PPS - Pulse Per Second

Overview

LinuxPPS provides a programming interface (API) to define in the system several PPS sources.

PPS means "pulse per second" and a PPS source is just a device which provides a high precision signal each second so that an application can use it to adjust system clock time.

A PPS source can be connected to a serial port (usually to the Data Carrier Detect pin) or to a parallel port (ACK-pin) or to a special CPU's GPIOs (this is the common case in embedded systems) but in each case when a new pulse arrives the system must apply to it a timestamp and record it for userland.

Common use is the combination of the NTPD as userland program, with a GPS receiver as PPS source, to obtain a wallclock-time with sub-millisecond synchronisation to UTC.

RFC considerations

While implementing a PPS API as RFC 2783 defines and using an embedded CPU GPIO-Pin as physical link to the signal, I encountered a deeper problem:

> At startup it needs a file descriptor as argument for the function time_pps_create().

This implies that the source has a /dev/... entry. This assumption is OK for the serial and parallel port, where you can do something useful besides(!) the gathering of timestamps as it is the central task for a PPS API. But this assumption does not work for a single purpose GPIO line. In this case even basic file-related functionality (like read() and write()) makes no sense at all and should not be a precondition for the use of a PPS API.

The problem can be simply solved if you consider that a PPS source is not always connected with a GPS data source.

So your programs should check if the GPS data source (the serial port for instance) is a PPS source too, and if not they should provide the possibility to open another device as PPS source.

In LinuxPPS the PPS sources are simply char devices usually mapped into files /dev/pps0, /dev/pps1, etc.

PPS with USB to serial devices

It is possible to grab the PPS from an USB to serial device. However, you should take into account the latencies and jitter introduced by the USB stack. Users have reported clock instability around +-1ms when synchronized with PPS through USB. With USB 2.0, jitter may decrease down to the order of 125 microseconds.

This may be suitable for time server synchronization with NTP because of its undersampling and algorithms.

If your device doesn't report PPS, you can check that the feature is supported by its driver. Most of the time, you only need to add a call to usb_serial_handle_dcd_change after checking the DCD status (see ch341 and pl2303 examples).

Coding example

To register a PPS source into the kernel you should define a struct pps_source_info as follows:

```
static struct pps_source_info pps_ktimer_info = {
        .name           = "ktimer",
        .path           = "",
        .mode           = PPS_CAPTUREASSERT | PPS_OFFSETASSERT |
                          PPS_ECHOASSERT |
                          PPS_CANWAIT | PPS_TSFMT_TSPEC,
        .echo           = pps_ktimer_echo,
        .owner          = THIS_MODULE,
};
```

and then calling the function pps_register_source() in your initialization routine as follows:

```
source = pps_register_source(&pps_ktimer_info,
```

```
                        PPS_CAPTUREASSERT | PPS_OFFSETASSERT);
```

The pps_register_source() prototype is:

```
int pps_register_source(struct pps_source_info *info, int default_params)
```

where "info" is a pointer to a structure that describes a particular PPS source, "default_params" tells the system what the initial default parameters for the device should be (it is obvious that these parameters must be a subset of ones defined in the struct pps_source_info which describe the capabilities of the driver).

Once you have registered a new PPS source into the system you can signal an assert event (for example in the interrupt handler routine) just using:

```
pps_event(source, &ts, PPS_CAPTUREASSERT, ptr)
```

where "ts" is the event's timestamp.

The same function may also run the defined echo function (pps_ktimer_echo(), passing to it the "ptr" pointer) if the user asked for that... etc..

Please see the file drivers/pps/clients/pps-ktimer.c for example code.

SYSFS support

If the SYSFS filesystem is enabled in the kernel it provides a new class:

```
$ ls /sys/class/pps/
pps0/  pps1/  pps2/
```

Every directory is the ID of a PPS sources defined in the system and inside you find several files:

```
$ ls -F /sys/class/pps/pps0/
assert     dev        mode       path       subsystem@
clear      echo       name       power/     uevent
```

Inside each "assert" and "clear" file you can find the timestamp and a sequence number:

```
$ cat /sys/class/pps/pps0/assert
1170026870.983207967#8
```

Where before the "#" is the timestamp in seconds; after it is the sequence number. Other files are:

- echo: reports if the PPS source has an echo function or not;
- mode: reports available PPS functioning modes;
- name: reports the PPS source's name;
- path: reports the PPS source's device path, that is the device the PPS source is connected to (if it exists).

Testing the PPS support

In order to test the PPS support even without specific hardware you can use the pps-ktimer driver (see the client subsection in the PPS configuration menu) and the userland tools available in your distribution's pps-tools package, http://linuxpps.org , or https://github.com/redlab-i/pps-tools.

Once you have enabled the compilation of pps-ktimer just modprobe it (if not statically compiled):

```
# modprobe pps-ktimer
```

and the run ppstest as follow:

```
$ ./ppstest /dev/pps1
trying PPS source "/dev/pps1"
found PPS source "/dev/pps1"
ok, found 1 source(s), now start fetching data...
source 0 - assert 1186592699.388832443, sequence: 364 - clear  0.000000000, sequence: 0
source 0 - assert 1186592700.388931295, sequence: 365 - clear  0.000000000, sequence: 0
source 0 - assert 1186592701.389032765, sequence: 366 - clear  0.000000000, sequence: 0
```

Please note that to compile userland programs, you need the file timepps.h. This is available in the pps-tools repository mentioned above.

Generators

Sometimes one needs to be able not only to catch PPS signals but to produce them also. For example, running a distributed simulation, which requires computers' clock to be synchronized very tightly. One way to do this is to invent some complicated hardware solutions but it may be neither necessary nor affordable. The cheap way is to load a PPS generator on one of the computers (master) and PPS clients on others (slaves), and use very simple cables to deliver signals using parallel ports, for example.

Parallel port cable pinout:

```
pin     name    master      slave
1       STROBE    *------     *
2       D0        *     |     *
3       D1        *     |     *
4       D2        *     |     *
5       D3        *     |     *
6       D4        *     |     *
7       D5        *     |     *
8       D6        *     |     *
9       D7        *     |     *
10      ACK       *     ------*
11      BUSY      *           *
12      PE        *           *
13      SEL       *           *
14      AUTOFD    *           *
15      ERROR     *           *
16      INIT      *           *
17      SELIN     *           *
18-25   GND       *-----------*
```

Please note that parallel port interrupt occurs only on high->low transition, so it is used for PPS assert edge. PPS clear edge can be determined only using polling in the interrupt handler which actually can be done way more precisely because interrupt handling delays can be quite big and random. So current parport PPS generator implementation (pps_gen_parport module) is geared towards using the clear edge for time synchronization.

Clear edge polling is done with disabled interrupts so it's better to select delay between assert and clear edge as small as possible to reduce system latencies. But if it is too small slave won't be able to capture clear edge transition. The default of 30us should be good enough in most situations. The delay can be selected using 'delay' pps_gen_parport module parameter.