Karlsruher Institut für Technologie

Generic Industrial Ethernet Communication for the Linux Kernel

Bachelorarbeit
von

cand. inform. Ahmad Fatoum

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr.-Ing. Torsten Kröger
Zweitgutachter:	Prof. Dr.-Ing. habil. Björn Hein
Betreuender Mitarbeiter:	M. Sc. Denis Štogl

Bearbeitungszeit: 15. November 2017 – 14. März 2018

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 19. März 2018

Motivation

The Linux networking subsystem has evolved greatly over the 26 years of its existence to become a major selling point for using Linux for home, office, server and high performance computing.

Industrial Ethernet, however, remains largely decoupled from the Linux networking subsystem. In order to minimize delays and jitter, user mode raw sockets as well as underlying kernel infrastructure are often avoided and existing drivers are replaced with reimplementations that directly communicate with the in-kernel protocol implementation. This entails discarding time-tested drivers, duplicating functionality and code getting increasingly out-of-sync with upstream.

Refactoring efforts for bridging infrastructure in the kernel provide APIs, which may also be leveraged to implement a generic interface for industrial Ethernet communication: One that reuses mainline drivers, uses well-defined hook points in the network flow and is less prone to breakage.

This thesis examines the way existing Industrial Ethernet stacks interact with the kernel, formulates the kernel-side API required to support them and details actual implementation of a generic openPOWERLINK driver 'adapter' that hooks cleanly into the networking subsystem.

The implementation is then evaluated in how it performs in comparison with the existing custom drivers and raw socket based communication.

Acknowledgments

Many thanks to Dr. Andreas Bihlmaier of robodev GmbH for motivation, guidance and support. Thanks to Alexander Ziegler of Siemens AG for two and a half years of networking mentorship. Thanks to my parents, not only because they *suggested* an acknowledgement section. Thanks to the Linux, openPOWERLINK and other free software contributors, who were generous enough to publish code for me to study and learn from. To them, as well as all my educators, I am greatly indebted.

License

An errata sheet is available at http://a3f.at/ba/errata.html. The author is grateful for every reported erratum. :-)

Contents

List of Figures

Acronyms

API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
AVB	Audio Video Bridging
CAL	Communication Abstraction Layer
CN	Controlled Node
CoS	Class of Service
CPU	Central Processing Unit
CSMA/CA	Carrier Sense Multiple Access: Collision Avoidance
CSMA/CD	Carrier Sense Multiple Access: Collision Detection
DLL	Data Link Layer
DMA	Direct Memory Access
DoS	Denial of Service
eBPF	Enhanced Berkeley Packet Filter
edrv	openPOWERLINK Ethernet Driver
EPL	Ethernet POWERLINK
GbE	Gigabit Ethernet
hrtimer	High Resolution Timer
IEEE	Institute of Electrical and Electronics Engineers
IOCTL	Input/Output Control
IOMMU	I/O Memory Management Unit
IP	Internet Protocol
IPC	Interprocess Communication
IRQ	Interrupt Request
ISA	Industry Standard Architecture
ISR	Interrupt Service Routine
KL	Kernel Layer
LAN	Local Area Network
MAC	Media Access Controller
MDIO	Management Data Input/Output
mmap	memory map

MMIO	memory-mapped input/output
MN	Managing Node
NAPI	New API
NDIS	Network Driver Interface Specification
NIC	Network Interface Controller
PCI	Peripheral Component Interconnect
PCP	Priority Code Point
PDO	Process Data Object
PollReq	Poll Request
PollRes	Poll Response
PTP	Precision Time Protocol
qdisc	Queuing Discipline
RAR	Receive Address Register
RCU	Read-Copy-Update
RFC	Request for Comment
RT	Real Time
Rx	Receive
SKB	Socket Buffer
SoA	Start of Async
SoC	Start of Cycle
softirq	Software Interrupt Request
TAP	Terminal Access Point
TDMA	Time Division Multiple Access
TSN	Time Sensitive Networking
Tx	Transmit
UL	User Layer
VLAN	Virtual Local Area Network
XDP	eXpress Data Path

Chapter 1

Introduction

1.1 Ethernet

Ethernet, standardized as IEEE 802.3 [5], is an ubiquitous standard for Local Area Networks (LANs). In its original configuration, Ethernet is used over a half-duplex shared medium, which is prone to collisions, i.e. multiple participants transmitting at the same time. Ethernet handles this with Carrier Sense Multiple Access: Collision Detection (CSMA/CD): If a device senses another transmission while it is transmitting, transmission is stopped for a random period of time before retrying. This *binary exponential back-off* algorithm introduces a degree of nondeterminism that makes it unsuitable for real-time communication [30], where communicating nodes are expected to adhere to deadlines.

Instead, contemporary Ethernet networks operate in a Carrier Sense Multiple Access: Collision Avoidance (CSMA/CA) fashion. Switches store ingress frames to packet buffers, until the determined destination port clears, at which time, they may be forwarded, allowing for full-duplex communication. As an optimization, switches also inspect source and destination Media Access Controller (MAC) addresses to learn the network topology and to cut down on flooding all ports with ingress traffic. However, switches come with their own set of problems: Packet buffer is a limited and often shared resource, excessive loads from other ports may delay packets or even deplete packet buffers, forcing the switch to drop packets. Packet drops can be combated by instructing Network Interface Controllers (NICs) to withhold transmission until instructed otherwise. This form of flow control, standardized as Institute of Electrical and Electronics Engineers (IEEE) 802.3x [3], solves the frame drop problem at cost of increased latency and nondeterminism, making fulfillment of real-time guarantees difficult [28].

This issue is partially addressed by the IEEE 802.1Q [32] Class of Service (CoS) mechanism, which defines a new Virtual Local Area Network (VLAN)

Ethertype and a 2 octet long VLAN tag, which includes a VLAN tag protocol identifier (VLAN ID) and a Priority Code Point (PCP). VLAN-enabled switches take the VLAN tag into account when forwarding frames and, ideally, the switch provides a separate hardware queue for each of the eight PCPs. This, coupled with a scheduling policy favoring higher priority traffic and a guarantee that the network will not be overwhelmed, ensures determinism for the high priority traffic classes. Further refinements to IEEE 802.1Q are ongoing (AVB and now TSN) to address the need for a general time sensitive networking standard. Ethernet has been around long enough, however, to prompt emergence of a number of non-standard real-time extensions over the years. This is only natural, as with Ethernet becoming *the* LAN used in the home, office and IT sectors, the cost of using Ethernet in the field has decreased dramatically.

1.1.1 Real-Time Ethernet Implementations

To aid with identification of NIC and driver requirements for a Linux implementation, some of the more common real-time Ethernet approaches will be briefly discussed.

PROFINET IO

PROFINET [26], defined by several manufacturers (including Siemens), leverages VLAN and Time Division Multiple Access (TDMA) for prioritization between streams in switched 100 Mbit/s networks. A central engineering software (I/O-Supervisor) calculates the largest non-periodic time slice and further divides it into phases, which are then programmed into each device (e.g. data length and offset from the start of the cycle). A bandwidth limit is observed to avoid overwhelming the switches and unallocated time slices are dedicated to best-effort (non-real-time) traffic. By adhering to time slices, the time each packet spends in packet buffers is minimized, and deadlines can be met because it was determined a-priori that the switch will be able to forward the packets in time.

A major complication with TDMA is the need for continuous clock synchronization [55]. Even after devices agree on a common time base for the start of the cycle, fluctuations in voltage and temperature will inevitably lead to the clocks drifting apart, necessitating recurring synchronization, especially with sub-millisecond cycle times. PROFINET IRT extends PROFINET RT with periodic clock synchronization based on 802.1AS [50] (gPTP) to achieve cycles in the order of a few tens of microseconds, but requires custom switches.

While PROFINET experiences heavy use in industry, its patent restrictions [21] make it less viable for this thesis as no open source PROFINET implementation

exists to build upon. Siemens offers PCIe Communication Processors (CPs) for use in Linux PCs [54], but with a proprietary userspace protocol stack.

Sercos III

In a Sercos [52] system, the master detects connected slaves, measures their latencies and assigns them time slices and slots within the Master Data and Acknowledge Telegrams (MDT & AT). The communication cycle is initiated by the master sending a MDT with synchronization information and device-specific control commands. Afterwards, the master sends an AT, which is an empty frame into which slaves insert their data. In addition to the Real Time (RT) channel, a Unified Communication Channel (UC) is provided for non-RT communication. If UC communication occurs within the RT interval, the Sercos node is responsible for delaying frame forwarding till UC.

Bosch Rexroth provides an open source SERCOS III Softmaster. Raw socket based Linux integration is also available [53].

EtherCAT

In Beckhoff's EtherCAT [37], the master is the only node allowed to send a new EtherCAT frame. This frame passes through the daisy-chained slaves, with each slave processing the frames and inserting their own data as the frame is moving downstream. For maximum performance, slaves use Application Specific Integrated Circuits (ASICs) which integrate a two-port switch and update the frame directly via Direct Memory Access (DMA). This means that only hardware propagation delay times affect the latency, making communication deterministic.

From the viewpoint of the master, a normal Ethernet frame is sent and received, which means that a standard Ethernet MAC may be employed. Open source implementations are available, for example EtherLab [35] and SOEM/SOES [49]. Linux integration is provided via raw sockets and a few custom Ethernet drivers. EtherLab also provides a generic driver, which may be relevant to this thesis.

Ethernet POWERLINK (EPL)

B&R's Ethernet POWERLINK [22] extends the old Ethernet over shared medium with polling based access control. The master, Managing Node (MN) in EPL parlance, periodically sends a poll request to each Controlled Node (CN), which grants bus access for a predetermined time. This allows implementation of MNs and CNs completely in software, but for isochronous real-time communication, hardware support is still required [60]. This is usually in the form of a specialized MAC that transmits a buffer as soon as the poll request is received.

With openPOWERLINK [7], a full-featured open source implementation of both MN and CN is available. Linux integration is done via raw sockets and a few custom Ethernet drivers. This thesis will mainly concentrate on EPL and its openPOWERLINK implementation, as the maturity and open source license of the openPOWERLINK project as well as the possibility to use it with stock NICs make it an attractive platform to build upon.

1.2 Ethernet POWERLINK

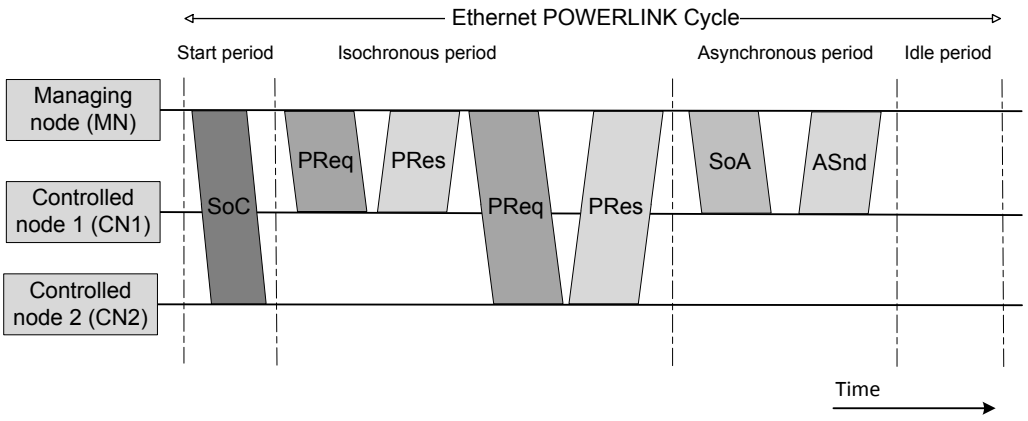

Figure 1.1: Ethernet POWERLINK timing, based on [60]

As with most other real-time protocols, communication occurs in cycles. The MN arms a timer and sends out a multicast Start of Cycle (SoC) frame on every timer expiration. Devices sample their process state as soon as they receive the SoC. The master then polls each active CN by sending a Poll Request (PollReq). The CN replies with a Poll Response (PollRes) containing the predefined Process Data Object (PDO) payload. Unlike the unicast PollReq, the PollRes is a multicast frame, which allows direct CN-to-CN communication. CNs may also request permission to send acyclic data by setting the *RequestToSend* flag inside their PollRes. After the MN finishes polling the CNs, a Start of Async (SoA) frame is sent out, which signals the beginning of the Asynchronous Period. The SoA may grant a specific CN the permission to transmit a single packet during this period. Two types of asynchronous transmission are possible: An EPL Async Send (ASnd) frame or a "normal"[1] non-EPL Ethernet frame.

