PPS signal setup on BeagleBone Black platform

Used Linux distribution:

The PPS signal generation was tested on the BeageBone Black device using the Debian 9.3 2018-01-28 4GB SD LXQT distribution, that can be downloaded from the official BeagleBone webpage. (http://beagleboard.org/latest-images)

The codes and scripts used in this documentation can be found in the https://github.com/t-tibor/msc_thesis github repository. A complete Debian based SD card image can be found here too, which has all the components installed except from the linuxptp init script.

Linuxptp

System requirements

The linuxptp is a free Linux implementation of the Precision Time Protocol defined by the IEEE 1588 standard. It supports Ordinary Clock and Boundary Clocks, hardware and software timestamping, Ipv4, IPv6 and raw Ethernet transport.

The following requirements have to be fulfilled to install the linuxptp:

1. Supported Ethernet MAC or PHY
    a. The NIC of the BeageBone Black device is supported.
2. Linux Kernel support for network packet timestamping (SO_TIMESTAMPING socket option) and clock control.
    a. These features are part of the main line kernel since version 3.0.
3. The linux kernel has to be compiled with the following configuration parameters:
   CONFIG_PPS=y (required)
   PTP_1588_CLOCK=y (required)
   CONFIG_NETWORK_PHY_TIMESTAMPING=y (optional - not used in BeageBone Black)
   The first parameter enables the PPS subsystem in the kernel, the second adds support for PTP clocks as character devices (e.g */dev/ptp0* - -this represent the HW clock in the Ethernet controller on the BeagelBone), and finally the third allows timestamping of the network packets by PHYs (this can be more accurate compared to MAC time stamps, however it introduces some other difficulties, like communication with the PHY or timestamp - packet association…).
   These parameters can be checked in the */boot/config-<kernel version>* file, and if any of the first two are not active, than the kernel has to be rebuild. (Not necessary with the official Debian BBBlack image.)
4. Ethtool Support: Ethtool is a utility for displaying and modifying the parameters of the Network Interface Controllers and their device drivers. It can be used for example to query the time stamping capabilities of the network interface, this feature is used by the linuxptp too.
    a. Many distributions contain this tool by default, if not (this is the case with the official BeageBone Debian distribution) it has to be built from the source code. The source files can be downloaded from the website https://www.kernel.org/pub/software/network/ethtool/.

After decompressing run *make* and *sudo make install* from the base directory, these commands will build and install the tool.
To test start it with the following parameters:

```
root@beaglebone:-# ethtool -T eth0
Time stamping parameters for eth0:
Capabilities:
	hardware-transmit     (SOF_TIMESTAMPING_TX_HARDWARE)
	software-transmit     (SOF_TIMESTAMPING_TX_SOFTWARE)
	hardware-receive      (SOF_TIMESTAMPING_RX_HARDWARE)
	software-receive      (SOF_TIMESTAMPING_RX_SOFTWARE)
	software-system-clock (SOF_TIMESTAMPING_SOFTWARE)
	hardware-raw-clock    (SOF_TIMESTAMPING_RAW_HARDWARE)
PTP Hardware Clock: 0
Hardware Transmit Timestamp Modes:
	off                   (HWTSTAMP_TX_OFF)
	on                    (HWTSTAMP_TX_ON)
Hardware Receive Filter Modes:
	none                  (HWTSTAMP_FILTER_NONE)
	ptpv2-event           (HWTSTAMP_FILTER_PTP_V2_EVENT)
```

The result contains the time stamping capabilities: SOF_TIMESTAMPIG_RAW_HARDWARE, SOF_TIMESTAMPIG_TX_HARDWARE, SOF_TIMESTAMPING_RX_HARDWARE. These flags signal that the eth0 interface is capable of timestamping in hardware the IEEE 1588 compliant PTP packages, which in necessary for high precision clock synchronisation.

Installation

The source files can be downloaded from the linuxptp official website:

http://linuxptp.sourceforge.net/

The code is maintained using the git version control system, to clone the repo run the following command:

```
git clone git://git.code.sf.net/p/linuxptp/code linuxptp
```

The code can be built and installed with the standard make commands:

```
root@beaglebone:-# make

root@beaglebone:-# make install
```

Usage

The linuxptp consist of 3 programs:

- **ptp4l:** This is the implementation of the PTP protocol and supports both *Ordinary (OC)* and *Boudary clock (BC)* roles. It is responsible for the grandmaster clock election (with the Best Master Clock algorithm) and for the actual clock synchronisation.

