PPSi Manual

July 2014 (ppsi-v2014.07)

Documentation about “PTP Ported to Silicon”

Alessandro Rubini and Aurelio Colosimo for CERN (be-co-ht)

Table of Contents

Introduction

PPSi (PTP Ported to Silicon) is an application which, in its basic operation, implements IEEE 1588 specification in a way that is portable to several architectures, including OS-less ones.

This manual is mainly aimed at developers: people who are working with PTP and/or White Rabbit and need to use the protocol in their own hardware or software environments. Users who simply run PPSi on the host may want to read Chapter 10 [Command Line], page 13 and little else.

1 Project Overview

This project originates in the context of *White Rabbit*, which is shortened to *WR* in this manual.

WR is a multi-lab research project that aims at synchronizing thousands of I/O devices distributed in a network several kilometers wide; its software protocol is an extension of PTP. A WR network is made up of two devices: the *WR Switch* and the *WR Node*. White Rabbit is developed at ohwr.org: `http://www.ohwr.org/projects/white-rabbit`.

The WR switch is an 18-ports Gigabit Ethernet switch that runs WR-PTP as a Linux process, synchronizing each Ethernet link as either a PTP master or a PTP slave (it is a *boundary clock*). The WR node is an I/O device that includes a soft core that runs WR-PTP in *freestanding* mode (i.e., without an operating system).

PPSi, as a software package, is designed to be the PTP implementation used in WR, running both standard PTP and WR-PTP, in all possible use cases.

The algorithm and computation routines regarding the basic IEEE 1588 are derived from the *PTPd* project, v.2.1.0 (see AUTHORS for details about copyright); but as of March 2013 very little remains of the original code base. In addition to the basic feature set we inherited from *PTPd*, we support raw Ethernet frames (`ETH_P_1588`), according to Annex F of the specification and support for protocol extensions.

We thank Danilo Sabato for fixing the *bare* architectures (see Section 2.2 [Architectures], page 2).

The home page of the PPSi project and the source repository are:

```
http://www.ohwr.org/projects/ppsi
git://ohwr.org/white-rabbit/ppsi.git
```

2 Status, Features, Bugs

This document tries to summarize the status of the project, but we are working a lot on the package, so information here may be slightly out of date with respect to code.

The software package is designed to be modular: each of architectures, protocols and timing engine can be replaced by providing a proper subdirectory.

2.1 Build-time Configuration

After release 2013.11 we added to PPSi the Kconfig configuration engine. Configuration is now performed by running "`make menuconfig`" or equivalent command.

A number of ready-to use configuration files are distributed in the `configs/` directory: to use any of them name it as target of *make* (for example: "`make wrs_defconfig`").

The default configuration is *unix_defconfig*, that builds a Unix daemon running the standard PTP version 2 protocol.

2.2 Architectures

When building PPSi, the user can specify which architecture to build for, by selecting it in `make menuconfig` or equivalent configuration command. When cross-compiling, you need to tell your cross-compiler prefix in the configuration step, or override it at build time by passing `CROSS_COMPILE=` on the commabnd line.

The package currently supports the following architectures:

unix

This is the default architecture and is the normal *hosted* environment. The architecture is usually tested over the Linux kernel using the standard GNU libraries. Most of the code relies on standard POSIX conventions, so this architecture most likely works on BSD and other variants as well. We may change naming accordingly, after identifying the weak (i.e. unportable) points. This architecture supports the standard protocol on both UDP and raw Ethernet, also both at the same time thanks to the per-link split of I/O methods. The architecture relies on *time-unix*, but it supports building with different time engines.

wrs

The White Rabbit switch build of PPSi is designed to be a separate architecture, even if the switch itself is a Linux system running on an ARM processor. The switch is a complex system, with several daemons cooperating through an IPC mechanism: the standard main loop for a standalone Unix daemon doesn't apply to the WR switch, and modelling it as a separate architecture is the simplest and cleanest approach, in our opinion (even if it leads to some code duplication). The architectures relies on *time-wrs*.

