Karlsruher Institut für Technologie

Masterthesis

Ethernet-basierte Echtzeitkommunikation auf einem Mikrocontroller-RTOS

Entwurf und Implementierung am Beispiel der modularen Realisierung eines 3D-Druckers mit Ethernet POWERLINK und NuttX

Thomas Keh

09.03.2017

Karlsruher Institut für Technologie
Fakultät für Informatik
Institut für Anthropomatik und Robotik (IAR) -
Intelligente Prozessautomation und Robotik (IPR)

Erstgutachter: Prof. Dr.-Ing. habil. Björn Hein
Zweitgutachter: Prof. Dr.-Ing. habil. Thomas Längle
Betreuer: Dr.-Ing. Andreas Bihlmaier

Karlsruher Institut für Technologie
Fakultät für Informatik
76128 Karlsruhe

robodev GmbH
Engler-Bunte-Ring 8
76131 Karlsruhe

Diese Arbeit wurde in Kooperation mit der robodev GmbH angefertigt.

Zusammenfassung

Die Verwendung von Echtzeit-fähigem Ethernet als Feldbussystem in Industrieanlagen ist eine zukunftsweisende Technologie und stellt einen branchenübergreifenden Trend unter den Stichworten *Industrial Ethernet* (IE) und *Industrie 4.0* dar. Auch in Fahrzeugen wird Ethernet eine wachsende Bedeutung zuteil. In dieser Arbeit wird beleuchtet, wie man mit günstiger, generischer und leistungsschwacher Hardware sowie quelloffener Software Ethernet-basierte Echtzeitkommunikation durchführen kann, ohne auf dedizierte Chips und proprietäre Lizenzen angewiesen zu sein. Dabei werden die Möglichkeiten eines Echtzeitbetriebssystems für eingebettete Systeme (RTOS) ausgeschöpft, um eine einfache Interoperation zwischen verschiedenen Anwendungen zu ermöglichen. Besonders die Möglichkeit der Integration von bestehender Netzwerksoftware in ein eingebettetes System, wie etwa einem Webserver zur Statusüberwachung, macht diese Lösung attraktiv. Die Arbeit liefert zunächst eine Argumentationshilfe zur Auswahl einer adäquaten und zukunftssicheren IE-Technologie sowie eines geeigneten RTOS. Darauf aufbauend wird eine konkrete Implementierung des *Ethernet POWERLINK*-Protokolls (EPL) gezeigt, die basierend auf dem RTOS *NuttX* und der *openPOWERLINK*-Softwarebibliothek auf einem *ARM Cortex-M4*-Mikrocontroller mit nur 136kB RAM lauffähig ist und Antwortzeiten von deutlich unter einer Millisekunde aufweist. Damit reicht die Leistungsfähigkeit des nur wenige Euro teuren Mikrocontrollers aus, um selbst anspruchsvolle Bewegungssteuerungen mit EPL zu realisieren. Einen Beweis dafür liefert der zusätzlich im Rahmen dieser Arbeit entwickelte 3D-Drucker, der auf vier gleichartigen Modulen basiert, die ihrerseits die oben genannte Kombination aus Hardware, Betriebssystem und Anwendungssoftware verkörpern. Die Module treiben jeweils eine der Antriebsachsen an und werden über ein EPL-Netzwerk mit einer Zykluszeit von acht Millisekunden von einem herkömmlichen Arbeitsplatzrechner gesteuert. Der 3D-Drucker ist ein anspruchsvoller Anwendungsfall zur Evaluierung der Echtzeiteigenschaften der entwickelten Plattform, denn durch seinen Aufbau als sogenannter Linear-Delta-Roboter können schon leichte Asynchronitäten Schaden an Druckobjekt und Maschine verursachen. Die Auswertung zeigt jedoch eine ansprechende Druckqualität, die konkurrenzfähig ist zu der eines nichtmodularen Druckers. Nebenbei zeigt die Entwicklung des 3D-Druckers exemplarisch, wie eine Maschinensteuerungssoftware, die für unmittelbar angeschlossene Hardware entwickelt wurde, zur Verwendung mit einem latenzbehafteten Feldbussystem angepasst werden kann. Er ist damit auch separat betrachtet ein zentraler Beitrag dieser Arbeit und liefert Erkenntnisse, die auch für andere Anwendungsfälle und Technologien Relevanz besitzen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AHB	Advanced High-performance Bus
ARP	Address Resolution Protocol
ASIC	Application-specific Integrated Circuit
ASnd	Asynchronous Send (EPL)
B210E	Micromint Bambino 210E
CAL	Communication Abstraction Layer
CiA	CAN in Automation
CIP	Common Industrial Protocol
CN	Controlled Node
CSMA/CD	Carrier Sense Multiple Access with Collision Detection
DLL	Data Link Layer (openPOWERLINK)
EIP	EtherNet/IP
EMC	External Memory Controller [61, S. 594ff]
emcmot	Enhanced Machine Controller: Motion (MK)
EPL	Ethernet POWERLINK
EPSG	Ethernet POWERLINK Standardization Group
ESC	EtherCAT Slave Controller
FE	Following Error
FPGA	Field Programmable Gate Array
FSN	Fully Switched Network
GPIO	General Purpose Input/Output
HAL	Hardware Abstraction Layer
IDE	Integrated Development Environment
IE	Industrial Ethernet
IOCTL	Input/Output Control
IoT	Internet of Things
IPC	Inter-process Communication
KL	Kernel Layer (openPOWERLINK)
MAC	Media Access Control
MK	Machinekit
MMU	Memory Management Unit
MN	Managing Node
MOSFET	Metall-Oxid-Halbleiter-Feldeffekttransistor
MTU	Maximum Transmission Unit
µC	Mikrocontroller
NMT	Network Management (EPL)
OC	openCONFIGURATOR
OPLK	openPOWERLINK
OSI	Open Systems Interconnection Model
OV	Objektverzeichnis
PDO	Process Data Object (EPL)

PReq	Poll Request (EPL)
PRes	Poll Response (EPL)
PWM	Pulsweitenmodulation
RC	Reduced Cycle
RPDO	Receive Process Data Object
RTOS	Real-time Operating System
SDO	Service Data Object (EPL)
SDRAM	Synchronous Dynamic Random Access Memory
SHM	Shared Memory
SLOC	Source Lines of Code [14]
SoC	System on Chip oder Start of Cycle (EPL)
SPI	Serial Peripheral Interface
SPIFI	SPI Flash Interface
SRAM	Static Random-Access Memory
TPDO	Transmit Process Data Object (EPL)
TUN/TAP	Network Tunnel / Network Tap
UL	User Layer (openPOWERLINK)
UVP	unverbindlicher Verkaufspreis
VETH	Virtual Ethernet
VLAN	Virtual Local Area Network
XDD	XML Device Description

1. Motivation

Ethernet ist ein weit verbreiteter Standard, der ein Datenübertragungsprotokoll auf der Netzzugangsebene sowie die zugehörige Hardware definiert [21]. Ethernet wird vorwiegend für lokale Netzverbindungen (*Local Area Network*, LAN) verwendet. Vorteile von Ethernet sind neben hohen verfügbaren Datenraten vor allem die der großen Verbreitung geschuldete umfassende Unterstützung durch Betriebssysteme und kostengünstig verfügbare Hardware.

Es gibt Bestrebungen, diese Vorzüge auch im industriellen Umfeld nutzbar zu machen und Ethernet als Bussystem unter anderem in Fertigungsanlagen einzusetzen. Damit tritt es in Konkurrenz zu bereits etablierten Bussystemen wie *CAN* [87, S.278], *PROFIBUS* [87, S.274], *INTERBUS* [87, S.290] und vielen weiteren. Die daraus entstandenen Technologien werden oft unter den Begriffen *Industrial Ethernet* (IE) oder *Echtzeit-Ethernet* zusammengefasst. Als ein zentraler Teil der fortschreitenden Automatisierung, Digitalisierung und Vernetzung von Produktionsanlagen werden Anstrengungen dieser Art auch mit dem Begriff *Industrie 4.0* charakterisiert, der ursprünglich auf ein gleichnamiges Projekt des *Bundesministeriums für Bildung und Forschung* (BMBF) zurückgeht [7]. Auch in Fahrzeugen erfreut sich Ethernet aus den genannten Gründen wachsender Beliebtheit[40].

Die hohen Datenraten von Ethernet stellen jedoch besondere Ansprüche an die beteiligten Geräte. Bei *Fast Ethernet* [20] müssen theoretisch bis zu 12,5 MB/s an Rohdaten verarbeitet werden (vgl. dazu max. 125kB/s bei CAN). Viele Hersteller von eingebetteten Geräten mit Ethernet-basierter Echtzeitkommunikation setzen deswegen auf *ASIC*s (*application-specific integrated circuit*), bzw. *FPGA*s (*field-programmable gate array*), die die Protokollverarbeitung übernehmen, ohne den Prozessor zu belasten. Der Prozessor kümmert sich dann ausschließlich um die für ihn bestimmten Nutzdaten.

Dies setzt spezielle Hardware voraus und verursacht nicht nur höhere Kosten, sondern erschwert auch den Einstieg in die Technologie im universitären und im privaten Umfeld, wo mehrfach verwendbare Hardwareplattformen gerne bevorzugt werden und wo oft auch die Mitarbeit an bestehenden Systemen im Vordergrund steht. Die Anschaffung spezieller Hardware kommt bei gegebenen Alternativen daher meist nicht in Frage. Echtzeit-Ethernet ist deswegen im Low-Cost-Bereich noch wenig verbreitet.

Aus diesem Grund soll in dieser Arbeit untersucht werden, ob und in welcher Qualität Ethernet-basierte Echtzeitkommunikation auf leistungsschwacher und unspezialisierter Hardware möglich ist. Konkret werden Mikrocontroller betrachtet, die auf dem ARM Cortex-M4 Prozessor [55] basieren und somit keine Speicherverwaltungseinheit (engl. *Memory Management Unit*, MMU) besitzen. Solche Mikrocontroller sind mit umfangreicher Schnittstellenausstattung bereits für unter 8€ erhältlich[1].

Es existieren bereits Implementierungen von Echtzeit-Ethernet-Protokollen, die auf gewöhnlicher PC-Hardware und Linux bzw. Windows lauffähig sind, sowie Implementierungen, die ohne Betriebssystem (*bare-metal*) direkt auf einem Mikrocontroller aufsetzen. Ersteres kommt für kostensensitive und energieeffiziente Anwendungen im Embedded-

[1] z.B. NXP LPC4330, Stand: Februar 2017

Bereich nicht in Frage. Letzteres erschwert die Integration von komplexen Anwendungen, da die gleichzeitige Ausführung von verschiedenen Submodulen ohne Betriebssystem umständlich, wenn nicht unmöglich ist. Auch die in vielen Echtzeit-Ethernet-Protokollen vorgesehene Verwendung eines TCP/IP-Seitenkanals, z.B. für den Betrieb eines Webservers zur Statusüberwachung, ist damit schwer umsetzbar. Der Code ist durch fehlende Abstraktionsschichten außerdem nicht portabel.

Der Beitrag dieser Arbeit soll deswegen eine portable Umsetzung auf einem geeigneten Echtzeitbetriebssystem für eingebettete Systeme (*Real Time Operating System*, RTOS) sein. Konkret wird dabei nach einer kurzen Abwägung der möglichen Alternativen die Portierung eines *Ethernet POWERLINK*-Stacks auf das RTOS *NuttX* gezeigt, zusammen mit den Anpassungen und Vorkehrungen, die nötig sind, um praxistaugliche Zykluszeiten zu erreichen. Die Praxistauglichkeit wird anschließend anhand der Realisierung eines modularen 3D-Druckers evaluiert. Bei diesem erfolgt Steuerung über vier unabhängige Module, die ausschließlich über Ethernet POWERLINK (EPL) mit einem handelsüblichen PC als Steuerrechner kommunizieren. Die Aufgabe ist anspruchsvoll, da die Motoren absolut synchron angesteuert werden müssen, um Beschädigungen zu vermeiden und um ansprechende Druckergebnisse zu erhalten.

Die Arbeit ist wie folgt strukturiert. Kapitel zwei widmet sich dem Thema Ethernetbasierte Echtzeitkommunikation im Allgemeinen. Verschiedene Verfahren werden miteinander verglichen und besonders daraufhin untersucht, ob mit ihnen die Ziele dieser Arbeit erreichbar sind. Wichtig ist dabei vor allem die Verfügbarkeit von kostengünstiger Hardware und freier Software. Die Wahl fällt schließlich auf Ethernet POWERLINK, dessen Eigenschaften im Detail erläutert werden. Es folgt ein Kapitel zu frei verfügbaren Echtzeitbetriebssystemen für eingebettete Systeme. Mehrere Vertreter werden anhand der Anforderungen für den Einsatz als sogenannter *EPL Controlled Node* untersucht. Auf NuttX, das im Vergleich als Sieger hervorgeht, wird näher eingegangen. Besonderes Augenmerk liegt dabei auf dem Netzwerksubsystem. Kapitel vier stellt die Hardwareplattformen vor, die zur Realisierung eines EPL Controlled Nodes verwendet werden. Das fünfte Kapitel präsentiert schließlich die Realisierung von EPL-basierten Controlled Nodes mit NuttX auf der zuvor vorgestellten Hardware. Ein großer Teil widmet sich dabei openPOWERLINK, einer quelloffenen EPL-Implementierung, die für diesen Zweck auf NuttX portiert wird. Eine ausführliche Evaluierung der Leistungsfähigkeit der hier entwickelten Controlled Nodes rundet das Kapitel ab. Auch wenn der Fokus dieser Arbeit auf der Entwicklung von Controlled Nodes liegt, ist der Managing Node (MN) doch ein essenzieller Bestandteil eines EPL-Netzwerks und seine Konfiguration ist nicht trivial. Trotz wesentlich höherer Rechenleistung kann ein MN die Leistungsfähigkeit des Gesamtnetzes begrenzen. Kapitel sechs widmet deshalb diesem Thema im Detail. Kapitel sieben stellt als zweiten großen Beitrag dieser Arbeit den oben genannten 3D-Drucker vor. Von Hardware und Aufbau über die Verwendung und Entwicklung verschiedener Softwarekomponenten, die Konfiguration, die Lösung von auftretenden Problemen bis zur Evaluierung des Gesamtprodukts wird der gesamte Entwicklungsprozess eines derartigen Geräts dargelegt. Schließlich fasst Kapitel acht die Ergebnisse der Arbeit kurz zusammen und gibt Anregungen für zukünftige Entwicklungen.

6 Bytes	6 Bytes	0/4 Bytes	2 Bytes	46/42-1500 Bytes	4 Bytes
Ziel-Adresse	Quell-Adresse	VLAN-Tag (optional)	Ethertype/ Länge	Daten	CRC

Gesamtlänge: 64-1518/1522 Bytes

Abbildung 1: Schema eines Ethernet-Frames. Das CRC-Feld (*cyclic redundancy check*, Prüfwert) wird üblicherweise von der Ethernet-Hardware angefügt.

2. Ethernet-basierte Echtzeitkommunikation

In industriellen Anwendungen werden meist strenge Echtzeitanforderungen gestellt. Damit sind im speziellen zwei Eigenschaften gemeint: Sowohl die *Latenzzeit*, als auch der *Jitter*, also die Varianz der Latenzzeit, müssen nach oben beschränkt sein [87, S.301]. Bei Missachtung der Grenzen sind im schlimmsten Fall Schaden an Mensch und Maschine möglich. Anwendungen, in denen die Grenzen überschritten werden dürfen, jedoch nur hinreichend selten und/oder innerhalb eines bestimmten Rahmens, stellen sogenannte *weiche Echtzeitbedingungen* [87, S.321] an das verwendete Bussystem.

Dieses Kapitel legt dar, warum Ethernet standardmäßig keine harten Echtzeitbedingungen erfüllt. Nach einer kurzen Einführung in die Grundlagen von Ethernet werden einige Grundkonzepte erörtert, die dazu fähig sind, Ethernet um Echtzeitfähigkeiten zu erweitern. Anschließend werden die verbreitetsten Protokolle und Lösungen für Echtzeit- und Industrie-Ethernet vorgestellt.

2.1. Ethernet

Ethernet ist als *IEEE 802.3* international standardisiert [21]. Es deckt im OSI-Schichtenmodell [89] die Bitübertragungs- und die Sicherungsschicht ab. Ethernet existiert vor allem auf physikalischer Ebene in vielen verschiedenen Varianten. Aktuell verbreitet sind vor allem 100Mbit-Ethernet (*100BASE-TX*, standardisiert in *IEEE 802.3u*) sowie Gigabit-Ethernet (*1000BASE-T*, *IEEE 802.3ab*) über Kabel aus verdrillten Kupferdrähten (*Twisted Pair*).

Die Datenübertragung ist paketorientiert. Ein sog. Frame ist mindestens 60 Bytes lang und ist wie in Abb. 1 dargestellt aufgebaut. Auf Bitübertragungsschicht kommen hierzu noch eine Präambel, die *Start-of-Frame*-Sequenz und eine definierte Pause bis zur Übertragung des nächsten Pakets (*Interpacket gap*, entspricht zwölf Bytes). Das optionale VLAN-Tag kann für virtuelle lokale Netze (sog. VLANs) verwendet werden und ist in *IEEE 802.1Q* standardisiert.

In einem Ethernet-Netzwerk besitzt jeder Teilnehmer eine eindeutige 48-Bit-Kennung, auch MAC-Adresse genannt (*Media Access Control* [76, S.]). Diese Kennung wird im Ethernet-Frame zur Adressierung verwendet. Es existieren spezielle MAC-Adressen, um alle Teilnehmer im Netzwerk (*Broadcast*) oder einen Teil der Teilnehmer (*Multicast*) anzusprechen. Eine Übertragung an ein einzelnes Gerät wird mit *Unicast* bezeichnet. Der sog. Ethertype ist ein 16-Bit-Wert, der das Protokoll in der nächsthöheren Schicht identifiziert.

Ethernet wurde als Bussystem für ein geteiltes Medium entworfen. Den dabei auftretenden Kollisionen wird mit dem Buszugriffsverfahren *Carrier-Sense Multiple Access/ Collision Detection* (CSMA/CD) begegnet [54, S. 86]. Das bedeutet, dass ein Busteilnehmer zunächst lauscht, ob der Bus belegt ist. Ist er frei, beginnt der Teilnehmer zu senden. Indem er gleichzeitig das auf dem Bus liegende Signal belauscht, können etwaige Kollisionen entdeckt werden, da das Signal eines weiteren sendenden Teilnehmers mit dem eigenen interferieren würde. Beide Teilnehmer brechen daraufhin die Übertragung ab und werden nach einer zufälligen Wartezeit erneut versuchen zu senden.

Es ist leicht einzusehen, dass dieses Zugriffsverfahren aufgrund des inhärenten Nichtdeterminismus mit garantiertem Echtzeitverhalten nicht vereinbar ist. Das ist nicht nur beim Einsatz als Feldbussystem ein Problem, sondern führt besonders bei hoher Busauslastung auch Allgemein zu einer suboptimalen Ausnutzung des Übertragungsmediums.

In der Vernetzung von Bürocomputern hat sich deshalb inzwischen die Verwendung von Switches zur Verkleinerung der Kollisionsdomäne durchgesetzt. Dabei wird eine Stern- oder Baumtopologie aufgebaut, bei der jeder Switch ein Paket zielgerichtet nur an denjenigen Anschluss weiterleitet, an dem der Teilnehmer angeschlossen ist, für den das Paket bestimmt ist. Ein Netz, das vollständig auf diese Weise aufgebaut ist, wird auch *voll-geswitchtes Netzwerk* (engl. *fully switched network*, FSN) genannt. Hier können keine Kollisionen mehr auftreten und damit führt CSMA/CD auch nicht mehr zu Wartezeiten von zufälliger Dauer. Ein FSN ermöglicht auch den sog. *full-duplex* Betrieb, also das gleichzeitige Senden und Empfangen von Paketen [76, S.76].

Es muss jedoch die zusätzliche Latenz durch die Verwendung von Switches berücksichtigt werden. Ein Switch muss zunächst die ersten Bytes eines ankommenden Pakets empfangen und verarbeiten, um die MAC-Adresse des Zielteilnehmers herauszufinden und es entsprechend einem Ausgang zuzuordnen. Erst im Anschluss kann es weitergereicht werden. Das führt unweigerlich zu einer Latenz. Die Latenzen mehrerer Ebenen in einer etwaigen Baumstruktur akkumulieren sich. Auch beim gleichzeitigen Eintreffen zweier Pakete, die an den selben Ausgang am Switch weitergereicht werden sollen, entstehen Verzögerungen unbekannter Dauer. Zusätzliche Maßnahmen sind also erforderlich, um ein FSN für Echtzeitanwendungen fit zu machen.

Mit *EtherCAT* [13], *EtherNet/IP*, *Profinet*, *SERCOS III*, *Modbus/TCP* und *Ethernet POWERLINK* seien die nach [1] den Weltmarkt dominierenden Systeme genannt (vgl. Abb. 2), die antreten, um geringe Latenzen und harte oder weiche Echtzeitfähigkeit durch Protokollanpassungen, durch Einschränkungen, oder durch höherschichtige Erweiterungen mit Ethernet zu vereinen. Eine ausführliche Auflistung der weltweit im Einsatz befindlichen Protokolle und Systeme findet sich in [69].

Nach einer kurzen Einführung in die zugrundeliegenden Prinzipien, sollen die oben genannten Systeme in den folgenden Abschnitten vorgestellt und auf ihre Eignung als echtzeitfähiges Feldbussystem überprüft werden. Eine zusammenfassende Gegenüberstellung findet sich in Tabelle 1 auf Seite 10. Eine detaillierte Gegenüberstellung der IE-Systeme mit besonderem Augenmerk auf deren Wirtschaftlichkeit findet sich auch in [39].

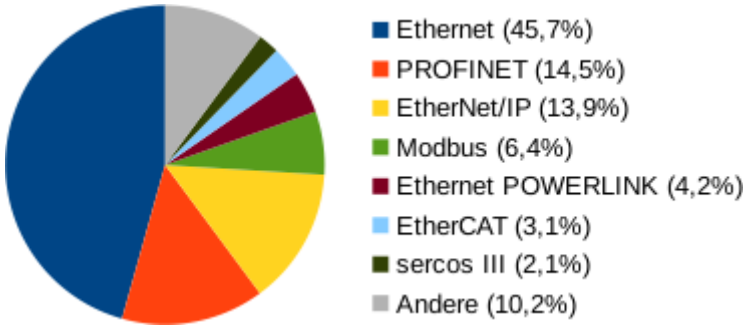


Abbildung 2: Weltmarktanteile von 2015 für neu installierte Industrie-Ethernet-Knoten. Quelle: [1]

2.2. Grundlegende Verfahren

Die meisten Protokolle, die auf einem Bus als geteiltes Medium harte Echtzeitbedingungen erfüllen wollen, haben gemeinsam, dass sie auf einem der im Folgenden genannten Grundprinzipien beruhen, um Kollisionen, die eine Auflösung in unbekannter Zeit notwendig machen würden, gar nicht erst auftreten zu lassen.

Zeitschlitzverfahren

Beim Zeitschlitzverfahren (engl. *time slicing*) wird eine feste Zykluszeit definiert, innerhalb derer je mind. ein Zeitschlitz für jeden Busteilnehmer vorgesehen ist. Ein Busteilnehmer darf ausschließlich in dem für ihn vorgesehenen Zeitschlitz senden. Dabei muss darauf geachtet werden, dass der Sende- und Übertragungsvorgang innerhalb dieser Zeit auch vollständig abgeschlossen werden kann. D.h. unterschiedliche Signallaufzeiten und andere Verzögerungen (bei Ethernet z.B. durch Hubs und Switches) müssen berücksichtigt werden. Es ist bei diesem Verfahren essentiell, dass die Zeitgeber der Teilnehmer synchron und präzise sind. Deswegen wird das Zeitschlitzverfahren oft kombiniert mit einem Verfahren, das die Teilnehmer regelmäßig synchronisiert. Unterschiedliche Bandbreitenanforderungen verschiedener Busteilnehmer können entweder durch unterschiedliche Längen der Zeitschlitze oder durch abwechselnde Verwendung eines Zeitschlitzes durch mehrere Teilnehmer (*Multiplexing*) berücksichtigt werden.

Polling

Polling, d.h. gezieltes Abfragen von Teilnehmern, ist eine Möglichkeit, Kollisionen auf dem Bus zu vermeiden, ohne synchrone Uhren aller Teilnehmer vorauszusetzen. Dabei sendet ein Master-Knoten Nachrichten an je genau einen anderen Knoten und gibt diesem damit Senderecht. Diese Nachricht kann auch bereits Nutzdaten enthalten. Es wird so lange keinem anderen Teilnehmer Senderecht gegeben, bis der Master ein Paket des entsprechenden Teilnehmers empfangen hat oder die zulässige Zeit für das Wahrnehmen des Senderechts vergangen ist. Oftmals wird auch das Polling zyklenweise ausgeführt,

um eine untere Schranke dafür definieren zu können, wie oft ein Teilnehmer Senderecht bekommt. Das heißt, dass der Master beispielsweise garantiert, jeden Teilnehmer mindestens einmal pro Zyklus abzufragen. Um den Teilnehmer nicht zu überlasten, lässt sich auch eine obere Schranke definieren (entspricht dem Versprechen, jeden Knoten *höchstens* einmal pro Zyklus anzusprechen). Ein Nachteil des Polling-Verfahrens ist, dass der Master im Allgemeinen einen sog. *Single Point of Failure* darstellt, d.h. ein Betrieb ist ohne ihn nicht möglich. Einige Protokolle, die Polling verwenden, sehen deswegen den Einsatz von mehreren redundanten Mastern vor, die im Fehlerfall einspringen können.

Token-Passing und Summen-Rahmen

Beim Token-Passing wird das Senderecht nicht zentral durch Nachrichten von einem Master-Knoten vergeben, sondern von einem Teilnehmer zum nächsten weitergereicht. Es bietet sich an, dafür eine Ringtopologie zu verwenden. Das Token kann dabei eine dedizierte Nachricht zur Zuteilung des Senderechts sein. Eine weitere, *Summenrahmen-Verfahren* genannte Möglichkeit ist, dass die Nutzdaten an sich das Token bilden, an die jeder Knoten seine Daten anhängt oder entgegennimmt und anschließend das geänderte Paket weiterleitet. Möchte man analog zu den anderen vorgestellten Verfahren ein regelmäßiges Senderecht garantieren, muss für jeden Teilnehmer eine obere Schranke der Bearbeitungszeit gegeben sein. Bei einem Ausfall eines einzelnen Teilnehmers bricht im Allgemeinen die gesamte Kommunikation zusammen.

Weitere Verfahren

Es gibt mit einigen Einschränkungen noch weitere Verfahren, den Buszugriff echtzeitfähig zu machen. So sorgt beispielsweise die bitweise Arbitrierung bei CAN dafür, dass Pakete des Knotens mit der höchsten Priorität harten Echtzeitbedingungen genügen. Zusammen mit einer Begrenzung der Senderate aller Teilnehmer, sodass eine Überlastung des Busses ausgeschlossen wird, lassen sich auch für die anderen Teilnehmer obere Grenzen für die Latenzzeit angeben.

2.3. Überblick

EtherCAT

EtherCAT (*Ethernet for control automation technology*) ist ein Ethernet-basiertes Protokoll, das harten Echtzeitanforderungen genügt und besonderes Augenmerk auf hohe Leistungsfähigkeit legt. So wird damit geworben, dass beispielsweise 12000 digitale Ein- und Ausgänge in 350µs gelesen bzw. gesetzt werden, oder 100 Servo-Achsen in 100µs angesteuert werden können [36, S.6]. Außerdem werden anders als bei vielen der konkurrierenden Systeme beide Senderichtungen bei Full-Duplex-Ethernet ausgenutzt, was zusammen mit der Tatsache, dass Daten mehrerer Busteilnehmer in einem einzelnen Frame übertragen werden, zu einer ausgezeichneten Nutzdaten-Auslastung von bis zu 90% der theoretisch verfügbaren Bandbreite führt [24, S.9].

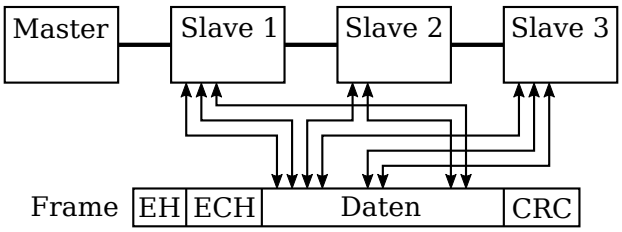


Abbildung 3: Illustration der Adresszuteilung von Prozessdaten mehrerer Slaves im EtherCAT-Frame. Angelehnt an [36, S.12]. EH: Ethernet-Header, ECH: EtherCAT-Header, CRC: Prüfsumme

EtherCAT basiert auf dem Master-Slave-Prinzip und arbeitet mit Summentelegrammen, in denen jedem Slave jeweils ein Adressbereich vom Master zugeteilt wird (vgl. Abb. 3). Die Knoten sind in Reihe geschaltet und geben das Frame von einem zum nächsten weiter, nachdem sie aus dem zugeteilten Adressbereich ihre Eingangsdaten eingelesen und ihre Ausgangsdaten abgelegt haben. Sobald der letzte Knoten in der Reihe erreicht ist, wird das Frame über den Rückkanal von Full-Duplex-Ethernet in Gegenrichtung unverändert weitergereicht, bis es wieder den Master erreicht. Es wird also virtuell eine Ring-Topologie gebildet. Die Verkabelung ist dank Hin- und Rückkanal nicht auf eine Linientopologie beschränkt, sondern kann über Doppelweichen auch Stern-, Baum- und anderen Topologien inklusive beliebiger Stichleitungen folgen, während virtuell immer eine Ringtopologie ausgebildet wird. Die Slaves erkennen dabei, je nachdem welche Anschlüsse belegt sind, automatisch ihre Rolle im Netz. Viele EtherCAT-Geräte besitzen auch mehr als zwei Anschlüsse, um unkompliziert Abzweigungen hinzufügen zu können.

Die hohe Leistungsfähigkeit von EtherCAT trotz sich aufsummierender Latenzen in einer Ringtopologie wird dadurch erreicht, dass die gesamte Protokollabwicklung der Slaves effizient in Hardware durchgeführt wird (im sog. *EtherCAT Slave Controller* (ESC)). Dabei beschränkt sich die Arbeit im wesentlichen auf den Abgleich eines Frame-Abschnitts auf einen Adressbereich im Gerät. Dadurch ist die Leistungsfähigkeit eines Geräts vollkommen unabhängig von Software und Mikrocontroller. Letzterer kann bei einfachen I/O-Geräten sogar ganz entfallen, wenn die Bits im Frame-Abschnitt direkt den Ein- und Ausgängen zugeordnet sind. Eine weitere Maßnahme zur Erhöhung der Leistungsfähigkeit ist die Tatsache, dass die EtherCAT-Geräte ein Frame bereits weiterleiten, während es noch empfangen wird.

Diese Maßnahmen sind jedoch auch der große Kritikpunkt an EtherCAT, denn sie verhindern den Einsatz von gewöhnlicher Ethernet-Hardware in Slave-Geräten mit einer EtherCAT-Implementierung in Software. Die EtherCAT Technology Group wirbt zwar selbst mit der Kosteneffizienz von EtherCAT und der Verfügbarkeit von kostengünstigen ESC-Chips (eine Übersicht verfügbarer Chips findet sich in [25], Stand: Oktober 2016), im Vergleich zu Standard-Ethernet-MACs, wie sie in vielen Kleinstrechnern und Mikrocontrollern bereits vorhanden sind und in großer Stückzahl produziert werden, liegt der Preis jedoch um Größenordnungen höher. Insbesondere ist in vielen Fällen die Anfertigung einer individuellen Platine erforderlich.