- **phy2sys:** It can be used to synchronise two different clocks in the system: generally one of these is a PTP hardware clock (PHC - synchronised by ptp4l) and the other is the system clock.
- **pmc:** This program is the implementation of the PTP management client according to the IEEE standard 1588.

To start the clock synchronisation on the local Ethernet network run the *ptp4l* program with the given parameters:

```
root@beaglebone:-# ptp4l -i eth0 -m
```

This starts the PTP stack on the eth0 interface over UDP/IPv4 protocol with the default hardware time stamping, and sends the messages to the standard output (-m).

A more convenient way to configure the *ptp4l* tool is to save all the parameters in a configuration file and then let the program configure itself from that. The configuration file can be given with the -f option.

```
root@beaglebone:-# ptp4l -f <ptp_config_file_location>
```

An example config file (that is equivalent to the previous command)

```
[global]
# The priority2 attribute of the local clock.
# It is used in the best master selection algorithm, lower values take precedence.
# Must be in the range 0 to 255. The default is 128.
priority2 126

Print messages to the standard output if enabled.
# The default is 0 (disabled).
verbose 1

# clock_servo

# -i eth0
[eth0]
```

The global section provides general parameters, for example associated with the BMC (Best Master Clock) algorithm. This algorithm is used to decide the grandmaster clock on the local network, and uses the following parameters to compare the capabilities of the clocks (from higher priority to lower):

1. Priority One
2. Clock Class
3. Clock Accuracy
4. Clock Variance
5. Priority 2
6. Source Port ID

The current configuration file contains a *Priority 2* field setting, which was used to select the clock that should become the grandmaster in our 2 device set-up, where all the other parameters were equivalent.

More information about the configuration file syntax and the possible parameters can be found in the ptp4l man(8) page.

The *ptp4l* process can be started automatically after system start using an init script, for which an example can be seen bellow:

```
#!/bin/sh
### BEGIN INIT INFO
# Provides: ptp4l
# Required-Start: $local_fs $network
# Required-Stop: $local_fs
# Default-Start: 2 3 4 5
# Default-Stop: 0 1 6
# Short-Description: ptp4l
# Description: linuxptp ptp4l service
### END INIT INFO

# processname: ptp4l
#
DAEMON="/usr/local/sbin/ptp4l"
ARGS="-f /root/dipt1/ptp4l.conf"
OPTIONS="--chuid root --oknodo --quiet"
#background
#PIDFILE=/var/run/ptp4l.pid
###

d_start() {
        ### Start daemons, perhaps with the daemon function
        tart-stop-daemon --start $OPTIONS --exec $DAEMON -- $ARGS
}
 ###
d_stop() {
        #Stop daemons, perhaps with the killproc function
        start-stop-daemon --stop $OPTIONS --exec $DAEMON

}

case "$1" in
        start)
                echo -n "Starting ptp4l services: "
                d_start
        ;;
        stop)
                echo -n "Shutting down ptp4l services: "
                d_stop
        ;;
        status)
                #<report the status of the daemons in free-form format,
                #perhaps with the status function>
        ;;
        restart | reload)
                # Restart the daemons, normally with '$0 stop; $0 start'
                d_stop && sleep 2 && d_start
        ;;
        *)
                echo "Usage: ptp4l {start|stop|status|reload|restart}"
                exit 1
        ;;
esac
exit 0
```

This script has to be placed in the */etc/init.d* folder (with the correct access rights), and then a symbolic link should be created with the

```
root@beaglebone: -# update-rc.d <script_name> defaults
```