wrpc

The *White Rabbit PTP Core* architecture is a port of PPSi to run on our I/O devices. The *wrpc* is a "*core*", i.e. a logic block, that runs in WR-capable I/O cards. Most such cards are developed as open hardware on *ohwr.org*; examples are the SPEC (a PCIe carrier for FMC devices) and the SVEC (a VME carrier, with two FMC slots driver by the same WR core). The *wrpc* includes a soft-core CPU that runs the WR-PTP daemon. PPSi is designed to be able to be linked from *wrpc-sw* as its own *wr-ptp* engine. PPSi in this environment currently supports only raw Ethernet, not UDP. The architecture uses *time-wrpc*.

sim

This is a simulator. It uses special time and network operations to simulate a master and a slave exchanging ptp frames. Arch-specific configuration options are implemented in *arch-sim/sim-conf.c*. Use of the simulator is briefly descibed in Section 3.2 [Configuring the Simulator], page 6.

bare-i386

This architecture uses system calls towards the Linux kernel but does not rely on standard libraries. The port only supports raw Ethernet frames and is thought to be a validation for our *freestanding* ports. The process built as *bare-i386* runs on conventional x86 Linux hosts and demonstrates that PPSi works properly with no added dependencies on external libraries; freedom from dependencies is mandatory to retain the ability to build and run in *freestanding* environments like *wrpc* or microcontrollers.

bare-x86-64

Like *bare-i386*, this architectures makes direct system calls without relying on external libraries. The host in this case is the a 64-bit PC instead of a 32-bit one. Both bare ports use *time-bare* for their timing operations.

We plan, over time, to support microcontrollers (a beta version for ARM7 is working, and we are considering Cortex-M) and *bare-arm* (to be tested on the WR switch).

2.3 Protocols

At build time, the user can select the standard protocol (selected by default by most architectures) or an extension. The code base only includes one extension, White Rabbit, which is the default choice when building for one of the WR architectures. Nothing in PPSi prevents our fellow developers to support their own *ptp* extension using PPSi.

Each extension lives in a subdirectory called `proto-ext-`*name*. Source files in that directory may override the implementation for the standard protocol (which lives in `proto-standard`) to provide their own functionalities. To simplify writing extensions, the *proto-standard* functions already provide *hooks* so the extension itself can provide callbacks while still using the basic PTP engine. The set of callbacks is currently based on the needs of WR, but we are willing to accept patches to provide more hooks as needed.

If you plan to write your own protocol extension within PPSi, please refer to how *proto-ext-whiterabbit* is implemented.

2.4 Time Functions and Network Operations

PPSi supports custom implementations of time functions, so you can easily port the daemon to your own timing primitives.

The subdirectories named *time-sth* are used to implement timing functions; timing includes the methods that are part of two data structures: *pp_time_operations* and *pp_network_operations*. The former structure deals with getting and setting system time, while the latter deals with frame tx and rx. Network operations are concerned with timestamping the actual I/O, and that's why they are considered part of the "timing" of PPSi.

Each architecture defines its own `TIME=` choice in the specific `Makefile`. The user can override the default by either setting the variable on the command line of PPSi, or by setting an environment variable if the architecture supports it. For example, the *bare* architectures force `TIME=bare`, while the *unix* architecture suggests Unix timing code (`TIME?=unix`). The choice for `TIME=` is not currently selected using Kconfig, because nobody is expected to diverge from the default choice for the architecture being built.

If you want to support a different timing engine within the Unix build system, you can use "`make TIME=xyz`" to request building the *time-xyz* subdirectory. Please note that the Unix time structures are built anyways for `CONFIG_ARCH=unix`, so you can piggy-back on those functions, either within your own methods or by replacing the `ppi->t_ops` and `ppi->n_ops` for the communications links that do not include your own hardware support.

2.5 Features

This is a summary of current and planned features for the PPSi package. As said, we are working on it these months, so the following list me be not up to date with software developments by the time you read it.

The following list if features doesn't consider known bugs, which are listed in the following sections. Please consider such bugs transient mishaps, as we are working on them right now; anyways, it would be unfair to hide them.

Support standard PTP and extensions.
: Each protocol extension should fall back to standard PTP when it detects that its peers are not able to speak the extended protocol. We don't plan to support more than one extensions in a single build: it would be just an academic exercise until somebody contributes a new protocol extension to PPSi.

Support both hosted and freestanding environments.
: This is already supported, though we still lack a microcontroller implementation, and our only freestanding environment is *wrpc*, running on an LM32 processor.