EtherNet/IP

EtherNet/IP [6], oder kurz EIP - IP steht hierbei für *Industrial Protocol*, ist eine Ethernet-Technologie, die auf dem im Internet verbreiteten TCP/IP-Protokollstapel aufbaut. Es stellt eine Implementierung des *Common Industrial Protocol* (CIP) [73] dar, ein vom Übertragungsmedium unabhängiges und objektorientiertes Protokoll für industrielle Automatisierungsaufgaben. CIP wird beispielsweise auch bei den Feldbussen *DeviceNet* und *ControlNet* eingesetzt.

EIP spezifiziert eine Vielzahl verschiedener Kommunikationsmechanismen, Datenobjekten und Vernetzungsstrategien. EIP existiert deshalb je nach Konfiguration und Anwendungsfall in verschiedenen Ausprägungen. Im Zusammenhang mit Echtzeitkommunikation wird typischerweise ein Switch in einer Sterntopologie eingesetzt, um Kollisionen durch CSMA/CD zu verhindern [86, S.7]. Für zeitkritische Daten wird außerdem meist ein zyklischer UDP-basierter Kommunikationskanal verwendet, bei dem die Senderate begrenzt sein sollte. Zu guter Letzt wird auch der Einsatz von priorisierenden Switches nach *IEEE 802.1q* empfohlen, um Echtzeitkommunikation und andere Datenübertragungen zu trennen [86, S.9].

Mit diesen Maßnahmen erfüllt EIP weiche Echtzeitanforderungen mit Zykluszeiten um die 10ms [39, S.9]. Mit *CIP Sync* und *CIP Motion* gibt es auch Bestrebungen noch strengere Anforderungen zu erfüllen.

Profinet

Profinet (*Process Field Network*) basiert wie EtherNet/IP auf dem bekannten TCP/IP-Protokollstack und ist unterteilt in zwei Teilstandards Profinet IO (*Input-Output*) und Profinet CBA (*Component Based Automation*). Das System ist in drei Schichten für jeweils unterschiedliche Echtzeitanforderungen unterteilt [26, S.35]:

- nicht zeitkritische Kommunikation über TCP/IP
- lokale Echtzeitkommunikation (RT): Bei Zykluszeiten von 5ms und mehr genügt eine VLAN-Priorisierung in Switches und die Limitierung der Sendebandbreite um hinreichendes Echtzeitverhalten zu erreichen.
- isochrone Echtzeitkommunikation (IRT) mit Zykluszeiten von max. 1ms und Jitter von unter 1µs. Hierfür werden spezielle Switches verwendet, die die IRT-Kommunikation von anderen Paketen trennen. Durch Synchronisations-Nachrichten werden alle Komponenten, auch die Switches, präzise synchronisiert. Auf dieser Basis wird ein IRT-Zyklus etabliert, in dem die Profinet-Geräte nach einem definierten Offset exklusives Senderecht besitzen. Die IRT-Schicht von Profinet basiert also auf Time-Slicing und erfüllt harte Echtzeitandorderungen.

Durch die Verwendung von speziellen Switches, büßt Profinet in der IRT-Variante einige der Vorteile ein, die man sich von industriellem Ethernet verspricht, u. a. die Verwendung von etablierter Standardhardware.

Modbus/TCP

Modbus ist ein offenes Kommunikationsprotokoll und existiert bereits seit 1979. Es ist in der Industrie sehr weit verbreitet, basiert jedoch in seiner ursprünglichen Form nicht auf Ethernet. Modbus-Datenpakete lassen sich auch mittels TCP/IP übertragen. Seit 2007 ist diese Variante als Modbus/TCP teil der Norm *IEC 61158*. Modbus/TCP verwendet Polling in einer Master/Slave-Stuktur, gibt aber darüberhinaus keine Echtzeitgarantien. Nach [39, S.5] sind die Konzepte von Modbus/TCP inzwischen in EtherNet/IP integriert.

SERCOS III

SERCOS III [42] verwendet einen Ethernetring oder -Doppelring mit dem alle Geräte mit einem Master verbunden sind. Die Kommunikation erfolgt zyklenweise, wobei ein Zeitschlitz für nichtzyklische Pakete reserviert ist. Die zyklischen Pakete werden in für jeden Teilnehmer fest während der Initialisierungsphase vergebenen Zeitschlitze gesendet. Etwaige Verzögerungen werden während der Initialisierung gemessen und später kompensiert. SERCOS III erreicht damit einen Jitter von unter 1µs [87]. Bei Verwendung einer Ringtopologie entsteht durch Full-Duplex-Ethernet ein Doppelring, was die Ausfallsicherheit etwas erhöht, da ein einzelner Defekt einer Leitung oder eines Geräts nicht zum Zusammenbruch des gesamten Netzwerks führt. Die zyklischen Nachrichten der Slaves werden sukzessiv zu einem Summentelegramm zusammengeführt und anschließend an den Master zurückgegeben. Dieses Verhalten macht so wie bei EtherCAT spezielle MACs erforderlich und ist dabei weniger flexibel, was die Verkabelungstopologie angeht. Lediglich der Master kann aufbauend auf Standard-Ethernet-Hardware implementiert werden. Eine Besonderheit ist, dass in einem Zyklus jede Nachricht zweimal bearbeitet wird. Einmal auf dem Hinweg und einmal auf dem Rückweg zum Master. Das macht einen direkten Datenaustausch zwischen zwei Slaves innerhalb eines Zyklus möglich.

Ethernet POWERLINK

Ethernet POWERLINK (EPL) [22] erweitert Fast Ethernet (*IEEE 802.3u*) und bietet sowohl harte Echtzeitkommunikation mithilfe eines zyklischen Pollingverfahrens, als auch den Versand beliebiger Ethernetpakete in einer asynchronen Phase. EPL wurde ursprünglich von B&R [35] entwickelt und wird heute als Version 2 von der *Ethernet POWERLINK Standardization Group* (EPSG) betreut. Ein Alleinstellungsmerkmal von EPL unter den konkurrierenden IE-Technologien ist, dass es trotz harter Echtzeiteigenschaften in den OSI-Schichten 1 und 2 [89] vollkommen kompatibel zu Standard-Ethernet-MACs und Hubs ist. Master und Slaves können dadurch in Software unter Verwendung der Standard-Ethernet-Treiber in gängigen Betriebssystemen implementiert werden. Entsprechende Software ist als Open-Source verfügbar und unterliegt keinen Patentansprüchen. Für hohe Leistungsanforderungen existieren auch dedizierte Chips, die die Protokollverarbeitung effizient in Hardware umsetzen.

Die harten Echtzeiteigenschaften werden dadurch erreicht, dass durch die exklusive Buszuteilung (Polling) durch den Master (hier: *Managing Node* (MN)) keine Kollisionen auftreten können. Um unnötige Latenzen zu vermeiden, wird empfohlen, zur Vernetzung

Tabelle 1: Vergleich Ethernet-basierter Bussysteme für den industriellen Einsatz

	PROFI-NET IRT	EtherNet/ IP	Modbus	EtherCAT	SERCOS III	Ethernet POWER-LINK
harte Echtzeit	✓	✗	✗	✓	✓	✓
Standard-Hardware[3]	✗	✓	✓	✗	✗	✓
Topologien	alle (mit Switches)	alle, meist Stern	alle (mit Switches)	alle (Doppel-weichen, implizit Ring/Dop-pelring)	Linie / Ring (implizit Ring / Doppel-ring)	alle (mit Hubs)
Prinzip	Zeitschlitz, spezielle Switches	versch.; meist Polling, VLAN	voll ge-switchtes Netzwerk	Summen-rahmen	Zeitschlitz/ Summen-rahmen	Polling / Zeitschlitz

Hubs statt Switches zu verwenden, die theoretisch eine niedrigere Latenz aufweisen[2], da die Pakete nicht gelesen und zwischengespeichert werden müssen (*store and forward*, *Teilstreckenverfahren*), sondern sofort an alle Ausgänge weitergereicht werden [76, S.312]. Switches bringen hier keinen Mehrwert, da eine Unterteilung der Kollisionsdomäne mangels Kollisionen nicht notwendig ist.

Ethernet POWERLINK ist eng mit CANopen verbunden, einem Protokoll, das eine herstellerübergreifende Anwendungsschicht auf Basis des CAN-Busses definiert und einheitliche Profile für verschiedene Hardwaretypen mitbringt, z.B. für Antriebe (*CiA-402*: *drives and motion control*), I/O-Geräte (*CiA-401*: *generic I/O modules*), aber auch für ganze Geräteverbünde, wie z.B. Aufzüge (*CiA-417*: *lift control systems*). EPL hat damit Zugriff auf einen großen Pool an Hardware, welche mit nur sehr geringem Aufwand an EPL angepasst werden kann, z.B. mit einem CANopen-EPL-Gateway, auch EPL-Router Typ 2 genannt.

Aufgrund der genannten Vorteile, die besonders dann zur Geltung kommen, wenn nicht ohne Weiteres individuelle, hochspezialisierte und/oder teure Hardware und Lizenzen angeschafft werden können, kommt in dieser Arbeit Ethernet POWERLINK als Feldbusstandard zum Einsatz. In den folgenden Abschnitten sollen deshalb die Funktionsweise und die Eigenschaften von EPL genauer beleuchtet werden.

[2]Die Aussage muss im Kontext der Entstehungszeit (EPL-Einführung: 2001) betrachtet werden. Durch den technischen Fortschritt sind Switches immer leistungsfähiger geworden und haben Hubs nahezu vollständig vom Markt verdrängt. Im zur Zeit in Arbeit befindlichen Nachfolgeprotokoll *Gigabit-Powerlink* sind deshalb trotz der theoretischen Nachteile Switches zur Vernetzung vorgesehen [41].

[3]Die Verwendung von Standard-Ethernet-Chips ist für alle teilnehmenden Geräte möglich.

Abbildung 4: Ein typischer EPL-Zyklus mit drei CNs

2.4. Ethernet POWERLINK

Nachdem in Abschnitt 2.3 nach einem Vergleich verschiedener IE-Technologien bereits einige wichtige Eigenschaften von EPL genannt wurden, sollen hier weitere für die Arbeit mit EPL wichtige Details erläutert werden.

2.4.1. Kommunikationszyklus

Die Kommunikation erfolgt zyklenweise, wie sie in Abb. 4 beispielhaft illustriert ist: Der MN sendet zum Start des Zyklus ein *Start of Cycle*-Paket (`SoC`) via Multicast. Mit dieser Nachricht werden alle Teilnehmer synchronisiert, d.h. Prozessdaten sollen genau zu diesem Zeitpunkt gelesen und geschrieben werden, egal wann die eigentliche Übertragung stattfindet. Anschließend sendet der MN jedem Slave (hier: *Controlled Node* (CN)) eine *Poll-Request*-Nachricht (`PReq`, als Unicast). Diese enthält zum einen Daten, die der MM an den CN senden will und zum anderen signalisiert sie dem CN exklusives Senderecht. Dieser antwortet mit einer *Poll Response*-Nachricht (`PRes`). Da auch diese via Ethernet-Multicast versendet wird, kann sie nicht nur Prozessdaten enthalten, die an den MN gerichtet sind, sondern auch für direkte CN-CN-Kommunikation verwendet werden. Der MN sendet das nächste `PReq`-Paket erst nach Erhalt der `PRes`-Nachricht. Dabei gilt ein konfigurierbares Timeout, nachdem von einem Defekt des CN ausgegangen wird und mit den anderen Knoten fortgefahren wird.

Die Erweiterung *EPSG DS302-C* sieht mit dem sog. *Poll Response Chaining* eine Beschleunigungsmöglichkeit für Anwendungen vor, die nur wenige Daten übertragen. Dabei wird auf die `PReq`-Pakete vollständig verzichtet, stattdessen sendet jeder Knoten, auch der MN, eine Poll Response an einem definierten Zeitpunkt innerhalb der synchronen Phase. Das Polling wird also durch ein Zeitschlitzverfahren ausgetauscht. Die Zeitpunkte werden nach einer Latenzmessung mit speziellen Paketen während der Initialisierungsphase vom MN festgelegt. Die Anforderungen an die CNs sind in dieser Variante etwas höher, da sie über präzise Timer verfügen müssen.

Nachdem die synchronen Daten mit allen CNs ausgetauscht wurden, beginnt mit der *Start of Asynchronous*-Nachricht (`SoA`) die asynchrone Phase. In dieser Phase darf höchstens ein einzelner Knoten senden. Welcher wird in im `SoA`-Frame signalisiert. Ein Knoten muss zuvor seinen Sendewunsch mit dem *RequestToSend*-Flag innerhalb der Poll Response angekündigt haben. Nachdem ein Knoten in der asynchronen Phase Senderecht bekommen hat, darf er jedes beliebige Ethernet-Frame senden, auch nicht EPL-spezifische Pakete. Damit ist TCP/IP-Kommunikation ohne Tunnelung möglich. Asynchrone EPL-Nachrichten werden in einem *Asynchronous Send*-Paket (`ASnd`) übertragen. Diese gibt es in verschiedenen Varianten, von denen einige im folgenden Abschnitt vorgestellt werden.

Protokoll	Byte	Inhalt
Ethernet (Typ 2)	0–5	Ziel-MAC-Adresse
	6–11	Quell-MAC-Adresse
	12–13	Ethertype
Ethernet POWERLINK	14	Nachrichtentyp
	15	Zielknoten
	16	Quellknoten
	17–n	Daten
Ethernet (Typ 2)	n+1–n+4	CRC

Abbildung 5: Aufbau eines EPL-Frames

Powerlink-Daten sind grundsätzlich in Ethernet-II-Frames eingebettet und bestehen mindestens aus je einem Byte für Ziel- und Quell-Knoten und EPL-Nachrichtentyp (vgl. Abb. 5).

Kommunikationsprimitive Die Anwendungsschicht lehnt sich stark an das CANopen-Protokoll [87, S.283–289] an. Die Kommunikationsprimitive *Process Data Object* (PDO) und *Service Data Object* (SDO) wurden übernommen, wobei die Größenbeschränkungen von z.B. lediglich 8 Byte bei CANopen-PDOs an die größere Bandbreite von Ethernet angepasst wurden. Auch das *Network Management* (NMT), bestehend aus einer Zustandsmaschine und Nachrichten zur Zustandsänderung und zur Fehlersignalisierung, und die sog. Objektverzeichnisse (OV, engl. *Object Dictionaries*), die den Gesamtzustand des Knotens inklusive Kommunikationsparameter, Typinformationen, Fähigkeiten, Geräte- und Prozessdaten enthalten, sind dem CANopen-Standard entnommen.

Alle Daten, die im Netzwerk ausgetauscht werden, beziehen sich auf die Objektverzeichnisse der Knoten, es wird also grundsätzlich ein Wert von einem Eintrag des OVs eines Knotens gelesen und in einen Eintrag des OVs eines anderen Knotens übernommen. Die Übertragung findet auf einem der folgenden Wege statt:

SDO SDOs (*Service Data Objects*) werden immer im asynchronen Zeitfenster übertragen und bestehen aus zwei Schichten, dem Sequence-Layer und dem Command-Layer. Der Sequence-Layer emöglicht die Aufteilung von Nutzdaten auf mehrere Pakete und damit mehrere Zyklen, was benötigt wird, wenn das asynchrone Zeitfenster nicht ausreicht um alle Daten in einem Rutsch zu übertragen. Im Command-Layer wird die auszuführende Aktion gefolgt von den eigentlichen Nutzdaten abgelegt. Ein Beispiel für eine Aktion ist *Write by Index*, womit ein Datum an einen bestimmten Index im OV geschrieben wird. Gegenüber CANopen erlaubt der EPL-Standard darüber hinaus auch einen OV-Zugriff anhand des Feldnamens sowie das Beschreiben mehrerer Felder in einem Zugriff.

SDOs werden üblicherweise ausschließlich zur Übertragung von Konfigurationsdaten beim Initialisieren des Netzwerks sowie zur Übermittlung von Statusinformationen verwendet.

PDO PDOs (*Process Data Objects*) können anders als in CANopen ausschließlich zyklisch übertragen werden und es existiert für Slaves auch je höchstens eine Transmit- (TPDO – Prozessdaten-Ausgänge) und eine Receive-PDO (RPDO – Prozessdaten-Eingänge). Sie unterscheiden sich von SDOs darin, dass sie ausschließlich Nutzdaten enthalten. Wie die Nutzdaten zu interpretieren sind, d.h. welche Bytes in der PDO welchem Eintrag im Objektverzeichnis entsprechen, muss zuvor im OV konfiguriert worden sein.

Die vom Master in jedem Zyklus an jeden CN gesendete `PReq`-Nachricht wird dabei vom diesem als RPDO interpretiert. Die `PRes`-Nachricht gilt als TPDO.

SDO via UDP/IP und PDO Eine unmittelbare Interaktion zwischen Hausnetzwerk und EPL-Geräten ist optional über die Einbettung von SDO-Nachrichten in UDP/IP-Frames möglich. Dazu muss das Hausnetz mit einem EPL-Router Typ 1 mit dem EPL-Netz verbunden sein. Über die jedem CN zugeorndete IP-Adresse ist so ein Kommunikationsweg vom Arbeitsplatzrechner zum CN ohne Umwandlung oder Tunnelung möglich. Eine weitere optionale Variante ist die Einbettung von SDO-Nachrichten in PDOs. Dies ist für sehr dynamische Anwendungen sinnvoll, deren Kommunikation sich nicht ohne weiteres mithilfe eines Prozessabbildes modellieren lässt.

Ident Mit den Pakettypen *Ident Request* (Teil eines `SoA`-Pakets) und *Ident Response* (Teil eines `ASnd`-Pakets) identifiziert der MN im Netzwerk vorhandene CNs.

Status Mit den Pakettypen *Status Request* (Teil eines `SoA`-Pakets) und *Status Response* (Teil eines `ASnd`-Pakets) ist eine Überwachung des CN-Zustands möglich. Das ist insbesondere für Knoten relevant, die nur in der asynchronen Phase kommunizieren, denn andernfalls würde ein Defekt nicht unmittelbar auffallen.

NMT Der MN nutzt die asynchrone Phase außerdem um *Network Management*-Pakete zu versenden. Das beinhaltet insbesondere Nachrichten zur Veranlassung von Zustandsänderungen in den CNs und das Neustarten der Kommunikation mit einem oder allen Knoten im Fehlerfall.

2.4.2. Virtuelles Ethernet

Da im asynchronen Zeitfenster beliebige Pakete versendet werden dürfen, lässt sich darüber ressourcenschonend ein herkömmliches Ethernet-Netzwerk simulieren. Gängige Implementierungen richten hierfür eine virtuelle Netzwerkkarte im System ein, für die das EPL-Netzwerk vollkommen transparent ist.

Von Vorteil ist, dass die Integration in ein IP-basiertes Netzwerk im EPL-Standard explizit vorgesehen ist. So ist jedem Knoten standardmäßig eine IP-Adresse zugeordnet, die sich aus der Knotennummer ableiten lässt (`192.168.100.`*`Knotennummer`*). Über das Objektverzeichnis lassen sich auch andere Adressen bzw. Subnetze definieren.

IP-Kommunikation mit Geräten außerhalb des EPL-Netzwerks wird ggf. durch einen sog. Typ 1-Router ermöglicht, der in seiner Grundfunktionalität nichts weiter darstellt,

Tabelle 2: Aufteilung eines EPL-Objektverzeichnisses

Index (hexadezimal)	Verwendung
0001 - 0FFF	Datentypen (rein deklarativ)
1000 - 1FFF	allgemeingültiges Kommunikationsprofil
2000 - 5FFF	herstellerspez. Felder
6000 - 9FFF	standardisierte Geräteprofile (bis zu 8 Stück)
A000 - BFFF	schnittstellenspez. Profile, z.B. dynamische Prozessabbilder nach *CiA 302-4*
C000 - FFFF	reserviert

als eine Netzwerkbrücke zwischen einer echten Netzwerkkarte und dem Virtuellen Ethernetgerät. Zusätzlich ist oft eine Authentifizierung und Network Address Translation (NAT) vorgesehen.

2.4.3. Datenorganisation und Anwendungsprofile

Alle Daten eines EPL-Geräts, egal ob Prozessdaten, Gerätedaten, oder Konfigurationsparameter, werden im OV abgelegt. Das OV besteht aus einfachen Datenfeldern, denen jeweils ein 16-bit Index und ein 8-bit Subindex zugeordnet ist. Es lassen sich also insgesamt über 16 Millionen Datenwerte ablegen. Der verfügbare Adressbereich ist wie in Tabelle 2 angegeben aufgeteilt.

Die grundlegenden Datentypen und Felder im allgemeingültigen Kommunikationsprofil sind im *EPSG Draft Standard 301*, dem EPL-Basisstandard, definiert. Die Geräteprofile sind mit denen des CANopen-Protokolls identisch. Eine Übersicht findet sich in [43]. Darüberhinaus definieren mehrere optionale Standarddokumente der EPSG und der CiA-Organisation [12] weitere Einträge im Objektverzeichnis (siehe [45] und [38]). Dazu zählen z.B. Standards zum Einsatz redundanter Managing Nodes (*EPSG DS302-A*) oder die Nutzung von dynamischen Prozessabbildern bei programmierbaren Knoten (*CiA 302-4*).

2.4.4. Gerätebeschreibung

Das Objektverzeichnis eines EPL-Geräts (insbesondere dessen Struktur, also die verfügbaren Felder) liegt üblicherweise in Form einer XML-basierten XDD-Datei vor (*XML Device Description*), wie sie in *EPSG DS-311* standardisiert ist. Das Format entspricht im Wesentlichen der CANopen-Variante, die in *CiA-311* definiert ist [67]. Mit der Speicherung dieser Datei im Objektverzeichnis sieht EPL optional eine vereinfachte Verteilung der XDD an Endkunden vor. Die XDD-Dateien der CNs können von einem Konfigurationsprogramm wie openCONFIGURATOR verwendet werden, um eine konkrete Netzwerkkonfiguration, d.h. eine Anfangsbelegung aller Objektverzeichnisse, zu erstellen. Mit der *Configuration Manager*-Funktionalität eines MN kann diese Konfiguration mittels einer einzelnen, dem MN vorliegenden Datei in der Initialisierungsphase auf das gesamte Netzwerk übertragen werden. Kapitel 6.3 gibt hierzu weitere Details.

3. Echtzeitbetriebssysteme für eingebettete Systeme

Es gibt grundsätzlich zwei verschiedene Herangehensweisen, Software für ein eingebettetes System zu entwickeln. Zum einen kann man sie direkt auf die eingesetzte Hardware zuschneiden. Schon der erste nach dem Start ausgeführte Maschinenbefehl gehört zur Anwendung. Es existiert kein Betriebssystem. Es werden nur die Teile der Hardware beachtet und initialisiert, die auch tatsächlich verwendet werden. Diese Art Software nennt man auch *bare metal*. Sie hat oft Vorteile in Bezug auf Speicherbedarf, Leistungsfähigkeit und Stromverbrauch.

Die historisch ersten Computerprogramme wurden alle auf diese Weise entwickelt. Seither entstanden in mehreren Evolutionsstufen Betriebssysteme, wovon die bekanntesten Vertreter eine ungeahnte Komplexität erreichen und aus mehreren Millionen Codezeilen bestehen[4]. Betriebssysteme stellen eine Abstraktionsschicht dar, die es ermöglicht, eine Anwendung mit geringem oder ohne zusätzlichen Aufwand auf unterschiedlicher Hardware auszuführen. Es erleichtert auch die Interaktion von verschiedenen Anwendungen und erlaubt deren (quasi-)gleichzeitige Ausführung.

Diese Abstraktionen sind nicht ohne Kosten in Form von Ressourcenverbrauch. Die meisten PC-Betriebssysteme stellen außerdem bestimmte Hardwarevoraussetzungen, wie z.B. das Vorhandensein einer Speicherverwaltungseinheit (MMU). Während im PC- und Server-Bereich die Bare-Metal-Entwicklung praktisch ausgestorben ist, findet sie wegen ebendiesen Anforderungen im Embedded-Bereich noch vielfach Anwendung.

Inzwischen versucht eine ganze Reihe von Projekten die Vorteile von beiden Welten zu vereinen und Betriebssysteme zu entwickeln, die auch auf einfachen Mikrocontrollern lauffähig sind, wie sie im Embedded-Bereich oft vorkommen. Dabei sollen sie sehr wenig Overhead im Vergleich zu einer Bare-Metal-Anwendung aufweisen, die in diesem Bereich oft gestellten Echtzeitanforderungen erfüllen und dabei angenehme Eigenschaften wie Multitasking und Portierbarkeit von bekannten PC-Betriebssystemen übernehmen. Auch wenn der Name nur einen Teilbereich der Eigenschaften ausdrückt, wird diese Art Betriebssystem oft RTOS (*Real-Time Operating System*, *Echtzeitbetriebssystem*) genannt. Eine weitere Bezeichnung ist *eingebettetes Betriebssystem*. Die Erstellung einer Anwendung, die auf ein solches RTOS aufsetzt, stellt die zweite Herangehensweise dar, Software für ein eingebettetes System zu entwickeln.

Eine beliebte Strategie ist es, Betriebssystem und Anwendung gemeinsam zu kompilieren und dabei eine monolithische Binärdatei zu erzeugen. Mit geeigneten Konfigurationsparametern und bedingter Kompilierung werden alle Teile des Betriebssystems, die nicht benötigt werden, ausgelassen. Die Konfiguration zur Kompilierzeit bedeutet eine bessere Anpassung an die Hardware und vermeidet an vielen Stellen die Notwendigkeit, Kompromisse einzugehen. Der Verzicht auf dynamisch geladene Binärdateien und Laufzeitbibliothekten ermöglicht effizientes Inlining und reduziert den Bedarf an Indirektionen. Abstraktionen werden so idealerweise schon zur Kompilierzeit aufgelöst. Das Endprodukt nähert sich so einer speziell entwickelten *Bare-Metal*-Anwendung an, gewährt jedoch auf Quelltextebene einen hohen Grad an Portierbarkeit.

[4] z.B. Linux-Kernel 4.9.10: 14.846.549 *Source Lines of Code* [14], gemessen mit `sloccount` 2.26 [82]

In den folgenden Abschnitten sollen einige Betriebssysteme für eingebettete Systeme betrachtet werden und die Entscheidung für NuttX als RTOS der Wahl in dieser Arbeit begründet werden.

3.1. Überblick

Die Anzahl der am Markt verfügbaren Echtzeitbetriebssysteme ist riesig. Allein die englische Wikipedia listet 178 Stück ([19], Stand: Dezember 2016). Aus diesem Grund kann hier leider nur auf eine kleine Auswahl eingegangen werden.

µClinux *µClinux* ist ein Projekt, das den Linux-Kernel so weit angepasst hat, dass es auf Mikrocontrollern ohne MMU lauffähig ist [15]. Große Teile sind heute im offiziellen Linux-Kernel enthalten, welcher seither auf einigen Architekturen für Systeme ohne MMU übersetzt werden kann. Durch die tiefgreifenden Änderungen am Kernel sind viele Linux-Programme jedoch nicht ohne weiteres lauffähig. Das liegt unter anderem daran, dass die auf den meisten Linux-Systemen eingesetzte C-Bibliothek `glibc` nicht verwendet werden kann.

ChibiOS/RT ChibiOS/RT ist ein RTOS, das nach eigener Aussage besonders Wert auf kompromisslos hohe Leistung legt [10]. Anwender können, wenn sie den vollen Funktionsumfang von ChibiOS nutzen wollen unter drei verschiedenen Lizenzen wählen: *GPLv3* [29], *Free Commercial* und *Full Commercial* [11]. Bei der Free Commercial-Lizenz bestehen keine Einschränkungen für die Anwendung, es dürfen jedoch keine Änderungen am Betriebssystem vorgenommen werden, was bei eingebetteten Systemen problematisch sein kann, wo oft Anpassungen für individuelle Hardwarevarianten nötig sind. Bei der Open-Source-Lizenz GPLv3 sind solche Änderungen möglich, Anwendungen gelten jedoch nach den Bestimmungen der GPLv3 als abgeleitetes Werk und müssen dann ebenfalls unter dieser Lizenz veröffentlicht werden [49]. Das ist beispielsweise auch dann problematisch, wenn bestehende Open-Source-Software, die unter einer inkompatiblen Lizenz steht, integriert werden soll. Die Full Commercial-Lizenz verursacht Lizenzkosten und widerspricht damit den Voraussetzungen dieser Arbeit.

mbed OS [4] ist ein RTOS für ARM Cortex-M-Mikrocontroller, das speziell für das *Internet der Dinge* entworfen wurde und somit eine reichhaltige Sammlung von Konnektivitätsbibiotheken enthält. Es gibt auch eine eigene Entwicklungsumgebung und günstige offizielle Demo-Boards (z.B. [3]). Der Einstieg in die Entwicklung einer ARM Cortex-M-Anwendung wird so stark erleichtert und erlaubt es, sich voll auf die Entwicklung der Kernapplikation zu konzentrieren. Nachteilig ist, dass die speziell auf die Plattform zugeschnittenen Bibliotheken die Integration von Fremdsoftware stark erschweren.

FreeRTOS FreeRTOS ist nach eigener Aussage Marktführer und De-Facto-Standard unter den Echtzeitbetriebssystemen [30]. Es unterstützt eine Vielzahl von Mikrocontrollern und Prozessoren von offiziell mindestens 20 verschiedenen Herstellern ([31], Stand:

Januar 2017). FreeRTOS steht unter der GPLv2 mit einer Ausnahme, die es ermöglicht, ähnlich wie beim Linux-Kernel[5], Anwendungsprogramme mit beliebiger Lizenz zu verwenden. Das System bietet Multithreading und Netzwerkunterstützung. Der TCP/IP-Stack ist jedoch nur für drei ausgewählte Mikrocontrollertypen unter freier Lizenz verfügbar [32].

NuttX NuttX ist ein BSD-lizenziertes RTOS, welches eine beeindruckende Vielzahl an verbreiteten APIs nachbildet, darunter POSIX [68], C/C++-Standardbibliotheken, Linux- und VxWorks-APIs [60]. Gleichzeitig ist NuttX hochgradig anpassbar mit einem Konfigurationssystem, das so auch beim Linux-Kernel verwendet wird. Damit ist es möglich, sehr kleinteilig alle nicht benötigten Teile auszulassen und die meisten Parameter zur Compile-Zeit festzulegen, was in dem gegeben Fall mit stark begrenzten Ressourcen von Vorteil ist. Die Aussicht, bestehende und beliebig lizenzierte Linux/Unix-Software mit begrenztem Aufwand auf dieses RTOS zu portieren, ist das ausschlaggebende Argument, warum für die weitere Arbeit an dem Thema NuttX als Betriebssystem verwendet wird.

3.2. NuttX

Teil dieser Arbeit ist es, NuttX an verschiedene Mikrocontroller-Boards anzupassen, verschiedene Konfigurationen zu testen und Treiber sowie Anwendungen zu entwickeln. Hierfür sind Kenntnisse über die Architektur von NuttX und insbesondere über die Funktionsweise des Netzwerksubsystems erforderlich. Die folgenden Abschnitte sollen diese vermitteln.

3.2.1. Architektur

Die Anzahl der unterstützten Prozessorarchitekturen ist groß und beinhaltet unter anderem bekannte Vertreter wie x86, MIPS und ARM in verschieden Varianten. Gerade im Embedded-Bereich ist die Diversität der Plattformen groß, auch innerhalb einer Prozessorfamilie. Deswegen ist der Quellcode von NuttX in mehreren Ebenen organisiert, die am Beispiel des Entwicklungsboards Micromint Bambino 210E (vgl. Kapitel 4) in Abb. 6 dargestellt sind.

