A Proposal for a portable PTP

August 2011

Alessandro Rubini

Introduction

The *White Rabbit* project is a multi-company multi-lab collaboration hosted at `ohwr.org`. Its aim is achieving sub-nanosecond synchronization among thousands of computers ("nodes") over distances of several kilometers.

Within *White Rabbit*, the implementation of the gory details is in hardware and gateware, but the communication protocol is managed by a software package. The software project is designed to be compatible with IEEE-1588 version 2 (a.k.a. *Precision Time Protocol, version 2*, or PTPv2 or simply PTP). The PTP specification allows for protocol extensions, but the published implementation doesn't do anything to ease writing them.

This is a proposal for a PTP implementation that better suits the needs of *White Rabbit* (which is a PTP extension) and possibly other users of the protocol, both with and without extensions, on diverse hardware platforms.

1 History and Requirements

The initial implementation of WR-PTP started as a fork of the official *PTPD* implementation for Posix environments, but the different requirements of the two projects led to massive changes in the code base.

To get the benefit of better maintainability of the code, but also to avoid code duplication, we are now aiming at unifying the code base of the two projects under a different layout. Our objective is designing a package that from a single set of source files can produce binaries to fulfill the 3 different PTPD binaries we are using:

- Standard PTP protocol within a Posix OS (i.e., GNU/Linux, BSD, possibly Windows);
- White-Rabbit PTP within a Posix OS (we use GNU/Linux, but there is no intrinsic limitation);
- White-Rabbit PTP on a bare-metal, without a host Operating System.

As a side-effect, the code can also build a standard PTP that runs on bare metal.

What we need, in practice, is two configuration bits; the aim is building 4 different binaries from the same code base, to consolidate development efforts and improvements so anyone can exploit the advances. The design, however, should not limit protocol extensions to WR-PTP, it should arrange for any other extension to fit in the same source code package. For simplicity, only one extension (or none of them) can be enabled for each build.

The design suggested here is centered on WR-PTP, but other protocol extensions are expected to have similar requirements as White Rabbit. If new requirements emerge over time, they will call for new solutions, but I can't foresee them at this point in time.

1.1 Different Protocols

The WR implementation of PTP is described in the *White Rabbit Specification*; current release as I write this is version 2. The document can be downloaded from the *Documents* section of the *White Rabbit* project in `ohwr.org`. Currently the direct link to the section and the file are `http://www.ohwr.org/documents/21` and `http://www.ohwr.org/attachments/656/wrspec.v2-06-07-2011.pdf`.

Standard PTP and WR-PTP can work together: a WR-PTP daemon will talk the standard protocol with non-WR peers. This behaviour is expected from all PTP extensions, so the standard protocol should always be available in PTP binaries, whether or not an extension has been selected at compile time.

The protocol differences between the standard PTP and WR-PTP are mainly in three areas:

There are additional states in the state machine.
: The PTP synchronization protocol is designed as a state machine, where the exchange of protocol packets causes transitions in the internal state of the daemon, while data is exchanged between daemon instances. In White Rabbit, the state machine has more states than in the IEEE 1588 specification, because WR needs to calibrate constant latencies in the physical Ethernet device.

Standard PTP packets are augmented wit trailing data.
: The *ANNOUNCE* packet of PTP is a fixed-length packet; the WR-PTP implementation adds WR-specific information to the end of such packet, so WR nodes can identify their peer as a WR node. Further packets can carry extra information as trailing TLV tuples, which is the standard-supported extension mechanism.

WR-PTP runs an RPC server together with the state machine
: The hosted version of WR-PTP (not the one running on bare metal) needs to interact with other processes running on the host computer. This is accomplished through an RPC server that shares the main loop with the one of the state machine.

What we need to transparently support the extension in a PTP build, is

- Change the `switch` statement that runs the state machine;
- Call extension-specific functions at the end of standard processing for each state in the machine;
- Make the state-machine a callable function, so the main loop is external and other events may be multiplexed with it

Additionally, any extension will bring in its own data structure to keep track of its own status. Such data structure needs to be allocated, freed and passed over in all PTP-related calculations.

1.2 Different Host Environments

In order for the PTP code base (with or without extensions) to be able to run in a freestanding environment, we need to move all code that makes system calls to a different source file, providing alternative entry points to be filled by each port to a different freestanding environment.

This means that the base code can make no such call as *socket*, *recvmsg*, *sendto*. While the current upstream code base has already split all such calls to the `dep/` subdirectories, it isn't really prepared to be compiled for a freestanding environment. The PTP code uses `#ifdef` to make choices, and any port to a different environment will require patching most files within the `dep` subdirectory.