Support multi-link operation.
: The daemon manages several links at the same time, being master slave or auto-detect independently on each link. The *best master clock* algorithm is run globally, but communication and timeouts are managed per-link.

Support both UDP and raw Ethernet.
: We already do that. We plan use the multi-link operation to support both UDP and raw Ethernet on the same network interface. Also, we expect to support UDP in the WR switch, to allow progressive and seamless migration to WR if you already support PTP in your network with a UDP-only implementation.

Support fall-back master links.
: This is needed for WR, and is being worked on. We need to track more than one master in order to switch over from one to the other within a few milliseconds and with no time glitches.

Support hardware timestamping where available.
: This is not yet implemented for the generic protocol, but we plan to add at least *time-linux-tstamp* soon.

Support BSD specifics.
: Unfortunately the *hosted* part of the packages is slightly too much Linux-dependent. We plan to support the BSD variants as well, hoping to get interest and patches from some BSD developer.

Allow architectures to state their capabilities.
: Currently the command-line arguments are a an all-or-nothing thing. But, for example, *bare* architectures don't support UDP and other architectures may not support raw Ethernet. We plan to allow architectures to state their capabilities to report proper errors when the user tries to set up unimplemented configurations.

Support run-time enable/disable of diagnostics.
: We want to allow run-time modification of diagnostics flags with a per-link granularity. Currently we have configuration-based per-link diagnostic flags and global flags that can be changed at run time (for example, arch-wrpc offers that through a shell command). We think this feature is useful when you run more than a pair of interfaces and have problems on some of the links but not all of them.

2.6 Bugs

As of 2013-05 the project suffers from these known bugs:

- All frames must be sent according to a pseudo-random distribution; this is mostly in place but mut be audited project-wide.
- We removed *peer-delay* support. We plan to add it back, and actually move White Rabbit to use *peer-delay* PTP instead of *end-to-end* PTP.
- The servo for standard-PTP must be audited. We are doing it now using *arch-sim* support,
- UDP over IPV6 is not yet supported.

3 Configuration

PPSi support configuration files and individual configuration items passed on the command line. Such support is currently not available for freestanding architectures (the *bare* ones and *wrpc-sw*).

When PPSi starts it parses its own configuration. The command line can include a number of `-f <file>` and `-C <item>` options; they are processed in order, so later ones override earlier ones. A configuration "item" can include several directives, using the semicolon as a separator.

If no configuration file is specified, the program reads the default one, which is architecture-specific (thus, the default configuration file is processed after all the command line configuration items. The default file name is `/etc/ppsi.conf`, but `arch-wrs` tries to load `/wr/etc/ppsi.conf` first. The source tree of PPSi includes two example configuration files in its own `etc/` subdirectory.

Configuration is made of of global options and port-specific options. To configure a port, use `port <name>` followed by its options. The parser allocates a new PTP state machine for each new port, but allows changing configuration of an existing port. For instance, to enable frame diagnostics for a specific port, you can use:

```
./ppsi -f /etc/ppsi.conf -C "port eth1; diagnostics 02"
```

Each configuration item is made up of a keyword and an optional argument. The argument can be either a number or a string. The parser looks up keywords in three tables: a global table, an architecture-specific table and an extension-specific table. Currently, only *arch-sim* has specific configuration items.

The list of global configuration items is on show in `lib/conf.c`, as the `pp_global_arglines` array. Future versions of this manual may document the keywords, if the time allows it.

Note: most current command-line options are going to be turned into configuration options. This applies to the priorities, intervals and thresholds, as well as the *slave-only* flag.

3.1 Configuring Faults

Configuration, for `arch-unix` and `arch-wrs`, allows to provide some fault injection.

In particular, the program allows dropping frames, in both the TX and RX paths. The configuration values state how many frames are dropped every 1000. Dropping is randomized, but the user can set the seed to ensure a repeatable dropping sequence.

Dropping transmitted frames is performed by reporting success (and the timestamp), while no frame is actually sent. A diagnostic message is generated at `frames` level 1, but other than than that nothing happens. Actually, `arch-wrs` needs to actually send a frame in order to get a timestamp back; in this case the program modifies the frame, to use a wrong Ethernet type or a wrong UDP port.