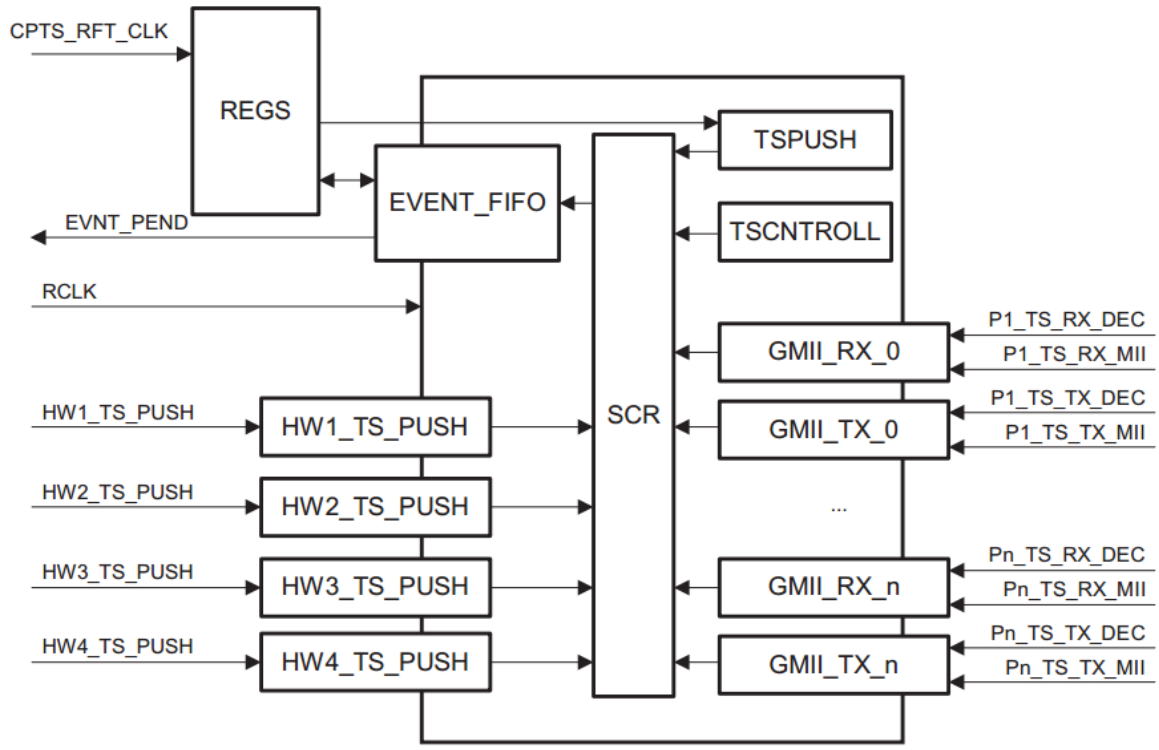
command.

PPS signal generation

The PPS- as its name suggests - is a high precision periodic signal that is asserted in every second with an arbitrary pulse length and is synchronised to the global time. The biggest challenge in implementing such a PPS signal on the BeagleBone Black development board is the fact, that the Ethernet controller does not support it by hardware. The suggested method is to use a different peripheral clock for the signal generation, which is synchronised to the internal counter of the Ethernet counter. The next sections contain a short introduction about the CPTS (Common Platform Time Synch), the peripheral timer and the signal generation algorithm.

CPTS

Let's examine the Ethernet controller time stamping unit



1. Figure Structural diagram of the CPTS module

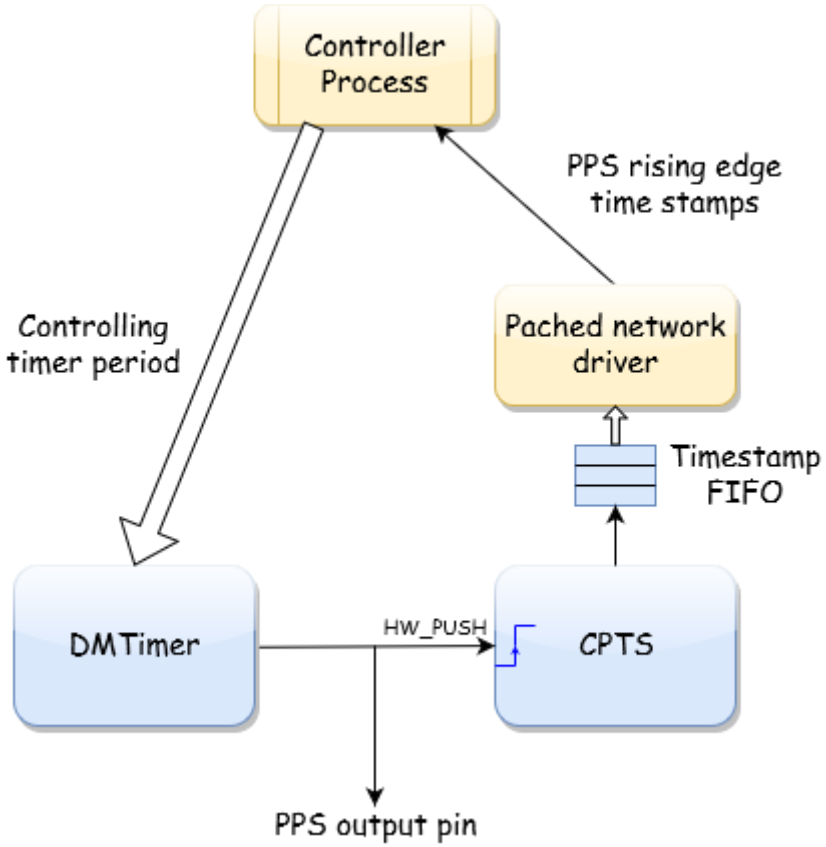
The role of the CPTS in the Ethernet controller is to provide high precision hardware time stamping ability. To put it simply the most important component of the CPTS is a free running timer (SCR) and an event FIFO for pushing the value of the counter into in response to some external events. These events can be:

- Incoming/outgoing IEEE 1588 Ethernet packets
- Software time stamp request
- Hardware time stamp request (4 independent lines)

The first 2 event types are used mainly for high precision time synchronisation.