Der schichtartige Aufbau des Quellcodes allein reicht nicht aus, um alle möglichen Hardwarevarianten abzubilden, denn die Ebenen stellen keine strikte Obermenge der Funktionen der darunterliegenden Ebene dar. Auch ist nicht immer für jede existierende Variante ein eigener Ordner vorhanden. NuttX arbeitet deswegen intensiv mit bedingter Kompilierung und nachträglichen Einsetzungen mithilfe des C-Präprozessors. Die konkreten Werte werden meist in einer Headerdatei aus einer spezifischeren Quelltextebene gesetzt. Einige Werte werden auch in einem Konfigurationsschritt vom Nutzer gewählt. Hierbei kommt das vom Linux-Kernel bekannte Kconfig [50] zum Einsatz.

[5] Der Linux-Kernel steht ebenfalls unter der GPLv2, Anwendungen, die nur auf gewöhnliche Systemaufrufe zurückgreifen, sind jedoch explizit von den Bestimmungen ausgeschlossen.

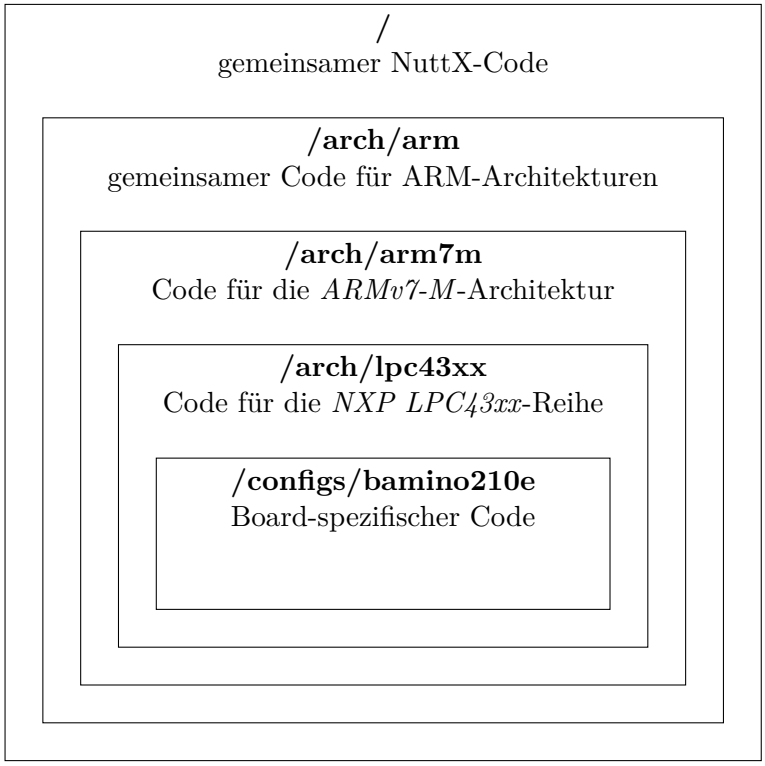


Abbildung 6: NuttX Codehierarchie für das Entwicklungsboard Micromint Bambino 210E

Dieses System ermöglicht eine recht geradlinige Portierung auf bis dato nicht unterstützte Hardwarevarianten und besteht gewöhnlich aus folgenden Schritten:

1. Anlegen eines neuen Ordners unterhalb von `/configs/`
2. Erstellen der Datei `/configs/`*`Boardname`*`/include/board.h`, die von allen relevanten Dateien inkludiert wird. Hier werden Board-spezifische Definitionen abgelegt, wie z.B. Pinbelegungen, Variantenauswahl, oder (De-)Aktivierung von Funktionen im Chip.
3. Erstellen der Datei `Kconfig`. Hier können Konfigurationsoptionen für den Nutzer angelegt werden. Außerdem werden hier ggf. Vorbedingungen und Empfehlungen in der Konfiguration anderer Teile von NuttX erzwungen bzw. vorausgewählt.
4. Anlegen einer Standardkonfiguration für Kconfig in der Datei `defconfig`. Dies geschieht üblicherweise durch Kopieren einer funktionierenden Konfiguration aus der Datei `.config` im NuttX-Hauptverzeichnis.
5. Viele Treiber verlangen nach einer Boardspezifischen Initialisierungsfunktion, im Falle des AD-Wandler-Treibers von NuttX beispielsweise die Funktion `board_adc_setup`. Sie müssen, wie alle anderen Boardspezifischen Funktionen, in einer Datei im `/configs/`*`Boardname`*`/src`-Ordner implementiert werden.
6. Makefiles und ggf. Linkerskripte müssen entsprechend angelegt oder auf Basis einer anderen Konfiguration angepasst werden.

Die Schritte wurden für die beiden in Kapitel 4 vorgestellten Plattformen durchgeführt.

3.2.2. Netzwerkstack

NuttX besitzt umfangreiche Netzwerkfähigkeiten und unterstützt wichtige Protokolle wie ARP, TCP, UDP, IPv4 und IPv6, ICMP und ICMPv6, und IGMP.

Die Schnittstelle zur Anwendung bilden dabei *Berkley Sockets*[6], wie sie auch bei Linux und vielen Unix-Systemen vorkommen. Die wichtigsten Funktionen dieser Schnittstelle sind die folgenden:

- Mit `socket()` wird Socket erstellt, der als Endpunkt eines Kommunikationskanals betrachtet werden kann.
- Mit `bind()` kann ein Socket auf eine Netzwerkadresse und andere Kommunikationsparameter eingeschränkt werden.
- Mit `read()`, `recv()` und `recvfrom()` werden Daten und Pakete empfangen.
- Mit `write()`, `send()` und `sendto()` werden Daten und Pakete versendet.

[6] als Teil der POSIX-Spezifikation auch POSIX-Sockets genannt [68]

- `close()` schließt den Kommunikationskanal.

Die Implementierung ist von dem *uIP Stack* von Adam Dunkels abgeleitet [83], der ursprünglich für leistungsschwache 8- und 16-bit-Mikrocontroller entworfen wurde. Durch die Reduktion auf die wesentlichen Bestandteile und Verzicht auf komplexe Routing-, Filter- und Puffer-Funktionalitäten ist der Netzwerk-Stack von NuttX trotz Standardkonformität wesentlich schlanker als das Gegenstück in Linux und vielen anderen Betriebssystemen. Dies hat jedoch, wie in Kapitel 5.2.2 sichtbar werden wird, direkte Auswirkungen auf die Implementierung eines Ethernet-basierten Protokolls wie EPL.

Ein großes Problem ist auch die an vielen Stellen fehlende Kapselung und Abstraktion der elementaren Protokolle, was Änderungen und Erweiterungen softwaretechnisch stark erschwert. So findet die Auftrennung anhand der Transport- und Vermittlungsprotokolle direkt im hardwarespezifischen Ethernet-Treiber statt. Das selbe Verhalten findet sich auch in den oben genannten Socket-Funktionen und an weiteren Stellen im System.

3.2.3. Datenempfang

Die grundlegende Funktionsweise des Netzwerkstacks lässt sich am Besten anhand der Eingangs-Prozessierung eines exemplarischen Ethernet-Pakets erläutern:

Im Falle des LPC43xx (vgl. Abschnitt 2.1) ist der Großteil des Ethernet-Protokolls auf Bitübertragungs- und Sicherungsschicht (OSI-Schichten 1 und 2) in Hardware implementiert. Der zugehörige Ethernet-Treiber `lpc43_ethernet` kümmert sich hauptsächlich um die Initialisierung und Konfiguration der Hardware. Bei Ankunft eines neuen Pakets wird ein Interrupt ausgelöst, woraufhin die Funktion `lpc43_receive` aufgerufen wird. Hier findet, wie eingangs erwähnt, eine erste inhaltliche Aufteilung des empfangenen Pakets statt. Zu große Pakete (entsprechend der *Maximum Transmission Unit* (MTU)) werden sofort verworfen. Wenn aktiviert, wird eine Referenz auf das gepufferte Paket dann an das *Raw-Socket*-Subsystem weitergegeben (`pkt_input`). Anschließend wird der Ethertype (vgl. Kapitel 2.1) betrachtet. IPv4- IPv6- und ARP-Pakete werden in jeweils verschiedene Subsysteme weitergeleitet. Zusätzlich wird der Empfang von IP-Paketen auch im ARP-Subsystem registriert, um den Adress-Cache zu aktualisieren.

Hier erkennt man die Schwäche dieser Architektur. Möchte man analog zu IP und ARP ein weiteres Protokoll implementieren, das direkt auf Ethernet aufbaut, wie z.B. Ethernet POWERLINK, muss der Ethernet-Treiber jeder verfügbaren Hardware-Plattform individuell angepasst werden. Dies ist insbesondere im Embedded-Bereich problematisch, wo viele Treiber für exotische Hardware außerhalb des Hauptentwicklungszweigs gepflegt werden.

Abbildung 7 zeigt den Prozess des Paketempfangs am Beispiel eines Ethernet-Pakets, das vom Anwender über einen Raw-Socket entgegengenommen wird. Aus Anwendersicht wird zunächst ein Socket entsprechenden Typs erzeugt. Intern wird daraufhin eine Verbindungs-Datenstruktur erstellt. Der Aufruf von `recv` hinterlegt dann einen Callback in der Verbindungs-Datenstruktur, der die Paketdaten kopieren und damit für die weitere Verwendung sichern soll. Mittels eines Semaphors wird gewartet, bis der Callback aufgerufen wurde. Anschließend wird er wieder gelöscht. Aus Treibersicht wird nach dem

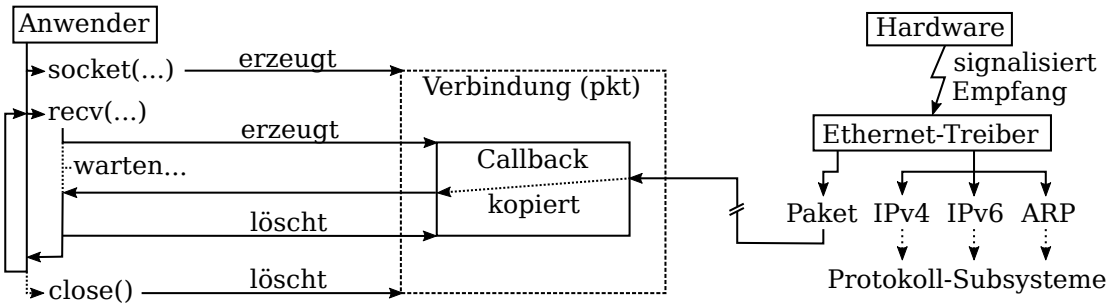


Abbildung 7: Empfang eines Ethernet-Pakets über einen Raw-Socket in NuttX

Aufruf von `pkt_input` (s. oben) eine passende Verbindung ausgewählt und darin – wenn vorhanden – der Callback aufgerufen.

Üblicherweise befindet sich der `recv`-Aufruf in einer Schleife, sodass fortlaufend Pakete angenommen werden können. Es sei hier anzumerken, dass dabei keine Pufferung stattfindet. In der kurzen Zeit zwischen Löschen und Neuregistrierung des Callbacks können Pakete also „verloren“ gehen. Das unterscheidet Raw-Sockets in NuttX von TCP-Sockets. Auch wenn sich dieses Verhalten von dem anderer Betriebssysteme unterscheidet und die Implementierung eines Protokolls erschwert, das verlässlichen Paketempfang voraussetzt, so ist sie doch nachvollziehbar, da sich die Semantik eines Raw-Sockets von der eines TCP-Sockets unterscheidet. TCP stellt einen Informationskanal dar, in dem die eigentlichen Nutzdaten gestückelt übertragen werden, in der Anwendungsschicht jedoch wie ein kontinuierlicher Datenstrom erscheinen (entspricht einem sogenannten *Stream Socket*). Es ergibt also Sinn, alle ankommenden Nutzdaten zu puffern und beim `recv`-Aufruf den bis dahin gesammelten Inhalt am Stück zurückzugeben. Bei Netzwerkkommunikation auf Ebene der Sicherungsschicht ohne Kenntnis höherer Schichten ist die Kenntnis der konkreten Paketgrenzen dagegen essentiell. Das macht eine korrekte Pufferung komplexer. Unter Berücksichtigung des Entwurfziels Ressourcenschonung ist ein Verzicht darauf vertretbar.

Trotzdem wird in Kapitel 5.2.2 ein gepufferter Paketempfang benötigt. Der folgende Abschnitt soll dazu eine Lösung bieten.

3.2.4. Gepufferter Paketempfang

Zur Ermöglichung von gepuffertem und damit verlässlichem Paketempfang kommen drei Varianten infrage:

1. Anpassen des NuttX-Raw-Sockets
2. Implementierung eines neuen Socket-Typs
3. Implementierung mit eigener API

Von Variante 1 wird Abstand genommen, um andere Raw-Socket-Anwendungen an einer derart performancekritischen Stelle nicht zu beeinträchtigen. Variante 2 wäre aus

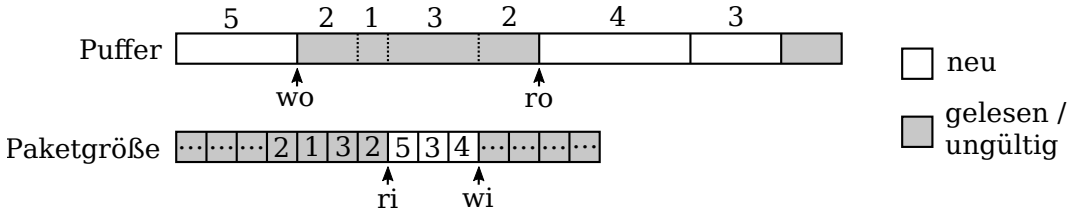


Abbildung 8: Gepufferter Paketempfang mit zwei Ringpuffern.
wo: Schreibposition, ro: Leseposition, ri: Leseindex, wi: Schreibindex

Anwendersicht eine elegante Lösung, aufgrund der Softwarearchitektur (s.o.) müssten dafür jedoch alle Netzwerktreiber und viele weitere Dateien in NuttX angepasst werden. Deshalb wird hier Variante 3 umgesetzt mit einer API aus den Funktionen `bufpkt_open`, `bufpkt_close`, `bufpkt_receive` und `bufpkt_send`.

Das System (fortan `bufpkt` genannt) greift auf Teile des Raw-Socket-Subsystems zurück und erscheint dem System wie ein gewöhnlicher Raw-Socket. Ansatzpunkt ist in `bufpkt_open` die NuttX-Funktion `pkt_callback_alloc`, die dafür sorgt, dass eigentlich für Raw-Sockets gedachte Pakete mit `pkt_input` an `bufpkt` weitergeleitet werden.

`bufpkt` arbeitet mit zwei Ringpuffern. Der erste speichert ankommende Paketdaten, der zweite die Paketgrößen, um beim Lesen die korrekten Paketgrenzen wiederherstellen zu können. Es wird bewusst auf dynamische Speicherallokationen verzichtet und Pakete werden nie fragmentiert abgelegt. Die Verarbeitung beschränkt sich deshalb auf einige wenige Vergleichs- und Additionsoperationen, In- und Dekrementieren eines Semaphors sowie zweimaliges Kopieren (Ethernet-Treiber $\rightarrow$ `bufpkt` und `bufpkt` $\rightarrow$ Anwender).

Abbildung 8 zeigt die zwei Ringpuffer mit beispielhafter Befüllung. Vier von links nach rechts wandernde Zeiger (wo, ro, wi, ri; vgl. Bildbeschriftung) sowie der Pufferinhalt bestimmen den Zustand des Systems. Ein Paket p der Größe $size_p$ wird verworfen, wenn $wo + size_p \geq ro$ oder wenn $wi + 1 = ri$. Der Puffer ist leer, wenn $wo = ro$ und $wi = ri$. Ein Semaphor zählt die aktuell verfügbaren Pakete und ermöglicht es einer Anwendung, auf eintreffende Pakete zu warten. Pseudocode zum Befüllen und Leeren des Puffers findet sich in Anhang A.3.

4. Hardware-Plattform

Zentrales Thema dieser Arbeit ist die Verwendung von Ethernet POWERLINK zusammen mit einem RTOS auf schwacher Hardware. Bevor es bearbeitet werden kann, muss zunächst der Begriff *schwache Hardware* näher definiert werden und konkrete Hardwareplattformen für die Umsetzung ausgewählt werden. Folgende Anforderungen sollen erfüllt werden:

1. Es handelt sich um einen Mikrocontroller (µC) ohne Speicherverwaltungseinheit
2. Die Leistungsaufnahme soll im Leerlauf sowie unter Last nicht größer als 1,5 Watt sein[7].
3. Es sollen keine speziellen FPGAs und ASICs verwendet werden, die über den Funktionsumfang eines marktüblichen µCs mit Ethernet-Anbindung hinausgehen. Insbesondere soll keine spezielle Hardwareunterstützung für Ethernet POWERLINK vorausgesetzt werden.
4. Der Preis pro µC incl. Ethernet-MAC und -PHY soll nicht über 10€ liegen.
5. Die Prozessorarchitektur und möglichst große Teile der Peripherie sollten von NuttX unterstützt werden. Die Anpassung von NuttX an das konkrete System sollte mit geringem Aufwand verbunden sein.

Da die Auswahl der am Markt verfügbaren Mikrocontrollern und SoCs überwältigend ist, beschränken wir uns zunächst auf die Prozessorfamilie. NuttX wurde auf viele Mikrocontroller portiert. Dazu zählen unter anderem Intel 80x86 kompatible µC, Atmel AVR-µC, µC, die auf dem MIPS-Befehlssatz beruhen und ganze 97 µC mit ARM-Kernen in verschiedenen Varianten (Stand Dezember 2016). In der großen Zahl der unterstützten ARM-µC spiegelt sich die große und weiter wachsende Verbreitung von ARM-Architekturen im Mikrocontroller-Segment wieder [74].

Die aktuellste ARM-Variante ist die 64-Bit-Architektur *ARMv8*, die u.a als. *ARM Cortex-A57* und *Cortex-A53* in High-End-Smartphones zu finden ist. Sie wird von NuttX nicht unterstützt (Stand: November 2016). Auch die Vorgänger-Architekturen *ARMv7* (seit 2004) und *ARMv6* (seit 2002) sind sehr weit verbreitet. ARM unterscheidet seit *ARMv6* zwischen den drei Architekturprofilen *Cortex-A*, *Cortex-M* und *Cortex-R*, die teils erhebliche Unterschiede aufweisen.

Das *A* in *Cortex-A* steht für *Application*. Ihr Einsatzgebiet ist die Ausführung komplexer Anwendungen und Betriebssysteme. Eine MMU ermöglicht den Betrieb von Linux und Android, schließt die *Cortex-A*-Prozessoren aber aufgrund von Anforderung 1 für die weitere Berücksichtigung aus. Das *Cortex-M*-Profil (M = *Microcontroller*) wurde für kosteneffiziente und stromsparende eingebettete Anwendungen entworfen. Prozessoren dieses Typs eignen sich also hervorragend für den hier betrachteten Anwendungsfall.

[7] Grund ist nicht nur eine lange Laufzeit bei Akkubetrieb sondern auch eine geringe Wärmeentwicklung. Das macht kleine Bauformen möglich und Lüfter ggf. verzichtbar. Ein Raspberry PI 3 Model B [27] hat zum Vergleich eine Leistungsaufnahme von ca. 1,2W im Leerlauf und ca. 2,4W unter Last.

Tabelle 3: Hardwareeigenschaften der CN-Plattform B210E

Prozessor	Dual core ARM Cortex-M4/M0 @ 204 MHz
RAM	200KB SRAM (intern lokal) 64KB SRAM (intern via AHB)
Flash	4MB (extern via SPIFI)
Schnittstellen	Ethernet, USB Host/Device, 2xSPI, 2xI2C, 3xUART, CAN, 6xPWM, 6xADC, GPIO
Coremark [8]	33 Punkte (ext. Flash @ 22 MHz) 78 Punkte (ext. Flash @ 68 MHz) 435 Punkte (int. SRAM)
Leistungsaufnahme	1,0W (Leerlauf) 1,1W (während der Ausführung von Coremark)
Preis	41,95$ (Herstellerpreis; Stand: Januar 2017)

Das *Cortex-M*-Profil unterscheidet sich deutlich von den A- und R-Profilen. Es hat einen kleineren Befehlssatz (*Thumb* und *Thumb2*) gewöhnlich keine Caches und weniger Pipeline-Stufen [88]. Auch innerhalb des *Cortex-M*-Profils gibt es große Diversität. Mit den Varianten *M0, M0+, M1, M3, M4* und *M7* wird ein großer Leistungsbereich abgedeckt. In [48] findet sich eine detaillierte Gegenüberstellung der drei ARM Cortex-Profile.

Eine passende Plattform für Experimente mit EPL und NuttX stellt das Entwicklungsboard *Bambino 210E* von Micromint (B210E, Preis: 41.95 $, Stand: November 2016) dar [2]. Grund ist, wie in Kapitel 5.3 ersichtlich wird, vor allem der mit 264kB überdurchschnittlich große Arbeitsspeicher. Es basiert auf dem Mikrocontroller *NXP LPC4330* mit einem *ARM Cortex-M4* Kern, getaktet auf 204 MHz. Unter der reichhaltigen Peripherieausstattung findet sich auch eine Ethernetschnittstelle. Außerdem kann der Mikrocontroller auf 4MB externen Flash-Speicher zugreifen, der ausreichend Platz für NuttX und mehrere Anwendungsprogramme bietet. Tabelle 3 und Abb. 9 geben einen detaillierten Überblick über die Hardwareausstattung des B210E. Drei dieser Boards kommen im Rahmen dieser Arbeit zum Einsatz.

Ein in Kapitel 7 benötigtes viertes Modul basiert auf einer Platine, die dem B210E sehr ähnlich, jedoch als Eigenentwicklung E4337 nicht frei verkäuflich ist. Es kommt der µC *NXP LPC4337* zum Einsatz, der bis auf die Speicherausstattung baugleich mit dem *LPC4330* ist. Die Größe des internen SRAMs ist mit 136kB um 128kB kleiner. Im Gegenzug ist auf dem Chip interner Flash-Speicher in der Gesamtgröße von 1MB vorhanden. Zusätzlich gibt es 8 MB via EMC (*External Memory Controller* [61, S. 594ff]) extern angebundenen SDRAM-Speicher. In Tabelle 4 finden sich Details und Benchmarks zum E4337.

[8] Punktzahl im Benchmark *CoreMark* (Version 1.0, kompiliert mit GCC 4.9.3 -O2) [18]

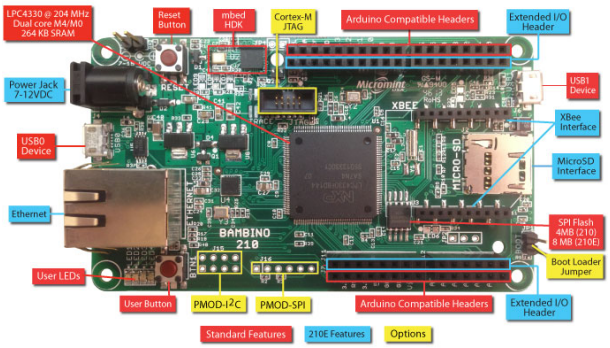


Abbildung 9: Schnittstellen der CN-Hardwareplattform B210E. Quelle: [2]

Tabelle 4: Hardwareeigenschaften der CN-Plattform E4337

Prozessor	Dual core ARM Cortex-M4/M0 @ 204 MHz
RAM	72KB SRAM (intern lokal) 64KB SRAM (intern via AHB) 8MB SDRAM (extern via EMC [61, S. 594ff])
Flash	1MB (intern)
Coremark	74 Punkte (ext. SDRAM) 435 Punkte (int. SRAM)

5. Ethernet POWERLINK mit NuttX und ARM Cortex-M4

Das Ziel dieser Arbeit ist, die praktische Umsetzbarkeit einer IE-Anwendung mit leistungsschwacher und generischer Hardware sowie einem portablen RTOS, das die Interoperabilität mit anderer Software gewährleistet, zu zeigen.

In den vorangegangen Abschnitten haben sich NuttX und Ethernet POWERLINK als geeignete Kandidaten für die Umsetzung der Ziele dieser Arbeit herauskristallisiert. In diesem Kapitel soll nun eine geeignete Umsetzungsstrategie und Implementierung gefunden werden, um daraufhin mit geringem Aufwand reale EPL-Projekte, wie den in Kapitel 7 vorgestellten 3D-Drucker, mit NuttX umsetzen zu können. Abschließend wird die Leistungsfähigkeit der Umsetzung ausführlich evaluiert.

5.1. Vorüberlegungen

Als ersten Schritt muss entschieden werden, in welcher Form einer Anwendung die EPL-Kommunikation ermöglicht werden soll.

Grundlegende Kommunikationsprotokolle sind in gängigen Unix-basierten Betriebssystemen direkt im Betriebssystem implementiert und werden über ein Socket-Interface verfügbar gemacht. Dies ist nützlich, wenn es sich um Transportprotokolle handelt, bzw. um Protokolle, die die OSI-Schichten 1–4 abdecken. Hier ist die innere Struktur der Nutzdaten anwendungsspezifisch und nicht weiter vorgegeben. Damit kann der Nutzer mit einem einheitlichen Interface (`send` und `recv`) einzelne Datagramme (z.B. bei Raw- und UDP-Sockets) oder Teile eines Datenstroms (so bei TCP) übermitteln bzw. empfangen. Besonders nützlich ist dabei die Eigenschaft, dass von verschiedenen Prozessen unabhängig voneinander auf das selbe Netzwerkgerät und in Teilen auf das selbe Transportprotokoll zugegriffen werden kann.

Ethernet POWERLINK ist jedoch kein reines Transportprotokoll. Auch die Anwendungsschicht ist fest vorgegeben und der Versand von Paketen im Netzwerk folgt exakt einem Muster. Es ist vorgesehen, dass die gesamte Anwendungsinteraktion in einem EPL-Gerät über das Objektverzeichnis stattfindet. Der Versand von einzelnen Paketen ist deswegen der EPL-Implementierung vorbehalten. Die OV-Interaktion mit Zugriffen über Index und Subindex hat wenig Ähnlichkeit mit dem Versenden und Empfangen beliebiger Daten über ein Transportprotokoll. Sie lässt sich damit auch nicht sinnvoll mit dem Socket-Interface abbilden.

Eine Ausnahme bildet der virtuelle Ethernetkanal. Dieser ist idealerweise gegenüber der Anwendung transparent und klinkt sich auf OSI Schicht 2 direkt ins System ein. Vom System bereitgestellte Protokolle oberhalb von Schicht 2 können so automatisch weiterverwendet werden.

Die naheliegende Alternative zu Sockets ist die Verwendung einer Funktionsschnittstelle. Damit lässt sich eine anwenderfreundliche und typsichere Abbildung aller Funktionalitäten erreichen, incl. Parametrisierung, Statusänderungen, OV-Zugriff und Prozessabbild-Zuordnung. Auch der openPOWERLINK-Stack verwendet für die Anwendungsinteraktion ein Funktionsinterface, bestehend aus 52 Funktionen[9].

[9] Version 2.4.1, ohne *deprecated*

Das reine Objektverzeichnis ließe sich auch elegant im Dateisystem darstellen, indem jedem OV-Eintrag eine virtuelle zeichenbasierte Gerätedatei zugeordnet wird, beispielsweise `/dev/epl/0x1018sub01` oder `/dev/epl/VendorId_U32` für den Eintrag VendorId_U32 an der Adresse `0x1018` (Subindex 1). Vorteile wären die Programmiersprachenunabhängigkeit und die systemweite Verfügbarkeit, nachteilig ist jedoch, dass für die zeichenorientierte Schnittstelle die interne Repräsentation der Datenwerte beachtet werden muss. Diese Variante lässt sich leicht aufbauend auf einem Funktionsinterface implementieren, ersetzt es jedoch nicht in Gänze. Deswegen soll fortan letzteres forciert werden.

Letztlich muss noch entschieden werden, ob eine gänzlich neue Implementierung des EPL-Protokolls erfolgen soll, oder ob auf vorhandene Arbeiten aufgebaut werden kann. Mit openPOWERLINK existiert derzeit (Stand: Januar 2017) nur eine einzige nahezu vollständige und quelloffene Implementierung des EPL-Protokolls. Darüber hinaus existieren einige software- und hardwarebasierte Lösungen, z.B. [66], [65], [37] und [75]. Diese finden hier aufgrund des Widerspruchs zu den Zielsetzungen aus Kapitel 1 keine Berücksichtigung.

Eine rein softwarebasierte EPL-Implementierung ist sehr umfangreich. Im Vergleich zu CANopen, das als Vorbild für große Teile von EPL gilt, ist wesentlich mehr spezifizierte Funktionalität zu berücksichtigen. Schon die Spezifikation des Grundprotokolls ist mit 356 Seiten (*EPSG DS 301 V1.3.0*) mehr als doppelt so umfangreich, wie das CANopen-Gegenstück mit 158 Seiten (*CiA 301 V4.2.0*). Einige Beispiele für den größeren Umfang sind die exakte Spezifikation des Zyklus und der Sendereihenfolge, die exakte Definition des Zustandsautomaten und der Knoten-Konfiguration, erweiterte OV-Interaktion mit Zugriff über Feldname, Stapelzugriff und Unterstützung für Binärdateien, standardisierter Konfigurationsprozess des Netzwerks beim Hochfahren, virtueller Ethernetkanal, und vieles mehr.

Dementsprechend umfasst der openPOWERLINK-Stack (OPLK) über 100.000 reine Code-Zeilen (physikalische *Single Lines of Code*, SLOC [14])[10]. Er unterstützt nicht nur das Basisprotokoll (*EPSG DS 301*) sondern auch die Erweiterungen *EPSG DS 302-A*, *302-B*, *302-C* und *302-F*. Eine ebenbürtige Eigenentwicklung diesen Umfangs erscheint deswegen nur dann sinnvoll, wenn dafür triftige Gründe vorliegen.

Glücklicherweise ist OPLK mit seinem modularen Aufbau bestens geeignet für eine Portierung auf neue Plattformen, was bereits für Linux und Windows (x86/x86-64) und für mindestens vier eingebettete Plattformen ohne Betriebssystem durchgeführt wurde. Im folgenden Abschnitt folgt deshalb eine detaillierte Einführung in die Architektur von openPOWERLINK sowie eine Erläuterung der nötigen Änderungen, um einen OPLK-basierten Controlled Node unter NuttX auf einem ARM Cortex-M-Mikrocontroller betreiben zu können.