Dropping received frames is performed by actually receiving (and timestamping), but returning a special error code to the caller. Again, PPSi creates a diagnostic message at `frames` level 1.

The following configuration lines are supported:

`rx-drop <value>`
`tx-drop <value>`
: Drop *value* frames every 1000 frames received or sent. The randomized sequence is global.

The randomization see can be passed by pre-setting the environment variable `PPSI_DROP_SEED` to a decimal numeric value. For example:

```
export PPSI_DROP_SEED=33
```

before starting the daemon.

3.2 Configuring the Simulator

To run the PPSi simulator you need to rely on diagnostics and specific configuration items. The configuration items are defined in *arch-sim/sim-conf.c* and are not individually documented here at this point.

After building with "`make sim_defconfig`", you can look at how PPSi behaves in different situation. For example, to see how the servo works with default parameters you can activate servo messages at level 2, and only look at the offset from master:

```
./ppsi -d 0002 | grep 'Offset from master'
```

The diagnostic values are specified in the range 0 to 2 and represent, in this order: state machine, time, frames, servo, bmc, extensions. See Section 5.1 [Diagnostic Macros], page 6 for details.

The simulator runs by default for one hour of simulated time (in a fraction of a second of running time), and the initial offset from master to slave is 0.9 seconds.

To pass configuration options, you can use the `-C` command line option. so, for example, to start with 0.1 seconds of offset and 1000 ns of transmission jitter, you can run like this:

```
./ppsi -d 0002 -C "sim_init_master_time .1; sim_jit_ns 1000"
```

4 PTP Clock Class

The clock class value (`clockClass`), a field of the "clock quality" structure, can be specified in the configuration file for the architectures that support such a file.

For the White Rabbit node (`arch-wrpc`) the class is defined according to build-time constants, set forth in `constants.h`.

When the node is configured as grandmaster (WRPC shell command `mode gm`), the class is set to 6 (a clock synchronized to a primary reference time source). However, if the internal PLL can not lock in a 60 seconds timeout, the class is set to 52: a clock that will never be a slave, but is not currently connected to a primary time source.

When the node is configured as master, it uses class 187, i.e. a clock that may be a slave of another clock.

5 Diagnostics

During development of PPSi, diagnostic support used several techniques, but finally we converged on the one described here, that is here to stay. The idea is that we need to add verbosity per-feature and per-port. This fine-grained control is expected to be important while developing features or while diagnosing problems on new architectures.

The `diagnostics` configuration keyword can be used both as a global item and as a port-specific configuration value.

5.1 Diagnostic Macros

The header file `<ppsi/diag-macros.h>` introduces the concept of diagnostics flags, which are hosted in both a global variable and *pp_instance* (i.e. one flags-set for each communication link). The macros use the logical OR of both flags, so developers can activate diagnostics on either a single link or globally.

The diagnostic flags are split into topics (called *things*). For each diagnostic thing the header defines a few bits; so we can have diagnostic levels for each of them, but we suggest only using level 1 and 2 – the rationale is in the header itself.

The *things* currently defined are:

- Finite State Machine: PPSi reports FSM state transitions.
- Time: at level 1 PPSi reports *time_set* operations and timeouts; at level 2 it also reports *time_get* operations.
- Frames: at level 1 PPSi reports any send and receive event; at level 2 it also shows the frame itself (using *ptpdump* code).
- Servo: report servo operation. At level 2 it also shows the individual timestamps and internal averaging.
- BMC: at level 1 PPSi reports BMC choices, at level 2 it reports addition of new masters as well.
- Extensions: extension-specific information.
- Configuration: at level 1 PPSi reports errors, at level 2 all configuration items being parsed (from either files or command line).

The user is expected to pass diagnostic flags as a string, specifying diagnostic levels for each of the things, where trailing zeroes are optional. So for example "`01`" specifies a diagnostic level 1 for time, and "`102`" specifies FSM at level 1 and frames at level 2. The header itself is more detailed about the conventions.

To parse the diagnostic string, PPSi offers *pp_diag_parse*. The function is used by both the code that reads the command line and code that parses configuration.