As the MN acts as bus arbiter, determinism is ensured as long as devices refrain from transmitting out of turn. For this reason, attaching non-EPL nodes is

[1] EPL frames are identified by Ethertype `0x88AB`.

not permitted. This also makes the use of switches to restrict collision domains unnecessary and hubs may be used instead. In practice however, switches may still be used despite the associated overhead, as they are easier to source.

1.2.1 openPOWERLINK

Subject of this thesis will be integration of the openPOWERLINK stack into the existing Linux networking subsystem in a way that makes writing custom drivers no longer necessary.

The openPOWERLINK stack is divided into a User Layer (UL) and Kernel Layer (KL), separated by a Communication Abstraction Layer (CAL) [58]. When hosting the KL in the kernel, direct function invocation can longer be used as CAL mode because KL and UL exist within different address spaces. Instead, the kernel module creates a `/dev/plk` device file, which may be opened by the UL for communication. The UL may memory map (mmap) the device file to get a shared memory region where the PDOs are placed. Sending commands to the KL is done by using the general purpose Input/Output Control (IOCTL) interface. Despite the name, the KL is not required to be hosted in-kernel, but can also be compiled as library which user applications may link against. Furthermore, the KL may even be implemented on a separate communication processor for maximum performance, in which case the CAL is used to configure the external device. While an integral part of openPOWERLINK, this thesis won't touch upon the UL and CAL, but on the networking flow inside the KL. For this, a brief familiarization with the concerned modules in openPOWERLINK, particularly on Linux, is in order.

Data Link Layer (DLL) The DLL is the component responsible for protocol handling in openPOWERLINK. It fills the role of the networking subsystem in Linux with the difference that it only supports POWERLINK. It also defines the Application Programming Interface (API) openPOWERLINK-capable Ethernet drivers need to export.

Virtual Ethernet While a CN may inject an arbitrary Ethernet packet during the asynchronous period, doing so using the normal socket operations is problematic: Sockets are decoupled from the EPL cycle and the packet may be delayed, so that it falls outside of the asynchronous period. Moreover, with custom Ethernet drivers, the NIC and its driver are decoupled from the Linux networking subsystem and no network interface exists that the sockets may be bound to. For this reason, the KL implements a virtual networking device (`veth`) that may be transparently used for asynchronous communication. In user mode, the Linux TAP API [2] is used for creating a virtual device that delivers incoming packets to a userspace

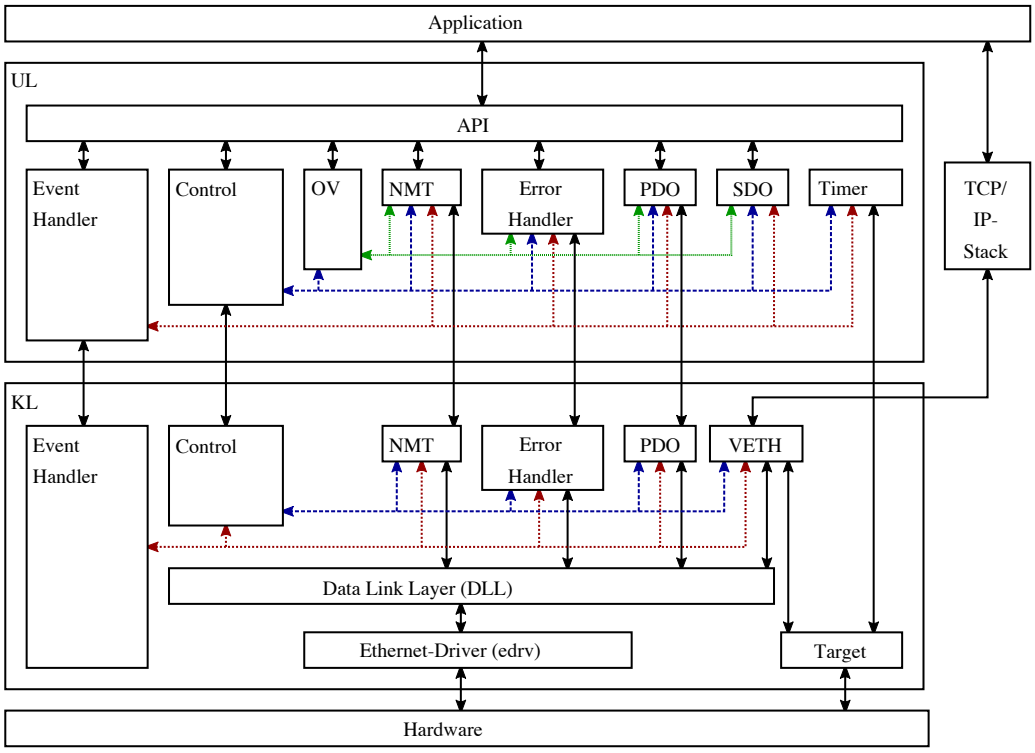

Figure 1.2: OpenPOWERLINK Architecture from [38], reused with permission.

process. When a packet is sent over the virtual interface, the DLL arranges for the appropriate flag in the PollRes to be set and buffers the packet until the SoA grants use of the asynchronous period.

`plkload` When building openPOWERLINK, each selected Ethernet driver will result in a separate self-contained kernel module (`*.ko`) which may be loaded with the `plkload` utility. `plkload` unbinds the NIC's original driver via SysFS and calls `insmod(1)` to insert the new kernel module. Original NIC drivers are detected by consulting `devices.txt` in the same directory. The file contains a mapping between the *edrv* name, the NIC name as shown by `lspci(1)` and the default kernel driver.

openPOWERLINK Ethernet Driver (edrv) The `stack/src/kernel/edrv` directory of the openPOWERLINK source tree contains the Ethernet drivers for all supported targets, including the libpcap based driver. *edrv*s are selected at configuration time and are each statically linked with the rest of the openPOWERLINK stack.

*edrv*s do not use the standard Linux networking API but implement a custom one, tailored for openPOWERLINK usage, in particular the DLL component. This is described here using the example of the Intel I210 NIC *edrv* and the `demo_mn_console` demo application.

Required *edrv* API

Initialization (`edrv_init`)

Initializes the Ethernet driver in the process context of the MN/CN calling `ioctl(2)`. This usually includes registering as Peripheral Component Interconnect (PCI) driver with the kernel, allocating transmission buffers and retrieving the MAC address, if the DLL has not specified any.

Because `plkload` unloads the previous NIC driver, registering with the PCI subsystem will directly invoke the PCI driver's `.probe` callback. The `.probe` callback does the usual tasks of mapping the required memory-mapped input/output (MMIO) regions, allocating descriptors and buffers, resetting the NIC and awaiting link up. Existing drivers depend on this and PCI hot-plugging may therefore lead to undefined behavior.

Transmission Buffer Allocation (`edrv_allocTxBuffer`)

Runs in the process context of the `eventThread` kernel-mode thread. When setting up a node, transmission buffers are allocated up-front for the different frame types, e.g. Poll- and Identification-Response.

Existing *edrv*s use the openPOWERLINK-provided stack-based `bufalloc` API to manage the transmission buffers previously allocated in `edrv_init`. As the buffers will usually be used in DMA, the usual precautions need to be taken: The buffer needs to be physically contiguous, caches need to be flushed before being handed to the NIC and invalidated after transmission completion and possibly more depending on the architecture [19]. This may be solved by either allocating consistent (cache-coherent) memory or using the Linux streaming DMA API.

Transmission Buffer Sending (`edrv_sendTxBuffer`)

Runs either in the process context of the `eventThread` kernel-mode thread for acyclic communication or in the interrupt context of the High Resolution Timer (hrtimer) for cyclic communication. Prepares a buffer for transmission by the NIC. This is usually done by initializing a transmission descriptor to point at the buffer. The driver's configuration may define `EDRV_USE_TTTX` to indicate hardware support for time triggered transmissions, which is the case with Intel I210 *edrv*.

If non-coherent memory is used, `dma_map_single` must be called on the buffer before finalizing the descriptor.

Transmission Completion Handler (`pfnTxHandler`)

Usually called from the interrupt handler for transmission completions. It signals the DLL that the buffer may be reused, which may result in calling `edrv_sendTxBuffer`. On Linux, entry into the openPOWERLINK DLL is protected by a spinlock and the completion handler runs with interrupts disabled. As locks are non-recursive on Linux, calling this handler in `edrv_sendTxBuffer` will cause a deadlock.

If non-coherent memory is used, `dma_unmap_single` should be called on the buffer.

Reception Handler (`pfnRxHandler`)

Usually called from the interrupt handler for packet reception.

If non-coherent memory is used, `dma_sync_single_for_cpu` must be called on the buffer before it is accessed.

MAC address retrieval (`edrv_getMacAddr`)

Runs either in the process context of the `eventThread` kernel-mode thread or the CN/MN calling `ioctl(2)`.

Can be readily implemented by returning a pointer to the MAC address configured during `edrv_init`.

Transmission Buffer Freeing (`edrv_freeTxBuffer`)

Runs in the process context of the `eventThread` kernel-mode thread. Called when removing nodes or prior to `edrv_exit`.

Deinitialization (`edrv_exit`)

Runs in the process context of CN/MN exiting by `ioctl(2)`. Undoes any initialization done within `edrv_init`.

Optional *edrv* API

These may be implemented as empty functions, provided the NIC is put into promiscuous or all-multicast mode. As promiscuous NICs forward all packets to the Central Processing Unit (CPU), this may affect real-time behavior on buses with many CNs.

`edrv_setRxMulticastMacAddr`

Configures the NIC to receive packets from the specified multicast address.

edrv_clearRxMulticastMacAddr
Configures the NIC to stop receiving packets from the specified multicast address.

edrv_changeRxFilter
Configures a response frame to be sent out when a received frame matches a filter.

Configurable *edrv* API

Implementation of this API is optional, unless configured otherwise in openPOWERLINK's CMake build system.

edrv_getDiagnostics
Populates a string with *edrv*-defined diagnostics.

edrv_releaseRxBuffer
For zero-copy use, pass ownership of receive buffers directly to the DLL instead of copying and export this function for deferred buffer release.

edrv_updateTxBuffer
Update the transmission buffer used with the auto-response filter.

edrv_getMacTime
Retrieve the NIC's current time in the time domain used for time-triggered transmissions.

1.3 Linux

In the project's own words [57]:

> Linux is a clone of the operating system Unix, written from scratch by Linus Torvalds with assistance from a loosely-knit team of hackers across the Net. It aims towards POSIX and Single UNIX Specification compliance.
>
> It has all the features you would expect in a modern fully-fledged Unix, including true multitasking, virtual memory, shared libraries, demand loading, shared copy-on-write executables, proper memory management, and multistack networking including IPv4 and IPv6.

> It is distributed under the GNU General Public License v2 - see the accompanying COPYING file for more details.

While the copyright for Linux source code lies with its respective contributors, its license grants end users the freedom to run, study, distribute and modify the software. This freedom has helped Linux establish itself as the leading UNIX clone, and the wide range of available software and supported hardware naturally made it a viable choice for use in embedded platforms, many of which have real-time guarantees to uphold.

Design

Like most Unix-like operating system kernels, Linux has a monolithic design. All kernel-provided services run within a privileged kernel mode in a separate kernel address space (termed kernelspace). Linux is modular and allows services to be disabled at configuration time or compiled as kernel modules, which are loaded at runtime. These kernel modules, however, cannot expect a stable API and require maintenance to be kept up-to-date with changes in the Linux kernel itself [40]. This allows a great deal of flexibility to kernel developers, as backward-compatibility is only guaranteed to userspace, but in the kernel, APIs may change freely if the need arises. In-kernel modules however, need not worry about API changes, as they are kept in-sync with the kernel's other subsystems. This doesn't apply to the out-of-tree openPOWERLINK, which may be compiled as Linux kernel module and would require regular maintenance to keep operating with newer kernels.

The kernel exists to serve the needs of user processes, which run within the less privileged userspace. Processes run each in a separate address space and are provided with Interprocess Communication (IPC) mechanisms for communications with each other and the outer world. These mechanisms are available via the system call interface, which is a special trap instruction that elevates the execution level and invokes a kernel-configured entry point. Linux guarantees the stability of this system call interface to user applications. Sockets are the primary IPC mechanism for interaction with the network. When doing a socket-related system call, execution eventually reaches the networking subsystem which interacts with the device drivers to Transmit (Tx) and Receive (Rx) network packets. Depending on the architecture, special device I/O instructions are provided or the devices are memory mapped and usable with normal memory loads and stores. The popular x86 architecture supports both modes. Additionally, devices may interrupt the processor to signal events that it needs to attend to. When such an Interrupt Request (IRQ) is asserted, the CPU checks whether that particular interrupt has been disabled (masked out) and, if not, invokes the previously associated Interrupt Service Routine (ISR) to handle the interrupt.

