openPOWERLINK in Linux Userspace: Implementation and Performance Evaluation of the Real-Time Ethernet Protocol Stack in Linux Userspace

Wolfgang Wallner
wolfgang.wallner@br-automation.com
Josef Baumgartner
josef.baumgartner@br-automation.com

Bernecker + Rainer Industrie-Elektronik Ges.m.b.H
B & R Strasse 1, 5142 Eggelsberg, Austria

Abstract

The RT-Preempt patch for Linux turns the Linux operating system into a real-time operating system and therefore is an ideal platform to implement a real-time Ethernet protocol stack like openPOWERLINK. The initial implementation of the openPOWERLINK stack on X86 Linux was developed as a kernel module. The solution completely bypasses the Linux network stack and achieves maximum performance through the usage of its own network interface drivers. However, this limits the protocol stack to the few openPOWERLINK network interface drivers currently available and also makes the protocol very dependent on the used kernel version. To circumvent these drawbacks, the whole protocol stack was implemented as Linux user space library. As most of the necessary real-time features are also available in user space and many applications do not need the performance level of the kernel space implementation, this solution is adequate for a lot of applications.

This paper describes the porting of the openPOWERLINK stack to user space and examines the performance of the user space implementation. Therefore, the influence of the user space implementation on the network jitter and on the generated system load is analyzed and compared with the kernel space implementation. Due to the long term goal to integrate the lower level layers of the openPOWERLINK stack into the mainline Linux kernel, in this paper it is furthermore discussed how the protocol stack could be segmented into a kernel part that would be integrated into the Linux kernel and a user part that is provided as a user space library.

1 Ethernet POWERLINK

1.1 Overview

POWERLINK is a strict, deterministic hard real-time protocol based on Fast Ethernet (100 MBit). It supports time-isochronous data transfer along with asynchronous communication between network nodes. POWERLINK was originally developed by B&R in 2001. In 2002 the protocol was opened to the public and the Ethernet POWERLINK Standardization Group (EPSG) was founded. The EPSG is an organization which drives the further development of the protocol. Ethernet POWERLINK is a patent-free and open standard, and the specification[1] is freely available on the EPSG website. As Fast Ethernet is used unmodified as the physical layer, no proprietary hardware (e.g. ASICS) is needed for an implementation. The POWERLINK network can be built with standard Ethernet hardware. Network switches could be used, but should be avoided, as they have no upper bound for the forwarding delay of frames. Instead of switches, hubs are preferred, because they provide lower latencies and more deterministic behaviour. As all nodes in a POWERLINK
network have to support the timing rules, standard Ethernet devices may not be connected directly to a POWERLINK domain.

1.2 POWERLINK Layer Model

Figure 1 shows the generic Ethernet POWERLINK layer model. The POWERLINK protocol is located at OSI layer 2 and 7 (Data Link Layer (DLL) and application layer). The characteristic timing that is used to circumvent non-real-time attributes of standard Ethernet (mainly CSMA/CD) belongs to the DLL. The POWERLINK specification defines that the CANopen interface is used as application layer. The usage of CANopen as application layer makes it easy to integrate classical CANopen applications to POWERLINK. The CANopen concepts of Device Profiles, the Object Dictionary, Service Data Objects (SDOs), Process Data Objects (PDOs) and Network Management (NMT) are all reused in POWERLINK. This is the reason why POWERLINK is often referred to as ”CANopen over Ethernet”.

![FIGURE 1: Overview of the POWERLINK Protocol Layers.](image)

1.3 Communication Principle (Data Link Layer)

A POWERLINK device can be a Managing Node (MN) or a Controlled Node (CN). A POWERLINK network has exactly one active MN (active redundant MNs are possible to increase fault tolerance). The MN regulates the activity on the network. All other active devices in the network are CNs.

Communication in POWERLINK networks happens in cycles. Each cycle starts with the transmission of the Start of Cyclic (SoC) frame by the MN. The SoC frame is sent as a multicast and can be received and processed by all other POWERLINK stations in the network. No application data is transported in the SoC; it is only used for synchronization.

Immediately after transmitting the SoC, the MN addresses each CN in the network with a Poll Request frame (PReq). Each CN responds with a Poll Response (PRes). This frame is sent as multicast and can therefore be received by the MN as well as by all other CNs in the network. Therefore, the PRes can not only send input data from the CN to the MN, but also allows cross-communication among the CNs. Direct cross-communication allows the times for data exchange between stations to be reduced considerably, since the data need not be copied in the MN.