Within PPSi, developers should insert diagnostic messages by means of the *pp_diag* function:

```
        pp_diag(ppi, thing, level, ...)
```

The function acts like *printf*, with the leading arguments `ppi`, `thing` (which is one of the names fsm, time, frames, servo, bmc, ext) and `level`, which should be 1 or 2. For example, the code setting system time includes this diagnostic line:

```
         pp_diag(ppi, time, 1, "%s: %9li.%09li\n", __func__,
                 tp.tv_sec, tp.tv_nsec);
```

Finally, if you need to shrink the size of your binary file, you can build PPSi with no diagnostic code at all (i.e., the compiler won't even generate the function calls), you can define `PPSI_NO_DIAG` in `CFLAGS` while building. This can be achieved by setting `USER_CFLAGS`:

```
        make USER_CFLAGS="-DPPSI_NO_DIAG"
```

For more details please refer to the header file, which is throughly commented.

5.2 Older Diagnostics

We introduced the diagnostic macros described above at the beginning of March 2013. Earlier code used a less-structured approach, which has later been removed. If you used `PP_PRINTF`, it's gone by now.

5.3 Printf

The code uses *pp_printf* as a replacement for *printf*. This implementation comes from previous work by Alessandro, which in turn uses an older Linux implementation. Now *pp_printf* is published separately, as the "Poor Programmer's Printf" and included here in its own subdirectory. Please check *pp_printf/README* for more details about the size of this implementation and the different implementation choices.

By avoiding calls to the real *printf* function we allow the PPSi code base to be built for freestanding implementations without ugly `#ifdef` tricks in the code. Please note that *pp_printf* and *printf* can coexist: for example the hosted version of PPSi links with the standard library without any problem; in that case *pp_printf* relies on *fputs* to write to *stdout*.

If your run-time environment already includes an implementation of *pp_printf*, you can build with `CONFIG_NO_PRINTF` set. For example:

```
make USER_CFLAGS="-DCONFIG_NO_PRINTF"
```

The resulting `ppsi.o` will include undefined references to the `pp_printf` symbol, which must be provided externally. This is how we build for *arch-wrpc*, which already includes its own implementation if *pp_printf*. In that specific situation we still link the *libc* provided by the compiler, but we don't want to lift its own *printf* which would bring in most of *newlib* and would overflow our available RAM by a huge amount.

If your freestanding environment provides a *printf* that you want to use, and which is not called *pp_printf*, you may use the `PROVIDE` keyword in your linker script. You can find an example in the *wrpc-sw* package, which maps *mprintf* to *pp_printf* at link time, in order to accept external code that calls *mprintf*, which we don't provide any more.

6 Tools

The PPSi package includes some tools, mainly meant to help developers. Most of them live in the *tools* subdirectory and must be built separately with "`make -C tools`".

6.1 ptpdump

This is a sniffer for PTP frames. It reports all Ethernet frames and UDP datagrams that talk PTP, from a specific network interface. The output format is line-oriented to help running *grep* over log files. The number of blank lines between frames depends on how much time elapsed between them; this should help identifying sync/follow-up pairs at a glimpse of the eye.

The program receives one optional argument on the command line, which is the name of the interface where it should listen; by default it uses `eth0`.

This is, for example, the dump of two UDP frames:

```
TIMEDELTA: 977 ms
TIME: (1362504223 - 0x51362a1f) 18:23:43.958091
ETH: 0800 (00:22:15:d7:c0:ef -> 01:00:5e:00:01:81)
IP: 17 (192.168.16.1 -> 224.0.1.129) len 72
UDP: (319 -> 319) len 52
VERSION: 2 (type 0, len 44, domain 0)
FLAGS: 0x0002 (correction 0x00000000)
PORT: 00-22-15-ff-fe-d7-c0-ef-00-01
REST: seq 29, ctrl 0, log-interval 0
MESSAGE: (E) SYNC
MSG-SYNC: 1362504223.957872054
DUMP: payload (size 44)
DUMP: 80 02 00 2c  00 00 02 00  00 00 00 00  00 00 00 00
DUMP: 00 00 00 00  00 22 15 ff  fe d7 c0 ef  00 01 00 1d
DUMP: 00 00 00 00  51 36 2a 1f  39 17 f7 b6

TIMEDELTA: 0 ms
TIME: (1362504223 - 0x51362a1f) 18:23:43.958259
ETH: 0800 (00:22:15:d7:c0:ef -> 01:00:5e:00:01:81)
IP: 17 (192.168.16.1 -> 224.0.1.129) len 72
UDP: (320 -> 320) len 52
VERSION: 2 (type 8, len 44, domain 0)
FLAGS: 0x0002 (correction 0x00000000)
PORT: 00-22-15-ff-fe-d7-c0-ef-00-01
REST: seq 29, ctrl 2, log-interval 0
MESSAGE: (G) FOLLOW_UP
MSG-FOLLOW_UP: 1362504223.957953221
DUMP: payload (size 44)
DUMP: 88 02 00 2c  00 00 02 00  00 00 00 00  00 00 00 00
DUMP: 00 00 00 00  00 22 15 ff  fe d7 c0 ef  00 01 00 1d
DUMP: 02 00 00 00  51 36 2a 1f  39 19 34 c5
```