Bottom Handling Mechanisms

For the duration of the interrupt handler, the original application is suspended and Linux disables all other IRQs. It is thus paramount to keep interrupt handlers as short as possible in order to keep the system responsive. This can be achieved by splitting interrupt handlers in two: A top half that does time-critical work and runs with interrupts disabled as a direct consequence of the IRQ, and a bottom half that dequeues a work item added by the top half and processes it with interrupts enabled. Linux realizes this with the Software Interrupt Request (softirq)[2] mechanism. The *hardirq* handler does the time-critical processing and schedules the softirq to run. On return, the kernel polls for queued softirqs and runs their appropriate softirq handlers. Softirqs are also polled on exit from softirq handlers and when exiting critical sections where bottom half mechanisms were disabled. To avoid excessive processing delays, at maximum 10 softirqs are handled in a row, before handling is offloaded to the per-cpu `ksoftirqd` threads. `ksoftirqd` is also scheduled when raising a softirq in process context. `ksoftirqd` runs at the normal user-level process priority, so servicing a softirq may either be the second highest priority event in a system (after hardirqs) or it may be as important as any other user-level process, if `ksoftirqd` has been scheduled as a result.

Softirq levels and handlers are defined at compile-time and may run concurrently on different cores. Of primary importance to us are the networking subsystem's `NET_RX_SOFTIRQ` and `NET_TX_SOFTIRQ`. The `TASKLET_SOFTIRQ` is multiplexed between multiple *tasklets*[3], which may be created dynamically. No two tasklets of the same type may run concurrently. The third bottom half mechanism is the work queue, which is basically a simplified API for using kernel threads. This is for less time-critical work and is subject to the usual scheduling.

1.3.1 Real-Time considerations

In an EPL system, a timeout is associated with each PollReq, which defines the time span within which a CN may answer with a PollRes. If the CN exceeds its allotted time, the computed PollRes is discarded. If the device were to send out-of-turn, it might cause collisions ruining the determinism of the system. As packet reception is usually announced via interrupts, the following real-time constraint can be formulated: The time from reception of the PollReq-related interrupt until sending the PollRes may not exceed the deadline. This constraint is easily satisfied if the system is idle and only does work when interrupted. The SoC's interrupt handler samples the data and the interrupt handler of the PollReq just transmits

[2] This is distinct from hardware-provided software interrupts, which are a form of a kernel trap instruction.

[3] Despite the name, these are unrelated to tasks, which are always managed by the scheduler.

the precomputed frame and both interrupt handlers have a known upper bound on execution time. However this is not practical as nodes usually do other work besides communicating and dedicating a core exclusively to interrupt processing may not be economical. It is then the task of the operating system to preempt lower priority tasks to guarantee real-time operation for the higher priority ones.

Scheduling

The POSIX Real Time Extensions [33], which Linux implements, define two real-time scheduling policies:

`SCHED_FIFO` Fixed-priority preemptive first-in first-out scheduling.

`SCHED_RR` Same as `SCHED_FIFO`, but with a time quantum, after which the next task is scheduled in a round robin fashion.

POSIX requires at least 32 distinct priority levels, Linux supports 98 (1 (low) to 99 (high)). As normal threads are always at the lowest priority (0), only interrupt handlers and higher priority threads may preempt a real-time `SCHED_FIFO` thread. To avoid lockup of the system by a non-yielding real-time task, Linux may throttle long-running real-time tasks to give `SCHED_OTHER` (non-real-time) tasks a chance to run. The definition of long-running may be configured by the user or disabled completely.

Preemption

Interrupts, by definition, interrupt the processor and thus, unless masked out, enjoy the highest priority. This, however, may be at odds with the operation of a real-time system [10], as a real-time userspace task may be more important than e.g. a diagnostic serial port's interrupts. Linux solves this with threaded IRQs: Ideally, the top half does only the bare minimum of checking whether the handler is responsible[4], acknowledges it to the hardware and returns. The kernel then automatically wakes up a dedicated IRQ handler thread for bottom half processing. This thread is then subject to normal kernel scheduling. In our scenario, it allows the real-time task to preempt even the less important serial port bottom half. Threaded IRQs are optional[5], however, as the increased determinism they provide comes at cost of throughput. The same applies for many other real-time enhancements: priority inheritance protocols avoid priority inversion at the cost

[4] IRQ lines may be shared.

[5] the `threadirqs=1` kernel parameter may be supplied at boot to force them.

of increasing context switching time; replacing spin-locks with mutexes that may sleep improves the kernel's reaction time, but with considerable overhead.

For this reason, the kernel and the PREEMPT RT patchset offer multiple preemption models, which users may select at configuration time to strike their preferred balance between responsiveness and throughput:

CONFIG_PREEMPT_NONE No forced preemption, preemption happens only on return from system calls and due to interrupts. This is useful for batch job execution.

CONFIG_PREEMPT_VOLUNTARY Additional explicit "preemption points" are added throughout the kernel. This is the default "Desktop" preemption model shipped in most Linux distributions.

CONFIG_PREEMPT The highest degree of preemption possible in an unmodified kernel. All kernel code outside of spinlock-protected regions and interrupt handlers may be preempted. Explicit preemption points are added after critical sections. Some distributions ship this as a separate low-latency kernel package [16].

CONFIG_PREEMPT_RTB Threaded IRQs are forced. Softirqs only run within the thread context and are preemptible. Timers are moved to a separate `ktimersoftd` thread. Raising a softirq in thread context guarantees that this softirq and no other will be handled when exiting a context where softirqs were disabled. This is part of the PREEMPT_RT patchset and is mainly used for its testing and debugging [44].

CONFIG_PREEMPT_RT_FULL Extends CONFIG_PREEMPT_RTB. Most spinlocks are replaced by rtmutexes, which are special mutexes that implement a priority inheritance protocol. This is part of the PREEMPT_RT patchset.

While most of the patchset has found its way into mainline Linux, the last two `CONFIG_PREEMPT_RT*` configuration options require applying the patch. PREEMPT_RT is an official Linux Foundation sponsored project and generally patchsets are made available for long term support releases of Linux.

Berkeley Socket API

Linux, as virtually all modern networked operating systems, exposes network access through the Berkeley Sockets IPC API. Therefore a brief introduction of the socket API is in order:

socket(2) At socket creation time, the user selects socket domain, protocol and type. Different protocol implementations may register with the kernel to handle their respective protocol. For generic communication (user specifies all layers of the communication), raw sockets can be used. When those are selected (type=`SOCK_RAW` or domain=`PF_SOCKET`), the user is given complete control over the data link layer and up. The resulting file descriptor may then be used for all further operations.

bind(2) Binds a socket to a network address.

write(2),send(2),sendto(2),sendmsg(2) For sending data. The data buffer is wrapped by the lower protocol layers configured in the socket call and sent out via the specified interface(s).

read(2),recv(2),recvfrom(2),recvmsg(2) For receiving data. When a packet arrives at a bound interface, the protocol handler receives the data and strips the lower level layers before handing the packet back to the network stack for further delivery to the user.

setsockopt(2) For configuring socket and networking subsystem parameters, similar to IOCTLs.

1.3.2 Transmit/Receive Path in Linux

Adding inter-network routing to the equation greatly complicates the design and setup of real-time networks. For this reason, industrial Ethernet protocols are generally non-routable and are situated directly above the data link layer, which corresponds to the Linux `PF_PACKET`[6] address family. Data sent from a raw socket is output on the medium as-is; reading from the raw socket reads all data entering or exiting the networking subsystem through the device driver. The *libpcap* library exports a cross-platform API offering a raw socket-like interface and is used in network analyzers like tcpdump and Wireshark.

Use of *libpcap*, and its underlying raw socket, is the basis for the current NIC-agnostic openPOWERLINK *edrv*, which requires no kernel-side modifications. However, this is not without disadvantages: Reaction times to packets and timer expirations suffer because of context switches and scheduling delays, while the in-kernel openPOWERLINK DLL may react immediately as soon as the interrupt fires.

[6] Frames and packets are often used interchangably, which is the case in this thesis as well.

To aid with the later identification of possible hook points, we'll take a look at a packet's flow through the kernel when transmitting and receiving.

openPOWERLINK Initialization

`pcap_create`, which is passed the interface's name, is used to create two pcap instances, one for sending and the other for reception. The interfaces are put into promiscuous mode and immediate mode is set. A new thread is created to poll for new packets. It is assigned a medium priority and scheduled according to the `SCHED_FIFO` real-time policy. The newly created thread blocks waiting for new packets. The packet handler examines the source MAC address of the ingress packet and either calls the Tx completion or Rx handler.

PCAP and Socket Interface Each created `pcap_t` instance wraps a `PF_PACKET` raw socket[7]. The immediate mode disables use of the `TPACKET_V3` mode even if available, which is currently the only mode which may involve buffering of packets. The underlying mechanism, PACKET_MMAP, addresses a major annoyance when sending a great number of packets with sockets: The need to call into the system for transmitting or receiving each individual packet. With PACKET_MMAP, users may request allocation of Tx and Rx buffer rings in userspace, which the kernel may asynchronously read or write to. System call use can then be restricted to synchronization, e.g. when polling the kernel for the arrival of new packets. Even with PACKET_MMAP being geared towards high throughput applications, latency-sensitive applications may benefit too: Userspace incurs only one system call when receiving a packet burst. As of v1.8, libpcap only allocates Rx rings and each transmitted packet results in a system call.

Packet Reception

After the NIC receives a packet of interest (i.e. with a MAC address matching the configured filters), the packet is copied into a ring buffer in kernel memory. Many newer NICs support multiple independent hardware queues, which could be used to prioritize ingress traffic. This is however not necessary in a polled system such as Ethernet POWERLINK. The NIC then raises an interrupt with the CPU responsible for the queue. The Rx interrupt handler of the driver runs and wakes up the New API (NAPI) subsystem by raising a `NET_RX_SOFTIRQ`[8].

The softirq, which runs with interrupts enabled, repeatedly calls the driver's `poll` function to harvest received packets from the ring buffers. Received packets

[7] On older Linux kernels, the older `SOCK_PACKET` type may be used.

[8] This is true for most network drivers, which have been ported to the New API.

are wrapped as Socket Buffers (SKBs) and passed up the networking stack by means of `__netif_receive_skb_core`[9].

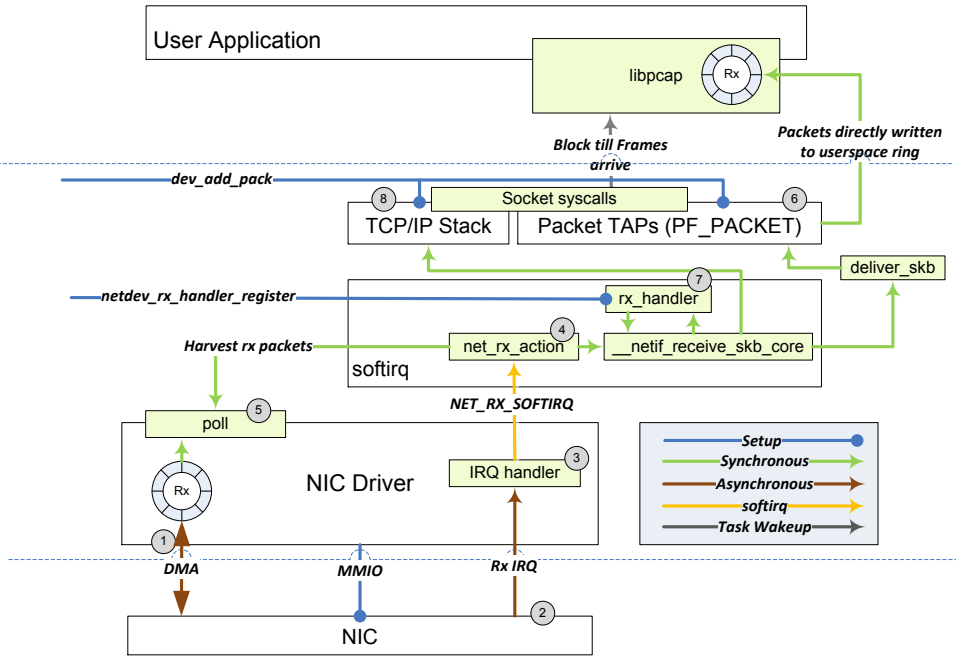

Figure 1.3: Linux Networking Receive Path

Numbering describes the chronological order

`__netif_receive_skb_core` is the main driver-side entry point into the networking subsystem. First, global and device-local packet TAPs are checked and if TAPs are installed, `deliver_skb` is called to pass them the packet. This is the case here (1.3) as *libpcap* is listening on the interface. Afterwards, if previously registered, the interface's `rx_handler` is called. The `rx_handler` API allows the registrant to easily control further processing of ingress packets, e.g. by consuming it or changing the destination device and looping it once more through the receive path. Afterwards the packet's Ethertype is compared with registered protocol handlers (added by means of `dev_add_pack`) and the matching protocol is delivered the SKB. Otherwise, frames are discarded and memory is reclaimed.