[10] Version 2.4.1, gemessen mit sloccount 2.26 [82]

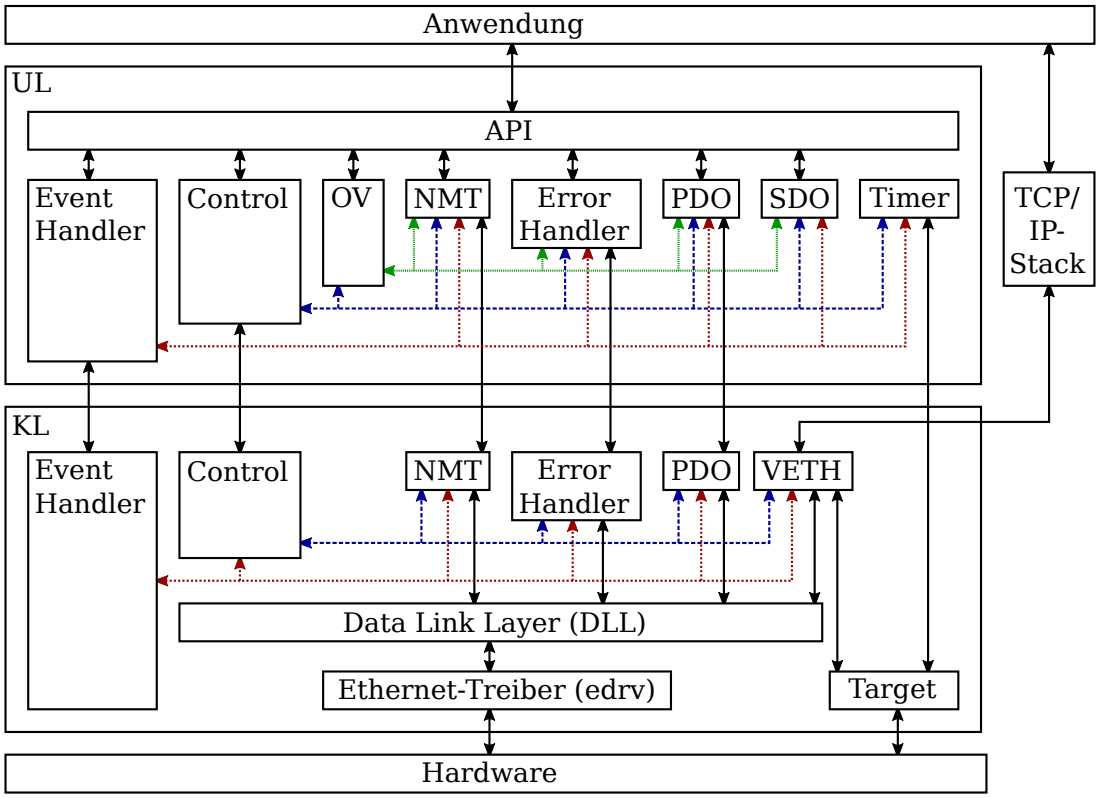


Abbildung 10: Architektur von openPOWERLINK aus Sicht eines Controlled Nodes. Angelehnt an [63].

5.2. openPOWERLINK

5.2.1. Architektur

Der openPOWERLINK-Stack ist modular aufgebaut. Alle Bestandteile sind, wie in Abb. 10 dargestellt, strikt in einen *User Layer* (UL) und in einen *Kernel Layer* (KL) eingeteilt. Diese Trennung wurde in Version 2.0 eingeführt, nachdem der Stack zuvor nur mit speziellen Treibern als Linux-Kernel-Modul lief und in [80] für eine plattformunabhängige Anwendung in den Userspace portiert wurde [81].

openPOWERLINK erlaubt durch die Auswahl verschiedener Quelldateien in den Makefiles eine Form der Compile-Time-Polymorphie. So können verschiedene Bausteine ausgewählt werden, die zur eigenen Plattform passen, es können neue hinzugefügt werden, und nicht benötigte Teile können ausgelassen werden (bspw. MN-Funktionalitäten oder virtuelles Ethernet).

Die Abstraktionsschicht, die UL und KL trennt, wird *Communication Abstraction Layer* (CAL) genannt. Je nach Plattform bzw. Konfiguration kann diese Abstraktionsschicht eine Hürde darstellen, die mit speziellen Kommunikationskanälen umgangen werden muss und unter Umständen mit Laufzeitkosten verbunden ist. Wird beispiels-

weise auf einem x86-System der KL als Linux-Kernelmodul kompiliert und der UL als Laufzeitbibliothek im Userspace betrieben, hat jeder Wechsel des Kontrollflusses von einer in die andere Schicht einen Kontextwechsel zur Folge. Es sind dann auch keine direkten Aufrufe einer Funktion aus der anderen Schicht möglich. Es gibt im Allgemeinen verschiedene Möglichkeiten, die Grenze zwischen UL und KL zu überwinden:

IOCTL IOCTL steht für *Input/Output Control* und ist eine Möglichkeit, Geräte- bzw. Treiberspezifische Operationen im Betriebssystemkernel auszuführen, für die kein eigener Systemaufruf vorgesehen ist. Die Funktion ist in den meisten Unix-basierten Systemen als `ioctl()` vorhanden. Windows bietet mit `DeviceIoControl` eine analoge Funktionalität. In openPOWERLINK steht diese Schnittstelle zur Verfügung, wenn der KL als Kernelmodul kompiliert wird.

Memory Mapping *Memory Mapping* (`mmap`) ist ein Verfahren, bei dem eine Datei unmittelbar in den Adressbereich des Programms eingeblendet wird. Diese Datei muss jedoch keine Datei im Dateisystem sein, sondern kann auch ein Gerätetreiber oder anonym sein. Im Falle des Gerätetreibers implementiert dieser eine Funktion, mit der ein beliebiger Speicherbereich auf die vom Nutzer gewünschte Adresse abgebildet werden kann. Wenn größere Datenmengen ausgetauscht werden, ermöglicht `mmap` eine sehr effiziente Form der Interprozesskommunikation (IPC), bei der Kopieroperationen vermieden werden, womit auch die Cache-Effizienz erheblich verbessert wird. Es sei darauf hingewiesen, dass die Verwendung von `mmap` bei kleineren Datenmengen nachteilig sein kann, da sich der zugeordnete Speicherbereich an den Seitengrenzen orientiert und damit immer ein Vielfaches der Seitengröße beträgt. Der OPLK-Stack verwendet diese Methode deswegen ausschließlich für den Austausch von PDO-Nutzdaten.

POSIX Shared Memory *POSIX Shared Memory* (SHM) ist dem Memory Mapping sehr ähnlich, mit dem Unterschied, dass ein Gerätetreiber hierfür keine spezielle Schnittstelle implementiert. Stattdessen öffnen alle Komponenten, die auf den Speicherbereich zugreifen wollen, mit `shm_open` den selben SHM-Bereich, welcher durch einen eindeutigen Namen identifiziert wird. Man erhält einen sog. *File Descriptor*, auf den entweder mit `read` und `write` oder wiederum via Memory Mapping zugegriffen werden kann. Fast alle CAL-Module von OPLK unterstützen diese Art der Kommunikation.

Funktionsaufruf Direkte Funktionsaufrufe stehen nur zur Verfügung, wenn sich beide Seiten im selben Modus befinden, also beide im Kernel- oder im Usermode. Und ohne Umwege (d.h. ohne `dlopen`/`LoadLibrary`) auch nur dann, wenn KL und UL beide in eine einzige Bibliothek kompiliert werden. Bei der Übersetzung in eine einzelne Bibliothek ergibt sich zudem weiterer Spielraum für Compiler-Optimierungen, wie z.B. Inlining.

Hardware-Interface Aus Performance-Gründen kann die Verarbeitung des Kernel Layers in Hardware erfolgen. Hierfür existieren neben frei programmierbaren FPGAs auch EPL-spezifische Lösungen von verschiedenen Herstellern (z.B. [37] [75]). Im OPLK-Stack gibt

es zwei Varianten, wie die Kommunikation mit der Hardware erfolgen kann. Bei eingebetteten Systemen, die einen FPGA für den KL verwenden, wird das *Host Interface* eingesetzt. An die Stelle von IOCTL, Memory Mapping, etc. treten dann spezielle Kommandos an einen Host Interface-Treiber. Es werden direkte Funktionsaufrufe verwendet, deswegen kann diese Methode nicht bei Trennung von KL und UL verwendet werden, was jedoch in eingebetteten Systemen auch nicht üblich ist. In (Industrie-)PCs können alternativ PCIe-Interfacekarten für Ethernet POWERLINK eingesetzt werden. Dabei wird ein angepasstes Kernelmodul eingesetzt, das einige der Kommandos über PCIe an die Hardware weiterreicht.

An vielen Stellen des CAL kann ein Ringpuffer eingesetzt werden, um die Auslastung zu erhöhen und um kurzfristige Belastungen abzufedern. Im Stack existiert hierfür eine eigene Ringpufferbibliothek, die analog zu den oben erörterten Schnittstellen in verschiedenen Varianten vorliegt. Es ist zu beachten, dass der Einsatz eines Ringpuffers das Debugging erheblich erschwert, da Ursache und Wirkung eines Fehlers so u.U. nicht durch einen einzelnen Kontrollfluss verbunden sind.

5.2.2. Module und deren Anpassung an NuttX

Die Portierung des openPOWERLINK-Stacks erfordert die geeignete Auswahl der Modulvarianten sowie die Neuentwicklung einiger Komponenten. Der Stack besteht im Wesentlichen aus folgenden Teilen:

Data Link Layer Der *Data Link Layer* (DLL) übernimmt den Großteil der Protokollverarbeitung. Das Modul allein besteht aus über 7.000 SLOC[11]. Hier werden alle Pakete angenommen, analysiert, delegiert und verarbeitet. Zu sendende Pakete werden hier zusammengebaut und in die Sendewarteschlange eingereiht. Hier wird auch der gesamte Zustand der EPL-Kommunikation festgehalten. Verschiedene Pakete, wie z.B. *Ident-* und *Status Response*, werden vorberechnet und in einem Puffer festgehalten, um schneller auf Anfragen reagieren zu können. Der *Data Link Layer* stellt eine eigene kleine Zustandsmaschine dar, um den verschiedenen Phasen der EPL-Kommunikation gerecht zu werden.

Das DLL-Modul besteht aus einem KL- und einem UL-Teil. Im KL findet die eigentliche EPL-Protokollverarbeitung statt, an einigen Stellen ist es jedoch nötig, aus dem UL Einfluss auf die Kommunikation zu nehmen, sei es wegen eines Sendewunsches für ein beliebiges Ethernet-Paket, einer Status- oder Identifikationsanfrage oder für die Änderung von Kommunikationsparametern. Die Kommunikation zwischen den Teilen findet zum einen mithilfe des Event-Layers statt und zum anderen durch vier unterschiedlich priorisierte Sende-Warteschlangen für NMT-, allgemeine EPL-, Sync- und VETH-Pakete.

OPLK erlaubt den Transfer der Datenpakete zw. UL und KL standardmäßig via IOCTL, via Funktionsaufruf oder über einen Ringpuffer. Da NuttX keinen Speicher-

[11] Version 2.4.1, gemessen mit sloccount 2.26 [82], Verzeichnisse `stack/src/kernel/dll`, `stack/src/user/dll` und `stack/src/common/dll`

schutz kennt und es die beste Leistung verspricht, wird für die NuttX-Portierung die Variante mit direkten Funktionsaufrufen verwendet (`dllcal-direct`).

Control Dieses Modul übernimmt die Initialisierung und alle vom Nutzer veranlassten Zustandsänderungen des OPLK-Stacks. Außerdem findet hier die Verknüpfung verschiedener Module statt. Ein Beispiel für eine solche Verknüpfung ist, dass nach dem NMT-Befehl `ResetApplication` das Zurücksetzen einiger Einträge im Objektverzeichnis veranlasst werden muss. Entscheidend ist auch, dass hier auf Fehlermeldungen aus dem Event-Modul reagiert wird. Das *Control*-Modul besteht aus einem KL- und einem UL-Teil. Zur Verknüpfung stehen alle in Abschnitt 5.2.1 genannten Varianten zur Verfügung. Für die NuttX-Portierung greifen wir aus bereits genannten Gründen auf die Variante mit direkten Funktionsaufrufen zurück.

Error Handler Der Error Handler bildet eine zentrale Stelle, wo Fehler in der Protokollausführung gemeldet werden können. Dabei werden Toleranzen berücksichtigt, wie sie im EPL-Standard vorgesehen und im Objektverzeichnis hinterlegt sind. Beispielsweise führt bei entsprechender Konfiguration das Fehlen eines `PRes`-Pakets erst nach mehrmaligem Auftreten zur Behandlung des Fehlers. Die Fehler werden an das Event-Modul weitergeleitet, sodass auch andere Teile des Stacks darauf reagieren können. Auch der Error Handler ist in KL und UL geteilt und steht in den Varianten `IOCTL`, `SHM`, `Host Interface` und `Direct` zur Verfügung. Die Direct-Variante verwendet eine statische Variable für die Speicherung von Fehlerobjekten, auf die aus beiden Schichten zugegriffen wird. Aus zur Verwendung von direkten Funktionsaufrufen analogen Gründen verwendet die NuttX-Portierung die Direct-Variante.

Event Handler Durch Verwendung von Polling ist Ethernet POWERLINK ein ereignisorientiertes Protokoll. Das spiegelt sich auch in der Implementierung des openPOWERLINK-Stacks wieder, in der eine zentrale Ereignisverwaltung eine wichtige Rolle spielt. Fast alle Verknüpfungen im Stack werden mit einem Event-Objekt realisiert, das den *Event Handler* passiert. Das Modul enthält vier unabhängige Ringpuffer, jeweils einen für Events, die innerhalb von KL und UL weitergeleitet werden und jeweils einen für die Richtungen KL $\rightarrow$ UL und UL $\rightarrow$ KL. Durch die Verwendung von Ringpuffern wird eine lose Kopplung zwischen UL und KL erreicht, sodass der KL nicht unmittelbar auf den ggf. langsameren UL warten muss. Das hat jedoch einen erheblichen Nachteil. Ursache und Wirkung eines ggf. auftretenden Fehlers sind so nicht durch einen einzelnen Kontrollfluss verbunden. Das erschwert das Debugging mithilfe von beliebten Hilfsmitteln wie Backtracing. Das ist insbesondere deswegen wichtig, da Fehler im OPLK-Stack üblicherweise durch entsprechende Rückgabewerte signalisiert werden, die oft wie folgt kommentarlos zum Abbruch führen: `if (ret != kErrorOk) goto Exit;`

Ethernet-Treiber Im OPLK-Stack übernimmt der Ethernet-Treiber (`edrv`) die Rolle Schnittstelle zur Netzwerkhardware und steht aufgrund der hier sehr großen Plattformdiversität in vielen Varianten bereit. Die auf PCs am einfachsten zu verwendende Variante

ist `edrv-pcap`. Diese verwendet die *PCAP*-Bibliothek [78], die auf verschiedenen Plattformen (Linux, Windows und weitere) eine einfache und einheitliche Möglichkeit bietet, Ethernet-Pakete mit einem Userspace-Programm zu senden und zu empfangen. Durch die Verwendung der Betriebssystem-eigenen Netzwerkschnittstellen ist openPOWERLINK damit auf nahezu allen PCs lauffähig.

Die Verwendung von *PCAP* hat jedoch den Nachteil, dass beim Transfer der Paketdaten vom Kernel in den Userspace ein Kontextwechsel und i.A. Kopieren stattfindet und damit wertvolle Zeit für die Paketbearbeitung und Reaktion verloren geht. Auch ist solch ein Kontextwechsel eine Quelle für Jitter. OPLK bietet deshalb eine Hand voll spezieller Ethernet-Treiber an, die ohne Umwege und unmittelbar mit der Netzwerkhardware kommunizieren. Da der gesamte Netzwerk-Stack des Betriebssystems umgangen wird, ist damit die kleinstmögliche Verzögerung möglich. In Abschnitt 6.1 findet sich hierzu ein Vergleichstest. In Version 2.4.1 werden die Chips Realtex 8111 und 8139, Intel 8255x, 82573 und I210, die Ethernet-MACs verschiedener SoCs und der openMAC IP-Core [33, S.8–9] unterstützt.

Auf Windows ist auch im Kernel ein hardwareunabhängiger Betrieb des OPLK-Stacks möglich. Dort wird ein *Network Driver Interface Specification* (NDIS [79]) *Intermediate Driver* [9] eingesetzt, der über eine standardisierte Schnittstelle Zugriff auf die Netzwerkhardware hat.

Auch in Linux gibt es weitere Alternativen. So lässt sich der Aufwand der Übertragung in den Userspace stark verringern, wenn Memory Mapping verwendet wird, statt Pakete zu kopieren. Aktuelle Linux-Kernel bieten hierfür die Raw-Socket-Option `PACKET_TX_RING`/`PACKET_RX_RING` an, die je nach Konfiguration auch von *PCAP* verwendet wird. Außerdem ist mit `dev_add_pack` ein Zugriff auf Pakete direkt im Kernel möglich. Eine entsprechende Implementierung in OPLK könnte Gegenstand zukünftiger Entwicklungen sein.

Für NuttX wurde eigens ein `edrv-nuttx`-Modul erstellt. Wie in Kapitel 3.2.2 erläutert, ist es leider nicht möglich, Raw-Sockets für diese Aufgabe zu verwenden. Stattdessen wird der eigens entwickelte `bufpkt`-Treiber eingesetzt (vgl. Kapitel 3.2.4). Das Modul empfängt in einem eigenen Thread fortlaufend Pakete über `bufpkt_receive` und leitet sie an den Stack weiter. Ausgehende Pakete werden über `bufpkt_send` gesendet.

Network Management Das *Network Management*-Modul (NMT) implementiert die Zustandsmaschine des EPL-Knotens. Beim Betrieb als Managing Node wird auch der Zustand der Controlled Nodes überwacht. Es existiert ein KL- und ein UL-Teil. Die Teile kommunizieren nicht direkt miteinander, nur indirekt über das Event-Modul. Das Modul kann deshalb unverändert in der NuttX-Portierung weiterverwendet werden.

PDO Die Durchführung der PDO-Kommunikation ist eine äußerst zeitkritische Aufgabe. In der synchronen Phase müssen fortlaufend Teile der PDO-Nachricht dem korrekten PDO-Kanal bzw. dem passenden OV-Eintrag zugeordnet werden. Die Daten müssen zum Synchronisationszeitpunkt immer aktuell sein und in jedem Zyklus zwischen Kernel und Anwendung ausgetauscht werden. Der Empfang einer Poll Request muss nach möglichst

kleiner Verzögerung zum Versenden dieser Daten führen. Die Bearbeitung empfangener Daten muss zum nächsten Synchronistationszeitpunkt abgeschlossen sein.

Dedizierte EPL-Chips können deshalb meist einen großen Teil der PDO-Verarbeitung in Hardware ausführen. Entsprechende Adaptervarianten des PDO-Moduls (vgl. Abschnitt 5.2.1) sind vorhanden. Das PDO-Modul benötigt einen gemeinsamen Speicherbereich zwischen KL und UL. In der *local*-Varante, die bei der NuttX-Portierung zum Einsatz kommt, ist dieser ohne Umwege über eine globale Variable erreichbar.

Der Speicherbereich wird für einen Dreifachpuffer (`pdokcal-triplebufshm`) verwendet. Er sorgt dafür, dass Schreib- und Leseoperationen sowie die eigentliche Übertragung der als PDO zyklisch übertragenen Daten entkoppelt sind, indem die Rolle der Puffer (Lese-/Schreibpuffer/leer) jeweils atomar getauscht wird. Beim Übernehmen des Prozessabbildes aus der Anwendung heraus mit `oplk_exchangeProcessImageIn` bzw. `oplk_exchangeProcessImageOut` wird genau auf diesen Dreifachpuffer zurückgegriffen.

Timer Der openPOWERLINK-Stack unterscheidet zwischen zwei verschiedenen Zeitgebern. *High Resolution Timer* (`hrestimer`) werden beim Betrieb als MN benötigt, um den Zyklus exakt in gleichbleibenden Zeitabständen mit dem Versand der `SoC`-Nachricht zu beginnen (wichtig für niedrigen *Jitter*). Es existiert eine ganze Reihe verschiedener `hrestimer`-Implementierungen in OPLK, die entweder hochauflösende Kerneltimer verwenden, wie sie u. a. in Linux und Windows verfügbar sind, oder direkt konkrete Hardwaretimer ansprechen. *User Timer* werden ausschließlich im UL verwendet, z.B. um im NMT-Modul nach definierten Zeitabständen Zustandswechsel zu veranlassen (vgl. OV-Eintrag `NMT_CNBasicEthernetTimeout_U32`). Neben betriebssystemabhängigen Varianten gibt es hier ein generisches, auch für NuttX geeignetes Modul, das entweder in einem Timer-Thread oder durch regelmäßiges Aufrufen einer Funktion die vergangene Zeit misst und ggf. eine Reaktion auslöst. Die vergangene Zeit wird durch eine Funktion im *Target*-Modul bereitgestellt.

Virtual Ethernet *Virtual Ethernet* (VETH) ist ein optionales Modul und ist als Gateway für beliebige Netzwerk- bzw. Internetsoftware gedacht, sodass diese über ein Powerlink-Netzwerk kommunizieren kann. Wie in Kapitel 2.4.2 erläutert, geschieht dieser Netzwerkverkehr in der asynchronen Phase, nachdem die Erlaubnis dazu erteilt wurde. Aufgabe des VETH-Moduls ist also, von Fremdsoftware Netzwerkpakete entgegenzunehmen, sie in einer Warteschlange abzulegen, den Sendewunsch zu melden, und die Pakete dann im richtigen Moment abzuschicken. Umgekehrt müssen eingehende Nicht-Powerlink-Pakete erkannt werden und unverändert an die Anwendung weitergereicht werden. Dies soll möglichst so geschehen, dass keine Änderungen an der Fremdsoftware nötig sind, d.h. dass das Gateway völlig transparent erscheint.

OPLK bietet in Version 2.4.1 u. a. folgende Varianten: `linuxkernel`, `linuxuser`, `linuxpcie` und `ndisintermediate`. `linuxkernel` bindet ein virtuelles Netzwerkgerät mit `netdev_register` direkt im Linux-Kerne ein. Bei der `linuxuser`-Variante wird dagegen das TUN/TAP-Subsystem[12] [16] verwendet, mit dem solche Netzwerkgeräte im

Userspace implementiert werden können. `linuxpcie` überlässt die VETH-Behandlung einer EPL-PCIe-Karte und `ndisintermediate` ist die Windows-Kernel-Implementierung.

Für die Portierung auf NuttX bietet sich an, entweder analog zur `edrv`-Implementierung einen Raw-Socket nachzubilden, oder auf den TUN-Treiber von NuttX aufzubauen. Letzteres liefert die sauberere, da generische Lösung. Bei VETH handelt es sich jedoch um eine OSI-Layer 2-Angelegenheit, der TUN-Treiber ist aber – wie auch in Linux – für IP-Kommunikation (OSI-Layer 3) ausgelegt. Deswegen wird im Rahmen dieser Arbeit ein TAP-Treiber erstellt, der bis auf wenige Zeilen identisch mit dem TUN-Treiber ist. Die wesentlichen Unterschiede sind der Verzicht auf die IP-Konfiguration und die korrekte Registrierung als Netzwerkgerät in Nuttx[13].

Für einen reibungslosen Betrieb müssen unbedingt folgende Bedingungen erfüllt werden:

1. Die MAC-Adresse des virtuellen Netzwerkgeräts muss der MAC-Adresse der realen Ethernet-Schnittstelle entsprechen. Andernfalls sind bei ARP (*Address Resoultion Protocol*) die Informationen in Header und Nutzdaten nicht konsistent. Das kann die TCP/IP-Kommunikation erheblich stören. In NuttX wird der Adressabgleich mit den Funktionen `netlib_getmacaddr` und `netlib_setmacaddr` durchgeführt.

2. Die reale Ethernet-Schnittstelle darf keine IP-Adresse aufweisen. Ankommende IP-Pakete werden so verworfen. Das ist notwendig, da andernfalls über den virtuellen Ethernetkanal versendete IP-Pakete zweimal im System ankommen würden. Zum einen auf dem Weg *Ethernettreiber* → ***pkt_input*** → ***bufpkt*** → *OPLK* → *TAP* → ***ipv4_input*** → *System* und zum anderen auf dem Weg *Ethernettreiber* → ***ipv4_input*** → *System*.

Target Im *Target*-Modul werden einige Betriebssystem- und architekturabhängige Helferfunktionen zusammengefasst. Dazu zählen Mutexe, Zeitstempel, Sleep-Befehl, Debug-LED-Ansteuerung und Funktionen zum Setzen der IP-Adresse und des IP-Gateways. Alle Funktionen wurden entsprechend für NuttX implementiert.

Objektverzeichnis Das Objektverzeichnis ist in OPLK statisch alloziert. Es wird eine Header-Datei inkludiert, in der mithilfe von Makros die Struktur des OVs festgelegt ist. Die vier OV-Bereiche *Generisch* (0x1000 – 0x1FFF), *herstellerspezifisch* (0x2000 – 0x5FFF), *standardisierte Geräteprofile und Schnittstellen* (0x6000 – 0x9FFF bzw. 0xA000 – 0xBFFF) befinden sich jeweils in einem eigenen Speicherbereich. Beim Zugriff auf einen OV-Eintrag wird zunächst anhand des Index der Bereich bestimmt, von dem die Speicheradresse und die Anzahl der Einträge bekannt ist. Mittels binärer Suche wird dann die Adresse des konkreten Eintrags bestimmt, woraufhin der Eintrag gelesen oder geändert werden kann.

[12] TUN steht für *Network Tunnel* und arbeitet auf OSI-Layer 3, TAP steht für *Network Tap* und arbeitet auf OSI-Layer 2.

[13] neuer Netzwerkgerätetyp `NET_LL_TAP` statt `NET_LL_TUN` mit Anpassung von `netdev_register`

Grund für die Einbettung im Programmcode ist die Kompatibilität mit Bare-Metal-Implementierungen, denn es wird kein Dateisystem benötigt, um etwa eine OV-Struktur aus einer Datei zu laden. Zukünftige Entwicklungen könnten OPLK jedoch dahingehend erweitern und die Anwendungsentwicklung damit erheblich flexibler gestalten. Technisch ist das möglich, da der Zugriff auf das statische OV über einen zur Laufzeit festgelegten Zeiger stattfindet.

SDO Das SDO-Modul kümmert sich um die SDO-Kommunikation und besteht nur aus einem UL-Teil. Es sind deshalb keine betriebssystemabhänigen IPC-Varianten nötig. Lediglich wenn Unterstützung für *SDO over UDP* gewünscht ist, kommt ein plattformabhängiger Quelltext zum Einsatz. Hier kann jedoch die Linux-Variante `sdoudp-linux` nahezu unverändert für NuttX übernommen werden. Sie verwendet einen Datagram-Socket, wie er auch in NuttX verfügbar ist.

5.2.3. Threads

Im OPLK-Stack müssen einige Aufgaben zeitlich entkoppelt und unabhängig voneinander ausgeführt werden. Die folgende Aufzählung ergibt einen Überblick. In Klammern ist der Thread angegeben, der die Aufgabe übernimmt.

- Empfang von Paketen im Ethernet Treiber (T_{edrv})
- Bearbeitung von Timer-Events, jeweils für *User Timer* und `hrestimer` (nur MN) (T_{timer} und $T_{\text{hrestimer}}$)
- Entgegennehmen von Event-Objekten aus den Ringpuffern jeweils in KL und UL (T_{eventk} und T_{eventu})
- Das Anwendungsprogramm (T_{main})
- optional Entgegennehmen von Paketen im VETH-Treiber (T_{VETH})
- optional Entgegennehmen von SDO over UDP-Paketen (T_{SDO})

5.3. Speicherbedarf

Auch nach der Portierung von openPOWERLINK auf NuttX, müssen noch erhebliche Anstrengungen unternommen werden, um einen darauf aufbauenden Controlled Node auf den Plattformen B210E und E4337 (vgl. Kapitel 4) lauffähig zu machen. Das größte Problem ist der Speicherverbrauch. Allein das Beispielprogramm `demo_cn_console`, das in OPLK mitgeliefert wird und ein Gerät mit 8 digitalen Ein- und Ausgängen simuliert, belegt 777098 Bytes und verwendet im Betrieb mindestens 4252KB Arbeitsspeicher. Verfügbar sind jedoch nur maximal 264kB auf dem B210E bzw. nur 136kB auf dem E4337. Dieser Arbeitsspeicher wird außerdem auch für etliche Betriebssystemsfunktionen verwendet und es besteht ein Aufteilungskonflikt zwischen Stacks fixer Größe, Heap und statischen Variablen, der Fragmentierung und mangelnde Auslastung verursachen

kann. Der Programmcode selbst konkurriert dagegen nicht um diesen Speicherbereich. Er befindet sich beim B210E im externen Flash-Speicher, beim E4337 im externen RAM[14].

Im Folgenden sollen die Maßnahmen im Einzelnen erläutert werden, die für eine erhebliche Reduktion des Bedarfs an Arbeitsspeicher sorgen.

Reduzierung der Threadanzahl Auch wenn in OPLK viele Aufgaben logisch und zeitlich voneinander unabhängig erledigt werden müssen, so ist ihre Ausführung auf einem Mikrocontroller mit nur einem Prozessorkern doch sequentiell. NuttX setzt hierfür einen Round-Robin-Scheduler ein, der den Prozessor alle 200ms[15] einem anderen Prozess oder Thread zuordnet. Jeder Thread hat dabei einen eigenen Speicherbereich als Stack reserviert, damit Zustand bei einem Wechsel erhalten bleibt. Unter bestimmten Voraussetzungen kann der Speicherplatz eines Stacks gespart werden, indem zwei Threads T_1 und T_2 durch einen Thread T_3 ersetzt werden. T_1 und T_2 müssen wie folgt darstellbar sein:

`while (1) { process_`T_i`(); }`

Außerdem dürfen `process_`T_1`()` und `process_`T_2`()` nicht blockierend auf ein Ereignis des jeweils anderen warten. Dann ersetzt ein Thread T_3 auf folgende Weise die beiden anderen Threads:

`while (1) { process_`T_1`(); process_`T_2`(); }`

Diese Methode wird angewandt, um die Threads T_{eventk}, T_{eventu} und T_{timer} zu einem gemeinsamen Thread T_{oplk} zusammenzufassen. T_{oplk} ist Teil der Anwendungsbibliothek `oplkcnlib`, die in Abschnitt 7.4 vorgestellt wird.

Stack- und Puffergrößen Bei der Erzeugung eines Threads mittels `pthread_create` wird in NuttX ein sog. *Thread Control Block* sowie ein Stack im Heap alloziert. Die Größe des Stacks ist über die gesamte Laufzeit des Threads konstant. Sie sollte deswegen mit Bedacht gewählt werden. Ein zu großer Stack verschwendet dauerhaft wertvollen Arbeitsspeicher, während ein zu kleiner Stack zu Abstürzen führen kann. Als Strategie zur Optimierung der Stackgrößen kann entweder iteratives Verkleinern angewandt werden, oder man benutzt Hilfsmittel, wie *Stack Coloration*, bei der Stacks bei der Erstellung mit einem leicht erkennbaren Muster initialisiert werden. Nach Beendigung des Programms kann dann an der Größe des Bereichs, in dem das Muster unverändert geblieben ist, das Verkleinerungspotential abgelesen werden. Besondere Vorsicht ist geboten bei der Verwendung von Rekursion, da die benötigte Stackgröße stark von der Rekursionstiefe abhängig ist. Die in Tabelle 5 aufgelisteten Stack-Größen erweisen sich als ausreichend und minimal[16] für einen stabilen Betrieb in den Anwendungsszenarien aus den Kapiteln 5.5 und 7.7. T_{idle} steht für den Idle-Thread des NuttX-Schedulers.

[14] Ein Bootloader kopiert den Programmcode beim Hochfahren vom Flash-Speicher in den externen RAM.