6.2 pps-out

The tool outputs a pulse-per-second signal to a parallel port or a serial port. Most likely it only works on an x86 Linux system because of direct access to I/O ports – no such portability problem is expected for serial ports. For this reason it is not built by default in the distributed `tools/Makefile`

The program receives a single argument on the command line, which is either an hex port number (with or without leading `0x`) or an absolute filename:

```
./tools/pps-out 378
./tools/pps-out /dev/ttyS0
```

When passed a port number, the program toggles all bits assuming it refers to a parallel port. When passed a pathname, the program assumes it is a serial port and it toggles the DTR and RTS modem control signals (on pins 4 and 7 of the DB9 male connector).

By setting the `VERBOSE` environment variable you ask the program to report how late it was before and after generating the rising edge:

```
 morgana$ VERBOSE=1 ./tools/pps-out /dev/ttyS0
gettimeofday takes 0.2 usec
udelay takes 4.0 usec
delayed between   0 and   2 usecs
delayed between   0 and   4 usecs
[...]
```

`pps-out` can be used to verify on a scope the level of synchronization of two or more computers, but please note that the user-space software-only approach shows some jitter; on my systems the delay is usually a few microseconds, up to around a dozen (in general, use of the parallel port has less delay and less jitter). In any case this offers a second source to check what NTP or PTP daemons report.

6.3 jmptime

The program uses *settimeofday* to make the local time jump forward or backward by some amount, specified as floating-point seconds:

```
# date +%H:%M:%S; ./tools/jmptime 1.32; date +%H:%M:%S
12:24:28
Requesting time-jump: 1.320000 seconds
12:24:29
```

6.4 chktime

This program monitors changes in the current time, but comparing the output of `clock_gettime(CLOCK_REALTIME)` and `clock_gettime(CLOCK_MONOTONIC)`. The delay between successive checks, expressed in milliseconds, can be specified on the command line and defaults to 10ms. The program only reports observed changes that are bigger than 0.5ms, to avoid excessive reporting of false positives that are simply due to process latencies induced by the system workload.

By running this program you can see the effect of *tools/jmptime*, or the insertion of a leap second (which is why I wrote this program in June 2012).

For example, this is what *chktime* reports when running "`jmptime .002`". The program doesn't need superuser privileges:

```
% ./tools/chktime 50
./tools/chktime: looping every 50 millisecs
13-03-12-12:29:41:       1996 us
```

6.5 adjtime

This program works like *jmptime*, but it requests a slow adjustment of the time. It receives the requested adjustment on the command line, as a floating point number just like *jmptime* described above. Additionally, it reports what was the ongoing adjustment using *adjtime*. See the `adjtime(2)` man page for details.

Adjustments requested by this program cannot be reported by *tools/chktime*, even when the overall change integrates to more than half a millisecond, because the change in clock speed affects both `CLOCK_REALTIME` and `CLOCK_MONOTONIC`.

You can see adjustment in one host by comparing with the time of another host, for example using *tools/mtp*, described next.

The following example shows how on my host the kernel adjusts the time by 15ms every 30s (i.e. 0.05%):

```
# ./tools/adjtime .15; sleep 30; ./tools/adjtime 0
Requesting adjustment: 0.150000 seconds
Previous adjustment: 0.000000 seconds
Requesting adjustment: 0.000000 seconds
Previous adjustment: 0.135000 seconds
```

6.6 adjrate

The program reads and optionally changes the clock rate of the host system using the Linux-specific *adjtimex* system call (the same being used in the core PPSi program). It is meant to check adjustment and get acquainted with the involved values.