The approach currently taken by WR-PTP is to identify whether or not the compiler is configured for an hosted environment. If it is, the `Makefile` makes the final pass building an executable, pulling in glue code that maps everything to system calls; if the environment is a freestanding one, the big object file is left alone.

In our opinion, the final link for non-hosted environments is better done by a different package where the complete freestanding application lives. Each microprocessor and each freestanding environment have their own peculiarities, and the related memory maps are highly dynamic; we don't think it's worth cluttering the PTP code base with such low-level details – in some way, it's not unlike what happens with dynamic linking: the executable being built is incomplete, and the final packing for execution is left to somebody else.

2 Portability Tools

In my experience, most stuff that used to be done by means of `#ifdef` can now be done much better with linker tricks.

Actually, the linker is nowadays the preferred way to achieve portability; it is heavily used as such by major boot loaders, by the Linux kernel and other projects. The kernel, in particular, uses it in several different ways and is a good source of information. Please be advised that my view of the world is biased, as I work mainly with kernel code.

The most useful features that can be used towards portability are described here.

2.1 Use of Named Sections.

By placing data in named sections, you ask the linker to coalesce such data in the final object file (or executable). In other words, you can build static arrays at compile time, and the content of such arrays comes from different source files.

In this way you don't need a central header that prototypes everything, nor you need to `#ifdef` the members of the array. Each data structure in the array may include call-backs to it own source file, while the array itself may be used to parse parse input data.

With named sections, by simply adding or removing object files in your `Makefile`, you can add or remove features from the final program. Without patching a single line of code. Each new extension is just a set of files that are added to the source directory.

Example:

```
struct action __attribute__((__section__(".actions"))) myact = {
      .command = 10, .function = eat_cmd10,
};

for (p = &act_first; p < &act_last; p++)
    if (packet->command == p->command)
        p->f(packet);
```

Please note that coalescing of named sections requires a few lines in your linker script. A full, as-simple-as-possible example of using named sections is in my *povacca* package (see Chapter 6 [Source Code Repositories], page 6).

2.2 Use of Weak Symbols

A *weak* symbol is used by the linker to resolve an undefined symbol if no *strong* symbol with the same name is available in the set of input object files. All symbols are *strong* by default.

You can use a *weak* symbol to provide a default implementation for a function or a data structure, but such default can be overridden by any file in the link-set that exports a symbol with the same name.

This can be used to allow `#ifdef` around calls to optional code or use of optional pointers. Whether or not the extra feature is available, the main source file is unchanged. For example, some global pointers may be defined as `NULL` in a *weak* symbol, so the link step may instantiate a different value.

Example:

```
int __attribute__((weak)) run_extra_code(struct ourobj *p)
{
    return 0;
}
```

2.3 Use of Aliases

You can define a symbol to be an alias of another symbol. For example, a function can be defined as a weak alias of another function. If no strong symbol is present in the link step, the alias will be used. Aliases can be used to provide several defaults that to the same thing (i.e., nothing) without writing several do-nothing functions, or provide a default implementation that falls back on another implementation without duplicating the code – and still allowing a *strong* function to take over, if available at link time.

Example:

```
int __attribute__((alias(normal_proto))) extended_proto(struct ourobj *p);
```

2.4 Use of Make Variables

The tools described up to now allow to build different executable files without resorting to `#ifdel`, and without patching global files when a new extension is added to the source tree.

Selection of which files are to be compiled (and thus linked) should then be performed by means of *make* variable, which may be pre-set either on the command line or in the environment.

I strongly advise against modifications to the Makefile, as people who must adapt the Makefile to local needs will invariably have issues when committing or fetching from repositories used by other people, and local bits tend to leak to global places. What I'd like to use for compilation is something like this:

```
make CROSS_COMPILE=arm-linux PTP_EXT=white-rabbit
```

Clearly users can choose to export environment variables in order to simply call "`make`", or can write their own scripts that pass proper arguments to the commands, without the risk to commit such local script.

3 A Design Guide

My proposal for a portable PTP, supporting WR-PTP and freestanding compilation as well as standard PTP operation is based on the following technical choices for the reasoning put forth so far. Some of the points below are for diagnosis and debugging, but rely on the same tools as the portability stuff.