[15] konfigurierbar mit `CONFIG_RR_INTERVAL`

[16] $\pm$ 1KB

Tabelle 5: Stackgrößen in openPOWERLINK und NuttX

T_{idle}	512B
T_{main}	2kB
T_{oplk}	6kB
T_{edrv}	1kB
T_{VETH}	1kB
T_{SDO}	deaktiviert

Tabelle 6: Puffergrößen in der OPLK-Portierung

`bufpkt` Paketdaten	512B
`bufpkt` Paketgrößen	$16 \cdot 4$B
Events KL $\rightarrow$ UL	512B (Standard: 32kB)
Events KL $\rightarrow$ KL	32B (Standard: 32kB)
Events UL $\rightarrow$ KL	deaktiviert (Standard: 32kB)
Events UL $\rightarrow$ UL	deaktiviert (Standard: 32kB)
DLL Sendepuffer (NMT/Generic/Sync/VETH)	deaktiviert (Standard: 32/32/8/8kB)
DLL Requests (NMT/Generic/Ident/Status)	nur MN (Standard: je 32kB)

Eine Optimierung des Speicherverbrauchs ist auch durch die Verkleinerung etwaiger Puffer möglich. Pufferüberläufe sind dabei dringend zu vermeiden, da dann in der Regel Datenverlust droht und die Kommunikation gestört wird. Die minimalen Puffergrößen werden durch fortlaufende Registrierung des maximalen Füllstandes in realen Anwendungsszenarien ermittelt. Eine entsprechende Logging-Funktionalität wurde, wo noch nicht vorhanden, implementiert. Tabelle 6 listet die Größen der relevanten Puffer auf. Einige der genannten Puffer sind schon aufgrund der Modulwahl in Abschnitt 5.2.2 verzichtbar, da die Daten stattdessen direkt weitergeleitet werden.

5.4. Speicheranbindung

Eine wichtige Erkenntnis ist, dass ein in Software implementierter EPL-Stack auf der ARM Cortex-M-Architektur in hohem Maße speichergebunden ist (engl. *memory bound*).

Das liegt zum einen am Protokoll, bei dem in jedem Zyklus eine große Menge an Daten empfangen und damit in den Speicher geschrieben werden muss. Nur `PReq`-Pakete sind direkt an einen einzelnen Controlled Node gerichtet und können bei anderen CNs – wenn entsprechend konfiguriert – schon im Ethernet-Chip ausgefiltert werden. Alle anderen Pakete werden als Multicast an alle Netzwerkteilnehmer gesendet.

Am Beispiel des 3D-Druckers aus Kapitel 7 lässt sich wie folgt zeigen, welche Datenmenge mindestens pro Sekunde in den Speicher transferiert wird. Nach *IEEE 802.3* ist ein Ethernet-Frame (ohne CRC) mindestens 60 Bytes lang, wovon bis zu 46 Bytes als Nutzdaten verfügbar sind. Im 3D-Drucker ist diese Menge ausreichend für die Pakettypen `SoC`, `PReq`, `PRes`, `SoA` und `ASnd`/SDO, welche somit die selbe Länge aufwei-

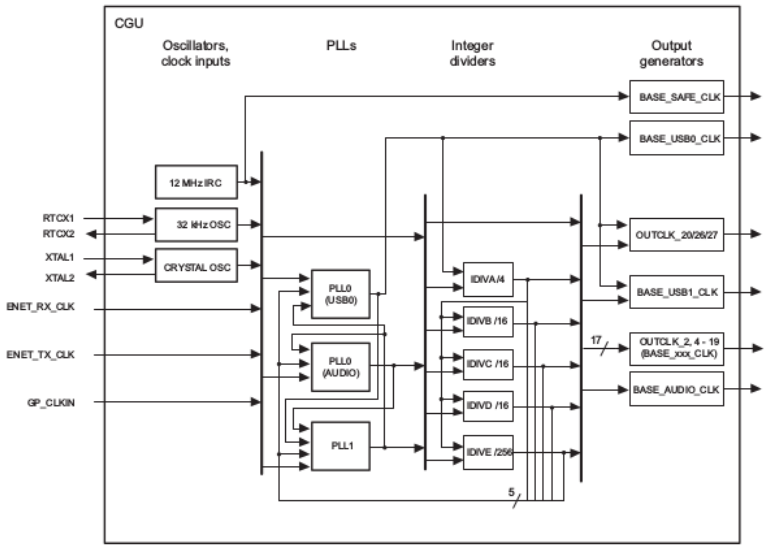


Abbildung 11: LPC43xx Clock Generation Unit. Quelle: [61, S. 171]

sen. Eine *Status Response* ist $L_{\mathrm{ASnd/Status}} = 72$ Bytes lang, eine *Ident Response* enthält $L_{\mathrm{ASnd/Ident}} = 176$ Bytes. Bei vier CNs und einer Zykluszeit von 8ms ergibt sich somit eine Datenrate von bis zu $\frac{1}{8ms} \cdot (L_{\mathrm{Soc}} + 3 \cdot L_{\mathrm{PRes}} + L_{\mathrm{SoA}} + \max L_{\mathrm{ASnd}}) = 59500 \frac{B}{s}$. Hinzu kommen das Zusammenstellen der eigenen `PRes`-Antwort ($60B$), der Austausch der Prozessabbilder ($19B + 13B$) und viele weitere Kopieroperationen im openPOWERLINK-Stack. Die genannten Schreib- und Leseoperationen finden hauptsächlich in Stack und Heap statt. Diese Bereiche sind deswegen im schnellen internen SRAM abgelegt.

Zum anderen enthält der verwendete Cortex-M4-Kern anders als ARM Cortex-A- und PC-Prozessoren keine Caches, weswegen der Mikrocontroller bei Schreib- und Leseoperationen unmittelbar auf den Speicher warten muss. Es sei hierbei angemerkt, dass auch für das reine Ausführen von Programmcode die Befehle zunächst aus dem Speicher gelesen werden müssen. Der Programmcode befindet sich aus Platzgründen im via SPIFI angebundenen Flash-Speicher, bzw. beim E4337 optional im externen SDRAM.

Es zeigt sich, dass die von NuttX standardmäßig durchgeführte Initialisierung der SPIFI-Anbindung für den Mikrocontroller *NXP LPC433x* nicht optimal ist. SPIFI steht für *SPI Flash Interface* [61, S.627f], SPI für *Serial Peripheral Interface Bus* – einen synchronen und seriellen Datenbus, mit dem verschiedene Komponenten einer digitalen Schaltung flexibel miteinander verbunden werden können [47]. Der Bus folgt einer Master-Slave-Topoplogie, wobei der Master ein Taktsignal ausgibt. In vorliegenden Fall entspricht der Mikrocontroller LPC4330/LPC4337 dem Master, dessen SPI-Taktfrequenz flexibel angepasst werden kann, indem der Takt über optionale Dividierer an verschiedene Taktquellen angebunden wird. Eine Illustration der Taktgenerierung findet sich in Abbildung 11.

Standardmäßig wird als Taktquelle für SPIFI ein dedizierter 12MHz-Quarzoszillator

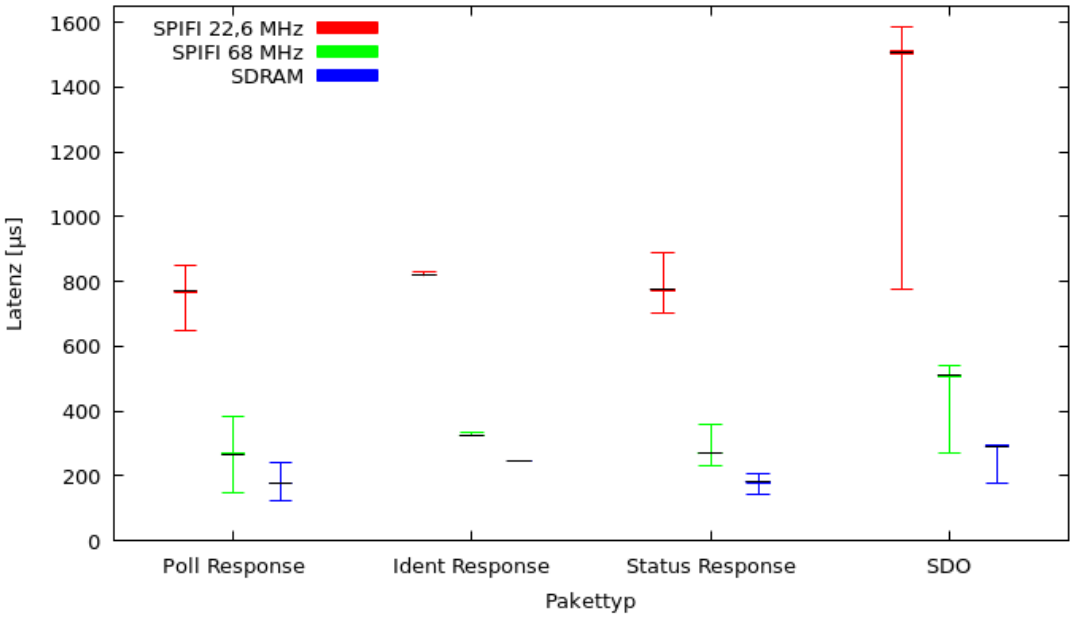


Abbildung 12: Box-Whisker-Diagramm zu den CN-Antwortzeiten in Abhängigkeit der Speicheranbindung und des Pakettyps. Die EPL-Konfiguration entspricht der in Szenario `1CN` aus Kapitel 5.5.2.

verwendet, der über die Phasenregelschleife `PLL1` (von engl. *Phase Lock Loop*) und das Divisionsmodul `IDIVB` (Divisor = 9) mit der SPI-Taktleitung verbunden ist. Die Regelschleife PLL1 ist mit einem Feedback-Teiler (Divisor = 17) ausgestattet, wodurch der Ausgangstakt vervielfacht wird. Es ergibt sich als Standard-Taktfrequenz:

$$f_{\text{spifi,default}} = \frac{12\text{MHz} \cdot 17}{9} = 22,\overline{6}\text{MHz}$$

Experimente zeigen, dass die Anbindung des Flash-Speichers mit einer bis zu drei mal höheren Taktfrequenz möglich und stabil ist. Hierfür wird der PLL1-Ausgangstakt über das Divisionsmodul `IDIVE` mit Divisor 3 umgeleitet, womit sich folgende Taktfrequenz ergibt:

$$f_{\text{spifi}} = \frac{12\text{MHz} \cdot 17}{3} = 68\text{MHz}$$

Eine Erhöhung der Taktfrequenz von standardmäßigen $22,\overline{6}$MHz auf 68MHz ergibt, wie in Abb. 12 zu sehen ist, eine Verkürzung der Antwortzeiten von durchschnittlich 790µs um 65% auf 275µs. Die Worst-Case-Antwortzeit reduziert sich im asynchronen Zeitfenster um 66% von 1587µs auf 540µs und in der synchronen Phase um 55% von 849µs auf 383µs. Wie man sieht ist der Leistungsgewinn nahezu linear.

Bein E4337 ergibt sich eine weitere Möglichkeit zur Beschleunigung der Ausführgeschwindigkeit, denn es steht ein 1MB fassender SDRAM-Speicher zur Verfügung, der

via EMC [61, S. 594ff] angebunden ist. Als flüchtiger Speicher ist dieser nicht für die persistente Ablage von Programmcode geeignet. Es ist jedoch möglich, beim Start des Mikrocontrollers den Programmcode vom Flash in den schnelleren SDRAM zu kopieren. Genau das wird beim E4337 umgesetzt, wodurch der Start leicht verzögert wird (um ca. 3s), die durchschnittliche `PRes`-Antwortzeit jedoch um 33% sinkt. Ebenso sinkt der Jitter der Antwortzeit von 45,3% auf 35,1%. Abb. 12 liefert auch hierzu statistische Werte aufgeschlüsselt nach Pakettyp.

5.5. Evaluierung

Die wichtigste Qualitätsmetrik eines EPL-Controlled Nodes ist seine Reaktionszeit auf Anfragen vom Managing Node, denn sie beeinflusst direkt die minimal erreichbare Zykluszeit. In diesem Zusammenhang muss man zwei Reaktionszeitklassen unterscheiden.

Erstens die Zeitspanne zwischen einem Poll Request und der zugehörigen Poll Response, im Folgenden l_{PRes} genannt. Sie entscheidet, wie viel Zeit im synchronen Zeitfenster für jeden CN reserviert werden muss. Im MN ist das als `NMT_MNCNPResTimeout` hinterlegt, ein Array, das für jeden CN die Angabe eines individuellen Werts erlaubt.

Zweitens gibt es die Latenz l_{ASnd} zwischen der Einladung zum Senden im asynchronen Zeitfenster (`SoA`) und der Ankunft des gesendeten Pakets (`ASnd`). Zu beachten ist hier, dass das asynchrone Zeitfenster für alle Knoten gleich lang ist. $\max l_{\mathrm{ASnd}}$ bestimmt also die minimale Dauer des asynchronen Zeitfensters.

In beiden Fällen muss insbesondere der Maximalwert der Reaktionszeiten betrachtet werden, denn bei Überschreitung der vorgesehenen Zeitfenster sind ernsthafte Störungen der EPL-Kommunikation möglich. Ggf. auftretender Jitter ist bei diesen Werten nicht kritisch, da EPL über einen separaten Synchronisierungsmechanismus verfügt. Trotzdem ist die zeitliche Varianz von Interesse, um Problemsituationen und Optimierunsmöglichkeiten zu erkennen.

In diesem Kapitel werden nun die genannten Werte herangezogen, um die Leistungsfähigkeit der erfolgreichen Ethernet POWERLINK-Implementierung für NuttX und ARMv7-M zu dokumentieren.

5.5.1. Powerlink Analyzer

Nicht nur die beiden oben genannten Latenzen, sondern die meisten interessanten Metriken im Zusammenhang von EPL-Kommunikation lassen sich allein aus dem entstandenen Netzwerkverkehr ermitteln: Antwortzeiten teilnehmender Geräte auf verschiedene Anfragetypen und ihre Varianz, Jitter der Zykluszeit, Anzahl fehlender Pakete und Zustandswechsel u.v.m. Da je nach Zykluszeit viele Tausend Pakete pro Sekunde versendet werden, ist eine manuelle Auswertung praktisch ausgeschlossen.

Im Rahmen dieses Projekts entstand deshalb das auf GitHub [34] verfügbare Open-Source-Programm *Powerlink Analyzer* [51], welches aus dem EPL-Netzwerkverkehr verschiedene Werte errechnet und übersichtlich anzeigt (vgl. Abbildung 13). Aus Gründen der Reproduzierbarkeit operiert es auf portablen Mitschnittsdateien im PCAPng-Format, die leicht mit einem Programm wie dumpcap [17] oder Wireshark [28] erzeugt

Tabelle 7: EPL-Parametrisierung in den Evaluierungsszenarien

	`1CN-A`, `Coremark-A`	`1CN-B`, `Coremark-B`	`4CN`, `Druck`
Anz. CNs	1	1	4
t_{cycle}	30ms	2ms	8ms
t_{async}	10ms	900µs	1,2ms
$t_{\text{PRes,i}} \forall i$	10ms	1ms	1,2ms
t_{wait}	10ms	100µs	2ms

werden können. Da das Programm fehlende Pakete erkennt sowie die Zustandsfolge aller Knoten ausgibt, ist es auch ein gutes Hilfsmittel um die Stabilität der Kommunikation zu beurteilen und damit die Parametrisierung des Managing Nodes (Zykluszeit, Timeouts, Fehlertoleranzen) zu optimieren. Folgende Liste gibt einen Überblick über den Funktionsumfang:

1. Erkennung von Fehlern in der Paketfolge. Z.B. fehlende Antworten oder unerwartete Pakettypen. Die Fehler werden gruppiert nach dem Tupel (Typ, betroffene Knoten-Nr., NMT-Zustand CN, NMT-Zustand MN), gezählt und übersichtlich angezeigt.

2. Ermittlung aller Zustandsübergänge und chronologische Auflistung anhand von Paket-Nr. und seit dem ersten Paket vergangener Zeit. Hieran lässt sich erkennen, ob das Netzwerk und alle Knoten nach der erstmaligen Initialisierung durchgehend verfügbar waren. Instabile Konfigurationen lassen sich so leicht feststellen. Auch die Gesamtdauer der Initialisierungsphase lässt sich leicht ablesen.

3. Statistische Betrachtung der Zykluszeit. Der absolute und relative Jitter-Wert ist ein wichtiges Maß zur Beurteilung des Echtzeitverhaltens des MN.

4. Statistische Betrachtung der Antwortzeiten. Durchschnitt, Minimum, Maximum und Jitter (absolut und relativ) der Verzögerung zwischen Anfrage und Antwort werden berechnet. Es erfolgt eine optionale Aufschlüsselung nach antwortendem Knoten und Pakettyp. Unterstützt werden die Paketfolgen `PReq` → `PRes` und `SoA` → `ASnd`. Bei `ASnd` wird zusätzlich zwischen den Diensten *Ident Response*, *Status Response*, *SDO* und *NMT* unterschieden.

Alle Messungen in diesem Kapitel wurden mit dem *Powerlink Analyzer* durchgeführt.

5.5.2. Antwortzeiten der CNs

Für die Messung der Antwortzeiten wird ein Netzwerk mit vier nahezu identischen CNs aufgebaut. Tabelle 7 gibt einen Überblick über die zugehörige EPL-Parametrisierung. Folgende vier Szenarien werden betrachtet:

```
10:21:00 [INFO] powerlink_analyzer: Loading PCAP file ../environment/sniffs/161026_pci_noveth.pcapng.

Errors:
[  1]    3x pres_missing                  (CN:Operational MN:Operational)
[  3]   12x status_response_missing       (CN:Unknown MN:PreOperational1)
[  3]   25x unexpected_packet_after_soa   (CN:Unknown MN:PreOperational1)
[  3]    5x pres_missing                  (CN:Unknown MN:Operational)
[253]  634x ident_response_missing        (CN:Unknown MN:PreOperational1)
[253]    2x ident_response_missing        (CN:Unknown MN:PreOperational2)
[253] 1058x ident_response_missing        (CN:Unknown MN:Operational)
[253]    1x ident_response_missing        (CN:Operational MN:Operational)

State Changes:
    1              0ns [240] PreOperational1
1'776  2'155'831'181ns [240] PreOperational2
1'785  2'174'647'524ns [  1] PreOperational2
1'790  2'180'888'364ns [  1] ReadyToOperate
1'792  2'181'772'338ns [  3] PreOperational2
1'799  2'187'984'355ns [  3] ReadyToOperate
1'812  2'199'770'055ns [  2] PreOperational2
1'821  2'205'723'297ns [  2] ReadyToOperate
1'832  2'213'723'582ns [240] ReadyToOperate
1'875  2'243'709'362ns [240] Operational
1'879  2'246'647'784ns [  1] Operational
1'898  2'259'800'596ns [  2] Operational
1'915  2'273'039'656ns [  3] Operational

Statistics:
Cycle/SoC avg =  6'000'008ns  min =  5'872'078ns  max =  6'082'512ns  jitter_abs =   127'930ns  jitter_rel =   2.13%
Responses avg =    588'299ns  min =        679ns  max =  1'371'968ns  jitter_abs =   783'669ns  jitter_rel = 133.21%
├─1       avg =    967'312ns  min =    337'774ns  max =  1'371'968ns  jitter_abs =   629'538ns  jitter_rel =  65.08%
├─2       avg =    406'758ns  min =    319'024ns  max =    859'253ns  jitter_abs =   452'495ns  jitter_rel = 111.24%
├─3       avg =    404'204ns  min =    324'299ns  max =  1'042'215ns  jitter_abs =   638'011ns  jitter_rel = 157.84%
├─240     avg =     16'489ns  min =        679ns  max =     63'719ns  jitter_abs =    47'230ns  jitter_rel = 286.43%
├─PRes    avg =    594'865ns  min =    319'024ns  max =  1'371'968ns  jitter_abs =   777'103ns  jitter_rel = 130.64%
│  ├─1    avg =    975'448ns  min =    804'461ns  max =  1'371'968ns  jitter_abs =   396'520ns  jitter_rel =  40.65%
│  ├─2    avg =    405'764ns  min =    319'024ns  max =    706'562ns  jitter_abs =   300'798ns  jitter_rel =  74.13%
│  └─3    avg =    403'282ns  min =    324'299ns  max =    612'023ns  jitter_abs =   208'741ns  jitter_rel =  51.76%
├─Ident   avg =    745'001ns  min =    548'426ns  max =  1'042'215ns  jitter_abs =   297'214ns  jitter_rel =  39.89%
│  ├─1    avg =    830'140ns  min =    829'257ns  max =    831'024ns  jitter_abs =       884ns  jitter_rel =   0.11%
│  ├─2    avg =    599'763ns  min =    548'426ns  max =    651'101ns  jitter_abs =    51'338ns  jitter_rel =   8.56%
│  └─3    avg =    805'101ns  min =    567'988ns  max =  1'042'215ns  jitter_abs =   237'114ns  jitter_rel =  29.45%
├─Status  avg =    405'299ns  min =    331'391ns  max =  1'010'820ns  jitter_abs =   605'521ns  jitter_rel = 149.40%
│  ├─1    avg =    405'334ns  min =    337'774ns  max =  1'005'348ns  jitter_abs =   600'014ns  jitter_rel = 148.03%
│  ├─2    avg =    408'310ns  min =    331'391ns  max =    848'902ns  jitter_abs =   440'592ns  jitter_rel = 107.91%
│  └─3    avg =    400'674ns  min =    349'702ns  max =  1'010'820ns  jitter_abs =   610'146ns  jitter_rel = 152.28%
├─SDO     avg =    232'011ns  min =        679ns  max =    859'253ns  jitter_abs =   627'242ns  jitter_rel = 270.35%
│  ├─1    avg =    642'953ns  min =    401'439ns  max =    695'753ns  jitter_abs =   241'514ns  jitter_rel =  37.56%
│  ├─2    avg =    657'197ns  min =    402'617ns  max =    859'253ns  jitter_abs =   254'580ns  jitter_rel =  38.74%
│  ├─3    avg =    638'315ns  min =    405'245ns  max =    695'252ns  jitter_abs =   233'070ns  jitter_rel =  36.51%
│  └─240  avg =     16'974ns  min =        679ns  max =     63'719ns  jitter_abs =    46'745ns  jitter_rel = 275.38%
├─NMT     avg =      8'911ns  min =      1'819ns  max =     56'682ns  jitter_abs =    47'771ns  jitter_rel = 536.05%
│  └─240  avg =      8'911ns  min =      1'819ns  max =     56'682ns  jitter_abs =    47'771ns  jitter_rel = 536.05%
```

Abbildung 13: Analyse eines EPL-Netzwerkmitschnitts mit *Powerlink Analyzer*

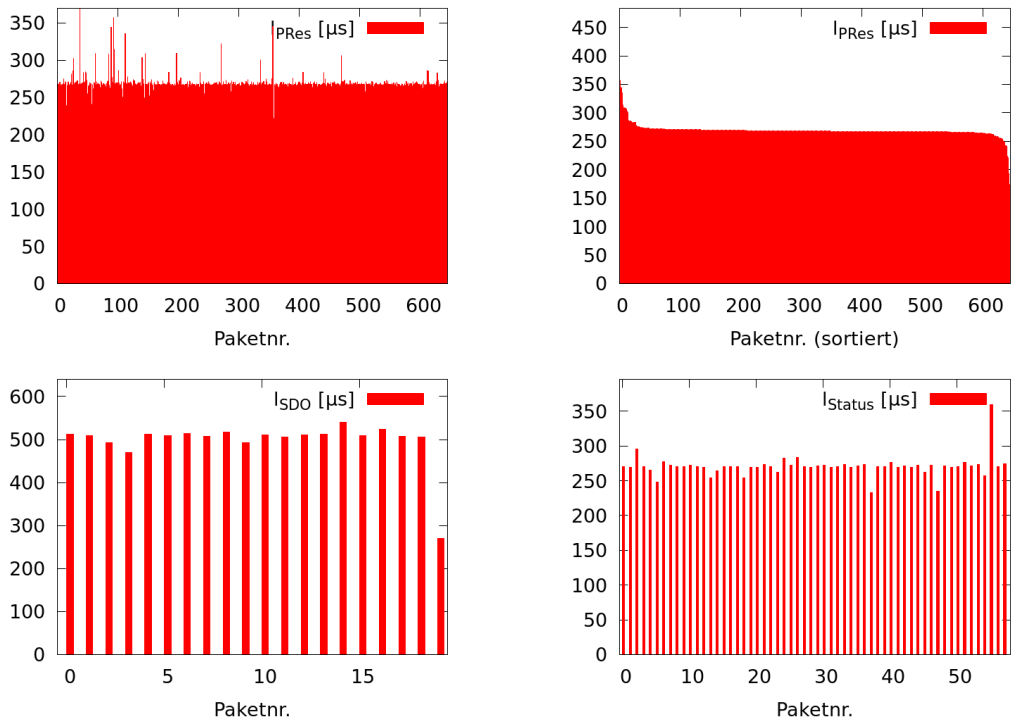


Abbildung 14: Antwortzeiten im `1CN-A`-Szenario. links oben: l_{PRes} als Zeitreihe, rechts oben: l_{PRes} absteigend sortiert, links unten: l_{SDO} als Zeitreihe, rechts unten: l_{Status} als Zeitreihe

1CN Der reine Rechenaufwand der verschiedenen Kommunikationsarten lässt sich am besten an einem einzelnen CN mit großer Zykluszeit ermitteln (Szenario `1CN-A`). Beeinflussungen durch andere CNs und durch SDO- und NMT-Kommunikation können so ausgeschlossen werden. Abb. 14 zeigt die Antwortzeiten eines einzelnen CNs über eine Aufnahmedauer von 22,7s (3000 Pakete) bei einer Zykluszeit von $t_{\mathrm{cycle}} = 30\mathrm{ms}$, sowie $t_{\mathrm{wait}} = 5\mathrm{ms}$. Es wird je synchrone Phase ein Byte an Daten gesendet und empfangen. Die Diagramme oben links und oben rechts stellen die `PRes`-Antwortzeiten dar, einmal als Zeitreihe und einmal absteigend sortiert. Die Werte liegen zwischen 147µs und 383µs mit einem Median von 267µs, dem Mittelwert 269µs und einem Quartilsabstand von 3µs. Die Antwortzeit ist damit sehr konstant und wird nur durch einige Ausreißer gestört, die durch Hintergrundprozesse im Betriebssystem und in OPLK verursacht werden. Unten links sind die CN-Antwortzeiten der SDO-Kommunikation in der Initialisierungsphase dargestellt. Es handelt sich um 20 Pakete mit einer Maximalverzögerung von 540µs. Unten rechts sind schließlich die Reaktionszeiten auf die während der Initialisierung regelmäßig stattfindenden Statusabfragen dargestellt. Sie liegen zwischen 233µs und 360µs.

Interessant ist nun, was die minimal mögliche Zykluszeit in dieser Konstellation ist. Leider ist die Bestimmung nicht so trivial, wie sie anfangs erscheinen mag – nämlich die Worst-Case-Latenzen als Timeoutwert festzulegen und die Zykluszeit gleich der Gesamt-

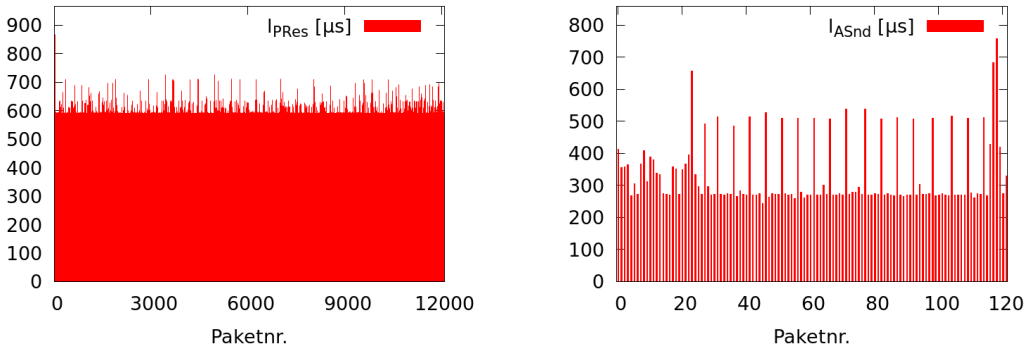


Abbildung 15: Antwortzeiten im `1CN-B`-Szenario. links: l_{PRes}, rechts: l_{ASnd}

summe zu setzen. Grund ist, dass die Reduktion der Zykluszeit die Ressourcenauslastung im CN erhöht und sich damit auch die Antwortzeit ändert. Vor allem in der rechenintensiven Initialisierungsphase (*Reduced Cycle*) wirkt sich das aus. Stattdessen muss die Zykluszeit sukzessive reduziert werden und jeweils mit einem Programm wie dem *Powerlink Analyzer* überprüft werden, ob Fehler aufgetreten sind.

In diesem Fall ergibt sich ein sinnvoller und stabiler Kompromiss bei $t_{\mathrm{cycle}} = 2\mathrm{ms}$ und $t_{\mathrm{async}} = 900\mu\mathrm{s}$ (Szenario `1CN-B`). Die Ergebnisse sind in Abb. 15 dargestellt. Es ist hier anzumerken, dass bei diesem Kompromiss während des *Reduced Cycles* insgesamt viermal nicht korrekt auf eine Statusanfrage geantwortet wird. Auf die Korrektheit und die synchrone Kommunikation wirkt sich das nicht aus. Mit einer Erhöhung der *Reduced Cycle Time* (`MinRedCycleTime_U32`, vgl. Abschnitt 6.2) ließe sich das Problem leicht beheben. Leider wird diese Funktionalität von OPLK noch nicht unterstützt. Alternativ kann t_{async} erhöht werden (z.B. auf 2ms wie in den Experimenten `4CN` und `Druck`). Damit verlängert sich jedoch die Zykluszeit auch nach der Initialisierungsphase.

Coremark Teil der Motivation für diese Arbeit ist, dass neben der reinen EPL-Kommunikation beliebige Drittsoftware ausgeführt werden kann. Hierfür sollte zunächst nachgewiesen werden, dass die EPL-Kommunikation dadurch nicht negativ beeinflusst wird, d.h. dass die Fähigkeiten des Echtzeitbetriebssystems korrekt eingesetzt werden. Außerdem muss genügend Rechenzeit und Arbeitsspeicher verfügbar sein, um überhaupt andere Software parallel ausführen zu können.

Zur Evaluierung wird der schon in Kapitel 4 eingesetzte Prozessorbenchmark *Coremark* verwendet. Das Experiment aus dem Szenario `1CN-A` wird wiederholt, diesmal wird jedoch nach einiger Zeit zweimal im Hintergrund Coremark ausgeführt (`Coremark-A`). Die Aufnahmedauer beträgt diesmal 72,2s (entspricht 10000 Paketen). Coremark wird einmal nach ca. 7s und einmal nach ca. 43s gestartet. Die Laufzeit beträgt jeweils 15s. In Abb. 16 sind links die `PRes`-Antwortzeiten zu sehen. Es ist mit bloßem Auge keine Änderung im Bereich 242 – 762 (7s – 22s) und 1489 – 2000 (43s – 58s) zu sehen. Der Durchschnittswert bleibt gegenüber dem `1CN`-Experiment mit 269µs exakt gleich. Lediglich der Maximalwert ist mit 528µs um 38% höher. Coremark erreicht 73 Punkte

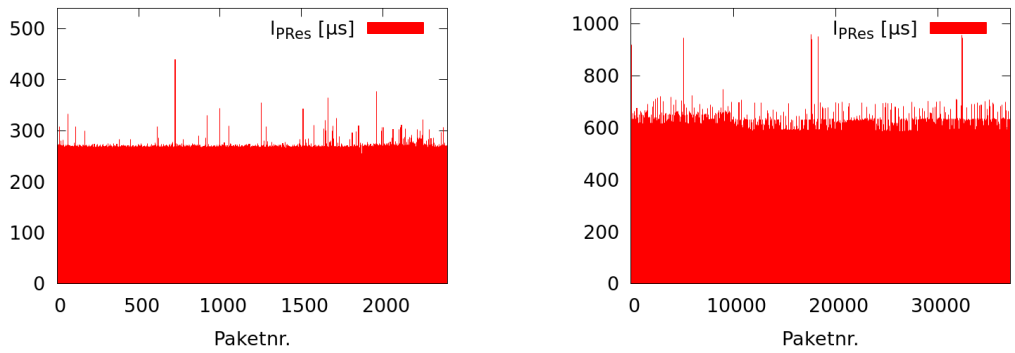


Abbildung 16: Antwortzeiten im `Coremark`-Szenario. links: l_{PRes} in `Coremark-A`, rechts: l_{PRes} in `Coremark-B`

gegenüber 78 Punkten wenn parallel keine EPL-Kommunikation läuft.