For us the hardware time stamp events are the most important, since the signal lines (HWx_TS_PUSH) are directly connected to the *portimerpwm* outputs of 4 of the DMTimers (Dual Mode Timer - this is the name of the peripheral timers). (This *portimerpwm* outputs can be used to generate a PWM signal.)

So one way to implement a PPS signal is that the PPS signal itself is generated as a regular PWM signal by a DMTimer on its *portimerpwm output*, but at every PPS rising edge we let the CPTS record the actual timestamp. After that a software controller program can monitor these timestamps and regulate the period and the value of the DMTimer in such a way, that the PPS signal gets and stays synchronised to the CTPS clock (which is in turn synchronised to the global time by the linuxptp).



2. Figure PPS generation method

Using this method we have to keep in mind, that the base frequency of the DMTimer introduces a new higher limit for the possible synchronisation precision: the DMTimers have a 24MHz base clock frequency / ~41.67 ns tick period and since the period of the PPS signal is an integer multiple of the base clock period, the PPS period cannot be more precise than ~+-20ns relative to the real 1 second, even with perfect and ideal IEEE 1588 synchronisation.

Kernel patching

The above described method needs access to the hardware time stamps of the PPS rising edges, however the CPTS driver of the mainline Linux kernel does not support the HW_TS_PUSH capability of the hardware. This problem can be solved by applying some kernel patches that extend the CPTS driver. The required patches:

- https://patchwork.kernel.org/patch/9784967/

- https://patchwork.kernel.org/patch/9784961/
- https://patchwork.kernel.org/patch/9784955/
- https://patchwork.kernel.org/patch/9784963/

Unfortunately these patches are written for the 4.12 kernel, so we need to change to this exact version.

Steps for the installation:

1. Upgrade the kernel of the BeageBone to the 4.12.14 version:

```
apt-get update
```

```
apt-get install linux-image-4.12.14.bone4
```

2. Clone the kernel building tool.

```
git clone https://github.com/RobertCNelson/bb-kernel
```

3. Check out to the 4.12 version and create a custom branch for the patched kernel.

```
git checkout origin/am33x-v4.12 -b tmp
```

4. Run the kernel building script. It downloads the cross-compiler for the ARM platform, clones the mainline Linux kernel from Linus' repository, applies some patches to it (not ours) and finally builds it.

```
build_kernel.sh
```

5. Copy the downloaded patches to the kernel directory and run the patch program. It will add or remove code lines from the kernel source.

```
patch -p1 < "patch_file_name"
```

6. Rebuild the patched kernel.

```
./tools/rebuild.sh
```

7. Copy the build outputs (from the *deploy* directory) to the BeagleBone, and run the *my_kernel_update.sh 4.12.14-bone4* bash script (from the location of the patches). This copies the new kernel image, the belonging device tree overlays, modules and firmware to the appropriate locations. (For more details check the script, which can be found in the git repo)

```
my_kernel_update.sh 4.12.14-bone4
```

8. Finally modify the used kernel version number in the */boot/uEnv.txt* file ('*uname_r=*' line), in order to inform the uBoot bootloader, that from now on the new kernel should be loaded on boot.

PPS pin configuration

To get the PPS out of the controller, we need to configure the GPIO mux so that the used DMTimer's *portimerpwm* signal appears on the associated pin. (**In our demo application the P8.09 pin is the PPS pin, and the used DMTimer is the fifth.**)

Basically there are two ways to do it:

1. Use the *config-pin* command.

```
config-pin P8.09 timer
```

2. Create a device tree overlayt to control the pin association. With this method the pin gets configured during boot automatically, so this is preferred.
    a. First build the overlay:

```
dtc -O dtb -o <filename>.dtbo -b 0 -@ <filename>.dts
```

    b. Copy the output file (.dtbo) to the */lib/firmware* location.
    c. Inform the uBoot about the new overlay, so that it adds that to the main device tree during boot. This can be done by adding the following line to */boot/uEnv.txt*:

```
dtb_overlay=/lib/firmware/<name of the .dtbo overlay file>
```

Controlling the PPS signal

A C program performs the synchronisation between the CPTS clock and the generated PPS signal by continuously reading the timestamps and correcting the timer value or frequency. (To be able to do that, it needs to be run with administrator rights.)