A CN only transmits when it receives a directly addressed request (PReq) from the MN. The MN waits for the response from the CN. This prevents collisions on the network and enables deterministic timing.

A fixed time is reserved in the network cycle for asynchronous data. Asynchronous data differs from cyclic data in that it need not be configured in advance. Asynchronous data is generated on-demand by a POWERLINK station. Examples are visualization data, diagnostic data, etc. One asynchronous frame can be sent per POWERLINK cycle. The CNs can signal the MN in the poll response frame that they would like to send asynchronous data. The MN determines which station is allowed to send, and shares this information in the Start of Asynchronous (SoA) frame. Any Ethernet frame can be sent as an asynchronous frame (ARP, IP, etc.). However, a maximum length (MTU = Maximum Transfer Unit) must not be exceeded.

The most important timing characteristic in an Ethernet POWERLINK network is the cycle time, which is measured between the start of two consecutive SoC frames. The worst case jitter of the cycle time is a quality attribute of the MN. A typical POWERLINK communication cycle is shown in Figure 2.
2 openPOWERLINK

2.1 Overview

openPOWERLINK is an open source implementation of the POWERLINK technology. It was originally developed by SYS TEC electronic GmbH [9] and later released under the BSD license in 2006. The openPOWERLINK project is hosted on the SourceForge website [10].

A main design goal of openPOWERLINK was portability. Current implementations include Linux, Windows, VxWorks, bare-metal devices and more.

2.2 Software Architecture

The software architecture of openPOWERLINK is very similar to the generic POWERLINK architecture as previously shown in Figure 1. A remarkable exception is the strict partitioning in two parts:

- **Kernel part:** The DLL and all layers below, like Ethernet driver or High Resolution Timers (HRTs), are contained in what is called the kernel part. This part contains the time critical modules of POWERLINK.
- **User part:** The CANopen specific modules (Object Dictionary, PDO, SDO, …) are grouped in this part.

These two parts exchange information through the Communication Abstraction Layer (CAL). The notations kernel part and user part are currently only naming conventions. In the current implementations these two parts are always located in the same memory space. However, this is one of the preparations for future implementations where this two parts are actually split apart.

2.3 Porting openPOWERLINK

To increase portability, platform dependent code is concentrated in a few isolated places. The porting process of the openPOWERLINK stack to a new platform typically consists of the adaptation or reimplementation of the following modules:

- **Multi-Tasking:** The openPOWERLINK stack requires some kind of concurrent execution for its modules. On bare-metal devices, this can be done using IRQs, on hosted platforms this is usually implemented using the platform specific thread API.
- **Shared Buffer:** These are message queues, which are internally used by the CAL to connect the kernel and user part of the stack.
- **Ethernet Driver:** In order for the DLL to be platform independent, an interface has defined to access the network. Each platform needs an implementation of an Ethernet Driver (Edrv) module to access the platform specific network interface that uses this interface.
- **Low Resolution Timers (LRTs):** Some parts of the stack need to watch timeouts in the range of milliseconds (i.e. SDO transfer timeout). These timeouts are not critical for real time, and the timers used for these purpose are referred to as Low Resolution Timers.
- **High Resolution Timers (HRTs):** The cyclic transmission of frames is controlled by the HRTs. These timers need to handle timeouts in the micro second range with a desired precision of a few nano seconds. To generate precise isochronous SoC frames, a POWERLINK MN implementation needs a very accurate system timer and low interrupt latencies.

While the first three points are usually straightforward, the last point poses a challenge on many platforms.

3 Userspace Implementation

3.1 Motivation

On the Linux platform, previously the only implementation of the openPOWERLINK stack was completely in kernel space, having only the application code in user space. This implementation is characterized by the following properties:

+ Provides high performance and precision
- Requires special Ethernet drivers
- Maintenance burden (not mainline)
- Hard to debug
The performance and precision reached by this implementation are satisfying (cycle times down to 250µs, jitter in the two-digit microsecond range).