Eine größere Konkurrenzsituation ist bei kleineren Zykluszeiten zu erwarten. Deswegen wird das `Coremark`-Experiment nun wie in Szenario `1CN-B` durchgeführt (`Coremark-B`). Abb. 16 zeigt unten links und unten rechts die Ergebnisse. Die Aufnahmedauer beträgt diesmal 75,5s, was insg. 150000 Paketen entspricht. Start der Coremark-Ausführung ist hier nach 15s bzw. 45s und sie dauert je 13s (`PRes`-Paketnr. ca. 7500 – 14000 und 22500 – 29000). Der Maximalwert ist ggü. `1CN-B` 93µs höher, zusätzlich kommen sieben *Poll Responses* (ca. 0,019%) gar nicht an. Der Coremark-Score beträgt 15 Punkte. Da die Zykluszeit schon in `1CN-B` äußerst knapp bemessen war, verwundert das Ergebnis nicht. Der exakt gleichgebliebene Mittelwert von 587µs und der niedrigere Coremark-Score zeugen davon, dass die OPLK-Priorisierung Wirkung zeigt. Die größeren Ausreißer scheinen ein Problem der vielen Kontextwechsel bei einer so kleinen Zykluszeit und so hohen Prozessorauslastung zu sein.

4CN Nach der Betrachtung eines einzelnen CNs soll nun ein realistischeres Szenario mit vier nahezu gleichartigen CNs (3 × B210E, 1 × E4337) analysiert werden. Die CNs führen bei einer Zykluszeit von $t_{\mathrm{cycle}} = 8\mathrm{ms}$ nichts außer der EPL-Kommunikation mit je 1B Ein- und Ausgangsdaten pro Knoten aus. Eine Aufnahme des Netzwerkverkehrs über 18,4s führt zu den Diagrammen in Abb. 17. Links oben ist die Latenz zwischen *Ident Request* und *Ident Response* zu sehen. Jeder CN wird zweimal angefragt, einmal ganz zu Beginn und einmal nach der Initialisierung. Die *Ident Response* ist mit 176 Byte die mit Abstand größte Nachricht in diesem Experiment. Dank Vorberechnung in OPLK hält sich die Antwortzeit mit durchschnittlich 432µs und maximal 816µs trotzdem in Grenzen. Die Antwortzeiten bei Status Requests und SDO-Kommunikation (Diagramme 2 und 3) sind durchschnittlich 285µs bzw. 175µs und maximal 413µs bzw. 765µs größer als im `1CN`-Experiment. Das vierte Diagramm fasst die Verzögerungen aller Nachrichten zusammen, die im asynchronen Zeitfenster von den vier CNs versendet werden.

Unten links und unten rechts folgt schließlich die Betrachtung der synchronen Kommunikation. Auffallend sind die im Vergleich zum `1CN`-Experiment die etwas höheren und

regelmäßig auftretenden Ausreißer, die mutmaßlich der wesentlich höheren Auslastung geschuldet sind. Durchschnitt und Maximalwert liegen bei 273µs bzw. 598µs.

Druck Ein noch realistischeres Szenario ist mit dem 3D-Drucker aus Kapitel 7 gegeben. Er besteht aus den selben vier CNs, jedoch mit komplexerer Software und umfangreicherem Datenaustausch (je CN 19 Byte eingehend und 13 Byte ausgehend). Neben der EPL-Kommunikation steuern die CNs auch einen Schrittmotor mit hoher Frequenz, lesen Eingänge aus und führen verschiedene Berechnungen durch. Für Details sei auf o.g. Kapitel verwiesen.

Die Initialisierung unterscheidet sich nicht wesentlich von der im `4CN`-Experiment. Deswegen wird hier nur die `PRes`-Latenz betrachtet. Es handelt sich um den in Abb. 18 dargestellten Werten um eine ca. zehnminütige Aufnahme während des Drucks von Testobjekt 2 aus Kapitel 7.7.

Auffallend sind die deutlich größeren Abweichungen bei der `PRes`-Antwortzeit. Hauptursache ist die zusätzliche Auslastung durch die Schrittmotoransteuerung, die den Prozessor je nach Geschwindigkeit sehr häufig unterbricht. Im vorliegenden Fall beträgt die Geschwindigkeit über fast den gesamten Druckprozess konstant $15\frac{mm}{s}$. Bei 40 Mikroschritten pro Millimeter entspricht das 600 Prozessorunterbrechungen pro Sekunde bzw. 4,8 Unterbrechungen pro Zyklus. Auch die anderen parallel laufenden Berechnungen wirken sich negativ auf den Determinismus aus.

Der Höchstwert beträgt 1101µs bei einem Durchschnitt von 321µs, einem Median von 344µs und einem Quartilsabstand von 63µs. Es besteht damit ein Puffer von knapp 100µs zu dem Timeout $\forall i : t_{\mathrm{PRes,i}} = 1{,}2\mathrm{ms}$. Der Timeout-Wert erweist sich auch bei anderen Objekten und längeren Druckprozessen als ausreichend groß. Schnellere Bewegungen sind mit dieser EPL-Parametrisierung mind. bis zur in der Druckersoftware konfigurierten Maximalgeschwindigkeit von $30\frac{mm}{s}$ problemlos möglich.

5.5.3. Rechenzeit

Nicht nur die absolute Latenz ist von Interesse, sondern auch, wie sie sich zusammensetzt. Insbesondere die Rechenzeit in den beteiligten Softwarekomponenten lässt Rückschlüsse auf zukünftige Optimierungsmöglichkeiten zu und zeigt gleichzeitig natürliche Limitierungen auf.

Die Daten sollen repräsentativ anhand der Kausalkette *Empfang einer Poll Request* $\rightarrow$ *Senden der Poll Response* ermittelt werden. Als Szenario dient ein einzelner CN, der mit der Software aus dem `Drucker`-Szenario auf einem `B210E` betrieben wird. Die Kommunikation erfolgt einmal mit $t_{\mathrm{cycle}} = 30ms$ und $t_{\mathrm{wait}} = 5ms$ sowie einmal mit $t_{\mathrm{cycle}} = 3ms$ und $t_{\mathrm{wait}} = 500\mu s$, um Einflüsse der Auslastung erkennen zu können. Das Diagramm in Abb. 19 fasst die Ergebnisse zusammen.

Für die Software beginnt alles mit einem Interrupt, mit dem die Ethernet-Hardware signalisiert, dass ein neues Paket verfügbar ist. Zu diesem Zeitpunkt hat das Paket schon nennenswerte Verarbeitungsschritte in Hardware hinter sich. Es musste zuvor deserialisiert werden, ggf. einen Filter durchlaufen und schließlich per DMA (*Direct Memory*

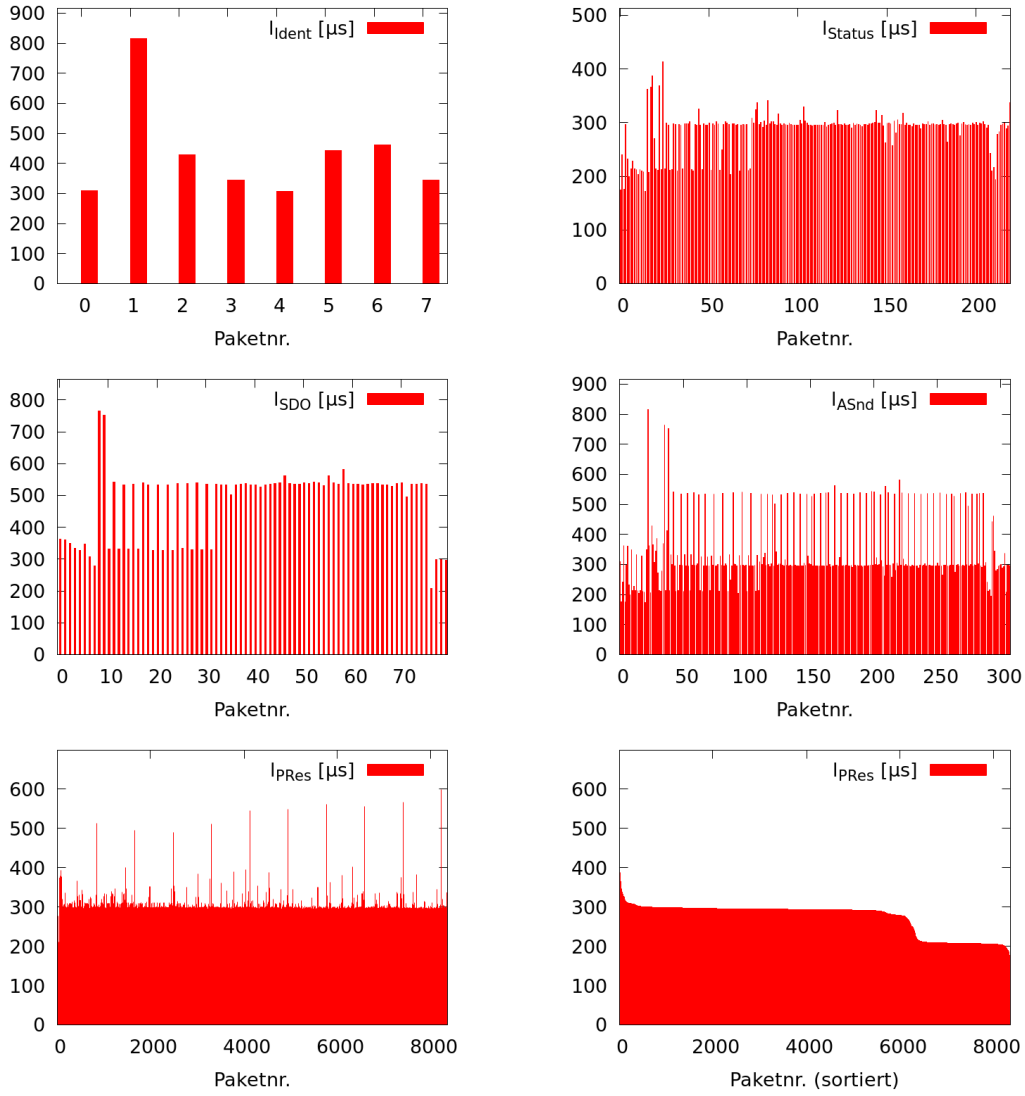


Abbildung 17: Antwortzeiten im `4CN`-Szenario. oben links: l_{Ident} als Zeitreihe, oben rechts: l_{Status} als Zeitreihe, Mitte links: l_{SDO} als Zeitreihe, Mitte rechts: l_{ASnd} als Zeitreihe, unten links: l_{PRes} als Zeitreihe, unten rechts: l_{PRes} absteigend sortiert

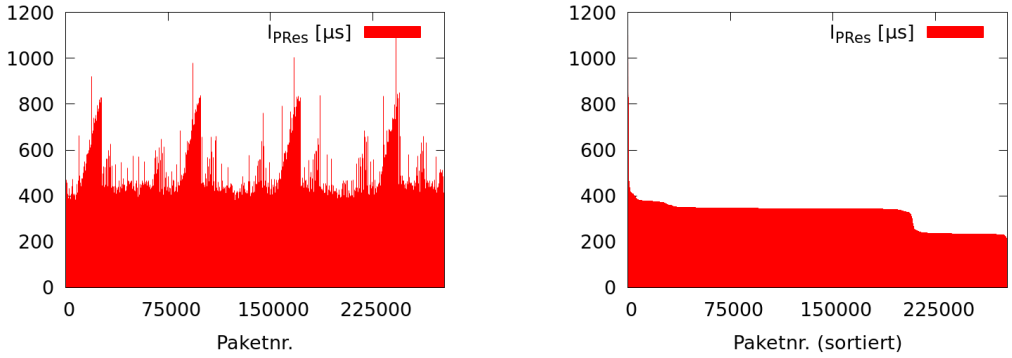


Abbildung 18: Antwortzeiten im `Drucker`-Szenario. links: l_{PRes} als Zeitreihe, rechts: l_{PRes} absteigend sortiert

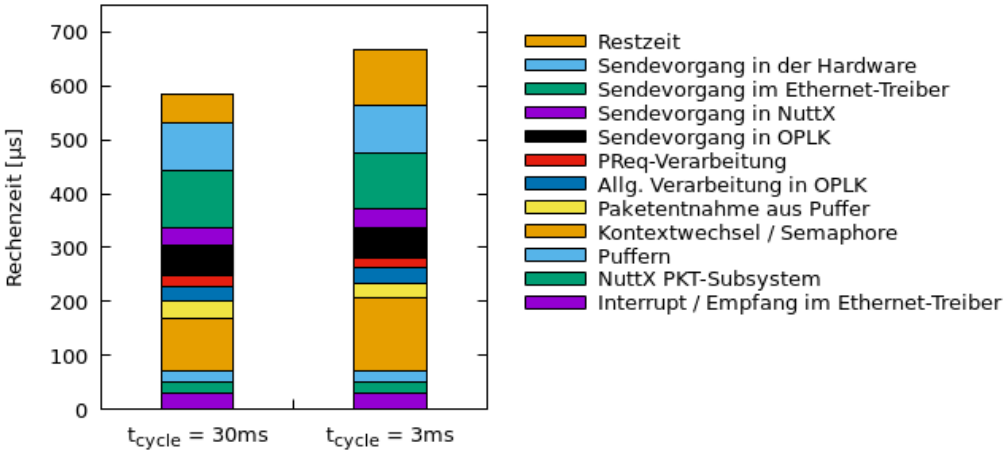


Abbildung 19: Rechenzeit

Access) in den Hauptspeicher kopiert werden. Im Diagramm ist die Zeit, die hierfür benötigt wird, als Teil der Restzeit dargestellt, denn sie lässt sich nicht ohne weiteres in Software ermitteln. Stattdessen dient die Differenz zur im Netzwerk gemessenen Latenz als Abschätzung. Ebenfalls Teil der Restzeit sind Latenzen, die durch Leitungen und den Hub verursacht werden.

Die Verarbeitung im hardwarespezifischen Ethernet-Treiber benötigt ca. 30µs bis zu dem Punkt, an dem das Paket an das `pkt`-Netzwerksubsystem von NuttX weitergegeben wird. Dort wird im Wesentlichen der passende Callback aufgerufen, in diesem Fall der des Treibers für gepufferten Paketempfang (`bufpkt`, vgl. Kapitel 3.2.4). Es folgt der Kopiervorgang in den Ringpuffer, der in diesem Fall 21µs in Anspruch nimmt.

An die Pufferung schließt ein Bruch im Kontrollfluss an. Alle Berechnungen fanden bis dahin im Interrupt-Kontext statt und sind direkt auf diesen zurückzuführen. openPOWERLINK hat jedoch einen eigenen Kontrollfluss. Der OPLK-Ethernet-Treiber (`edrv`) nimmt mit einem blockierenden Aufruf von `bufpkt_receive` in Endlosschleife Pakete entgegen. Die Synchronisierung zwischen den Kontrollflüssen erfolgt mit einem Semaphor. Das Verfahren ist mit ca. 96µs – 135µs teuer, da es globale Synchronisierung und das zeitweise Deaktivieren von Interrupts involviert. Es ist auch von der Prozessorauslastung abhängig, da andere laufende und höherpriore lauffähige Threads die Zuteilung des `edrv`-Threads nach dem Inkrementieren der Sempahore verzögern können.

Auf OPLK-Seite wird das Paket anschließend aus dem Ringpuffer entnommen (26µs – 32µs) und dann inhaltlich verarbeitet. Im vorliegenden Szenario steht nach 29µs fest, dass es sich um ein `PReq`-Paket handelt. Dank Vorberechnung benötigt die `PReq`-Verarbeitung bis zum Initiieren des `PRes`-Sendevorgangs nur 19µs. Es folgt der Sendevorgang in OPLK (56µs), im NuttX-Netzwerksubsystem (34µs) und im Ethernet-Treiber (104µs). Die Zeitdauer, bis die Ethernet-Hardware per Interrupt meldet, dass der Sendevorgang abgeschlossen ist, beträgt ca. 88µs.

Die meisten der genannten Verzögerungen sind nahe am theoretischen Minimum und unterliegen nur sehr kleinen Schwankungen. Eine Optimierungsmöglichkeit ergibt sich am ehesten durch engere Kopplung der beiden Kontrollflüsse. Das widerspräche jedoch in Teilen dem Entwurfsziel der losen und portablen Kopplung von Betriebssystem und Anwendung und der Verwendung von Standardfunktionalitäten. Besser ist es, die Ausführgeschwindigkeit im Allgemeinen zu erhöhen, beispielsweise durch Verlagerung von rechenaufwendigen Teilen des Programmcodes in den schnellen Chip-lokalen Arbeitsspeicher.

6. Managing Node

Auch wenn der Fokus dieser Arbeit auf der Realisierung eines NuttX- und Mikrocontroller-basierten Controlled Nodes liegen soll, so ist für einen effizienten Betrieb des EPL-Netzwerks die Berücksichtigung des Managing Nodes und seiner Konfiguration dennoch unerlässlich. Neben der Auflistung der verwendeten Hardware und einem Leistungsvergleich soll in diesem Kapitel vor allem auf die OV-Konfiguration eingegangen werden sowie wie sie an Controlled Nodes unterschiedlicher Leistungsklassen individuell angepasst werden kann.

6.1. Hardware

An den Managing Node werden etwas höhere Leistungsanforderungen gestellt, als an die Controlled Nodes, denn er sendet ein Vielfaches an Nachrichten (pro Zyklus standardmäßig mind. $n_{\mathrm{CN}}+2$ Pakete, n_{CN}: Anzahl CNs im Netzwerk) und verarbeitet typischerweise auch alle im Netzwerk gesendeten Pakete. Als zentraler Taktgeber ist die Jitterfreiheit der von ihm bestimmten Zykluszeit entscheidend für die Korrektheit vieler Echtzeitanwendungen.

Mit fortlaufender Optimierung der Controlled Nodes wurde im Laufe der Arbeit in einigen Fällen der MN zum limitierenden Faktor, und das obwohl die als MN eingesetzten Plattformen mit 3.749 bis 20.698 Punkten in Coremark theoretisch 48- bis 280-mal leistungsfähiger sind, als die *ARM Cortex-M4*-basierten CNs (74 bzw. 78 Punkte). Es lohnt sich daher, sich dem Thema MN-Leistungsfähigkeit ausführlicher zu widmen.

Drei Rechner kommen in insgesamt sechs verschiedenen Konfigurationen zum Einsatz: Ein *Raspberry PI 3 Model B* (`RPi3`) [27], ein ca. neun Jahre alter PC mit *Intel Core2-6400*-Prozessor (`PCCore2`) sowie ein moderner PC mit *Intel Core i5-6500*-Prozessor (`PCi5`). Details finden sich in den Tabellen 8 bis 10. Als Betriebssystem kommt bei allen Varianten Linux zum Einsatz (*Rabspian* auf `RPi3` und *Ubuntu 16.04* auf den PCs, *Ubuntu* optional auch mit einem aktuellen Echtzeit-Kernel (mit dem *Preempt-RT*-Patch von Ingo Molnar [84]; entspricht den Varianten `PCCore2-rt` und `PCi5-rt`)). `PCCore2` verfügt zudem über eine Netzwerkkarte, für die in openPOWERLINK ein nativer Treiber vorliegt. Die Variante `PCCore2-edrv` verwendet diesen Treiber. Großer Vorteil ist, dass damit die Bearbeitung der Netzwerkpakete vollständig im Kernel ablaufen kann und nicht auf die Userland-Bibliothek *PCAP* [78] zurückgegriffen werden muss. Außerdem können Pakete auf diese Weise unmittelbar nach Erhalt und ohne zusätzliche Kopieroperation verarbeitet und gegebenenfalls verworfen werden. Etliche Verarbeitungsschritte werden so nicht nur beschleunigt sondern unterliegen auch weniger zeitlichen Schwankungen.

Folgende Liste gibt einen Überblick über vom MN abhängige Werte, einen Einfluss auf die minimal erreichbare Zykluszeit und die Qualität der Kommunikation haben können:

1. Die Latenz zwischen `SoA` und den vom MN versendeten `ASnd/SDO`-Paketen ($l_{\mathrm{SDO,MN}}$)
2. Die Latenz zwischen `SoA` und den NMT-Kommandos `NMTResetNode` ($l_{\mathrm{ResetNode}}$, einmalig als Broadcast versendet), `NMTResetConfiguration` ($l_{\mathrm{ResetConf}}$), `NMTEn-`

Tabelle 8: Hardwareeigenschaften der MN-Plattform `RPi3`

SoC	Broadcom BCM2837
Prozessor	ARM Cortex-A53 (64-bit, 4 Kerne) @ 1,2 GHz
Speicher	1 GB LPDDR2 SDRAM @ 900 MHz
Coremark	3.749 Punkte (Single-Core)

Tabelle 9: Hardwareeigenschaften der MN-Plattform `PCCore2`

Prozessor	Intel Core2 Duo E6400 (2 Kerne) @ 2,13 GHz
Speicher	2 GB DDR @ 667 MHz
Netzwerkkarte	Intel Ethernet Pro 100 (82557x)
Betriebssystem (`PCCore2`)	Ubuntu 16.04 (Linux 4.4.0)
Betriebssystem (`PCCore2-rt`)	Ubuntu 16.04 (Linux 4.4.25-rt35)
Coremark (`PCCore2`)	6.989 Punkte (Single-Core)
Coremark (`PCCore2-rt`)	6.860 Punkte (Single-Core)

Tabelle 10: Hardwareeigenschaften der MN-Plattform `PCi5`

Prozessor	Intel Core i5-6500 (4 Kerne) @ 3,2 GHz
Speicher	8 GB DDR4 @ 2400 MHz
Betriebssystem (`PCi5`)	Ubuntu 16.04 (Linux 4.4.0)
Betriebssystem (`PCi5-rt`)	Ubuntu 16.04 (Linux 4.8.0-rt3)
Coremark (`PCi5`)	17.854 Punkte (Single-Core)
Coremark (`PCi5-rt`)	17.796 Punkte (Single-Core)

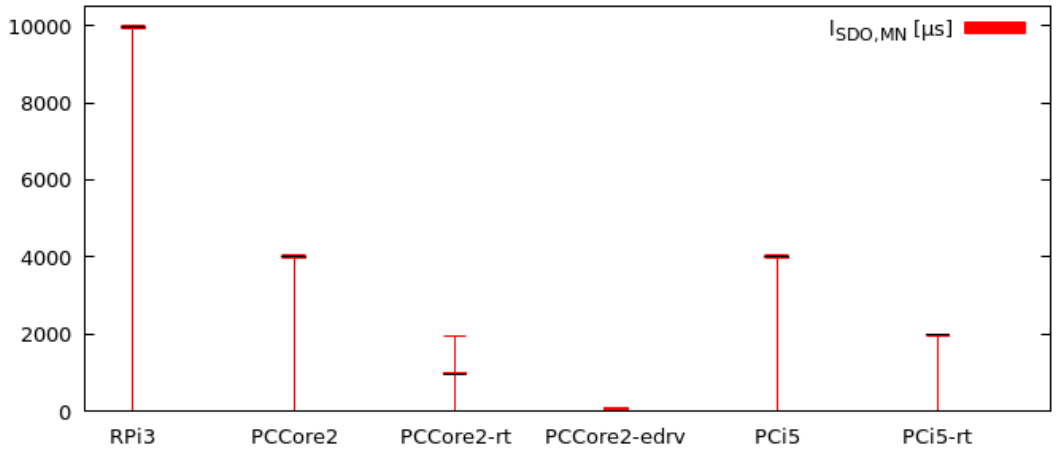


Abbildung 20: Antwortzeiten des MN bei SDO-Kommunikation in Abhängigkeit der Plattform

`ableReadyToOperate` ($l_{\text{EnableReady}}$) sowie `NMTStartNode` ($l_{\text{StartNode}}$, je 1× pro CN versendet)

3. Die Latenz zwischen `SoC` und erstem `PReq` (l_{PReq}, abzüglich t_{wait})
4. Jitter beim Versand der `SoC`-Pakete

Im Folgenden sollen die MN-Varianten `RPi3`, `PCCore2`, `PCCore2-rt`, `PCCore2-edrv`, `PCi5` und `PCi5-rt` miteinander verglichen werden. Als Software kommt ein auf dem OPLK-Demoprogramm `demo_mn_console` aufbauender Managing Node zum Einsatz. Die Netzwerkkonfiguration entspricht derjenigen in Szenario `1CN-A` aus Kapitel 5.5.2.

Abb. 20 zeigt zunächst $l_{\text{SDO,MN}}$ als Box-Whisker-Diagramm. Sofort sind die großen Unterschiede in der Worst-Case-Antwortzeit zu erkennen. Sie reicht von 64µs bei `PCCore2-edrv` bis zu über 10ms bei `RPi3` (Faktor 150).

Bei allen Varianten außer bei `PCCore2-edrv` liegt sowohl der Worst-Case- als auch der Medianwert mit mind. 991µs über dem CN-Maximalwert $\max l_{\text{SDO,CN}} = 540\mu s$ (vgl. Abschnitt 5.5.2). Der MN erzwingt also in diesen Fällen die Wahl eines größeren asynchronen Zeitfensters, als es für die CNs nötig wäre. Das Diagramm macht deutlich, dass reine Rechenleistung nicht entscheidend ist für den effizienten Betrieb eines Ethernet-basierten Feldbussystems ist. So ist die Variante `PCCore2-edrv` mit nativem Treiber und Kernelbetrieb im Mittel 574-mal schneller als der neun Jahre jüngere und theoretisch 2,6-mal leistungsfähigere `PCi5`. Auch bei *PCAP*-Betrieb bringt die Verwendung eines Echtzeitkernels größere Vorteile als ein leistungsfähigerer Rechner (Reduktion von $\max l_{\text{SDO,MN}}$ um 52% (`PCCore2-rt`) bzw. 51% (`PCi5-rt`) gegenüber der Variante mit ungepatchten Kernel).

Einen Einfluss auf die Zykluszeit können auch die vom MN versendeten NMT-Kommandos haben. Es handelt sich im Normalbetrieb um einige wenige und simpel aufge-

Tabelle 11: Latenz in µs zwischen `SoA`- und NMT-Kommando in Abhängigkeit der Plattform

	RPi3	PCCore2	PCCore2-rt	PCCore2-edrv	PCi5	PCi5-rt
$l_{\text{ResetNode}}$	4977	976	981	4	944	897
$l_{\text{ResetConf}}$	9951	4010	965	5	4017	1980
$l_{\text{EnableReady}}$	3	7	4	5	18	23
$l_{\text{StartNode}}$	4	7	4	4	18	22

Tabelle 12: Zykluszeit/SoC-Jitter in Abhängigkeit der Plattform

	RPi3	PCCore2	PCCore2-rt	PCCore2-edrv	PCi5	PCi5-rt
Jitter absolut [µs]	121	138	47	40	115	69
Jitter relativ [%]	0,40	0,46	0,16	0,13	0,38	0,23

baute Pakete, die im asynchronen Zeitfenster übertragen werden. Sie fallen jedoch in einigen Fällen mit Zustandsübergängen im MN zusammen, wodurch just zum Sendezeitpunkt eine Reihe von weiteren Berechnungen durchgeführt werden. Das trifft vor allem auf die Befehle `NMTResetNode` und `NMTResetConfiguration` zu. Tabelle 11 listet die Latenzen aller Konfigurationen auf. Der Höchstwert von `PCCore2-edrv` liegt bei 5µs. Die Höchstwerte der anderen Konfigurationen liegen zwischen 981µs und 9951µs.

Die Unterschiede beim Jitter der Zykluszeit sind weniger gravierend. Sie liegen durchweg unter 138µs, was 0,46% der Zykluszeit von 30ms entspricht. Den besten Wert erreicht hier ebenfalls `PCCore2-edrv`. Auch die Verwendung eines Echtzeitkernels sorgt für einen kleineren Jitter. Tabelle 12 können die Werte aller Konfigurationen entnommen werden.

Die hier erhobenen Daten legen nahe, dass, wenn möglich, ein MN mit nativ angebundener Netzwerkhardware zu bevorzugen ist, auch wenn im Netzwerk sonst nur deutlich leistungsschwächere CNs vorhanden sind. Dementsprechend werden alle anderen Experimente in dieser Arbeit mit der MN-Plattform `PCCore2-edrv` durchgeführt.

6.2. Konfiguration

Die Inbetriebnahme eines EPL-Netzwerks mit openPOWERLINK und selbst entwickelten CNs ist aufgrund von etwaig unbemerkten Timeout-Überschreitungen und in vielen Fällen uneindeutigen oder verzögerten Fehlermeldungen nicht trivial. Die folgende Liste soll deswegen einen Überblick darüber geben, welche Parameter im Objektverzeichnis des MN in diesem Zusammenhang von Bedeutung sind. Sobald alle genannten Einträge an die gegebenen Rahmenbedingungen angepasst sind, steht einem fehlerfreien EPL-Betrieb gewöhnlich nichts mehr im Wege.

1. `NMT_CycleLen_U32` (t_{cycle})

   Gibt die Zykluszeit, also exakt die Zeit zwischen zwei `SoC`-Frames an. Es ist Aufgabe des Systemintegrators, diese Zeit ausreichend groß zu wählen. Wenn die anderen

relevanten Timing-Parameter passend gewählt sind, kann die Zykluszeit t_{cycle} näherungsweise wie folgt gewählt werden:

$$t_{\text{cycle}} \geq \sum_{i \in \text{CNs}} t_{\text{PRes,i}} + \max t_{\text{async}} + t_{\text{wait}}$$

Da sich das Antwortverhalten der beteiligten Knoten jedoch mit veränderter Zykluszeit ebenfalls verändern kann, muss der Betrieb mit geänderter Zykluszeit unbedingt experimentell getestet werden.

2. `NMT_NodeAssignment_AU32`

   In diesem Feld wird angegeben, welche Knoten im Netzwerk erwartet werden, und ob sie essentiell oder optional sind. Das Feld besteht aus höchstens 255 Subindizes. Jeder Subindex steht dabei für eine EPL-Knotennummer und enthält ein Bitfeld, das die Eigenschaften des entsprechenden Knotens angibt. Das niederwertigste Bit gibt beispielsweise an, dass ein solcher Knoten im Netzwerk existieren soll. Eine Auflistung der möglichen Angaben findet sich in [22, S.232].

3. `NMT_MNCNPResTimeout_AU32` ($t_{\text{PRes,i}}$ für CN i)

   Damit nicht das gesamte Netzwerk zusammenbricht wenn ein einzelner CN ausfällt, muss bekannt sein, wie lange der MN in der isochronen Phase auf eine Poll Response warten soll, nachdem er ein Poll Request gesendet hat. Nach dieser Zeitspanne fährt er mit dem Polling aller anderen Knoten fort und registriert einen Fehler für den CN, der nicht geantwortet hat. Da im Netzwerk unterschiedlich leistungsfähige CNs existieren können, existiert für jeden Knoten ein Subindex im Feld `NMT_MNCNPResTimeout_AU32`, sodass das Timeout individuell konfiguriert werden kann. Damit können auch unterschiedliche Leitungslängen ausgeglichen werden. Viele Controlled Nodes geben im OV-Eintrag `PResMaxLatency_U32` die Worst-Case-Verzögerung einer `PRes`-Antwort an, was als Anhaltspunkt bzw. untere Grenze für die Konfiguration im MN-OV dienen kann.

