invalid	The object is created empty with *no* attributes.
Stale revert to defaults	The object is loaded from a pre-defined default. The source of these defaults is to be determined.

Multiple language support

All strings should use UTF-8 to encode extended character sets.

Binary support

Since PPS terminates lines with a linefeed, you need to escape this character. If you wish to store arbitrary binary data then you need to encode the binary data in much the same fashion as you would an email message.

Running PPS

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

Syntax:

```
pps [options]
```

Options:

`-b`	Do *not* run in the background. Useful for debugging.
`-l`	Load all objects on startup. Default is to load objects on demand.
`-m` *mount*	Specify the mountpath for PPS. Default is **`/fs/pps/`**
`-p` *path*	Set the path for backing up the persistent storage.
`-s` *size*	Set the maximum object size, in kilobytes. Default is 4 kilobytes; maximum is 16 kilobytes.
`-v`	Enable verbose mode. Increase the number of “v”s to increase verbosity.

Synopsis:

```
#include <stdlib.h>

int psparse( char **readbufp,
             char **attrs,
             char **valuep,
             char **encoding );
```

Arguments:

readbufp	The address of a pointer to the string of options that you want to parse. The function updates this pointer as it parses the options; see the "Description" below.
attrs	A vector of possible attributes; see "*attrs* vector" below.
valuep	The address of a pointer that the function updates to point to the first character of a value that is associated with an attribute; see the "Description" below.
encoding	The address of a pointer that the function updates to point to the first character of the encoding type.

Library:

This function will be placed in **libc**.

Description:

The function *psparse()* parses a string read from the QNX Publish/Subscribe resource manager (PPS). The general format of this string is:

```
@objectname
attrname:encoding:value
attrname:encoding:value
@objectname
attrname:encoding:value
attrname:encoding:value
...
```

Though typically *psparse()* returns only a single object, it may return multiple objects.

The *psparse()* function takes:

- the address of a pointer to the read string
- a vector of possible attributes
- the address of a value string pointer

- pointer to a single character to hold the type of quote around the value

It returns the index of the attribute that matched the next attribute in the input string, or `-1` if there was no match. It modifies the buffer referenced by *readbufp*.

attrs vector

The *attrs* vector is organized as a series of pointers to null strings. The end of the *attrs* vector is identified by a NULL pointer.

When *psparse()* fails to match an attribute with an attribute in the *attrs* array, the calling program should decide if this failure is an error, or if the unrecognized attribute should be ignored. Unless you are sure that all the attributes in an object are valid, it is recommended that you have the calling program ignore unknown attributes.

Returns:

The *psparse()* function returns `-1` when the attribute it is scanning is not in the *attrs* vector. The variable referenced by *valuep* contains a pointer to the first character of the attribute that wasn't recognized, rather than a pointer to a value for that token. The variable referenced by *readbufp* points to the next option to be parsed, or, if there are no more options, to a null character.

Examples:

The following code fragment shows a typical use case for *psparse()*:

```
enum {
    OBJ_NAME,
    ATTR_AUTHOR,
    ATTR_TITLE,
    ATTR_TIME,
    LAST_ATTR
};
static char *attrs[] ={
    [OBJ_NAME]    = "@",
    [ATTR_AUTHOR] = "author",
    [ATTR_TITLE]  = "title",
    [ATTR_TIME]   = "time",
    [LAST_ATTR]   = 0
};
char    *encoding;
char    *value, *ppsdata = strdup(testdata);

while(*ppsdata) {

    switch(psparse(&ppsdata, attrs, &value, &encoding)) {
    case ATTR_BEG:
        printf("OBJ_NAME:  %s\n", value);
        break;

    case ATTR_AUTHOR:
```

```
            printf("ATTR_AUTHOR:%s %s\n", encoding, value);
            break;

        case ATTR_TITLE:
            printf("ATTR_TITLE:%s %s\n", encoding, value);
            break;

        case ATTR_TIME:
            printf("ATTR_TIME:%s %s\n", encoding, value);
            break;

        default:
            printf("UNKNOWN: %s\n", value);
            break;
        }
}
```

Classification:

QNX

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.

See also:

getsubopt()

Index

O

P

R

S

T

V