On contemporary Linux systems, libpcap would have configured the raw socket to use `PACKET_MMAP` and the payload is copied into the shared buffer. Immedi-

[9] We assume Generic Receive Offload (GRO) and Receive Packet Steering (RPS) to be disabled, as they would negatively impact packet delivery latency in favor of throughput.

ately afterwards, a wakeup message is sent to the raw socket. The process is added to the run queue and will be scheduled according to the scheduling policy in place.

Cyclic Packet Transmission

Linux v2.6.16 introduced hrtimers, timers which may be configured at sub-jiffy[10] granularity. These are exposed to userspace as POSIX real-time timer functions [33]. For cyclic communication as in the case of the `demo_mn_console`, the cyclic thread block waiting on `clock_nanosleep(2)`[11], which wakes up after the pre-configured monotonic time span elapses. The prepared frame containing the PDO data is then sent out via `pcap_sendpacket`, which calls Linux' `send(2)` system call under the hood.

This shows the most profound problem with the approach: The common task of periodic transmission of a buffer incurs scheduling delay and a context switch, only to immediately switch into kernel context for the system call and back. In light of the recent Meltdown exploit [46] findings, this effect is amplified: Context switches into the kernel require page tables to be swapped and without special hardware support, the translation look-aside buffer (TLB) needs to be completely invalidated, with adverse effects on latency.

Calling into the kernel to transmit the packet will eventually drop to `tpacket_snd`, which calls `dev_queue_xmit` to hand over the packet to the `qdisc` mechanism[12]. `__dev_queue_xmit`, which must run with interrupts enabled, picks a transmission queue and if the Queuing Discipline (qdisc) doesn't do traffic shaping, the queue is empty and qdisc is not already running, tries to claim the transmission lock, which grants access to the driver's `.ndo_start_xmit` routine. If the lock is already claimed, the `NET_TX_SOFTIRQ` softirq is raised to retry transmission at a later time. This also happens if the qdisc couldn't be bypassed, in which case the new socket buffer is enqueued and transmission is delayed until softirq activation. Figure 1.4 illustrates the primary stages of the transmission flow.

[10] A jiffy is the default scheduling time quantum used by the kernel, usually a few milliseconds.

[11] With the PREEMPT RT patchset applied, signal delivery may not happen within interrupt context due to locking constraints. However, waking up threads blocked on `clock_nanosleep(2)` is possible.

[12] Alternatively, user programs may elect to bypass the qdisc by setting the socket option `PACKET_QDISC_BYPASS`.

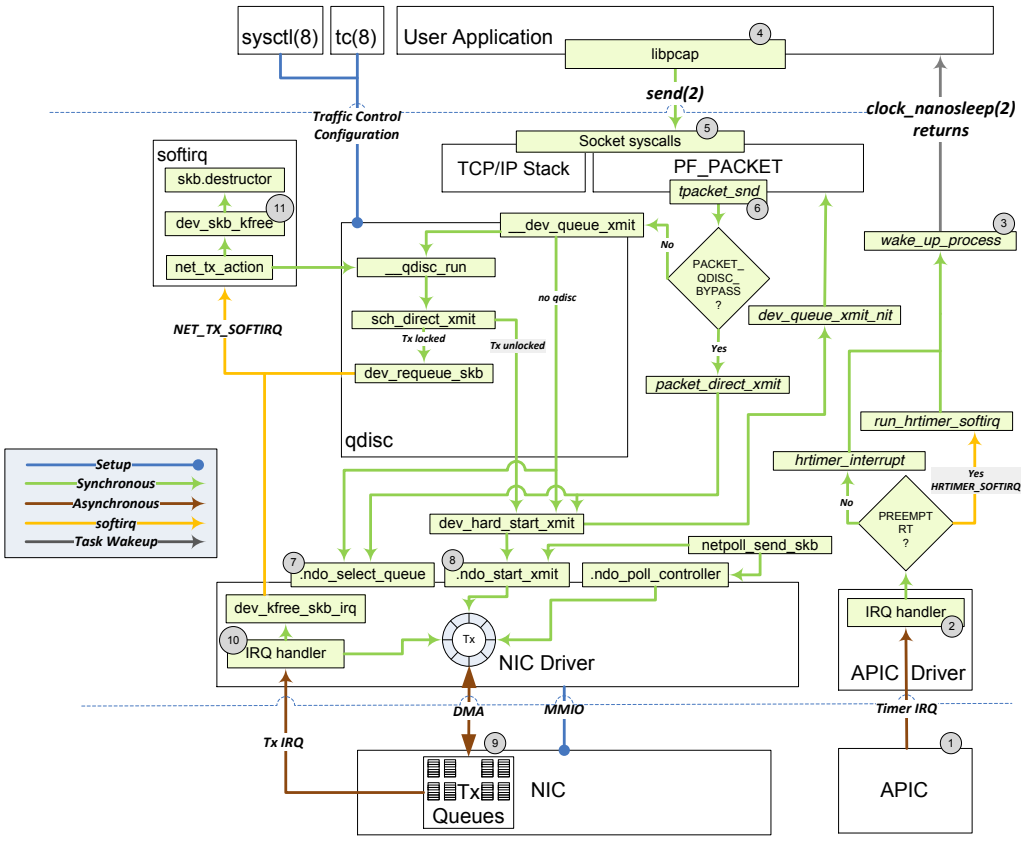

Figure 1.4: Linux Networking Transmit Path

Numbering describes the chronological order

Chapter 2

Related Work

While industrial Ethernet drivers have yet to find their way into the mainline kernel [1], Linux is ubiquitous and so are Linux systems in the field doing real-time Ethernet. In this chapter, available interfaces for integrating real-time Ethernet into Linux are described to aid with implementation of a generic openPOWERLINK driver.

2.1 Userspace Pass-through

The author's experience suggests that proprietary userspace implementations remain the status quo in commercial industrial Ethernet implementations. The NICs used are often specifically designed for accelerated handling of the protocol. A relatively small kernel module is inserted that initializes the NIC and allows mapping its MMIO regions into userspace. Afterwards, the driver is only invoked on reception of interrupts, which it passes through to userspace. The userspace protocol implementation then maps descriptors and buffers and does further configuration and protocol handling from userspace.

This dichotomy is to avoid being required to disclose source code, as the Linux license explicitly exempts reliant userspace applications from being derivative works [56]. Research into open source implementations or usage of commodity NICs in such a setup appears scarce, not least because often open source development is restricted or encumbered by patents on many of these technologies [21].

[1] openPOWERLINK v1 was briefly in the staging tree [41], but was evicted, partly because it duplicated Ethernet drivers.

2.1.1 Linux Kernel Facilities

UIO

Linux v2.6.23 introduced the UIO interface, which has been developed especially with industrial I/O cards in mind [39]. Implementors of the API may call `uio_register_device`, which goes on to create a file in the virtual SysFS with a standardized interface to access the device from userspace. The implementor registers an in-kernel IRQ handler that acknowledges the interrupt and calls `uio_event_notify` which wakes up user code doing blocking reads on the device. A `mmap(2)` function is implemented as well for DMA and mapping of MMIO regions.

Interaction with a UIO device is limited to processes with superuser privileges. UIO has no notion of IOMMU protection and as such userspace drivers may configure DMA-capable devices to write anywhere in memory [59].

VFIO

Linux v3.6 introduced the VFIO mechanism as an alternative to UIO. VFIO exploits I/O Memory Management Units (IOMMUs) and PCI I/O virtualization (SR-IOV) to allow unprivileged and safe access to devices. This is especially useful for pass-through of devices to virtual machines. Unlike UIO, VFIO can be used completely from userspace. The standard `vfio-pci` driver may be bound to the NIC and can then be configured with an `eventfd(2)` to relay IRQs as file descriptor reads. Further UIO features like `mmap(2)` are supported as well.

2.1.2 Userspace Packet Processors

Even without copyright and virtualization considerations, bypassing the kernel can bring tangible advantages. For narrowly defined packet processing tasks, such as switching, kernel socket buffers with their support for multiple consumers and non-contiguous buffers are wasteful [31], especially so in setups where multiple 10 Gigabit Ethernet (GbE) NICs need to be serviced [11]. Bypassing the kernel's networking stack in this scenario can substantially reduce latency. This led to an emergence of a number of high speed user-mode packet I/O frameworks. While these frameworks are tuned for throughput, not determinism, it is likely that at the comparatively low throughput industrial Ethernet operates at, these frameworks could satisfy the real-time requirement.

Data Plane Development Kit (DPDK)

Of these frameworks, DPDK is probably the most extensive. An official Linux Foundation sponsored project, it provides user-mode drivers for 25 NIC chipsets from more than 14 manufacturers [1]. DPDK can use UIO or VFIO if available and implements the NIC drivers in userspace. Packet processors may then be built on top of it.

As industrial Ethernet usually operates at a link speed of 100 Mbit/s and the vast majority of devices service a single NIC (usually behind a two port switch or hub for bus topology use[2]), throughput, DPDK's main selling point is not a deciding factor. Use of DPDK as-is is also complicated by its scope: Support for Linux targets such as MIPS and PowerPC is not available and supported devices tend to be PCIe cards [1], while industrial Ethernet nodes are often connected over platform NICs with the MAC hosted directly on-chip.

2.2 In-Kernel Infrastructure

2.2.1 Bridging Support

Looking at the *edrv* API detailed in 1.2.1, one could redefine openPOWERLINK as a two-port bridge: Packets may enter from the first port (userspace) and the bridge is responsible for forwarding them, possibly queuing them until the right time. Similarly, frames ingressing from the network port (*edrv*) are inspected before deciding where to forward them.

With this in mind, a generic `edrv` could be implemented, controlling the network driver as one port by means of the bridging API and exporting the openPOWERLINK *edrv* API on the other. For this we will briefly review three bridge-like components within Linux.

eXpress Data Path (XDP)

At a link speed of 10Gbit/s, the kernel has only $67.2ns$ to deal with a minimally sized packet before another is delivered. Failure to do so will result in denial of service. With v4.8, Linux gained a new mechanism for fast classification of packets: XDP. NICs implementing XDP allow eBPF bytecode supplied by userspace to run directly in response to ingressing packets. These eBPF programs may drop, mangle or forward the packet, even before the allocation of SKBs and any interaction with the networking stack. Because of the time-criticality of these eBPF programs, they are not allowed to loop or call arbitrary C routines. While this might change, as of

[2] Technically a daisy chain topology.

Linux Kernel 4.14, only 8 NICs are supported, less than DPDK and none of which openPOWERLINK provides *edrv*s for. Widespread adaptation of the XDP API to the often lower speed platform NICs seems unlikely, disqualifying XDP at its current state for our use case.

Distributed Switch Architecture (DSA)

The Distributed Switch Architecture is a feature in Marvell Ethernet switching ASICs, which allows the switch and the management CPU to exchange information via specially tagged frames. The switch, which is also connected to the CPU over an Management Data Input/Output (MDIO) bus, is configured to parse a tag that holds forwarding or configuration instructions. The DSA subsystem within Linux allows treating the switch ports as if they were local ports: `ifconfig(1)` configuration is transparently converted to management frames. Frames transmitted from a switch-bound interface are automatically tagged to egress at the correct port. Ingressing frames have a DSA tag inserted and are forwarded to the management CPU.

The DSA subsystem transmits by changing the egressing SKB's `.dev` field and calling `dev_queue_xmit`. Reception is done by registering a protocol handler for the DSA Ethertype[3].