4. `NMT_IsochrSlotAssign_AU8`

   Neben individuellen Timeouts für die Poll Response gibt es eine weitere Möglichkeit, unterschiedliche Leistungsvermögen der CNs im Netzwerk auszugleichen: Mit dem Feld `NMT_IsochrSlotAssign_AU8` lässt sich die Reihenfolge festlegen, in der die CNs vom MN abgefragt werden. So kann man schwächeren CNs mehr Zeit zur Verarbeitung des `SoC`-Pakets geben, welches durch seine Funktionweise als zentrales Synchronisationssignal üblicherweise einige Berechnungen auf den CNs zur Folge hat.

5. `WaitSoCPReq_U32` (t_{wait}, Teil von `NMT_MNCycleTiming_REC`)

   Um auch dem Knoten, der immer als erstes abgefragt wird, ausreichend Zeit zur `SoC`-Bearbeitung zu geben und vor allem um die `PRes`-Timeouts (vgl. `NMT_MNCNPResTimeout_AU32`) unabhängig von der Reihenfolge zu machen, lässt sich der

Versand des ersten Poll Requests nach dem SoC-Paket mit WaitSoCPReq_U32 (Einheit Nanosekunden) künstlich verzögern. Als Anhaltspunkt kann (falls vorhanden) das Maximum der Werte D_NMT_CNSoC2PReq_U32 (max. SoC-Bearbeitungszeit) der CNs herangezogen werden. Auch für den MN ist der Wert relevant, wenn zum Synchronisationszeitpunkt längere Berechnungen durchgeführt werden.

6. AsyncSlotTimeout_U32 (t_{async}, Teil von NMT_MNCycleTiming_REC)

   Dieser Wert bestimmt die Länge der asynchronen Phase in Sekunden. Da beim Hochfahren des Netzwerks essentielle Informationen im asynchronen Slot übertragen werden, ist es wichtig, diese Zeitspanne so groß zu wählen, dass keines der Pakete verloren geht. Dabei muss auf den langsamsten Knoten im Netzwerk Rücksicht genommen werden. Da außerdem die dort übertragenen Pakete durchaus größer sein können als die im Normalbetrieb übertragenen Prozessdaten (z.B. ist eine *Ident Response* 176 Bytes groß) und auch Pakete des virtuellen Ethernet-Kanals hier übertragen werden, kann die asynchrone Phase einen beträchtlichen Teil der Zykluszeit ausmachen. Es lohnt sich deshalb in besonderem Maße, die asynchronen Antwortzeiten von langsamen CNs zu optimieren. Eine Begrenzung der Paketgröße in der asynchronen Phase ist mit der Einstellung AsyncMTU_U16 (Teil von NMT_CycleTiming_REC) möglich.

7. SDO_SequLayerTimeout_U32

   Die Zeit in Millisekunden, nach der eine SDO-Übertragung abgebrochen wird, wenn sie bis dahin nicht abgeschlossen wurde. Da eine SDO-Übertragung üblicherweise aus mehreren Paketen besteht, die nacheinander in der asynchronen Phase ausgetauscht werden, sollte dieser Wert an die Zykluszeit angepasst sein. Dies ist insbesondere wichtig, wenn zu Testzwecken die Zykluszeit stark reduziert wird (etwa, um die manuelle Analyse des Netzwerkverkehrs zu erleichtern). Nach Ablauf der Zeit startet der MN mit der NMT-Nachricht NMTResetNode die Kommunikation neu.

8. MinRedCycleTime_U32

   Im sog. *Reduced Cycle* (RC), dh. wenn der MN sich im Zustand NMT_MS_PRE_-OPERATIONAL_1 befindet, entfällt die isochrone Phase, um die dort ausschließlich in der asynchronen Phase stattfindende Kommunikation zu beschleunigen. Dies kann die CNs überfordern, da weniger Zeit zur Verarbeitung des SoC-Frames bleibt. Aus diesem Grund lässt sich mit MinRedCycleTime_U32 eine Mindestlänge des RCs in Mikrosekunden angeben. Wenn sich garantieren lässt, dass nach der Initialisierungsphase keine weitere SDO- und VETH-Kommunikation mehr stattfindet, lässt sich diese Einstellung auch zur Reduzierung der Zykluszeit nutzen. Leider wird diese Einstellung von openPOWERLINK in Version 2.4.1 nicht unterstützt.

6.3. openCONFIGURATOR

Die Konfiguration der Knoten wird am Besten mit dem quelloffenen Programm openCONFIGURATOR (OC) durchgeführt. Es basiert auf der Eclipse-Plattform und wird

als Teil des openPOWERLINK-Projekts auf Sourceforge entwickelt [62]. Wie schon in Abschnitt 5.1 erwähnt, spiegeln sich in den Objektverzeichnissen der teilnehmenden Geräte gewöhnlich alle möglichen Konfigurationsoptionen wieder. Deshalb kann ein EPL-Netzwerk bequem über ein einziges Programm wie dem OC eingerichtet werden.

Ein Netzwerk wird wie folgt eingerichtet. Zunächst muss die Objektverzeichnisstruktur des MN im XDD-Format (vgl. Kapitel 2.4.4) geladen werden. Im OC wird eine für den OPLK-MN geeignete Datei bereits mitgeliefert. Anschließend werden Controlled Nodes mit je einer individuellen Knotennummer angelegt. Auch für sie muss eine XDD-Datei geladen werden, die üblicherweise bei einem EPL-Gerät mitgeliefert wird. Das Anlegen bzw. Ändern einer Struktur für ein selbstentwickeltes Gerät ist in OC derzeit leider nicht möglich (Version 2.1.1). XDD-Dateien lassen sich im XML-Format jedoch von Hand editieren.

Die Objektverzeichnisse der eingerichteten Knoten können dann mit konkreten Werten belegt werden. Für die wichtigsten Parameter in den OVs, wie Zykluszeit, Länge des asynchronen Zeitfensters und Poll Response-Timeout, stehen zusätzlich separate Eingabefelder zur Verfügung. Eine sehr wichtige Funktion des OC ist das Konfigurieren der PDO-Mappings. Mit wenigen Klicks werden OV-Einträge TPDOs und RPDOs zugeordnet, sodass sie in `PReq`- und `PRes`-Paketen regelmäßig übertragen werden, und so z.B. zwischen MN und CN abgeglichen werden.

Die fertige Konfiguration kann am Ende als CDC-Datei (*Concise Device Configuration*) exportiert werden, die eine komprimierte Form der OV-Belegungen darstellt. Ein MN lädt diese Datei und gleicht damit in der Initialisierungsphase alle OVs im Netzwerk ab. Das Verfahren ist unter der Bezeichnung *Configuration Manager* in [44, S. 37] standardisiert.

7. Realisierung eines modularen 3D-Druckers

Ein 3D-Drucker hat normalerweise mindestens drei Achsen, um einen Druckkopf und/ oder eine Druckplatte in drei Dimensionen bewegen zu können. Beim Druck wird dann eine vorher berechnete Trajektorie submillimetergenau abgefahren. Das stellt hohe Ansprüche an die Genauigkeit der Ansteuerung der Achsen, insbesondere an deren Synchronität. Angenommen, die Achsen stellen jeweils unabhängige Module dar, die nur über ein Feldbussystem verbunden sind, stellt solch ein Drucker ein anspruchsvolles Testobjekt dar, um das Echtzeitverhalten des Systems zu evaluieren.

7.1. Aufbau

Im Rahmen dieser Arbeit wird ein vorhandener 3D-Drucker mit bis dato zentraler Steuerung auf eben diese Weise umgebaut. Es handelt sich um einen Linear-Delta-Drucker der auf dem Kossel-Modell von Johann C. Rocholl [71] basiert. Der Drucker ist in Abb. 21 zu sehen. Bei Delta-Robotern [64] sorgt eine Aufhängung aus drei Parallelogrammen dafür, dass der Aktor stets in der selben Ausrichtung verbleibt. Der Druckprozess basiert auf dem Schmelzschichtungsverfahren (engl. *Fused Deposition Modeling* [52, S. 137]), bei dem ein Kunststofffilament durch eine heiße Düse gepresst wird, wobei das Objekt Schicht für Schicht von unten nach oben aufgebaut wird.

Ein im Schichtverfahren arbeitender 3D-Drucker auf Basis des Linear-Delta-Robotermodells stellt hohe Ansprüche an die Echtzeitfähigkeiten des Gesamtsystems. Die Kinematik eines Deltaroboters hat die Eigenschaft, dass eine simple lineare Bewegung in einer horizontalen Ebene eine hochsynchrone Bewegung aller drei beteiligten Achsen verlangt. Bei asynchroner Ansteuerung bzw. Verzögerungen, die nur einzelne Achsen betreffen, stellt besonders die dabei entstehende, ungewollte Bewegung in z-Richtung eine Gefahr dar, da die heiße und spitze Düse in bereits fertiggestellte Schichten des Werkstücks eindringen kann. Beim Auftragen der ersten Schichten sind sogar ernsthafte Beschädigungen des Druckers möglich, wenn die Düse auf das harte Heizbett auftrifft.

Der Drucker besitzt vier Schrittmotoren. Drei davon treiben jeweils eine der senkrechten Linearachsen an; der vierte arbeitet als Teil des sog. *Extruders*, zieht also Filament nach und presst es durch eine temperierte Düse (*Hot End*). Außerdem sind drei Endschalter, ein beheizbares Druckbett incl. Temperatursensor, ein Lüfter zur Kühlung des oberen Teils des Druckkopfes sowie Heizelement und Temperatursensor am Hot End Teil des Druckers. Jede Linearachse mit zugehörigem Endschalter hängt an je einem B210E (vgl. Kapitel 4), alle anderen Teile sind über das E4337 angebunden. Die vier Boards sind untereinander und mit einem PC über Ethernet Powerlink verbunden. Dieser Aufbau ist in Abb. 22 dargestellt.

Neben Demonstrationszwecken zur Leistungsfähigkeit unserer Implementierung, verspricht diese Modularität auch eine große Flexibilität und Interoperabilität. Möchte man den Drucker später erweitern, beispielsweise um ein zweites Hot End, genügt es, dieses analog zum ersten an das bestehende EPL-Netzwerk anzuschließen. Es ist nicht nötig, Änderungen an einer zentralen Hauptplatine vorzunehmen, bzw. diese zu ersetzen. Auch die Integration von Fremd-Hardware ist damit möglich, denn viele Hersteller bieten in-

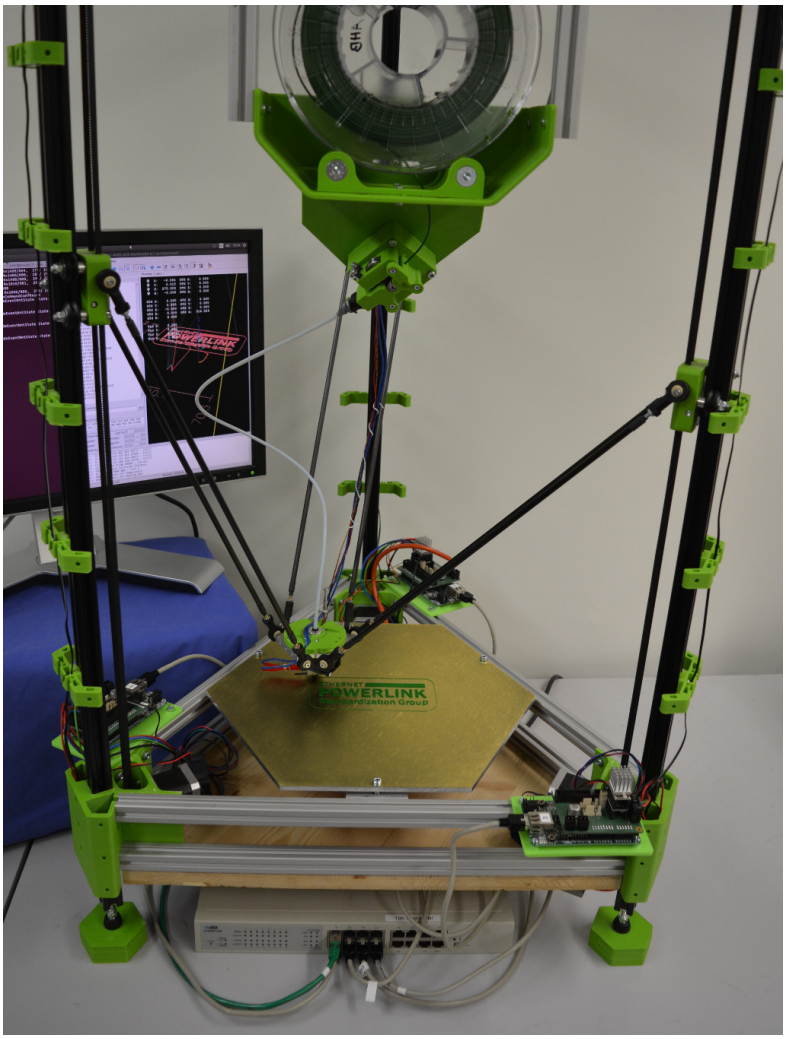


Abbildung 21: Fotografie des modularen 3D-Druckers

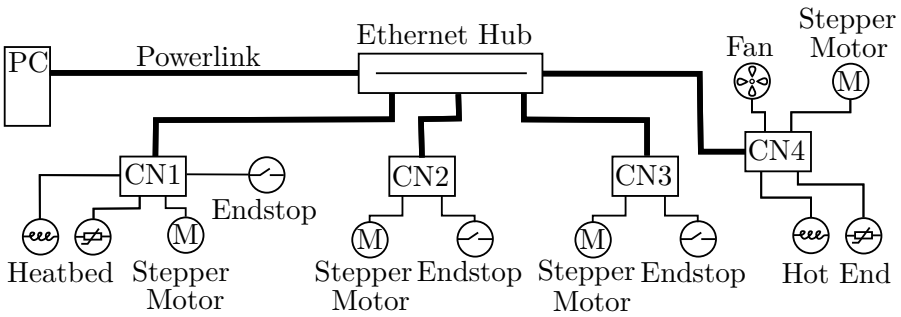


Abbildung 22: Schema des modularen 3D-Druckers

zwischen Motoren und ähnliches an, die ab Werk eine EPL-Schnittstelle besitzen. In [23] wird ein ähnlicher Ansatz verfolgt, um eine CNC-Fräsmaschine mit einem PC und EPL-Servo-Motoren zu betreiben.

7.2. Hardware

Für die mechanischen Bauteile eines Druckers diesen Typs sei auf das RepRap-Wiki [72] verwiesen. Eine Übersicht über die elektronischen Bauteile gibt folgende Liste:

- Motoren: 4× Nanotec ST4118M0906, 2-Phasen-Schrittmotor, 4,59W
- Heizbett mit eingebautem Thermistor, Durchmesser 28cm
- Lüfter zur Kühlung des Druckobjekts, Durchmesser 40mm
- Lüfter zur Kühlung des Druckkopfes, Durchmesser 30mm
- 3× Lüfter zur Kühlung der Schrittmotortreiber, Durchmesser 40mm
- Heizstab für die Druckdüse
- Temperatursensor auf Basis eines Typ-K Thermoelements und der Digitalisierungsschaltung MAX31855 von Maxim; Temperaturbereich: -270°C bis 1372°C, Auflösung: 0.25°C, Genauigkeit: ±2 °C, µC-Anbindung: SPI
- 3× B210E (vgl. Abschnitt 4)
- 1× RD4337 (vgl. Abschnitt 4)
- 4× Adapterplatine (s.u.)

Die Armlänge an der Druckkopfaufhängung beträgt je ca. 481mm. Der Radius des Kreises, der die drei Vertikalachsen schneidet, beträgt 251mm. Es ergibt sich ein zylindrischer Bauraum von ca. 14,8 Litern (Radius 15cm, Höhe 21cm). Die Gesamtgröße des Druckers beträgt 103×66×60cm.

Adapterplatine Zur Anbindung der Sensoren und Aktoren an den Mikrocontroller wurden Adapterplatinen (`printer_cape`) angefertigt, die für jedes der vier Module gleich aufgebaut ist. Zwei MOSFETs (*Metall-Oxid-Halbleiter-Feldeffekttransistoren*) ermöglichen die Ansteuerung größerer Lasten, wie z.B. Heizelemente, an einem GPIO-Pin des Mikrocontrollers (*general purpose input/output*). Dazu gehört auch ein Anschluss zur externen Spannungsversorgung (24V). Mehrere Buchsen reichen Pins des Mikrocontrollers unverändert nach außen weiter: Zwei Analogeingänge zur Verwendung für Temperatursensoren, ein GPIO-Pin zum Anschluss eines Endschalters, und zweimal die erforderlichen Pins zum Anschluss von Peripherie über den SPI-Bus.

Die Ansteuerung des Schrittmotors stellt einen Sonderfall dar. Prinzipiell wäre ein Betrieb über die MOSFET-Anschlüsse möglich. Stattdessen wird jedoch eine Treiberschaltung für zweiphasige Schrittmotoren verwendet, die jeden Umpolungsschritt in 16

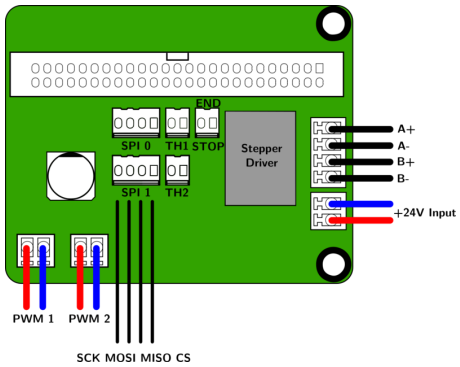


Abbildung 23: Schema der `printer_cape`-Platine

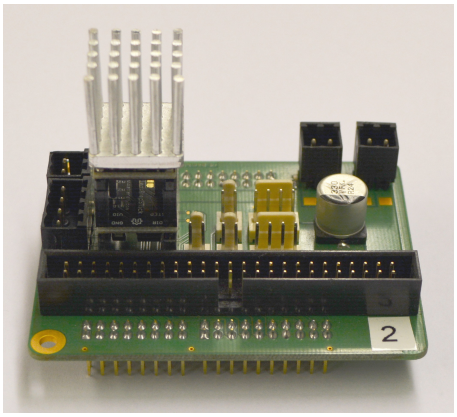


Abbildung 24: Fotografie einer der `printer_cape`-Platinen

Zwischenschritte (sog. *Micro Steps*) unterteilt, sodass das im Motor rotierende Magnetfeld pro Umdrehung nicht nur vier distinkte Positionen annimmt, sondern 64. Die Bewegung wird dadurch flüssiger und vor allem leiser. Als vorgefertigte Schaltung wird der *SilentStepStick TMC2100* von Watterott zusammen mit der Schutzschaltung gegen umgekehrten Stromfluss *SilentStepStick Protector* des selben Herstellers eingesetzt. Sie wird gesteuert über einen digitalen Richtungs- (`DIR`) und einen Schritteingang (`STEP`). Jede steigende Flanke am `STEP`-Eingang hat die Durchführung eines Mikroschritts zur Folge.

Die Verbindung mit dem B210E erfolgt über eine Stiftleiste, die auf Arduino-kompatible Buchsenleisten passt, wie sie auch der B210E bietet. Die Verbindung zum E4337 geschieht über ein Flachbandkabel. Die Abbildungen 23 und 24 zeigen die Adapterplatine als Schemenbild bzw. Fotografie.

7.3. Steuerungssoftware

Beim Betrieb eines 3D-Druckers sind gewöhnlich verschiedene Arbeitsschritte involviert, um von einem 3D-Modell zu den Bewegungsbefehlen eines jeden Motors zu gelangen. Ein weit verbreitetes Format für 3D-Modelle ist STL (*Stereo Lithography* [46]). Dieses muss zunächst in eine Bahnplanung, in unserem Fall für den Druckkopf, übersetzt werden, welche abhängig ist von Filamentdicke und gewünschten Objekteigenschaften. Diesen Prozess nennt man Slicen (von engl. *to slice*), da das Objekt virtuell in Schichten geschnitten wird, die dann nacheinander in einer gegebenen Bewegungsfolge aufgetragen werden sollen. In Abb. 25 findet sich die Visualisierung eines 3D-Objekts nach dem Slicing. Wir verwenden für diesen Schritt das Open-Source-Programm Slic3r [70]. Slic3r erzeugt G-Code [5], eine verbreitete NC-Programmiersprache (*Numeric Control*). Der G-Code enthält dann eine Befehlsfolge, die die Zielpositionen des Extruders vorgibt. Außerdem sind Befehle enthalten, die weitere Parameter des Druckers, wie z.B. die Temperatur des Hot Ends oder die Lüftergeschwindigkeit vorgeben. Diese Befehlsfolge

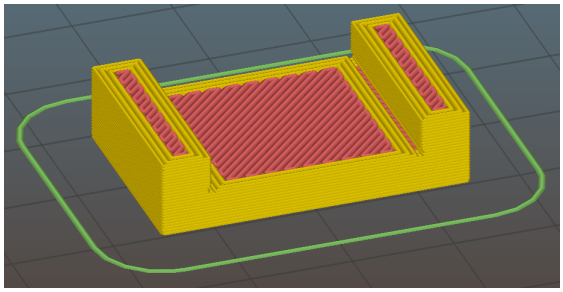


Abbildung 25: Visualisierung eines 3D-Objekts nach dem Slicing mit Slic3r [70]. Die Außenbahnen sind gelb dargestellt; die Innenfüllung ist rot.

muss in Aktionen umgesetzt werden. Die Zielpositionen des Extruders müssen hierfür zunächst in den Gelenkraum rücktransformiert werden (*inverse Kinematik*). Üblicherweise geschieht diese Verarbeitung direkt auf dem Drucker mithilfe einer Controllerplatine und einer Drucker-Firmware. Eine Übersicht über frei verfügbare Drucker-Firmwares liefert [85].

Es ist Teil des modularen Konzepts, dass alle Aktionen, die die Kenntnis des Gesamtsystems voraussetzen, von einem generischen und austauschbaren PC durchgeführt werden. Dazu zählt auch die eben erwähnte inverse Kinematik. Über das EPL-Netzwerk werden lediglich Modul-lokale Stell- und Messgrößen, wie Zieltemperatur und Achsposition, übertragen.

Es wird also ein PC-Programm benötigt, das zwar Teile der Aufgaben einer klassischen Druckerfirmware übernimmt, die Maschine aber nicht direkt steuert, sondern dazu fähig ist, Stell- und Messgrößen an ein Programm weiterzugeben, das den Datenaustausch mit den Modulen über ein Feldbussystem sicherstellt.

Wir haben uns hierbei für das Programm Machinekit (MK) [56], einen Fork von LinuxCNC [53], entschieden, da es die für diese Aufgabe erforderliche Flexibilität aufweist. Abb. 26 zeigt einen Screenshot des Programms. Über eine Hardware-Abstraktionsschicht (*Hardware Abstraction Layer* (HAL) [57]) kann man eine Vielzahl unterschiedlicher Maschinenmodelle auf einer Vielzahl von Plattformen steuern. Der Nutzer kann dabei aus über 150 Bausteinen (*HAL-Komponenten*) wählen [58], diese flexibel über virtuelle Pins und Signale verknüpfen und auch selbstgeschriebene Bausteine beisteuern. Letztere Möglichkeit wird in Form einer der HAL-Komponente `epl_shm` genutzt, die folgende Werte über Shared Memory [77, S.533ff] extern zur Verfügung stellt bzw. entgegen nimmt:

- Soll- und Istposition der drei vertikalen Achsen in Millimeter über dem kalibrierten Ursprung.
- Sollposition des Extruderantriebs in Bogengrad
- Sollgeschwindigkeit aller vier Antriebe in $\frac{\text{mm}}{\text{s}}$
- Endschalter-Signal in $\{0, 1\}$

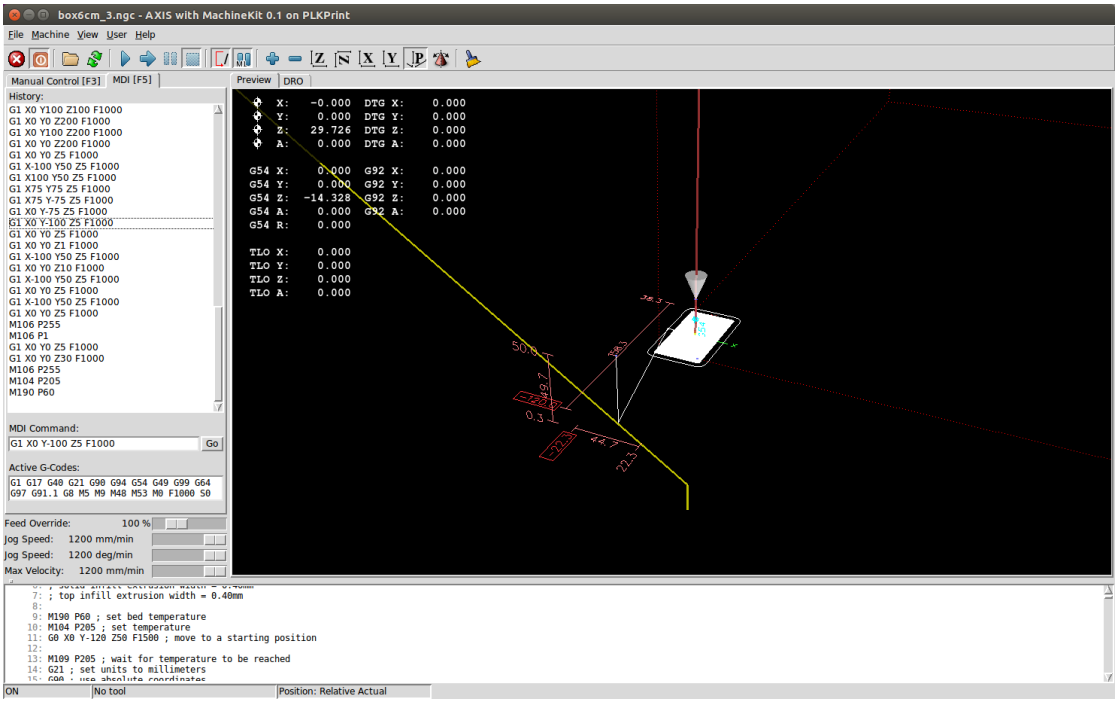


Abbildung 26: Screenshot der Machinekit-Steuerung

- Einschalt-Signal in $\{0, 1\}$
- Soll- und Isttemperatur von Heizbett und Hot End in °C
- Sollgeschwindigkeit des Lüfters als PWM-Wert (*Pulsweitenmodulation*) in $[0, 255]$

Auf diesen geteilten Speicherbereich greift ein weiteres Programm (`printermn`) zu, das als Managing Node (MN) unter Verwendung des openPOWERLINK-Stacks die Kommunikation mit den CNs übernimmt. Die Zugriffe sind über ein Semaphor synchronisiert. Gelesen und geschrieben wird ausschließlich direkt nach einem Sync-Event (vgl. Abschnitt 2.4.1).

Die gesamte Bearbeitungskette ist in Abb. 27 illustriert. Die konkrete Implementierung setzt ein tieferes Verständnis der Funktionsweise von Machinekit voraus, weshalb in Abschnitt 7.3.1 eine kurze Einführung folgt.

7.3.1. Machinekit

Wie schon in Abschnitt 7.3 erwähnt ist die Hardware-Abstraktrionsschicht der zentrale Teil von Machinekit, der es ermöglicht, das Programm sehr flexibel an individuelle Maschinen anzupassen.

Machinekit orientiert sich hierbei am klassischen Systementwurf. Es existieren Komponenten wie Regler, Schritterzeuger, Bewegungssteuerung, Kinematiken, Hardware-

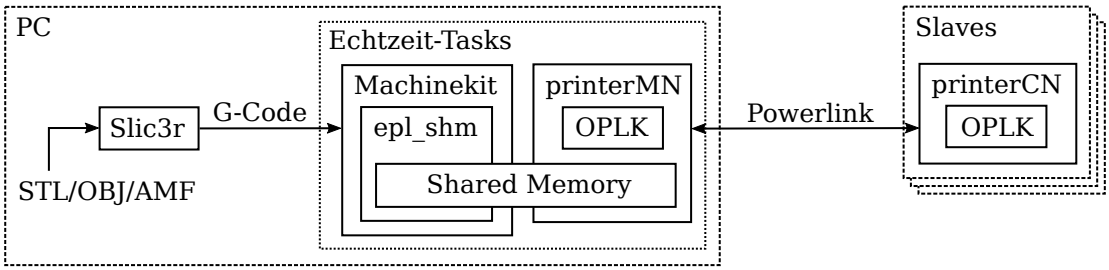


Abbildung 27: Software und Schnittstellen beim Betrieb des 3D-Druckers

```
newthread customservo [EMCMOT]SERVO_PERIOD nowait fp

loadrt lineardeltakins
loadrt tp
loadrt [EMCMOT]EMCMOT servo_period_nsec=[EMCMOT]SERVO_PERIOD num_joints=
[TRAJ]AXES tp=tp kins=lineardeltakins
loadrt ferror_round
loadrt rd_plk_shm num_motors=4
loadrt csv_dump num_columns=16 filepath="machinekit_dump.csv"

addf epl_shm.read customservo
addf motion-command-handler customservo
addf motion-controller customservo
addf epl_shm.write customservo
addf ferror-round.0 customservo
addf csv-dump.0 customservo

setp lineardeltakins.L [MACHINE]CF_ROD
setp lineardeltakins.R [MACHINE]DELTA_R

newsig e0.temp.set     float
newsig e0.temp.meas    float
newsig bed.temp.set    float
newsig bed.temp.meas   float
newsig fan.speed.set   float
newsig fan.speed.meas  float

net e0.temp.meas <= rd_plk_shm.e0-temp-meas
net bed.temp.meas <= rd_plk_shm.bed-temp-meas
net fan.speed.meas <= rd_plk_shm.fan-speed-meas
net e0.temp.set => rd_plk_shm.e0-temp-set
```

```
[EMC]
VERSION =   $Revision$
MACHINE =   EPLPrint7
DEBUG =     0

[MACHINE]
DELTA_R =   251.00
CF_ROD =    481

[TRAJ]
AXES =                          4
COORDINATES =                   X Y Z A
HOME =                          0 0 0 0
LINEAR_UNITS =                  mm
ANGULAR_UNITS =                 degree
CYCLE_TIME =                    0.010
DEFAULT_VELOCITY =              20
DEFAULT_ANGULAR_VELOCITY =      20
MAX_LINEAR_VELOCITY =           50
MAX_ANGULAR_VELOCITY =          50
MAX_ACCELERATION =              500
DEFAULT_ACCELERATION =          500

[AXIS_0]

TYPE =                          LINEAR
HOME =                          686
HOME_OFFSET =                   690
MIN_LIMIT =                     250
MAX_LIMIT =                     695
```

Abbildung 28: Ausschnitt aus der Machinekit-Konfiguration des 3D-Druckers (links: HAL-Datei, rechts: INI-Datei

Schnittstellen, ein virtuelles Oszilloskop und vieles mehr. Die Komponenten haben Anschlüsse (Pins), die über virtuelle Kabel (Signale) verbunden werden.