The input for the program are the timestamps generated by the CPTS module in response to the HW_TS_PUSH rising edge (that is actually the start of the PPS pulse). This data can be read from the */dev/ptp0* device file after some *ioctl* configuration (the file operations are implemented in the patched CPTS driver), and contains a second and a nanosecond field. Out purpose is to minimalize the latter, because it means, that the rising edges of the PPS signal happen exactly at the start of new seconds.

The controller program has 2 states. At the start only a rough offset correction is done to bring the error under a specific limit (in the current implementation 1us). After that a 3 stage PI clock servo takes control, which uses different coefficient sets (3 sets - one for each stage) according to the actual error. For more details check the source code.

The controller software can be built simply by applying *make BBonePPS* in the source directory. To run, don't forget to grant root permission, and since it writes no output to the terminal it can be run in the background.

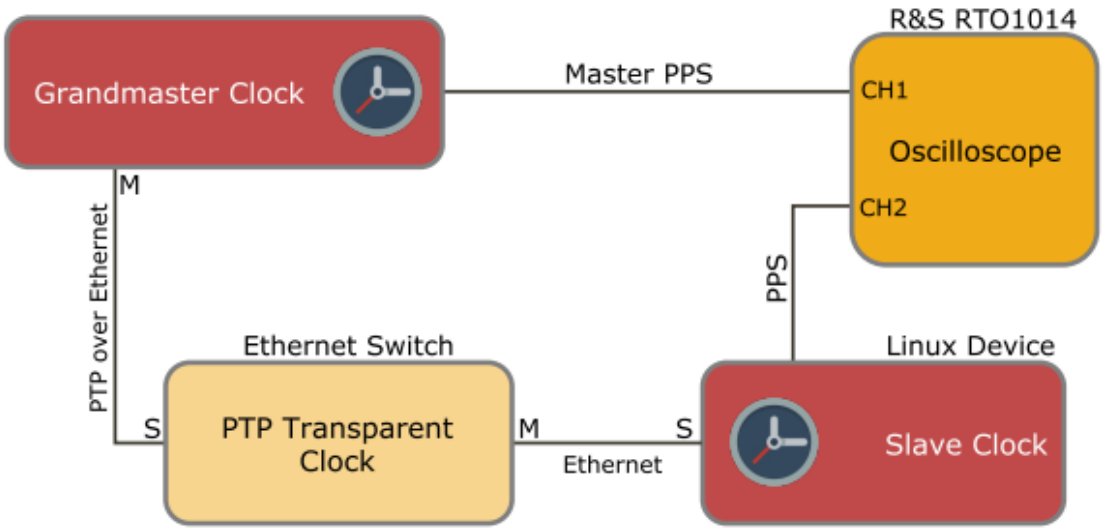
```
sudo ./BBonePPS&
```

Accuracy of the generated PPS signal

The following components can influence the final precision of the PPS signal:

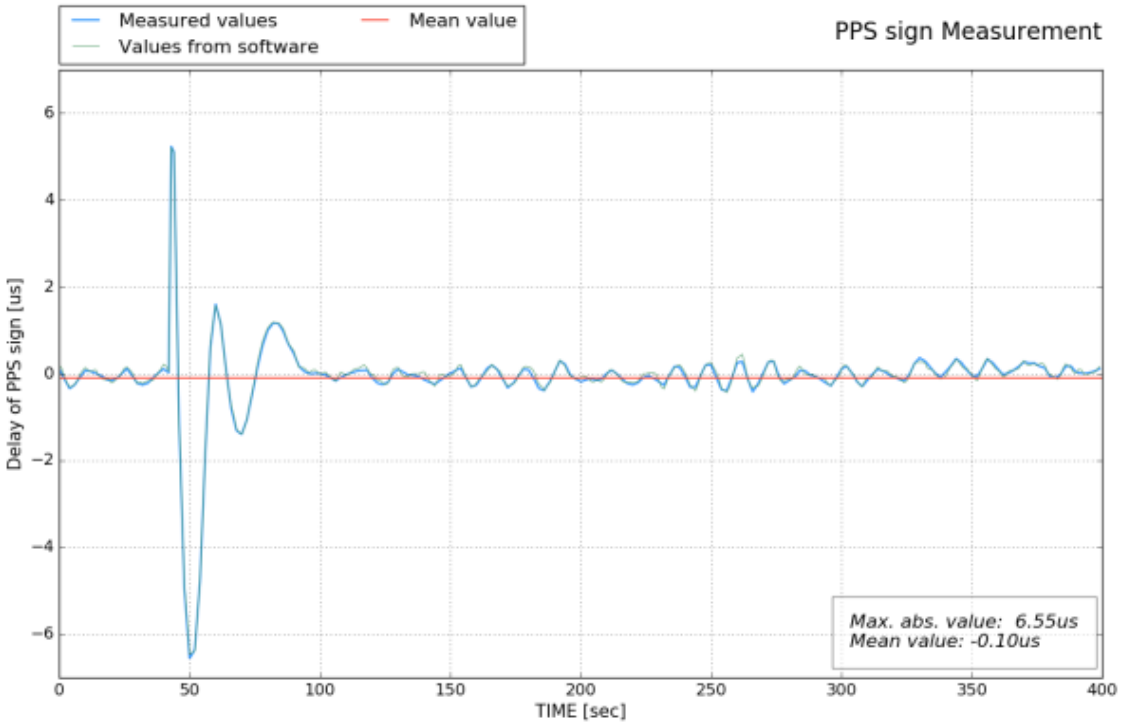
1. Resolution of the CPTS timer (250MHz / 4ns)
2. Resolution of the DMTimer (24Mhz / 42ns)
3. Precision of the clock synchronisation.

From the first 2 items we can see, that the theoretical lower limit for the PPS error is ~ 21 ns, so we cannot go below that. To determine the accuracy in practice, some measurements were performed using the arrangement illustrated on the 3. Figure.



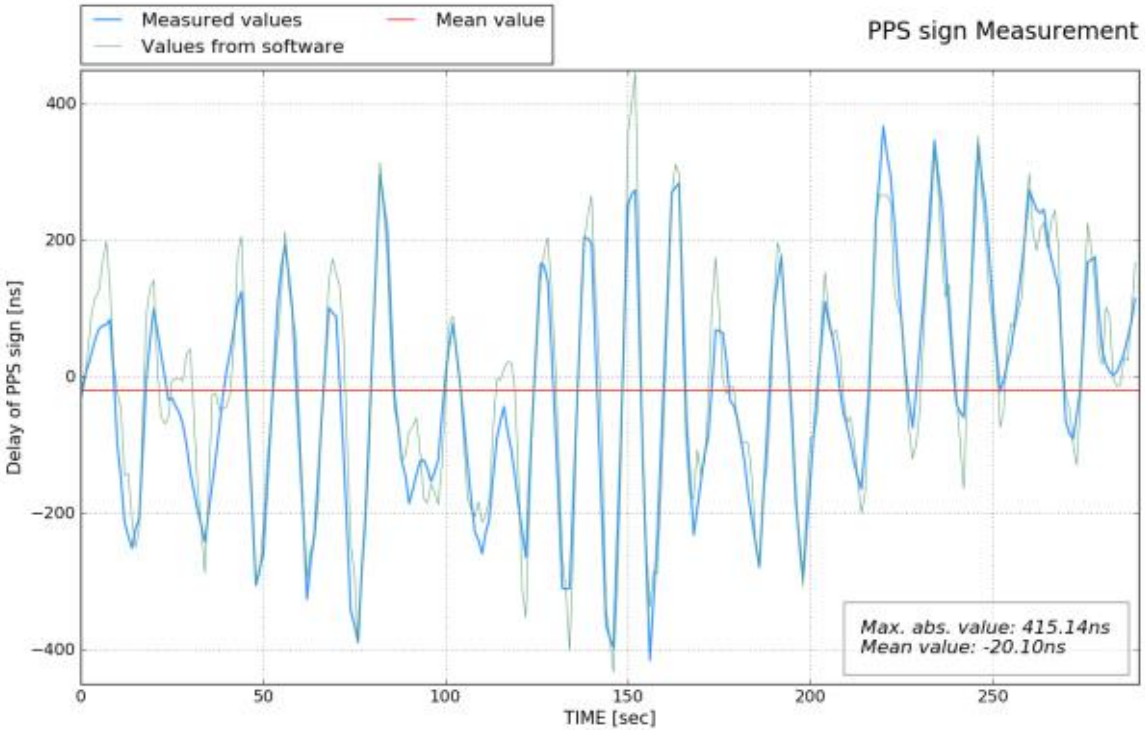
3. Figure Measurement set-up

The results:



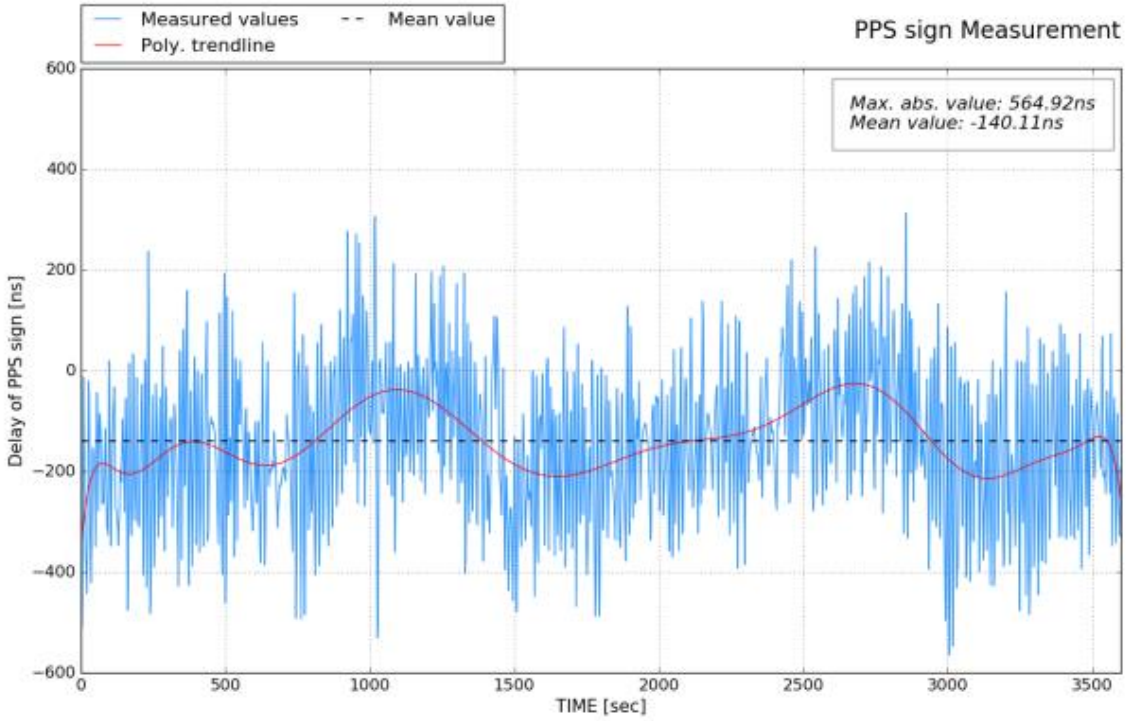
4. Figure Settling characteristic of the control loop after perturbation

From the error values measured by the oscilloscope and by the software (5. Figure) we can conclude that the latter values often differ in a magnitude of ~50-100ns from the former, however this fact does not influence the overall dynamics in a bad direction.