- The PTP engine (with or without extensions) is built as a big `.o` file. Glue code for standard system calls is only linked in for hosted compilation, otherwise leaving the object file alone (as in current WR-PTP).
- The state machine is described by a global table, assembled in a named section, so an extension can add new states. Each state is implemented by a function that receives a pointer to the machine's status and possibly the new input packet just received.
- The global table defines two functions for each state, and they are called one after another. Standard PTP will have one function only, but extensions may choose to use the standard function and a custom one (this allows, for example, to handle trailing data in protocol packets).
- The states are integer values, and each extension is free to use any positive number above 100 (the low numbers are reserved for standard states).
- Each state in the state machine returns an integer value, which is the next state, to be used in the next iteration. Thus, the extension can insert new states for its own work. For example, WR-PTP has extra states after the ANNOUNCE packet, entered only if the peer is another WR-PTP node.
- No global data is ever used. The status structure and any other data is allocated on request using specific callbacks (freestanding implementations will most likely return a global structure, but this is an implementation detail of the specific use case).

- The code should make a number of diagnostic calls, calling specific function names (the set of such functions may increase during development). No *printf* or other *varargs* functions should be used in the state machine itself, as it increases the code size of the caller even when configuration selects a do-nothing implementation.
- All diagnostic functions are implemented as weak aliases to a function that does nothing. In this way, the overhead is limited to parameter passing in the caller and one "`return 0;`" function. A size-constrained freestanding implementation can implement no diagnostic functions at all or a limited number of them.
- From the above, the protocol operation is thus a function being called by external code, which calls the two state-machine functions for the current state. Thus, the main loop of the application is external, and the process (or freestanding CPU) may concurrently follow other activities. In some way, we may say PTPD acts as a library.

4 A Prototype

No prototype is there at this point. You'll find it in a later edition of this document, in its own git tree, unless it is considered not worth writing.

This is from an informal email I wrote:

> The prototype I thought of is trivial protocol, where daemons meet by sending announce messages and running a state machine. The source code will compile for both hosted and standalone environments, as base protocol, with an extension or with a different extension.
>
> It is extected to actually run on a network, in order to see the technical approach in action. I also wanted to run it on the SPEC, but I haven't yet studied how packets are sent and received there, so I don't know if it's feasible or not.

5 Expected portability issues

The portability tools being used depend on features of the ELF binary formats, which is being used in most of the civilized world for both hosted and freestanding environments. However, some players in the OS marketplace are still using other obsolete formats, so we have extra issues to face because of them.

Microsoft Windows is a notable user of pre-ELF formats. Their choice of still using obsolete COFF files means that *weak* and *alias* may not be available there (I've not checked myself), but named sections (the most important feature) exits in COFF and can be used under Windows. Work is ongoing to determine how to use them with a simple `Makefile`, using *povacca* as test case.

The syntax being used in my code for named sections and the rest is *gcc* syntax. This should not be a portability issue as *gcc* is the compiler of choice in all systems being used for PTP. If other compilers must be supported, an approach like `<linux/compiler.h>` may be used – the kernel was born as a *gcc-only* project, but now other important compilers are supported as well. Actually, hiding the ugly `__attribute__` is good in any case, and simpler macros should be used in C code through some preprocessor help, thus paving the way to support for other compilers.

6 Source Code Repositories

The discussion in this document refers to code found in the following repositories:

- The official PTP package is at `http://ptpd.sourceforge.net`.
- The current WR-PTP implementation is temporarily hosted at `git://gnudd.com/ptp-noposix.git`.
- Code for a freestanding WR-PTP environment is at `git://github.com/twlostow/wr-core-software.git`
- The hosted implementation of WR-PTP is part of *wr-switch-sw* at `git://ohwr.org/white-rabbit/wr-switch-sw.git`.
- A minimal example of using sections is *povacca*, something I wrote for a short lesson and now I pushed at `git://gnudd.com/povacca.git`.
- The source for this document and associated code is at `git://gnudd.com/ptp-proposal.git`.

7 Acknowledgments

The PTP specification is an IEEE effort with a number of authors.

The official PTPD is the work of George V. Neville-Neil, Steven Kreuzer, Gael Mace, Alexandre Van Kempen, Kendall Correll, Aidan Williams.

The WR-PTP is hence forked and mostly written by Tomasz Wlostowski, Maciej Lipinski and Grzegorz Daniluk.

The effort of porting PTP and WR-PTP to be portable is taken care of by Marco Pizzamiglio, who's also addressing compatibility issues of the proposed approach with compilation under Windows (using *gcc*, though).

The *White Rabbit* project is a multi-company multi-lab collaboration hosted at `ohwr.org`. Many people have been involved in designing building and running software, hardware and gateware within the project.

Table of Contents