Bonding Driver

The Linux bonding driver provides a method for link aggregation and auto fail-over. Multiple network interfaces are "bonded" into a single interface, which userspace may use as any other, but internally offloads to its slaves. For example, a bonding driver configured for 802.3ad LACP mode would select a slave interface by hashing the packet and then instruct the slave to send it with `dev_queue_xmit`. Ingress frames are handled by registering a `rx_handler` for the slave interface that reassigns the SKB's `.dev` field to point at the bond. The ingress frame is then relooped into the network stack for normal delivery. Enslaving a device is possible at runtime and is done by registering a `rx_handler`, overriding hardware settings like MAC address and promiscuous mode to mirror those of the host, forwarding bond interface configuration like multicast settings to the slave and by branding it `IFF_SLAVE`, which suppresses some classes of multicast traffic.

2.2.2 In-Kernel servers

The socket API is also available kernel-side and can be used to implement network services, of which the TUX in-kernel web server [4] is probably the most

[3]The DSA Ethertype is configurable over MDIO as there is no IEEE-assigned Ethertype.

prominent example. Besides the performance advantage of reducing execution mode switches, being in-kernel allows for a great deal of flexibility, such as the use of zero-copy mechanisms and a tighter event loop integration. TUX does scatter-gather DMA directly from the page cache to the network and overrides the default TCP socket callbacks to directly insert incoming requests to the worker threads' input queue [43]. If the default socket callbacks are reused, the existing openPOWERLINK libpcap *edrv* can be readily converted to use the kernel API instead [25]. IgH's EtherCAT Master for Linux also features a kernel raw socket based generic driver [34].

2.2.3 NDIS

With the Windows NT kernel being closed source, it is necessary to export a well-defined API for device driver implementors. For network drivers, this is the NDIS API [47]. Of particular interest to us are the NDIS miniport and intermediate drivers. Miniport drivers are responsible for managing their NICs and intermediate drivers can be layered between the miniport driver and the rest of the networking stack. openPOWERLINK provides a generic NDIS intermediate driver, but Linux' ndiswrapper provides only support for integration of NDIS miniport drivers and intermediate drivers are not supported.

2.3 Possible Hook Points

In order to address the needs of the *edrv* API (cf. 1.2.1), a generic driver has to be able to transmit packets and to hook into the networking subsystem's Rx and Tx completion flow.

2.3.1 Receiving

Intercepting incoming packets is a task common to firewalls, bridges, protocol handlers and diagnostic tools. We will consider the different facilities the kernel offers to achieve this.

`rx_handler`

`rx_handlers` were first introduced with Linux v2.6.35 while refactoring `__netif_receive_skb`. Previously `__netif_receive_skb` had custom code to deal with virtual network devices, like bridges, bonding and macvlan. These virtual devices operate as software switches: They need to sit directly behind the networking driver and control how the packet is forwarded. The return type

of the handler indicates whether the packet should be dropped, looped once more through the receive path, delivered to an exact match or continue as usual.

The `rx_handler` is called in `NET_RX_SOFTIRQ` context, with interrupts enabled and holding the RCU lock.

Protocol Handlers (`dev_add_pack`)

Protocol handlers for new Ethertypes may be registered with `dev_add_pack`. Registering packet TAPs that receive all traffic is possible as well, which is how `PF_PACKET` raw sockets are implemented. Ingress packets are delivered to registered TAPs just before invoking a registered `rx_handler`. As of Linux v4.16, `PF_PACKET` seems to be the only remaining catch-all user of `dev_add_pack` [9]. Former users have been migrated to use `rx_handlers` and it has been suggested that the `rx_handler` route may be more efficient [15].

Netfilter

`net/netfilter/core.c` defines a static array of hooks, which are dispersed throughout the networking subsystem. At key points within the Internet Protocol (IP) [51] packet flow, networking code calls the `nf_hook` function, which checks whether the specified protocol family (e.g. `NFPROTO_IPv4`) and hook type (e.g. `NF_INET_POST_ROUTING`) has a hook registered. If so, it is called with the SKB as argument. The hook may inspect the SKB and decide whether the packet should be dropped or not. Netfilter hooks on the Rx path run within the `NET_RX_SOFTIRQ`. Netfilter is the mechanism underlying the Linux `iptables(1)` firewall configuration utility.

Netpoll

To be able to trace IRQ handlers, the Kernel GNU Debugger over Ethernet (KGDBoE) requires the ability to receive packets in the absence of interrupts. Netpoll used to contain a basic ARP and UDP packet reception functionality to support KGDBoE. As KGDBoE was never mainlined, the unused `netpoll_rx` code was finally removed in v3.15 [13].

Tracepoints

Tracepoints are markers within kernel code that can be used at runtime to alter behavior. In normal operation, these markers are no-ops (opcodes that do not alter system state). When instrumented, the no-ops are overwritten by a call to a kernel function; the original function is *hooked*.

2.3.2 Transmitting

Having identified the different methods for capturing incoming traffic, the available API for transmitting packets must also be reviewed.

`.ndo_start_xmit`

The Linux networking API equivalent of `edrv_sendTxBuffer`. May only be called with `dev->xmit_lock` held and while `netif_queue_stopped` doesn't return false. The network driver may call `netif_stop_queue` on link down or because transmission descriptors have been depleted. All network transmissions exit through this function. There is no requirement that `.ndo_start_xmit` is safe to call in an interrupt context.

`dev_queue_xmit`

Used in DSA, bonding, bridge and `PF_PACKET` modules, `dev_queue_xmit` is the default way to send packets in the Linux kernel. The documentation notes that `dev_queue_xmit` may only be called with interrupts enabled, as it disables bottom halves. This forbids calling it directly in the hrtimer callback[4]. If the queue is stopped or budget is exceeded, the packet is retried in the `NET_TX_SOFTIRQ`.

Netpoll

Netpoll is an API to implement minimal network clients in the kernel without involving the networking stack. This is useful in situations where the networking stack is unusable, like when the kernel crashes, is debugged or during an interrupt handler. Netpoll has its beginnings in v2.4, when netdump and netconsole first appeared [48]. Netdump is a kernel feature that allows sending core dumps over the network as UDP datagrams, as writing to persistent storage after a kernel panic may be destructive. Similarly, netconsole sends messages written by `printk`, which may also occur in hard interrupt contexts, over the network. Netpoll was merged in v2.6.5 to address this need. It features a simple ARP and UDP implementation, but also supports sending out arbitrary SKBs. Users interested just in `netpoll_send_skb` need only call `__netpoll_setup` on the target device, which sets up a transmission work queue. Afterwards packets may be sent in any context with `netpoll_send_skb`. `netpoll_send_skb` disables interrupts, determines a transmission queue, claims the `HARD_TX_LOCK` and directly invokes the driver's `.ndo_start_xmit` routine. If claiming the transmission lock fails

[4] Unless the PREEMPT RT patchset is applied, which runs hrtimer handlers in a softirq context.

or `netif_queue_stopped` returns true, it is retried every 50 µs and if implemented, the driver-provided `.ndo_poll_controller` routine is called to poll for reclaimable transmission descriptors. If a jiffy passes before transmission could be completed, it is added to the work queue for later delivery. Netpoll guarantees that packets are sent out in order.

Use of netpoll requires that `.ndo_start_xmit` is safe to call in an interrupt context. At the time of this writing (Linux v4.16), `drivers/net/ethernet` contained 261 definitions of `net_device_ops` objects, of which 135 had `.ndo_poll_controller` members. It is safe to assume that drivers that implement the optional `.ndo_poll_controller` have been vetted to allow `.ndo_start_xmit` to be called in hard IRQ context. This might not hold true for other drivers. The main complication with calling `.ndo_start_xmit` in interrupt context is error cases where `dev_kfree_skb` is called, as the SKB destructors may not be safe to run in an hard IRQ context. An easy workaround is calling `dev_kfree_skb_any` instead, which checks whether it runs in interrupt context and, if so, offloads the freeing of the SKB to happen during `NET_TX_SOFTIRQ`.

`PF_PACKET` with `PACKET_QDISC_BYPASS`

For high throughput traffic generation purposes, `PF_PACKET` sockets may be configured to bypass the qdisc mechanism entirely [14], in which case the packet is sent out via `packet_direct_xmit` in `net/packet/af_packet.c`. The control flow is largely similar to netpoll's, with the exception that drivers are not polled for empty descriptors and that failure to claim the `HARD_TX_LOCK` the first time results in dropping the packet.

2.3.3 Transmission Completion

As transmission completes asynchronously, we are dependent on Tx confirmations to reclaim Tx buffers and descriptors for reuse.

Manual confirmation

Registered Tx handlers are called directly after the driver's `.ndo_start_xmit` routine returns. This requires copying the SKB as transmission confirmation implies that reusing the transmission buffer is now possible. Manually confirming in the same context also requires the protocol stack to be reentrant, which the openPOWERLINK DLL on Linux is not. This can be solved by either running the Tx handler in a tasklet or by revising the locking scheme used in the DLL.

Timestamping

For high precision timestamping required for Precision Time Protocol (PTP) use, timestamp recording of egress packets has to be done as close to the medium as possible, preferably in hardware. If hardware support is not available, the driver may call `skb_tx_timestamp` directly before handing the packet to the MAC hardware. This function clones the SKB and attaches a timestamp. User code may then query the timestamp via `ioctl(2)` or request the kernel to forward all cloned packets to the socket's error queue. Receiving such a packet can be taken as a confirmation of transmission.

SKB destructor

For Denial of Service (DoS) protection, sockets are initialized with a maximum receive and send buffer size. When sending a packet, the socket's `.sk_wmem_alloc` field is incremented and if it exceeds the send buffer size, the system call blocks until space is made available. This happens when the NIC confirms transmission of the SKB's data, at which time the `dev_kfree_skb_any` is called. The function checks whether this is the last reference to the SKB (to avoid double crediting) and if so, the preset `.destructor` is called, which decrements the used memory.

If the destructor is called within hard IRQ context, the `NET_TX_SOFTIRQ` is raised and destruction is deferred.

Chapter 3

Design and Implementation

After reviewing the possible approaches outlined in the previous chapter, a prototype edrv-bridge driver has been implemented. Loading the driver still happens with `plkload`, but it has been slightly modified to not unload the existing driver. Hooking the receive path is done by registering a `rx_handler`, not much different than what bridging drivers do within the kernel. Three transmission paths have been implemented, netpoll, which is able to run with interrupts disabled, `dev_queue_xmit` which must enable interrupts and uses the qdisc mechanism and a fall-back solution that also runs with interrupts enabled, but skips the qdisc. The latter copies the `direct_xmit` path utilized by `PF_PACKET` sockets when configured with `PACKET_QDISC_BYPASS`. For memory management, standard Linux facilities like `kmalloc` & `build_skb` or `alloc_skb` are used. The SKB's destructor is used to notify the DLL of transmission completion. Following is a more detailed description of the implementation.

3.0.1 Claiming Interfaces

The standard `plkload` script works by unbinding the stock driver before loading the openPOWERLINK *edrv*, while a generic driver would reuse existing drivers. `plkload` was extended to skip unbinding drivers when loading a generic driver. Instead, the user may supply a module parameter to `plkload`, which is in turn passed to `insmod`. The edrv-bridge accepts the following load-time parameters:

`slave_interface`

Specify the slave interface to claim. As `edrv_init` runs in the context of the MN application initiating the `ioctl(2)`, it inherits its network namespace. The given interface string is searched for in that network namespace. Despite setting the `IFF_SLAVE` on the interface, multicast interference,

particularly IPv6, may still occur. Listing A.1 on page 51 details a shell script with which isolation may be realized.

use_qdisc

Whether the qdisc (through `dev_queue_xmit`) should be used. This defaults to false.

use_netpoll

Whether netpoll should be used. This defaults to true if the module was built with `CONFIG_NETPOLL` defined.

use_build_skb

Whether `build_skb` should be used instead of `alloc_skb` when wrapping Tx buffers in SKBs for transmission. This defaults to false, as we have experienced race conditions in our current implementation.

3.0.2 Rx

Frame interception is realized by registering a `rx_handler` for the `slave_interface`. The `rx_handler` runs in a softirq context. After calling the `.pfnRxHandler` function, the SKB is freed and `RX_HANDLER_CONSUMED` is returned to avoid further processing down the networking stack. Only one `rx_handler` may be registered by a device. This is deemed acceptable as currently, the other uses of `rx_handler` are not applicable to EPL. An added benefit is that local sniffing of ingress EPL traffic with `tcpdump(1)` or Wireshark is now possible. This might affect latency, however.

3.0.3 Memory Management

In absence of an IOMMU, special care might be required when allocating buffers for DMA, as NIC errata or bus limitations may restrict the address space usable for DMA with the device. As drivers can make their DMA preferences known by means of `dma_set_mask`, edrv-bridge could in theory allocate memory matching the device's DMA mask. Currently, "normal" memory is allocated via `kmalloc` with the flag `GFP_KERNEL`, which provides physically contiguous DMA-capable memory. Alternatively, `GFP_DMA` could be specified if the NIC is connected on a ISA[1] or a similarly restricted bus. The edrv-i210 and Linux' stock Ethernet drivers also use `GFP_KERNEL` allocation and streaming DMA for Tx buffers.