However, there are disadvantages: As it needs special device drivers, a new device driver has to be written for every additionally supported Ethernet chip. As these drivers are not part of the mainline Linux kernel, this will increase the amount of maintenance needed to keep them functional. Additionally, this implementation is not suitable for general purpose debugging of the openPOWERLINK stack (kernel space debugging is more difficult). This lead to the idea of porting the stack completely to user space. The result of the porting efforts should have the following advantages:

+ Support for all Ethernet chips by using some kind of standard network interface
+ Less maintenance effort (stable interfaces in user space)
+ Easier to debug than kernel space implementation
+ Still enough performance for many production applications
+ Possible first step to a later kernel space/user space hybrid solution (outlined in Section 5)

3.2 Linux platform overview

Figure 3 shows the general architecture of different openPOWERLINK stack implementations on the Linux platform. The first architecture shows the complete openPOWERLINK stack implemented in Linux kernel space. This implementation is described in detail in [2]. A long term performance and stability test of this implementation is run in the OSADL Realtime QA Farm[7]. The second architecture shows the current port to user space that is based on the pcap library, which is in the focus of this paper. The architecture on the right shows an implementation of the user space stack which uses the openPOWERLINK kernel space drivers. This implementation was developed to examine the influence of the libPCAP interface on the performance and determinism of the system. It is shown for comparison, but will not be further described in the rest of this paper.

3.3 Porting to user space

Section 2.3 sketched to general porting procedure, this section will describe the design decisions that were made for the Linux user space implementation.

3.3.1 Multi-Tasking

The user space implementation is based on the pthread library, which is used to provide concurrent execution of different openPOWERLINK modules.

3.3.2 Shared Buffer

The shared buffers in user space were implemented using plain malloc. This is possible because all parts of the stack not only reside in the same memory space, but also as threads inside the same process. POSIX semaphores and mutexes were used for synchronization between the different threads.

3.3.3 Ethernet Interface

To provide access to the Ethernet interface from user space two possible implementations could be used, either libpcap or raw sockets. Because PCAP based openPOWERLINK Edrv modules are already available for Windows XP and Windows CE, it was also
chosen as the basis for the Linux platform. As libP-CAP uses RAW sockets on Linux there is nearly no performance difference.

### 3.3.4 Timers

Implementations for both LRTs as well as HRTs use the POSIX timer API. Using POSIX timers for the needed HRTs is possible because of the high-resolution timers that were introduced by Thomas Gleixner and Ingo Molnar as part of the Linux kernel since 2.6.16. The new timer system does no longer depend on the periodic tick of the operating system and allows nanoseconds resolution. However, the resolution depends on the available timer hardware of the system. On an Intel X86 architecture there are different clock sources available (hpet, tsc, acpi) which provide a usable timer resolution in the microsecond range. These high-resolution timers can be used to increase the precision of POSIX user space timers, which is exactly what we needed.

A detailed overview of the new architecture is given in the paper *Htimers and Beyond: Transforming the Linux Time Subsystems* [3].

### 4 Performance Evaluation

#### 4.1 Test description

The different implementations of the openPOWERLINK stack on Linux were configured as MN and used to control a network of up to 30 CNs. While the timing of frames that were sent by the MN were monitored by an external PC, different load scenarios were run on the MN to analyze there influence. To simulate a real world application, each CN was equipped with digital I/O modules. The control application on the MN modified the outputs based on the input values in every cycle.

#### 4.1.1 Hardware wiring

Figure 4 shows the general hardware setup that was used for the performance and precision tests. The most interesting node in the test setup is the MN, which is shown in blue in the drawing. The MN is the Linux PC that was used to compare the different openPOWERLINK implementations. A variable amount of Ethernet POWERLINK CNs was connected to the MN using a standard Ethernet hub. Up to 30 CNs were used, partitioned as 3 daisy chains, each consisting of up to 10 nodes.

The Ethernet POWERLINK network in the figure is highlighted in orange, other connections that are shown in white indicate standard non-real-time Ethernet. As measuring network times using tools like Wireshark on a standard desktop PC suffers from larger jitter in the timestamps of individual frames, a B&R Network Analyzer X20ET8819 was used. The B&R Network Analyzer is equipped with two network ports, one for POWERLINK and one for standard Ethernet. It is able to capture frames on the POWERLINK network and timestamp them with a 20ns resolution. It packs the timestamped POWERLINK frames into UDP packets and sends them onto the Ethernet interface for further analysis. The PC that was used to collect the captured POWERLINK frames and later run statistical analyzes and create test protocol is shown in the upper right corner. To generate network stress, another external device was needed. For this purpose, another Linux PC was used, which is shown in the upper left corner.

#### 4.1.2 Node configuration

**Managing Node (MN)**

The hardware platform used for the MN was a B&R AutomationPC810[5] (APC810). Besides being designed as mechanically robust for harsh environments, it is not different from a standard X86 desktop platform.