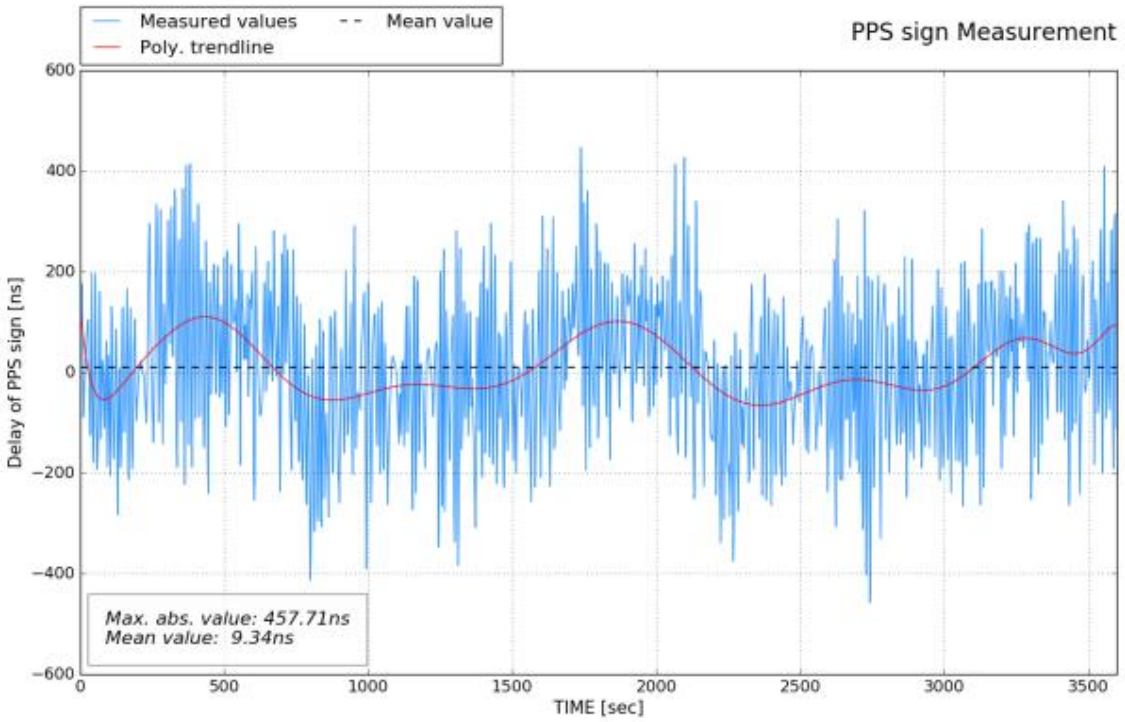
5. Figure Measured error values

A longer measurement (6. Figure) reveals, that the mean value of the error is different from zero, which can be a result of an imperfect clock synchronisation.



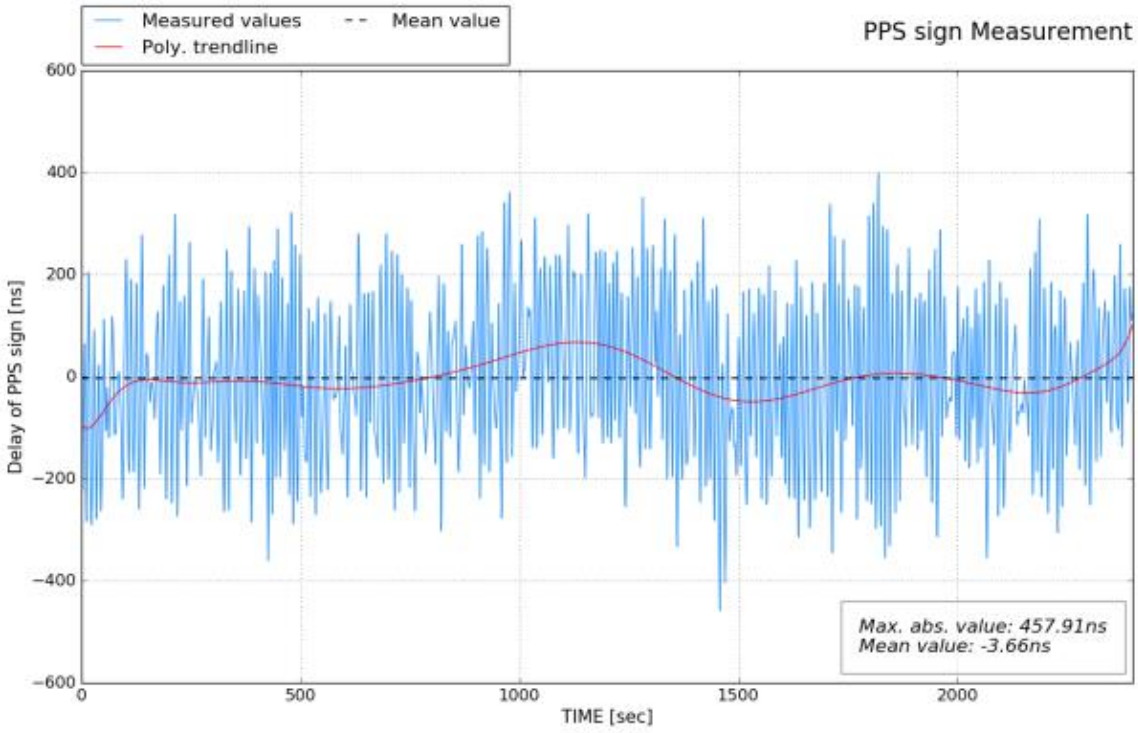
6. Figure PPS offset from the Master PPS signal (without correction)

This offset can be however easily corrected by the PPS controller software by simply adding an offset value to the measured error values. The result can be seen on the 7. Figure.



7. Figure PPS offset from the Master PPS signal (with correction)

The fluctuation can be further decreased by refining the 3rd controller stage, which operates when the highest precision is achieved. (8. Figure)



8. Figure PPS signal error after refinement of the 3 stage

From these data we can conclude, that circa +-400ns accuracy can be expected from this PPS implementation. This can be eventually decreased by improving the clock synchronisation algorithm (asymmetries etc.) and the clock controller program.