[1] ISA DMA has a 16 MB physical address limit.

An alternative to the use of `kmalloc` to allocate the SKB's data buffer is to `alloc_skb` including the data buffer as a whole upfront. In practice however, the SKB itself will not be in the cache anymore by the time the packet is actually transmitted, and allocating a fresh SKB from the (hot) slab cache would be less costly than incurring a cache miss. Additionally, the cyclic transmission's effect on the cache is reduced, as zeroing the SKB is no longer necessary [20]. The downside is a decrease in determinism, as the cache miss with its bounded latency is replaced by a possible, albeit improbable, memory allocation. Users may set `build_skb` to true if they want to override the use of `alloc_skb`.

3.0.4 Tx

As noted in 1.2.1, cyclic transmission occurs in the hrtimer's interrupt context. This is a hard IRQ context on normal Linux and a softirq context on systems with the PREEMPT RT patchset applied. Unless netpoll is used, if run within a hard IRQ context, either interrupts need to be reenabled for the duration of the transmission or the transmission needs to be offloaded to (preferably) the `HI_SOFTIRQ`[2], edrv-bridge's `edrv_sendTxBuffer` solves this by either using netpoll or enabling interrupts for the duration of the transmission. Regardless of the transmission mode in use, the buffer needs to be wrapped as SKB for further delivery. If `alloc_skb` was used, the `tEdrvTxBuffer` contains a pointer to the SKB, which is cloned. Otherwise, with `use_build_skb`, a new SKB is allocated on the fly to point at the data buffer. Afterwards, the SKB's destructor is set and the packet is then passed down to the configured delivery method.

3.0.5 Tx Confirmation

Tx confirmation occurs in the SKB's destructor, which is called when freeing the SKB. Afterwards data and the SKB itself are freed. Because we clone SKBs allocated by `alloc_skb`, the reference count of 2 ensures the packet's data is not freed unexpectedly. The same could be achieved manually by incrementing the reference counter while using `build_skb`. Also possible is setting the SKB's `.head` element to `NULL` which leads to an early-exit from the function. edrv-bridge uses the latter method.

3.0.6 Multicast Receive Filters

The Linux networking API provides the functionally equivalent `dev_mc_{add,del}` functions with which `edrv_{set,clear}RxMulticastMacAddr` can be

[2] This is the softirq used for tasklet_hi_schedule.

readily implemented.

Chapter 4

Evaluation

4.1 Testing Methodology

In real-time systems, correctness not only refers to the result being correct, but also to its timing behavior. This requires knowledge of the upper bound on latency. Ideally, this can be formally calculated, but with compiler optimizations, speculative execution, cache hierarchies and any full-fledged operating system's sheer size, this is prohibitively difficult. This leaves practical measurements of latencies in production-like environments as the only choice for evaluating real-time behavior. This is done by measuring the latency with which the system reacts to an external event and more importantly its variance. This is called *jitter*. For the evaluation of edrv-bridge we will consider two occurrences of jitter:

SoC-SoC Jitter

On reception of the timer interrupt, the MN sends out a SoC. The measured reaction times indicate the Tx portion's fitness for use as a MN or for TDMA in general.

PollReq-PollRes Jitter

On reception of a PollReq, the CN answers with a PollRes. The PollRes is precomputed when the SoC was received, so this tests the variance in both the Tx and Rx paths without involving much of the openPOWERLINK stack. The worst case latency measured here defines the shortest achievable cycle length.

4.1.1 Testing Environment

A key advantage of openPOWERLINK targeting Linux is the portability to the plethora of hardware supported by Linux. As such, it is important that the testing

reflects both ends of the spectrum: Industrial-grade automation PCs as well as consumer-grade Single Board Computers (SBCs). This is especially relevant in regard to the increasing use of the latter in industrial automation as alternative to specialized Programmable Logic Controllers (PLCs) (cf. Revolution Pi [42]).

`rpi`: Raspberry Pi 3 (smsc9512 NIC)

A popular SBC with a quad core ARM Cortex-A53 clocked at 1.2 GHz and with 1 GB of memory. Ethernet connectivity is over an external USB 2.0 Hub (smsc9512) with an integrated Ethernet PHY and MAC. As USB controllers, by design, don't interrupt the CPU, this adds a rather large latency to the system. USB 2.0 has a maximum polling frequency of 8000 Hz [17], which translates to a worst-case of $125\mu s$ of latency added by the USB hub alone. The system runs the v4.9.47 kernel fork provided by the Raspberry Pi Foundation with the PREEMPT RT patchset applied.

The bulk of the NIC management of the smsc9512 is done by the usbnet driver. The usbnet driver does not support netpoll, neither does it have a native *edrv* equivalent. For this reason, we will compare latencies of edrv-pcap and edrv-bridge with qdisc transmission mode.

`brpc`: B&R Automation PC 910 (82574L NIC)

A rugged PC for industrial automation. It features an 8-core Intel Core i7-3615QE 3.1 GHz CPU with 16 GB of RAM. The motherboard has an integrated Intel 82574L NIC. With B&R being the original force behind EPL, it should be safe to assume that the NIC's native openPOWERLINK *edrv* should give an accurate representation of the possible determinism. This system runs the newest stable kernel available, v4.14.20, with the PREEMPT RT patchset applied.

We compared the latencies of edrv-pcap, the native edrv-82573 and edrv-bridge with both netpoll and qdisc transmission mode.

Testing has been conducted with the `demo_mn_console` and `demo_cn_console` applications included in openPOWERLINK while `hackbench(1)` was running.

`hackbench(1)` is a scheduler stress test included in the rt-tests suite [45]. By default it forks to create 400 tasks that communicate with each other over pipes.

The cycle length was configured to $5ms$. Timestamps were recorded on a separate system connected to the same hub. To avoid inaccuracies introduced by software timestamping, an Intel I350 was configured to hardware timestamp ingress packets. These hardware timestamps have the precision of the MAC clock, $80ns$ at

a link speed of 100 Mbit/s [36]. The capturing software was `dumpcap(1)`, which is included with Wireshark v2.5.0[1].

[1] v2.5.0 is the first version to support querying Linux for hardware timestamps [23].

4.2 Time Measurements

4.2.1 Power Management Effects on Latency

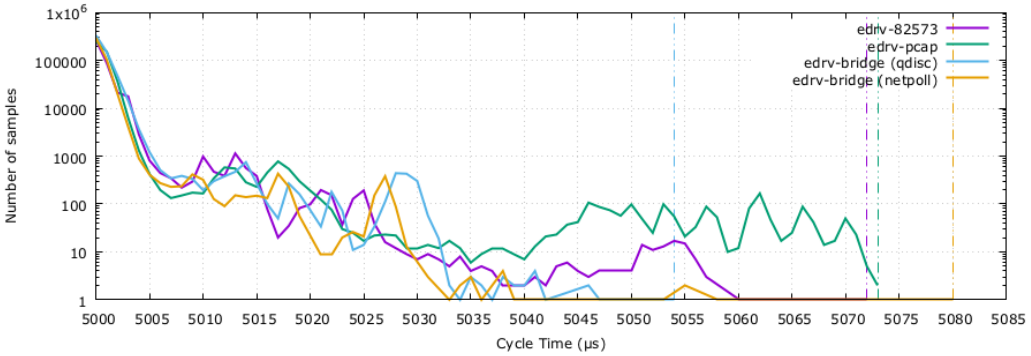

Figure 4.1: `brpc`: 5×10^5 cycles

Figure 4.1 shows that on average, the edrv-bridge (qdisc & netpoll) driver outperforms both the edrv-82573 and edrv-pcap driver. This decrease in average latency translates into a performance boost as less CPU cycles are expended for networking. However, for real-time use, the worst-case latency is of primary interest, where edrv-bridge (netpoll) is slightly worse than the existing solutions owing to the outlier at $5080\mu s$. Rerunning the same test, but with `hackbench(1)` in the background (4.2) shows a very interesting effect; Running a scheduler stress test actually *improves* the worst-case latency. This unexpected result helps explain the outliers in 4.1: Having to schedule 400 tasks would prevent the CPU from reaching deeper sleep states (C-states). While it is beneficial to turn off part of the CPU circuitry when idle, transferring out of a high C-state could increase DMA latency considerably [27].

To restrict high C-state transition, Linux offers a power management QoS interface. Applications may indicate to the kernel the maximum DMA latency they are willing to tolerate by writing to the `/dev/cpu_dma_latency` device file [6]. We conducted all further tests with an adjusted openPOWERLINK userspace library that writes a 0 to `/dev/cpu_dma_latency` indicating that no deep sleep states should be considered by the kernel. This increases the accuracy of shorter term measurements as we avoid contamination by power management jitter. However, this might not be a sensible value for general use in production [12] and should be decided on a case-by-case basis.

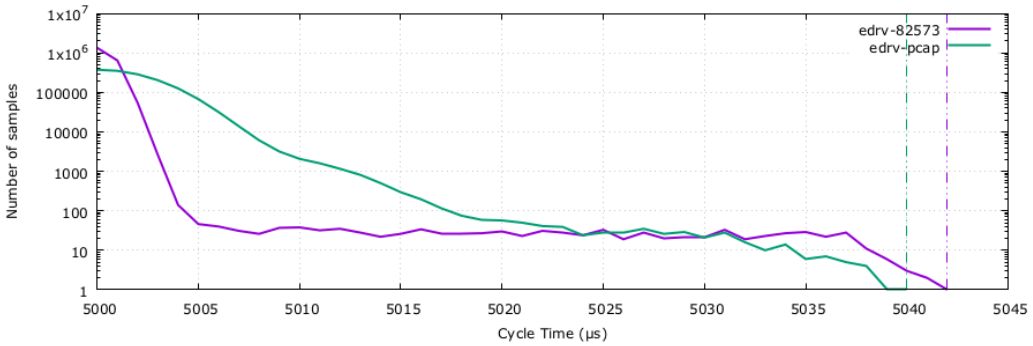

Figure 4.2: `brpc`: 5×10^6 cycles with hackbench

4.2.2 SoC-SoC Jitter

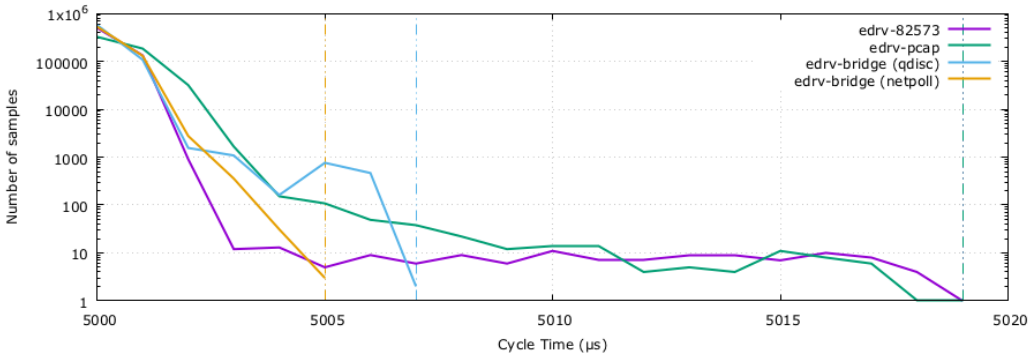

Figure 4.3: `brpc`: 7×10^5 cycles with `cpu_dma_latency=0`.

Disabling deep C-states smooths out the curve considerably in 4.3 and strongly suggests that the outliers found in 4.1 were indeed due to power management. On average, edrv-82573 has less overhead than edrv-pcap as expected, but both experience the same maximum jitter of $19\mu s$. This is still nearly a four-fold increase over the $5\mu s$ experienced by edrv-bridge (netpoll) over the same time span. edrv-bridge with netpoll is slightly better than with qdisc, which is understandable as the netpoll control flow is shorter (cf. 1.4). These results are not indicative of performance in production however, because we have no CNs attached to the bus and the MN is idle, except for `demo_mn_console`.

The results in 4.4 show that running `hackbench(1)` with its 400 tasks adds a $15\mu s$ to the worst-case latency, but otherwise doesn't affect the qualitative differences between the four *edrv*s.