The APC810 used in our tests was equipped with a Intel Core2Duo U7500 dual core processor running...
at 1.06 GHz, 1 GByte DDR2 PC2-5300 DRAM and a 40GB hard disk drive. The Intel 945GME chipset contains the Graphics Media Accelerator GMA 950. The APC is equipped with two on-board network interfaces, which use different Ethernet chips. One of these interfaces is based on the Intel 82573L, while the other uses a Realtek 8111B. A third interface was added as a PCI card, based on the Realtek 8139 chip.

These connections were used as follows:

- **Intel 82573L**: Used as POWERLINK interface. This interface was configured to have no IP address, to avoid interference between the POWERLINK and the Linux network stack.
- **Realtek 8111B**: Connected to the corporate network, used for TCP/IP communication.
- **Realtek 8139**: Directly connected to the PC that serves as flood ping generator (used for network stress test).

The operating system used on the MN was an Ubuntu 10.04 LTS (Lucid Lynx). We used the latest stable real-time kernel 2.6.33.7-rt30 as listed on the OSADL webpage[8]. The thread priorities were adjusted to the following values (demo_pi_console is the name of the used demo application):

<table>
<thead>
<tr>
<th>Thread</th>
<th>Priority</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sirq-hrtimer/0</td>
<td>-81</td>
<td>High-res timer</td>
</tr>
<tr>
<td>sirq-hrtimer/1</td>
<td>-81</td>
<td>High-res timer</td>
</tr>
<tr>
<td>demo_pi_console</td>
<td>-76</td>
<td>High-res timer</td>
</tr>
<tr>
<td>irq/29-eth1</td>
<td>-71</td>
<td>Network interface</td>
</tr>
<tr>
<td>sirq-net-rx/0</td>
<td>-61</td>
<td>Network handling</td>
</tr>
<tr>
<td>sirq-net-rx/1</td>
<td>-61</td>
<td>Network handling</td>
</tr>
<tr>
<td>demo_pi_console</td>
<td>-56</td>
<td>Shared Buffer K→U</td>
</tr>
<tr>
<td>demo_pi_console</td>
<td>-51</td>
<td>Shared Buffer U→K</td>
</tr>
<tr>
<td>demo_pi_console</td>
<td>-51</td>
<td>Edrv (PCAP)</td>
</tr>
<tr>
<td>demo_pi_console</td>
<td>-50</td>
<td>Low-res timer</td>
</tr>
<tr>
<td>demo_pi_console</td>
<td>-21</td>
<td>Startup thread</td>
</tr>
</tbody>
</table>

**TABLE 1:** Thread priorities used by the user space implementation.

The priorities of the system threads were increased using the tool `chrt` before the POWERLINK application was started. To adjust the priorities of the stack internal threads, the API call `sched_priority` was used during run time.

As stated earlier in section 2.3, the interrupt latencies for timer IRQs need to be as low as possible to increase precision. This is the reason why the timer related threads are set to the highest priorities. For the same reason, the priorities of network related threads has been increased. It is important that the real-time related openPOWERLINK threads all have a higher priority than the other system threads. The internal priority relation between the different openPOWERLINK threads is based on the stack architecture (timer threads higher than network threads, thread of kernel-to-user shared buffer higher than user-to-kernel shared buffer, ...). The Low-res timer threads have the same priority as the system SIRQs, as they are not critical to the real-time behaviour. The startup thread has a very low priority, because it is mainly used for initialization, but has nothing to do during cyclic operation.

**Controlled Nodes (CNs)**

A network of standard B&R POWERLINK bus couplers (X20BC0083[6]) was used as CNs. As the application on the MN should simulate a real world implementation, these CNs were equipped with input and output modules and exchanged new data in every cycle. These CNs were addressed as standard CANopen DS401: Generic I/O modules.

**Network Analyzer**

To generate high precision time stamps for the observed POWERLINK frames, a B&R network analyzer (X20ET8819) was used. This device is equipped with two Ethernet ports. One of these interfaces is used as a pure POWERLINK input port to analyze the received frames. It latches the time of reception with a precision of 20 ns. This information is packed in UDP packets and sent out on the second Ethernet port. On the PC this information can be received and further processed, i.e. to create high precision Wireshark traces. In our case these measurements were evaluated in our test program to measure the SoC jitter.

**Flood ping generator**

A standard desktop PC running Linux was used to create high amounts of network IRQs on the MN by sending flood pings.