The numerical argument of the system call is parts-per-million scaled by 16 bits. So for example half *ppm* is passed as 32768.

The following are example uses of the program on an ntp-driven host:

```
morgana% ./tools/adjrate --help
./tools/adjrate: use "./tools/adjrate [<adj-value> [ppm]]"
morgana% ./tools/adjrate
rate: -407582 (-6.219208 ppm)
morgana% ./tools/adjrate -6
./tools/adjrate: adjtimex(rate=-6): -1 (Operation not permitted)
morgana% sudo ./tools/adjrate -6 ppm
morgana% sudo ./tools/adjrate
rate: -393216 (-6.000000 ppm)
```

6.7 mtp

The directory *tools/mtp* includes a few example programs I wrote for an article about time synchronization. MTP means “mini time protocol”; it uses the T1, T2, T3, T4 idea to report the time difference between two hosts. The program comes in two flavors: UDP and raw Ethernet.

To run a listening server on one host, you can run the program in *listen* mode:

```
tornado% ./tools/mtp/mtp_udp -l
```

On the other host, you can run the client that reports the time difference it measures, you can pass either an IP address or an host name:

```
morgana% ./tools/mtp/mtp_udp tornado
0: rtt  0.000459000   delta   0.099351500
```

You can continuously monitor the difference by running the program in a loop:

```
morgana% while true; do ./tools/mtp/mtp_udp tornado; sleep 0.1; done
```

The program *mtp_packet* works in the same way by using raw Ethernet frames (`AF_PACKET`). It needs an interface name as first argument and superuser privileges.

6.8 MAKEALL

The *MAKEALL* script, in the top-level directory of PPSi builds the program for all known configurations, picking them from `configs/`. Developers are urged to run it before committing each patch, to ensure they are not breaking the program for configurations they are not using – but sometimes I forgot to do that myself and committed trivially-incomplete changes.

It may happen, however, that some developers experience errors or warnings that others didn't notice, because of differences in compiler version or library versions.

What follows is an older example run, limited to hosted compilation; currently, with the new Kconfig engine introduced after release 2013.11, `MAKEALL` ignores command-line arguments and always builds all configurations; but I plan to add the command line back.

```
% ./MAKEALL unix
###### Build for  arch "unix", ext "", printf xint
   text    data     bss     dec     hex filename
  15801     224     344   16369    3ff1 ppsi.o
  34370     984     380   35734    8b96 ppsi
###### Build for  arch "unix", ext "", all messages
   text    data     bss     dec     hex filename
  16850     224     344   17418    440a ppsi.o
  35410     984     380   36774    8fa6 ppsi
```

7 Build Details

This is a summary about the build process for PPSi.

The main *Makefile* creates directory names from `$ARCH` and `$PROTO_EXT`, each sub-*Makefile* then can augment `CFLAGS` with `"-ffreestanding"` or whatever it needs. Similarly, `CROSS_COMPILE` may be set by sub-Makefiles but please let the environment override it (as no custom pathnames should be edited before building and the pristine package can be used).

The basic state machine is in *./fsm.c*. It's a simple file released in the public domain as we'd like the idea to be reused and the code itself is not worth copylefting – even thought the file is not completely independent from PPSi itself.

All the rest of the package is built as libraries. The link order of libraries selects which object files are picked up and which are not. Additionally, `"CONFIG_PRINTF_XINT"` or one of the other *pp_printf* configurations can be set to override the default. By default PPSi builds the "full" implementation.

This state-machine source refers to a specific state machine by the name *pp_state_table*. The table is picked from a library: either from the extension being selected or the *proto-standard* one. The table includes pointers to functions, and such names will select which other object files are picked up from the libraries.

Individual architectures can add files to the `"OBJ-y"` make variable, in order to add their own stuff (like the *main* function or *crt0.o* for freestanding stuff). Similarly, architectures can add files to the `"all"` target. The main Makefile only builds *ppsi.o*, leaving the final link to the chosen architecture, so for example *arch-unix* adds `ppsi` to the `all` target.