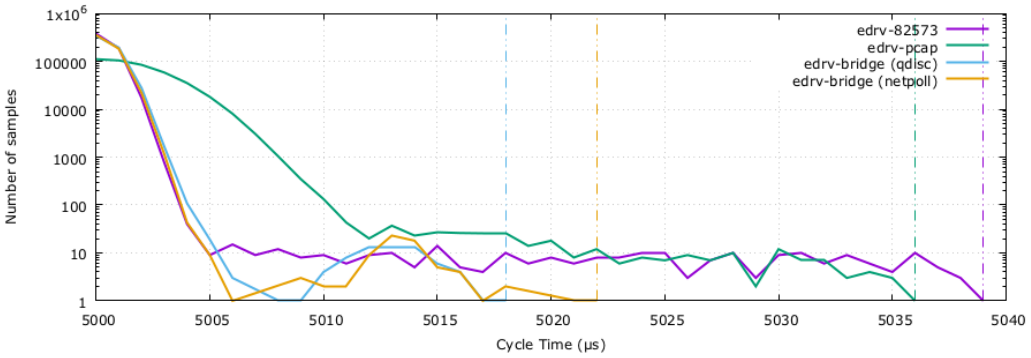

Figure 4.4: `brpc`: 7×10^5 cycles with `cpu_dma_latency=0` and hackbench

In 4.5, latency measurements on the idle `rpi` result in a maximum jitter difference of $100\mu s$ in favor of edrv-bridge (qdisc). With the scheduler stress test, the difference shrinks to just $5\mu s$. This might be nondeterminism introduced be the USB to Ethernet Bridge, because transmission has to be delayed until the USB Start of Frame (SoF) packet, which might take anywhere from 0 to $125\mu s$. We suspect that long term measurements would show a starker contrast.

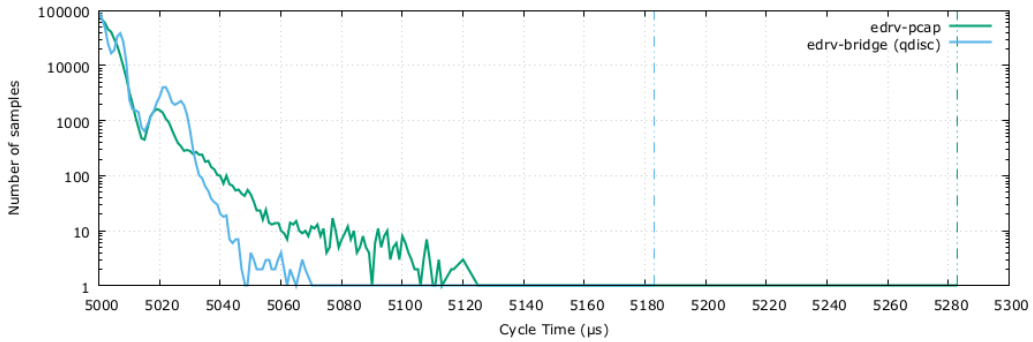

Figure 4.5: `rpi`: 7×10^5 cycles with `cpu_dma_latency=0`.

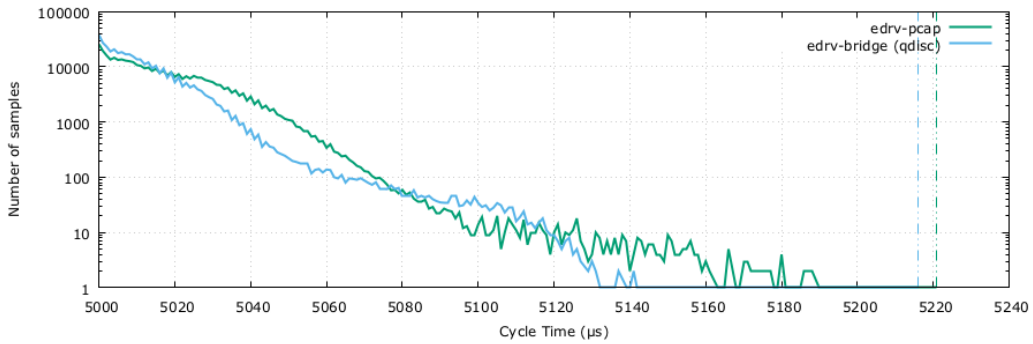

Figure 4.6: `rpi`: 7×10^5 cycles with `cpu_dma_latency=0` and hackbench.

4.2.3 PollReq-PollRes Jitter

For a more comprehensive evaluation, we connect a CN to an existing EPL network and measure each *edrv*'s reaction times to the MN's poll requests.

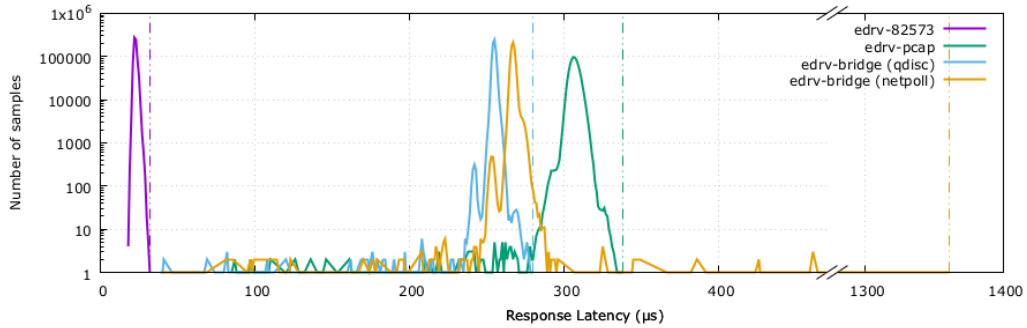

Figure 4.7: `brpc`: 7×10^5 cycles with `cpu_dma_latency=0` and hackbench.

Figure 4.7 shows that edrv-82573 strongly outperforms the other *edrv*s, as the custom driver is able to call the DLL Rx handler directly from hardirq context. edrv-bridge (qdisc) outperforms edrv-pcap by $58\mu s$ in worst-case latency, which seems to be the average latency incurred by the extra context switching and scheduling. Both can be avoided in edrv-bridge by calling into the DLL directly out of the softirq.

edrv-bridge (netpoll) suffers from a very large outlier at $1361\mu s$. Repeating the test yielded a similar result. We were unable to reproduce this effect with edrv-bridge (qdisc) however, which we ran for 7×10^6 cycles. We also ran edrv-bridge (netpoll) as a MN for 6×10^6 cycles and SoC-SoC cycle times confirmed our findings in 4.4 without any outliers. Applying a `SCHED_FIFO` real-time priority to the `ksoftirqd` did not affect the result. This could warrant further examination.

Figure 4.8 repeats the test for the `rpi`. The system as a whole is more prone to jitter and experiences higher latencies. Like with `brpc`, edrv-bridge outperforms edrv-pcap.

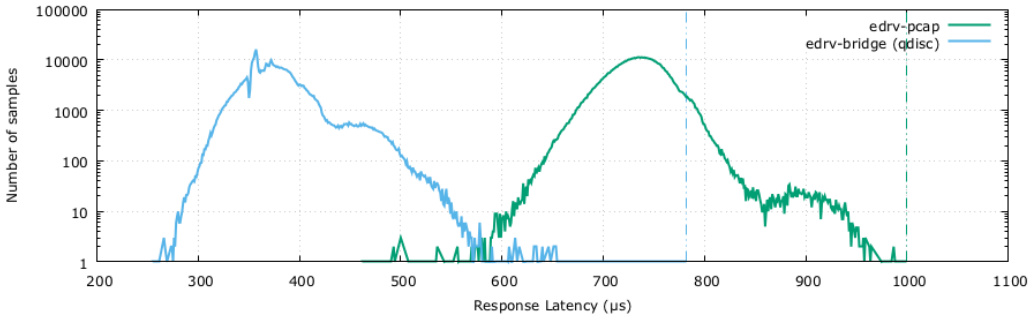

Figure 4.8: `rpi`: 7×10^5 cycles with `cpu_dma_latency=0` and hackbench.

Chapter 5

Conclusion and Outlook

5.1 Conclusion

The previously detailed evaluations show that edrv-bridge (qdisc) is a feasible alternative to development and maintenance of custom Ethernet drivers for openPOWERLINK without sacrificing performance or real-time behavior. Use of the underlying mechanisms is not restricted to EPL, but could be employed in implementing other industrial Ethernet nodes as well, as long they do not require custom hardware (e.g. EtherCAT masters or PROFINET RT). To ease adoption of edrv-bridge-like mechanisms for other real-time Ethernets, we implemented the free-standing micromanaging uman driver, which leverages the same mechanisms of edrv-bridge, but without interfacing with openPOWERLINK or requiring knowledge of its internals.

uman

uman is a simple two-port software switch. It exposes a DebugFS interface that can be configured at runtime with the interface to micromanage. Afterwards, the new `uman0` interface may be used to pass information to the uman driver, which it then passes to the micromanaged driver. Tx mode can be specified as module load parameter. Netpoll, qdisc and direct_xmit modes are supported. uman also hooks the micromanaged driver with an `rx_handler`, which injects received frames into the network stack as if they originated from the `uman0` interface. Users may strip uman of unneeded functionality and implement protocol-specific API on top. Having uman as self-contained example is also beneficial for testing purposes: Operations on sockets bound to uman and sockets bound to the underlying slave driver only differ in the Tx/Rx forwarding providing an easy and generic platform for profiling, optimizing and updating uman.

Driver Procurement

Both the edrv-bridge and uman driver are available under the GNU General Public License v2.0, at [8] and [24], respectively. We intend to submit edrv-bridge for upstream inclusion into openPOWERLINK as soon we are certain of long term stability in production use.

5.2 Support for Hardware Features

A major advantage of writing custom drivers is the possibility to use hardware features not exposed by stock drivers inside Linux. To completely substitute the existing *edrv*s, there needs to be a way for the edrv-bridge to support relevant hardware features.

Receive Address Filters

Unless put into promiscuous mode, a simple NIC would discard all ingress packets that don't match its unicast MAC address. As a middle ground, NICs feature hash tables that may be programmed with the hashes of additional, usually multicast, addresses. This conserves CPU time, as only frames, which were predetermined to be interesting, are forwarded. As an optimization, some NICs contain a small number of Receive Address Registers (RARs) that can be programmed with the full 6 bytes of the Ethernet address and can be promptly checked before hash table lookup. An example is the Intel 82573 that has 14 freely programmable RARs. While the edrv-82573 and edrv-i210 drivers support the RARs, support for the multicast hash table is not implemented, but the stock Linux driver has support for both, which may be readily leveraged by the edrv-bridge.

I210 Launch-times

The I210 is a PCIe network card by Intel, targeting the AVB and TSN markets. While the card doesn't support TSN's Time Aware Shaper (TAS), it provides a time triggered transmission mode, by which TAS can be emulated. In a true TSN-enabled NIC, the driver initializes the TAS by programming the NIC with the agreed-upon network time slices and afterwards the NIC is responsible for assuring that only traffic associated with the current time slice may egress. The I210 supports this mode of operation by allowing an optional *launch time* field in the Tx descriptor, with which the software can control when the packet is dequeued from the NIC-internal Tx queue. Launch times in the I210 have a granularity of

$32ns$ and may be specified up to half a second into the future. This makes it very suitable for use as the cylic transmission mechanism for the SoC.

Unfortunately, as of Linux v4.16, support for this feature has not been mainlined as existing RFCs were too tightly coupled with the I210. This was acceptable for openPOWERLINK, which had support for launch times since 2014. As the trend towards convergent networks and TSN is likely to continue, it is to be expected that this won't be the last consumer NIC to have special hardware support useful for real-time Ethernet. We will briefly discuss a possible approach for I210 launch time support:

`edrv_getMacTime`

The clock of the I210 is already exposed via the PTP subsystem, but there seems to exist no API to infer PTP devices associated with a NIC. As an alternative, userspace can be expected to supply the device file associated with the slave interface. edrv-bridge can use it to look up the associated `ptp_clock_info` object, which includes the `edrv_getMacTime`-equivalent `gettime64` routine.

`edrv_sendTxBuffer`

SKBs include a 48 byte long control buffer for storing private variables, which may be used freely as we pass the SKBs directly into the driver. The launch time can be stored to this control buffer. `.ndo_start_xmit` in the Intel IGB driver would then need to be modified to take the launch time into account when transmitting.

Shipping

The major complication is that changes to `.ndo_start_xmit` will need to be made available to a userbase that may run a wide variety of Linux and IGB versions and providing patches for all these versions may not be feasible. This would also be at odds with the concept of a generic driver, as upstream code is duplicated. Upstreaming complete support from device driver to userspace is probably not feasible either, because the user needs only device driver support and adjusting the whole networking subsystem as a side effect would not be economically viable. We speculate, however, that there is a middle ground: Device driver support can be upstreamed as a first step and the remainder may be left to future implementation. KGDBoE is an example of this: The previously discussed netpoll has had Rx support for 10 years without any in-kernel users [18] [13]. It is conceivable that a similar arrangement can be made for launch times and other hardware features.