**Measurement PC**

Another standard desktop PC running Linux that was used to dump the timing measurements sent by the network analyzer, do statistical calculations on them using GNU R, and create the test reports with LATEX.
4.1.3 Load scenario

- **Idle**: The first measurement was done on an idle system as a reference for the different stress tests.

- **CPU load**: For the CPU stress test, the tool `cpuburn` was used [11]. It is designed to load X86 CPUs as heavily as possible for the purposes of system testing.

- **Hard Disk I/O load**: The tool `dd` was used to read and write large amounts of data from and to the hard disk drive.

- **USB I/O load**: As for the hard disk, `dd` was used on an USB drive to produce USB I/O load.

- **Network load**: Heavy network stress was caused by an external flood ping on the first Ethernet interface.

- **Scheduling load**: Heavy process scheduling load was caused by `hackbench` [12]. It spawns over a hundred processes which are communicating by sending signals to each other.

- **Miscellaneous load**: To cause miscellaneous system load a Linux kernel compilation was started.

4.2 Results

**Precision**

For a comparison between the measured jitter values of the kernel space and the user space implementation, see Figure 5. The influence of the different load scenarios is very similar for both the user space and kernel space implementation. Notice however the different scale: in the range of 100 µs for user space and in the range of 40 µs for kernel space. High scheduling load has the greatest impact on the network latencies on both implementations.

**Performance**

The measured CPU load of the user space implementation on different configurations is visualized in Figure 6. The kernel space and user space implementation are compared in the following table (CPU load is given in percent of a single CPU core):

<table>
<thead>
<tr>
<th>Cycle time</th>
<th>CNs</th>
<th>User [%]</th>
<th>Kernel [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 ms</td>
<td>3</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>10 ms</td>
<td>10</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>10 ms</td>
<td>20</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>10 ms</td>
<td>30</td>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>5 ms</td>
<td>3</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>5 ms</td>
<td>10</td>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>5 ms</td>
<td>20</td>
<td>23</td>
<td>5</td>
</tr>
<tr>
<td>5 ms</td>
<td>30</td>
<td>32</td>
<td>8</td>
</tr>
<tr>
<td>2 ms</td>
<td>3</td>
<td>22</td>
<td>7</td>
</tr>
<tr>
<td>2 ms</td>
<td>10</td>
<td>36</td>
<td>10</td>
</tr>
<tr>
<td>2 ms</td>
<td>20</td>
<td>N/A</td>
<td>16</td>
</tr>
<tr>
<td>2 ms</td>
<td>30</td>
<td>N/A</td>
<td>20</td>
</tr>
</tbody>
</table>

**TABLE 2**: Comparison of the CPU load on different network configurations.

As can be seen from these numbers, the CPU load to drive the same network configuration may increase by a factor of 4-5. With the current user space implementation, it was not possible to handle more than 15 CNs with a cycle time of 2 ms. The reason for this limitation is currently unknown and must be further examined.

**FIGURE 6**: CPU load of the pcap based POWERLINK stack in different configurations.

5 Conclusion and Future Work

The measured values of performance and precision of the user space implementation are inferior to the kernel space variant, which was expected. While high performance application still need to be served by the kernel space implementation, the experiments have shown that the user space variant can be used for many applications with lower requirements. A noticeable benefit of the user space implementation is the portability. Through the use of the pcap library it can be used on any Ethernet chip that is supported by the mainline Linux kernel. In combination with
the RT-Preempt patch this implementation can be used to turn any standard X86 Linux box into a master for real time industrial networking based on Ethernet POWERLINK.

A possible and preferred future stack architecture is shown in figure 7. We would like to submit the time critical parts (mostly the Data Link Layer (DLL)) directly into the mainline kernel. The non time critical parts (mainly CANopen) could be implemented and distributed as a user space library. This setup would greatly reduce the amount of time and effort needed to turn a standard Linux installation into a hard real-time network master while still providing high performance.

The openPOWERLINK stack has already been prepared for the use as a kernel space/user space hybrid solution and many parts of the needed infrastructure are already in place. However, before we can finally split the two parts, more work needs to be done. There is ongoing effort in the openPOWERLINK community to realize the described architecture.

Additionally some enhancements in the Linux network stack architecture are needed for a high-performance POWERLINK stack. It is necessary to be able to capture and insert Ethernet packets with low latencies. These topics were already outlined by Thomas Gleixner in the document *Powerlink - Linux kernel support*[4]. As these functions are of general interest there is a good chance that they will be implemented in the mainline kernel.
References