Since code and data space is a problem in the freestanding world (for example, the whole ptp may need to fit in 64kB RAM including data and stack), each state in the state machine of the standard protocol must be implemented as a separate file. This allows an extension not using that particular function to save the overhead of binary size. Clearly an extension may implement several functions in the same file, if they are known to be all used in the final binary or of it uses `-ffunction-sections` and `-fdata-sections`).

8 Licensing

The code is licensed according to the GNU LGPL, version 2.1 or later. Some files are individually released to the public domain, when we think they are especially simple or generic.

Both the full and the partial printf code is distributed according to the GPL-2, as it comes from the Linux kernel. This means that any code using our diagnostics fall under the GPL requirements; you may compile and use the diagnostic code internally with your own proprietary code but you can't distribute binaries with diagnostics without the complete source code and associated rights. You may avoid the GPL requirements by using different printf implementations; if so we'd love to have them contributed back in the package.

The same issue about the GPL license applies to the *div64_32* function. We need this implementation in our *wrpc* code base because the default *libgcc* division is very big, and we are always tight with our in-FPGA memory space.

For binaries without diagnostic code and the size-optimized division, the LGPL applies, as detailed below.

For licensing purposes, any 'arch-name' or 'time-name' subdirectory "can be reasonably considered independent and separate works in themselves" (quoting the LPGL text).

Code in the directories provided here is all LGPL, and you may add your own drop-in subdirectories. The LGPL requirements for source distribution and associated permissions won't apply to any such subdirectory that you may add, even if technically such code is not linked as a library.

9 Coding Style and Conventions

The coding style is the one inherited from Linux kernel project (see *Documentation/CodingStyle* in the kernel sources). However, structures, constants and field names defined by IEEE 1588 are kept in the suggested "CamelCase" form. Similarly, the typedefs are left, even if they are really a pain to deal with. Most of this stuff is in include/ppsi/ieee1588_types.h file.

The mostly used prefix is `pp_`, the short prefix for 'Portable PTP', which is used for every function related to the algorithm itself (but not in the architecture-specific code).

Some prefixes to the IEEE naming are added, in order to improve readability:

`EN` means "enumeration type". For instance, Enumeration Time Source (defined in the spec at table 7, page 57) becomes `ENTimeSource`.

`PPM_` means "ppsi message", and is used for message types.

`Msg` means "message" and is used for message structures.

`PPS_` means "ppsi state" and is used for state machine's states definition.

`DS` means "data set", and is used for the standard Data Sets (e.g. *DSCurrent* is the "Current Data Set"). The concept of data sets is defined in the specification at chapter 8, page 63.

10 Command Line

The hosted build, as well as the two *bare* ones, include command line support.

The command:

```
./ppsi --help
```

will print help about command line options (we also support the question-mark like the original ptp, but it's a bad choice because it is a shell wildcard that should be escaped). Actually, all multi-char options will print the help at this point in time, because we only support since-char options; we don't want to rely on *getopt* which is not available for all architectures, nor we want to implement complex parsing.

Note: most current command-line options are going to be turned into configuration options. This applies to the priorities, intervals and thresholds, as well as the *slave-only* flag.

For standard operation, simply run `./ppsi` with no options. It will work like the PTPd executable, with the automatic choice of master/slave defined in IEEE specification (announce/timeout mechanism).

What follows is a list of the most important command line options. For a list of the other ones please see the help message.

`-f <file>`
: Read configuration file. The program can read several configuration files. If no `-f` is passed, it tries to read a configuration file from the standard places: `/wr/etc/ppsi.conf` if White Rabbit, and `/etc/ppsi.conf`.

`-C <config-item>`
: Pass a single configuration item, or several of them using the semicolon as separator. See Chapter 3 [Configuration], page 5 for details.

`-d`
: Diagnostics. This options receives the string of diagnostic levels for fsm, time, frames, servo, bmc, extension (in that order). See Section 5.1 [Diagnostic Macros], page 6 for details.

`-b <ifname>`
: Specify which interface to use, for Ethernet mode (default: architecture-dependent, but `eth0` for Linux builds). This option can only be used in single-port operation.

`-e`
: Run in Ethernet mode (by default PPSi uses UDP if the architecture supports it).

`-g`
: Run as slave only.