Linux v4.15 saw introduction of AVB's credit based shaper as qdisc mechanism [29]. It is likely that TAS support along with hardware acceleration will

follow. As edrv-bridge can already be configured to use the qdisc mechanism, we expect porting to use future mainline launch time support to be straightforward.

5.3 Outlook

While the initial results are promising, i.e. deliver better performance than the only currently available portable solution, namely pcap-edrv, long-term tests are a must to prove fitness for use in production. An interesting avenue for further research could be investigation of netpoll's outliers and analysis whether its $50\mu s$ waiting time before retrying transmission is an acceptable default for a real-time system. Further improvements could be made by fixing the race condition preventing use of `build_skb` or by dropping manual namespace isolation as in A.1 in favor of restricting multicast interference directly in edrv-bridge.

Appendix A

Helper Scripts

```
#!/bin/bash

IF=enp4s0
NS=epln s

if [ $(id -u) -ne 0 ]; then
  echo "This script must be run as root. Try running with sudo." 1>&2
  exit 1
fi

ip link set dev $IF up
ip address flush dev $IF
ip -6 address flush dev $IF
ip link set $IF arp off

ip netns add $NS
ip link set $IF netns $NS

for what in all default; do
    sysctl -w net.ipv6.conf.$what.disable_ipv6=1
    sysctl -w net.ipv6.conf.$what.autoconf=0
    sysctl -w net.ipv6.conf.$what.accept_ra=0
    sysctl -w net.ipv6.conf.$what.forwarding=0
done

ip netns exec $NS sysctl -w net.ipv6.conf.$what.disable_ipv6=1
ip netns exec $NS sysctl -w net.ipv6.conf.$what.autoconf=0
ip netns exec $NS sysctl -w net.ipv6.conf.$what.accept_ra=0
ip netns exec $NS sysctl -w net.ipv6.conf.$what.forwarding=0

ip netns exec $NS ip link set dev $IF up
ip netns exec $NS ip address flush dev $IF
ip netns exec $NS ip -6 address flush dev $IF
ip netns exec $NS ip link set $IF arp off
```

Listing A.1: Network Namespace Isolation

Bibliography

[1] DPDK: Supported NICs. http://dpdk.org/doc/nics.

[2] Universal TUN/TAP device driver. https://www.kernel.org/doc/Documentation/networking/tuntap.txt.

[3] IEEE Standards for Local and Metropolitan Area Networks: Supplements to Carrier Sense Multiple Access With Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications - Specification for 802.3 Full Duplex Operation and Physical Layer Specification for 100 Mb/s Operation on Two Pairs of Category 3 Or Better Balanced Twisted Pair Cable (100BASE-T2). *IEEE Std 802.3x-1997 and IEEE Std 802.3y-1997 (Supplement to ISO/IEC 8802-3: 1996; ANSI/IEEE Std 802.3, 1996 Edition)*, pages 1–324, , 1997.

[4] TUX 2.0 Reference Manual. http://www.stllinux.org/meeting_notes/2001/0719/tux/index.html, 2001.

[5] IEEE Standard for Information technology–Telecommunications and information exchange between systems–Local and metropolitan area networks–Specific requirements Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications. *IEEE Std 802.3-2008 (Revision of IEEE Std 802.3-2005)*, pages 1–2977, Dec 2008.

[6] Are hardware power management features causing latency spikes in my application? https://access.redhat.com/articles/65410, 2013.

[7] Release 2 of the openPOWERLINK protocol stack. https://github.com/OpenAutomationTechnologies/openPOWERLINK_V2, 2017.

[8] a3f/openpowerlink_v2, March 2018. https://doi.org/10.5281/zenodo.1195775.

[9] dev_add_pack - Elixir - Bootlin, 2018. https://elixir.bootlin.com/linux/v4.15.7/ident/dev_add_pack.

[10] Abbott, Doug. *Linux for Embedded and Real-time Applications*. Elsevier, 2nd edition, 2011.

[11] Barbette, Tom and Soldani, Cyril and Mathy, Laurent. Fast userspace packet processing. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for networking and communications systems*, pages 5–16. IEEE Computer Society, 2015.

[12] Benson, T. M. A system-level optimization framework for high-performance networking. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, Sept 2014.

[13] Biederman, Eric W. netpoll: Remove dead packet receive code (CONFIG_NETPOLL_TRAP), 2014. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=9c62a68d13119a1ca9718381d97b0cb415ff4e9d.

[14] Borkmann, Daniel. [net-next,3/3] packet: introduce PACKET_QDISC_BYPASS socket option. Linux Netdev Mailing List. http://lists.openwall.net/netdev/2013/12/06/175.

[15] Brodin, Arvid. Re: bridge: HSR support. Linux Netdev Mailing List. https://www.spinics.net/lists/netdev/msg182746.html.

[16] Canonical. Ubuntu - Package Search Results – linux-lowlatency. https://packages.ubuntu.com/search?keywords=linux-lowlatency, 2018.

[17] Compaq and Hewlett-Packard and Intel and Lucent and Microsoft and NEC and Philips. Universal Serial Bus Specification Revision 2.0, 2000.

[18] Corbet, Jonathan. Netpoll is merged. *LWN.net*, March 2004. https://lwn.net/Articles/75944/.

[19] Corbet, Jonathan and Rubini, Alessandro and Kroah-Hartman, Greg. *Linux device drivers : [where the kernel meets the hardware]*. OReilly, Beijing, 3. ed. edition, 2005.

[20] Dumazet, Eric. net: introduce build_skb(), 2011. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=b2b5ce9d1ccf1c45f8ac68e5d901112ab76ba199.

[21] Emde, Carsten. Open Source in Industrial Automation - already a Reality. ROS-Industrial Conference, 2016.

[22] Ethernet POWERLINK Standardization Group. EPSG Draft Standard 301, Ethernet POWERLINK Communication Profile Specification Version 1.3.0. 2016.

[23] Fatoum, Ahmad. Add hardware timestamping support, 2017. https://code.wireshark.org/review/gitweb?p=wireshark.git;a=commit;h=aca55a29f7b982e7a0bd9911d1d176561c8d7a84.

[24] Fatoum, Ahmad. a3f/uman, March 2018. https://doi.org/10.5281/zenodo.1195773.

[25] Fatoum, Ahmad. edrv-kernelrawsock_linux.c. https://github.com/a3f/openPOWERLINK_V2/blob/master/stack/src/kernel/edrv/edrv-kernelrawsock_linux.c, 2018.

[26] Feld, Joachim. PROFINET - scalable factory communication for all applications. In *Factory Communication Systems, 2004. Proceedings. 2004 IEEE International Workshop on*, pages 33–38. IEEE, 2004.

[27] Flajslik, Mario and Rosenblum, Mendel. Network Interface Design for Low Latency Request-Response Protocols. In *USENIX Annual Technical Conference*, pages 333–346, 2013.

[28] Garcia, Gerardo. Ethernet in Real time, Jan 2006. http://www.machinedesign.com/archive/ethernet-real-time.

[29] Gomes, Vinicius Costa. [Intel-wired-lan] [next-queue PATCH v9 6/6] igb: Add support for CBS offload. Intel Wired LAN Mailing List. https://lists.osuosl.org/pipermail/intel-wired-lan/Week-of-Mon-20171016/010456.html.

[30] Hanssen, Ferdy and Jansen, Pierre G. *Real-time Communication Protocols: An Overview*. Centre for Telematics and Information Technology, University of Twente, 2003.

[31] Honda, Michio. Offloading to yet-another software switch. In *Proceedings of netdev 0.1, Feb 14-17*, 2015. https://people.netfilter.org/pablo/netdev0.1/papers/Offloading-to-yet-another-software-switch.pdf.

[32] IEEE, http://www.ieee802.org/1/pages/802.1Q.html. *IEEE 802.1Q: VLAN*, 2005.

[33] IEEE Portable Applications Standards Committee and others. IEEE Std 1003.1 b-1993, Real Time Extensions, 1993.

[34] Ingenieurgemeinschaft IgH. EtherLab - EtherCAT Master - Supported Hardware. http://www.etherlab.org/en/ethercat/hardware.php, 2013.

[35] Ingenieurgemeinschaft IgH. IgH EtherCAT Master for Linux. http://www.etherlab.org/en/ethercat/, 2013.

[36] Intel. *Intel Ethernet Controller I350 Datasheet*. https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/ethernet-controller-i350-datasheet.pdf, 2017.

[37] Jansen, Dirk and Buttner, Holger. Real-time Ethernet: the EtherCAT solution. *Computing and Control Engineering*, 15(1):16–21, 2004.

[38] Keh, Thomas. Ethernet-basierte Echtzeitkommunikation auf einem Mikrocontroller-RTOS. Master's thesis, Karlsruhe Institute of Technology , Germany, 2017.

[39] Koch, Hans J and Linutronix GmbH. Userspace I/O drivers in a realtime context. In *The 13th Realtime Linux Workshop*, 2011.

[40] Kroah-Hartman, Greg. The Linux Kernel Driver Interface, Dec 2004. https://www.kernel.org/doc/Documentation/process/stable-api-nonsense.rst.

[41] Kroah-Hartman, Greg. Staging tree status for the .32 kernel merge. Sep 2009. https://lwn.net/Articles/350590/.

[42] KUNBUS GmbH. Revolution Pi - Industrial PC based on Raspberry Pi. https://revolution.kunbus.com.

[43] Lever, Chuck and Eriksen, Marius Aamodt and Molloy, Stephen P. An analysis of the TUX web server. Technical report, Center for Information Technology Integration, 2000.

[44] Linux Foundation. Preemption Models. https://wiki.linuxfoundation.org/realtime/documentation/technical_basics/preemption_models, 2018.

[45] Linux Foundation. RT-Tests. https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/rt-tests, 2018.

[46] Lipp, Moritz and Schwarz, Michael and Gruss, Daniel and Prescher, Thomas and Haas, Werner and Mangard, Stefan and Kocher, Paul and Genkin, Daniel and Yarom, Yuval and Hamburg, Mike. Meltdown. *ArXiv e-prints*, January 2018.

[47] Microsoft Hardware Dev Center. *NDIS Intermediate Drivers*. https://msdn.microsoft.com/en-us/windows/hardware/drivers/network/.

[48] Moyer, Jeff. Programming with the Netpoll API. Linux Kongress, 2005.

[49] Open EtherCAT Society. Open EtherCAT Society. https://github.com/OpenEtherCATsociety, 2017.

[50] PROFIBUS & PROFINET International, https://www.profibus.com/index.php?eID=dumpFile&t=f&f=51694&token=e5507607a61a4e34b9493731abdc8a1c4f43c788. *PROFINET-Feldgeräte*, 2016.

[51] Rosen, Rami. *Linux Kernel Networking : Implementation and Theory*. SpringerLink : Bücher. Apress, Berkeley, CA, 2014.

[52] Schemm, E. SERCOS to link with Ethernet for its third generation. *Computing and Control Engineering*, 15:30–33(3), April 2004. http://digital-library.theiet.org/content/journals/10.1049/cce_20040205.

[53] Schiffler, Andreas. Sercos Softmaster for MachineKit. https://github.com/aschiffler/linuxcnc-sercos3, 2017.

[54] Siemens AG. CP 1626, Product Details - Industry Mall - Siemens WW, 2018. https://mall.industry.siemens.com/mall/en/WW/Catalog/Product/6GK1162-6AA01.

[55] Sivrikaya, Fikret and Yener, Bülent. Time synchronization in sensor networks: a survey. 18(4):45–50, 2004.

[56] Torvalds, Linus. GPL 2.0 with Linux Syscall note, 2000. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/COPYING.

[57] Torvalds, Linus. Linux kernel release 4.x <http://kernel.org/> README, 2016. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/admin-guide/README.rst.

[58] Wallner, Wolfgang and Brucknerr, Dietmar and Baumgartner, Josef. openPOWERLINK 2.0: A split kernel/user space implementation of the real-time Ethernet protocol POWERLINK, 2013.

[59] Williamson, Alex. VFIO - "Virtual Function I/O", Jul 2012. https://www.kernel.org/doc/Documentation/vfio.txt.

[60] Zurawski, Richard. *Industrial Communication Technology Handbook*. CRC Press, 2nd edition, 2015.