Der System-Integrator erzeugt für eine Maschine eine HAL-Datei und eine INI-Datei. Ein Ausschnitt aus der für den 3D-Drucker verwendeten Konfiguration findet sich in Abb. 28. In der HAL-Datei stehen Befehle zum Laden der verschiedenen Module, es werden passende Parameter gesetzt und Pins werden über Signale mit anderen Pins verbunden. Wenn Teile der Maschine sich nicht mit den vorhandenen Modulen abbilden lassen, gibt es die Möglichkeit, unkompliziert ein selbst entwickeltes C- oder Python-Modul zu laden und diese genau wie bei vorgefertigten Modulen über Pins zu vernetzen.

Die INI-Datei enthält Parameter, wie Anzahl und Typ der Achsen, Beschleunigungen und Geschwindigkeiten und auch Metadaten, wie Name und Version der Konfiguration und ein Verweis auf die HAL-Datei.

Machinekit ist echtzeitfähig und unterstützt dabei verschiedene Plattformen. Für den System-Integrator ist das dank der Echtzeitabstraktionsschicht `rtapi` transparent. Er er-

zeugt in der HAL-Datei einen Echtzeitthread und ordnet ihm Funktionen zu, die zyklisch aufgerufen werden sollen. Funktionen sind wie Pins immer Teil einer HAL-Komponente und die Zuordnung von Funktionen zu Threads ähnelt der Verbindung von Pins zu anderen Pins.

Im Folgenden sollen einige Begriffe und Komponenten aufgelistet werden, die in dieser Arbeit Verwendung finden.

emcmot `emcmot` (*Enhanced Machine Controller: Motion* [59]) ist als Bewegungssteuerung das zentrale Modul von Machinekit bei der Steuerung einer Maschine. Es ist dafür zuständig, anhand einer Folge von Zielpositionen zyklisch Bewegungsparameter wie Zielposition und Geschwindigkeit auszugeben und dabei alle Rahmenbedingungen einzuhalten. Als Rahmenbedingungen existieren z.B. Soll- und Maximalgeschwindigkeiten sowie Maximalbeschleunigungen bzw. Bewegungsprofile. Dabei findet eine Interpolation statt. Das Modul arbeitet als reine Steuerung, d.h. tatsächliche Gelenkpositionen fließen nicht in die Berechnungen ein[17].

Kinematik Mit *Kinematik* ist hier eine HAL-Komponente gemeint, die dazu fähig ist, Weltkoordinaten in Gelenkwinkel bzw. Stellungen aller Freiheitsgrade der Maschine zu transformieren und umgekehrt. Die Kinematik wird an die Bewegungssteuerung `emcmot` übergeben. Für den 3D-Drucker wird die Komponente `lineardeltakins` verwendet, die eine Linear-Delta-Maschine anhand ihres Radius und der Armlänge zwischen Aufhängung und Endeffektor modelliert.

HAL-Thread Mit `newthread` *`Name Periode`* wird ein neuer Echtzeitthread erzeugt, der dann zur Erledigung von Aufgaben anderer HAL-Komponenten verwendet werden kann. Mit `addf` *`Funktionsname Threadname`* wird ihm eine Funktion einer Komponente zugeordnet, die regelmäßig mit der genannten Periode ausgeführt wird. Wenn mehrere Funktionen zugeordnet sind, werden sie strikt in der angegeben Reihenfolge ausgeführt. Ein Thread kann mit der Zusatzoption `nowait` erzeugt werden. Die Ausführung folgt dann nicht einer festen Taktrate, sondern beginnt nach jedem Durchgang erneut von vorne. Das ist sinnvoll, wenn eine externe Synchronisierung verwendet wird.

Signal Ein Signal ist eine Verbindung zwischen zwei oder mehr Pins und wird mit dem Befehl `net` *`Signal Quell-Pin Ziel-Pin`* erzeugt. Der Wert eines Signal wandert in einem Schritt immer nur eine Komponente weit. Ein Schritt ist dabei durch die Ausführung der verbundenen HAL-Komponente definiert. Ist beispielsweise Komponente `A` über ein Signal mit `B` verbunden und `B` wiederum mit `C` und alle Komponenten (bzw. deren Funktion, die die Pins verwendet) sind dem selben Thread zugeordnet, dann ist ein in Zyklus x geänderter Wert am Ausgangs-Pin von `A` erst in Zyklus $x+2$ am Eingangs-Pin von `C` verfügbar.

[17] Theoretisch ist auch eine Regelung mithilfe eines zusätzlichen PID-Moduls möglich. Dies funktioniert jedoch nicht ohne weiteres bei verzögertem Empfang der Ist-Position. Vgl. hierzu Abschnitt 7.6

Tabelle 13: Einträge im OV, die über das Netzwerk übertragen werden

Variable	**Adresse**	**Funktionalität**
ActualExtruderTemp	0x2000	Ist-Wert Extruder-Temperatur in °C
ActualBedTemp	0x2001	Ist-Wert Heizbett-Temperatur in °C
EndStop	0x2010	Enschalter gedrückt - ja/nein?
Enabled	0x3000	Motor aktiviert - ja/nein?
Fan1Speed	0x3010	Lüftergeschwindikeit 1 (Objektkühler)
Fan2Speed	0x3011	Lüftergeschwindikeit 2 (ungenutzt)
TargetExtruderTemp	0x3020	Sollwert Extruder-Temperatur in °C
TargetBedTemp	0x3021	Sollwert Heizbett-Temperatur in °C
PositionActualValue	0x6063	Motorposition in Schritten
TargetPosition	0x607A	Zielposition in Motor-Schritten
TargetVelocity	0x60FF	Zielgeschwindikeit in Schritten pro Sekunde

7.4. Controlled Node Software

Das auf den vier Controlled Nodes laufende Programm `printercn` verwendet den portierten openPOWERLINK-Stack und ist auf allen Knoten identisch. Leicht unterschiedliche Verhaltensweisen ergeben sich lediglich durch eine Rollenzuweisung anhand der Knotennummer, die den logischen Funktionen ihre Schnittstelle zuweist (beispielsweise *Extruder Temperatur* zu `SPI1` auf Knoten 4). Eine Übersicht der Zuweisungen findet sich in Tabelle A.1. Zur einfacheren Entwicklung von CN-Anwendungen auf Basis von openPOWERLINK und NuttX wurde die Bibliothek `oplkcnlib` entwickelt, die neben der umfangreichen, aber in dieser Konstellation meist gleichbleibenden Initialisierung des Stacks auch Aufgaben, wie das zyklische Aufrufen der Event- und Timer-Module (vergleiche Kapitel 5.2.2) und die Verwaltung des Prozessabbildes übernimmt. Die typische OPLK-Interaktion eines Anwendungsprogramms reduziert sich unter Verwendung von `oplkcnlib` so auf das Bereitstellen von Knotennummer und einer beliebigen Funktion, die im ganzen Netzwerk synchron aufgerufen wird[18], sowie der Zuordnung Einträgen im Objektverzeichnis zum Prozessabbild. Letzteres kann entfallen, wenn Daten mit `oplk_readLocalObject()` und `oplk_writeLocalObject()` direkt mit dem Objektverzeichnis ausgetauscht werden.

7.5. Ethernet POWERLINK Kommunikation

Alle Stell- und Messgrößen des Systems lassen sich mit den in Tabelle 13 aufgelisteten Variablen darstellen, die im Objektverzeichnis aller verwendeten Module abgelegt sind. Die Werte werden mit je vier Receive- und Transmit-PDOs zyklisch zwischen MN und den CNs ausgetauscht. Unter Verwendung von *Dynamic Index Assignment* [44, S. 51] kann auf MN-Seite in Form eines Arrays von Prozessabbildern auf die Werte aller CNs im Netzwerk zugegriffen werden.

[18] Hier sollte die Aktualisierung bzw. Übernahme von echtzeitkritischen Werten erfolgen.

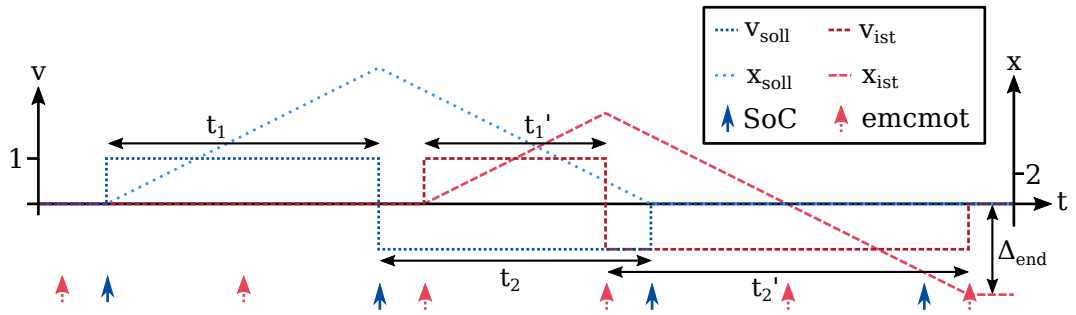


Abbildung 29: Geschwindigkeits- und Bewegungsprofil einer Achse (Ist- und Sollwerte) bei einer naiven Implementierung mit $T_{\text{EPL}} = 6 \neq 4 = T_{\text{MK}}$.

7.6. Synchronisierung

Eine der größten Herausforderungen bei der Realisierung des 3D-Druckers mit der oben genannten Softwarearchitektur ist die korrekte und möglichst verzögerungsarme Synchronisierung zwischen dem CNC-Kern `emcmot` von Machinekit und der EPL-Kommunikation.

Ziel ist ein möglichst exaktes Folgen einer geplanten Trajektorie. Die Differenz zwischen Soll- und Istposition wird Schleppfehler oder *Following error* (FE) genannt. Bei der in MK üblichen direkten Ansteuerung der Motoren, z.B. über den Parallel-Port des PCs, ist der FE gewöhnlich klein und nach auch oben begrenzt, wenn Maschineneigenschaften wie Elastizität und Motorkraft sich in den definierten Rahmenbedingungen (max. Beschleunigung etc.) wiederspiegeln.

Schrittmotoren können bei zu großer Beschleunigung oder zu großem Widerstand Schritte verlieren. Dieser Fehler fällt in Software nur auf, wenn ein eigenständiger Encoder verwendet wird. Wenn diese Fehler auftreten, ist der FE nicht mehr beschränkt. Erneutes Homing (vgl. Anhang A.4) und eine Anpassung der Parameter ist nach solch einem Fehler unerlässlich. Mit der Angabe des maximalen FEs (auf Wunsch in Abhängigkeit zur Geschwindigkeit) lässt sich erreichen, dass das Programm pausiert, wenn die Abweichung zu groß wird, sodass Beschädigungen vermieden werden.

Ein spezieller Typ Schleppfehler entsteht, wenn durch ein Feldbussystem größere Latenzen entstehen, besonders, wenn diese nicht konstant sind. Im folgenden Abschnitt soll dieser Typ genauer betrachtet werden.

Feldbusanbindung Im vorliegenden System kommen mit EPL ein zyklisch arbeitender Feldbus (Periode T_{EPL}, Phase φ_{EPL}) und mit Machinekit eine zyklisch arbeitende Steuerung (T_{MK}, φ_{MK}) zum Einsatz. EPL garantiert zusätzlich, dass übertragene Daten zu Beginn des nächsten Zyklus (`SoC`) im ganzen Netzwerk zeitgleich gültig werden.

Eine naive Umsetzung lässt beide Zyklen unabhängig voneinander laufen ($\varphi_{\text{EPL}} \neq \varphi_{\text{MK}}$). Die Verzögerung von der Berechnung einer Zielposition und -Geschwindigkeit bis zur Berücksichtigung dieser Werte im CN liegt in diesem Fall zwischen der Dauer eines

Zyklus ($l_{min} = T_{\mathrm{EPL}}$) und der Dauer zweier Zyklen ($l_{max} = 2 \cdot T_{\mathrm{EPL}}$)[19]. Sind zusätzlich die Perioden verschieden ($T_{\mathrm{EPL}} \neq T_{\mathrm{MK}}$), unterliegt die Latenz einer Schwingung, die sich in nicht nur in der zeitlichen Verschiebung der Bewegung äußert, sondern diese auch räumlich ändert. Diese Situation ist in Abb. 29 dargestellt:

Es sei $T_{\mathrm{EPL}} = 4$ und $T_{\mathrm{MK}} = 6$. `emcmot` starte mit einer anfänglichen Phasenverschiebung von $ps = 1$. Die Ereignisse *`emcmot` wird ausgeführt* und *`SoC`-Sync-Event* sind als Pfeile dargestellt. Neu berechnete Werte werden immer mit dem folgenden Sync-Event übernommen und daraufhin übertragen. Mit dem übernächsten Sync-Event hat der CN die neuen Werte erhalten (`PReq` wurde empfangen und Prozessabbild wurde aktualisiert) und aktualisiert seine Bewegungsparameter.

Es soll eine einfache Hin- und Rückbewegung durchgeführt werden, indem die Sollgeschwindigkeit für sechs Zeiteinheiten ($t_1 = 1 \cdot T_{\mathrm{MK}}$) auf 1 gesetzt wird und danach für die gleiche Zeitspanne ($t_2 = t_1$) auf -1. Die Übernahme der Werte an verschiedenen Zeitpunkten innerhalb von T_{MK} und die Tatsache, dass Bewegungsparameter auf dem CN immer für die Dauer eines ganzzahlig Vielfachen von T_{EPL} gültig sind, sorgt dafür, dass die reale Hinbewegung die Dauer $t_1' = 4 \neq t_1$ hat und die Rückbewegung $t_2' = 8 \neq t_2$. Das Resultat ist ein deutlich verändertes Bewegungsprofil und eine um $\Delta_{\mathrm{end}} = -2$ verschobene Endposition. Man kann dieses Verhalten auch als eine Unterabtastung eines diskreten Signals betrachten.

Bei einer starken Überabtastung, d.h. $T_{\mathrm{EPL}} \ll T_{\mathrm{MK}}$, nähert sich das Resultat bei wachsender Differenz einer Lösung mit unmittelbar angeschlossener Hardware an. Dies ist jedoch schwer zu erreichen, da zum einen der Zykluszeit technologisch untere Grenzen gesetzt sind und zum anderen der 3D-Druckprozess nicht mit beliebig großen T_{MK} funktioniert, da er an vielen Stellen schnelle Richtungswechsel voraussetzt (z.B. beim sogenannten Retract, vgl. Anhang A.4). Machinekit arbeitet standardmäßig mit $T_{\mathrm{MK}} = 1ms$; negative Auswirkungen auf den Druckprozess wurden bereits ab $T_{\mathrm{MK}} \geq 10ms$ experimentell festgestellt.

Exakte Synchronisierung Zufriendenstellende Ergebnisse sind nur mit einer exakten Synchronisierung der beteiligten Komponenten erreichbar. Die Schwierigkeit dabei ist, zwei sich zyklisch wiederholende Tasks in Phase zu bringen, ohne Zeitbeschränkungen zu verletzen. Die Phase der EPL-Kommunikation sei als gegeben angesehen, denn auch kleinste Verletzungen der im Objektverzeichnis angegebenen Timeout-Werte werden mit einem Neustart der Kommunikation bestraft. Also muss sich Machinekit an openPOWERLINK, genauer am Synchronisationszeitpunkt des EPL-Netzwerks, orientieren. Standardmäßig ruft Machinekit jede Millisekunde (T_{MK} frei wählbar) in einem dedizierten HAL-Thread das Modul `emcmot` auf, interpoliert die Bewegung also in T_{MK}-Schritten.

In der vorliegenden Implementierung, die in Abb. 30 illustriert ist, wird der Thread durch einen `nowait`-HAL-Thread ersetzt (vgl. Abschnitt 7.3.1), um dann wie folgt selbst die korrekte Taktung zu gewährleisten:

[19] Die Zeit zwischen `SoC` und der ersten `PReq`-Nachricht, in der an das Sync-Event gekoppelte Aufgaben erledigt werden, wird hier vernachlässigt.

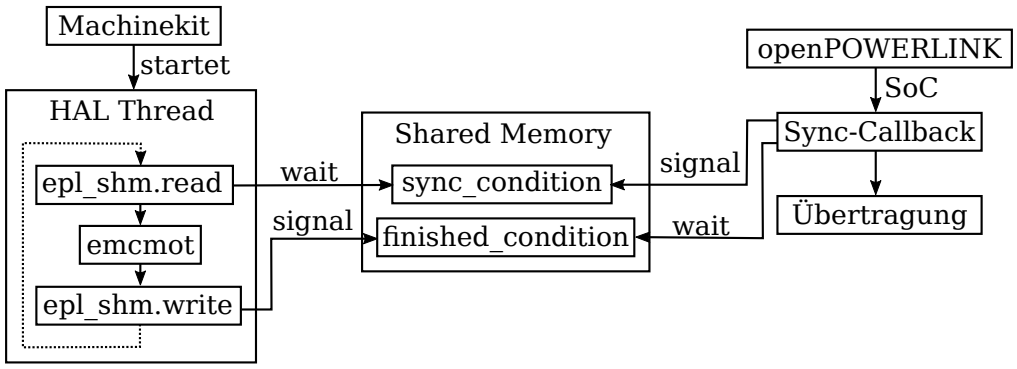


Abbildung 30: Synchronisierung zwischen Machinekit und openPOWERLINK-MN

Dem Thread werden zwei zusätzliche Funktionen `epl_shm.read` und `epl_shm.write` zugeordnet. `epl_shm.read` wartet mithilfe der POSIX Bedingungsvariable [8, S.70] `sync_condition` auf ein Signal vom MN, das direkt nach einem Sync-Event gesendet wird. Daraufhin werden alle empfangenen Daten, also das Eingangs-Prozessabbild, über einen gemeinsamen Speicherbereich (*POSIX Shared Memory*) an Machinekit weitergeleitet und als HAL-Pins zur Verfügung gestellt. Über Signale verbunden, stehen diese dann im folgenden Zyklus zur Schleppfehlerberechnung zur Verfügung. `epl_shm.write` nimmt über HAL-Pins die in `emcmot` berechneten Sollwerte entgegen und sendet sie über den gemeinsamen Speicherbereich an den MN. Zusätzlich wird der Abschluss der Berechnungen über die Bedingungsvariable `finished_condition` mitgeteilt. Das Ausgangs-Prozessabbild ist damit für die Übertragung freigegeben. Es sei darauf hingewiesen, dass zwischen dem Bereitstellen eines neuen Sollwerts und der Übertragung zum MN aufgrund der Verwendung von HAL-Signalen ein Zyklus vergeht.

Die zusätzlichen Verzögerungen durch die HAL-Architektur stellen kein Problem dar, da sie konstant sind und lediglich in der Schleppfehlerberechnung berücksichtigt werden müssen. Wichtig ist dagegen, dass die Berechnungsdauer in Machinekit, welche sich leicht an den HAL-Pins `epl_shm.read.tmax`, `epl_shm.write.tmax`, `motion-command-handler.tmax` und `motion-controller.tmax` ablesen lässt, abgestimmt ist auf die im MN konfigurierte Wartezeit zwischen Zyklusbeginn und Versenden der ersten `PReq`-Nachricht (OV-Eintrag `WaitSoCPReq`). Nur so ist garantiert, dass ein neues Ausgangs-Prozessabbild noch im selben Zyklus übertragen wird.

Diese Implementierung stellt sicher, dass der Schleppfehlerberechnung immer die Ist-Werte vorliegen, die genau am zwei Zyklen zurückliegenden Synchronisationszeitpunkt im CN gültig waren und dass neu berechnete Soll-Werte genau am übernächsten Synchronisationszeitpunkt im CN übernommen werden. In Abb. 31 ist ein Zyklus dargestellt, in dem neben der Gesamt-Latenz auch ersichtlich wird, wie die Übertragung immer genau zwischen zwei Interpolationsschritten stattfindet.

Tatsächlicher Schleppfehler Es liegt in der Natur des Verfahrens, dass die zum Stellwert passenden Ist-Werte immer um fünf Zyklen versetzt bei Machinekit ankommen.

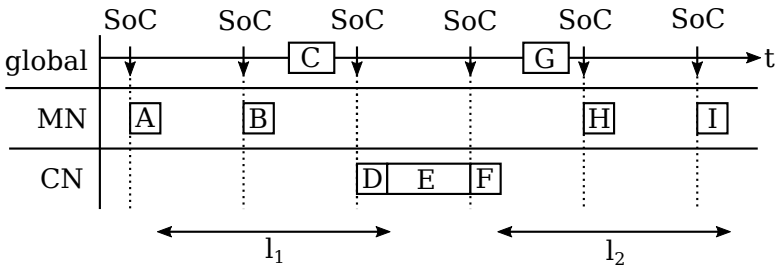


Abbildung 31: Zeitliche Abfolge einer EPL-Übertragung von der Berechnung einer neuen Soll-Position in Machinekit bis zur Rückmeldung des Wertes als neue Ist-Position und der folgenden Berechnung des Schleppfehlers.

l_1 und l_2: Latenz durch die EPL-Übertragung und HAL-Verknüpfung
A: `emcmot` berechnet neue Zielposition x_i
B: x_i wird über HAL-Pin an den MN weitergegeben
C: x_i wird übertragen
D: x_i ist im CN verfügbar
E: Bewegung wird ausgeführt
F: x_i wurde erreicht. Halte fest als neue Ist-Position x_i'
G: x_i' wird übertragen
H: x_i' ist im MN verfügbar und wird an HAL übergeben
I: x_i' ist in Machinekit verfügbar. Berechne Schleppfehler $x_i' - x_i$

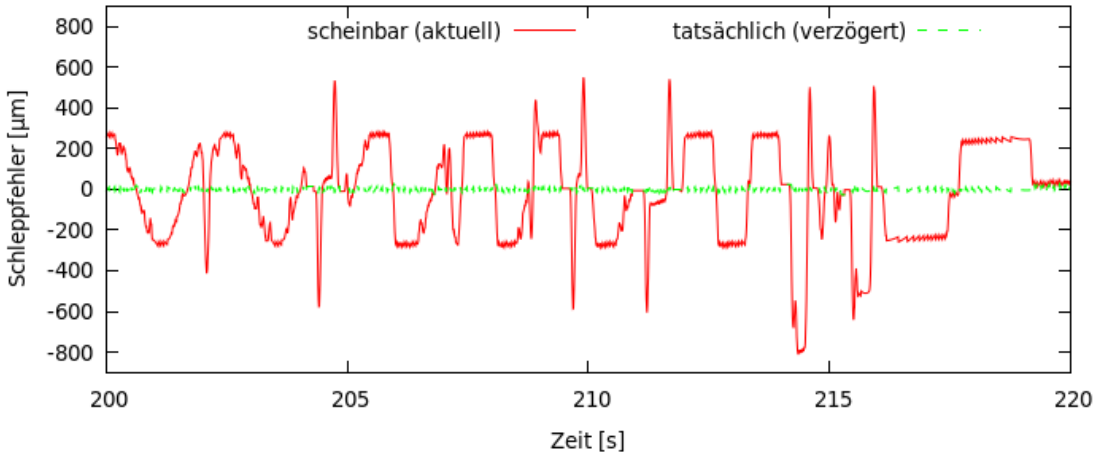


Abbildung 32: Vergleich des scheinbaren Schleppfehlers, der von Machinekit durch Differenzbildung von akutellem Soll- und Istwert berechnet wird, mit dem tatsächlichen Schleppfehler unter Berücksichtigung der Rückmeldungs-Latenz von 5 Zyklen. Es ist ein 20s-Ausschnitt des Schleppfehlers von Achse α während des Drucks von Testobjekt 3 zu sehen (vgl. Abschnitt 7.7).

Ein Zyklus Verzögerung ist inhärent durch die zyklussynchrone Interpolation gegeben, da in dieser Zeit der (scheinbar) diskrete Bewegungsschritt ausgeführt wird. Zwei Zyklen Verzögerung sind der EPL-Übertragung von Stell- und Ist-Wert geschuldet. Schließlich verursacht die Verwendung der Machinekit-HAL weitere zwei Zyklen Verzögerung. Da MK davon ausgeht, dass alle Messwerte unmittelbar zur Verfügung stehen, hat die zusätzliche Latenz zur Folge, dass der von MK berechnete Schleppfehler wesentlich höher ausfällt, als er in Wirklichkeit ist. In Abb. 32 werden beide Varianten gegenübergestellt: Zum einen der Schleppfehler, wie er von Machinekit angezeigt wird und zum anderen der tatsächliche Schleppfehler der insgesamt verzögerten Bewegung.

7.7. Evaluierung

Die Leistungsfähigkeit des 3D-Druckers wird zum einen praktisch anhand der Qualität verschiedener gedruckter Testobjekte gezeigt und zum anderen anhand des auftretenden Schleppfehlers exakt bestimmt. Zusätzlich wird in diesem Kapitel auch die Synchronität der Schrittmotoren überprüft.

7.7.1. Datenerhebung und Diskretisierungsfehler

Der Schleppfehler wird mit einem eigens entwickelten HAL-Modul registriert. In jedem Zyklus werden die Soll- und Istpositionen aller beteiligten Achsen in eine CSV-Datei geschrieben.

Der Schleppfehler ist eine theoretische Größe, die verfahrensbedingte Abweichungen erfasst. Jitter und Diskretisierung sind dabei die Hauptquellen für derartige Fehler. Me-

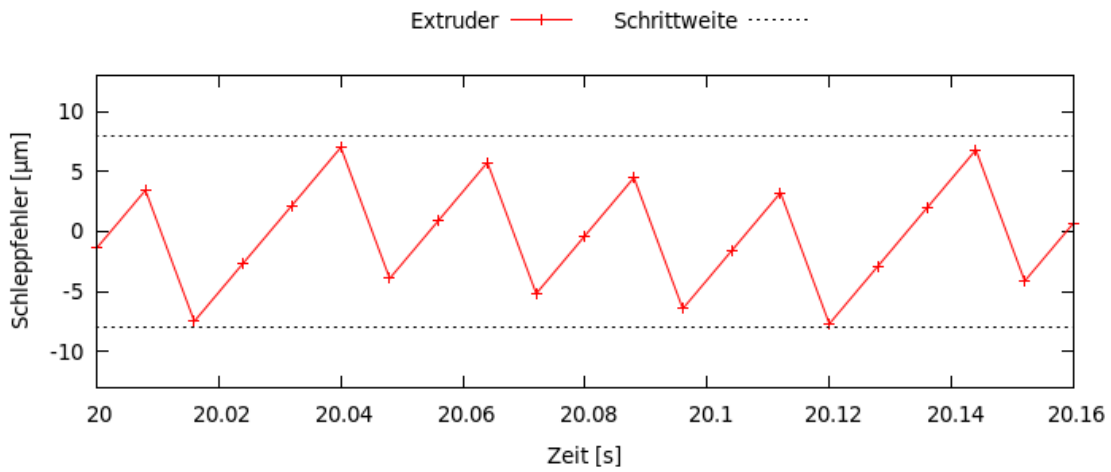


Abbildung 33: Visualisierung des Diskretisierungsfehlers über 20 Zyklen. Es handelt sich um einen 16ms-Ausschnitt des Extruder-Schleppfehlers beim Druck von Testobjekt 3.

chanisch bedingte Abweichungen werden hiermit ausdrücklich nicht erfasst. Diese entstehen z.B. durch ungenaue Bestimmung der Druckerparameter, durch temparaturbedingte Schwankungen und durch mangelnde Steifigkeit und Vibrationen. Die Berücksichtigung solcher Abweichungen würde den Rahmen dieser Arbeit sprengen. Sie werden deshalb als hinreichend klein angenommen.

Da Schrittmotoren verwendet werden, sind Bewegungen nur in diskreten Einheiten möglich (Mikroschritte, vgl. Abschnitt 7.2). Für die drei Vertikalachsen beträgt die Schrittweite ca. 25µm, bei der Extruderachse sind es ca. 16µm. Jede Abweichung lässt sich also auch in Mikroschritten ausdrücken.

Machinekit arbeitet intern mit Fließkommazahlen in der Einheit Millimeter. Bei der Umrechnung in Motorschritte muss gerundet werden. Im Bereich eines Mikroschritts ist eine Abweichung vom Sollwert deswegen als Diskretisierungsfehler unvermeidlich. Abb. 33 stellt diesen Fehler anschaulich in einem 16ms-Ausschnitt des Drucks von Testobjekt 3 (siehe unten) dar. Nur Abweichungen, die betragsmäßig über der Hälfte der Schrittweite liegen, können als vermeidbare Fehler angesehen werden.

7.7.2. Testdrucke

Es kommen insgesamt drei Testobjekte zum Einsatz. Die bei den Tests eingesetzten Kommunikationsparameter sind in Tabelle 14 aufgelistet. Die wichtigsten Werte der Slic3r-Konfiguration stehen in Tabelle 15.

Testobjekt 1: Würfel Als erstes einfaches Testobjekt kommt ein Würfel der Größe 1×1×1cm zum Einsatz. Durch die hohe Füllrate ist das Objekt sehr stabil und unanfällig für Verformungen beim Abkühlen. Mit dem Druck von Außenwand, Deckschicht,

Tabelle 14: EPL-Konfiguration beim Druck der Evaluierungsobjekte (vgl. Kapitel 6.2)

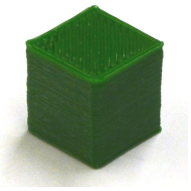
t_{cycle}	8ms
t_{async}	1,2ms
$t_{\text{PRes,i}} \forall i$	1,2ms
t_{wait}	2ms

Tabelle 15: Slic3r-Parameter für die Evaluierungsobjekte

Filamentdurchmesser	1,75mm
Düsendurchmesser	0,4mm
Düsentemperatur	205°C
Schichtdicke	0,2mm
Geschw. Ränder	$15\frac{mm}{s}$
Geschw. kleine Ränder	$10\frac{mm}{s}$
Geschw. sichtbare Ränder und Flächen	$10\frac{mm}{s}$
Geschw. Füllung	$25\frac{mm}{s}$
Geschw. Eilgang	$30\frac{mm}{s}$
Geschw. Ränder	$15\frac{mm}{s}$

Zick-Zack- sowie Honigwaben-Füllung und Retract sind die typischen Druckbewegungen enthalten. Größere Abweichungen in der Ansteuerung würden anhand von ungeraden Linien, falscher Größe, Winkeln ungleich 90°, oder schwankender Schichtdicke sichtbar werden. Breite und Tiefe entsprechen genau dem Sollwert (Messgenauigkeit 0,05mm). Der Würfel ist mit 10,2mm etwas zu hoch. Das liegt an dem konservativ kalibrierten Abstand zwischen dem Druckkopf im Ursprung und der Druckplatte. Dadurch wird die erste Schicht etwas dicker als geplant. Der Effekt lässt sich verringern, indem der Abstand bei Betriebstemperatur regelmäßig neu kalibriert wird. Da der Wert nicht von der Objektgröße abhängig ist, wird ein verfahrensbedingter Fehler ausgeschlossen. Alle anderen Werte treffen genau die Erwartungen. Der verfahrensbedingte Schleppfehler beträgt höchstens zwei Mikroschritte (0,32% der Messwerte). Abb. 34 zeigt ein Foto des Würfels und das zugehörige Schleppfehlerdiagramm.

Testobjekt 2: Wand Als zweites wird eine Wand mit der Stärke einer einzelnen Linie gedruckt. Damit lässt sich ermitteln, ob die Bewegung des Druckkopfes deterministisch ist. Es handelt sich um ein Quadrat mit abgerundeten Ecken. Das Profil der Wand wird fortlaufend nachgefahren in einer nach jeder Umrundung um 0,2mm nach oben verschobenen Ebene. Variierende Fehler wären hier leicht erkenntlich, da die Schichten dann nicht perfekt aufeinander passen. Im vorliegenden Objekt sind jedoch keine nennenswerten Unebenheiten zu ertasten. Der maximale Schleppfehler beträgt 69µm. Wie im Diagramm aus Abb. 35 zu sehen ist, befindet sich die hohe Abweichung ganz am Anfang. In diesem Bereich findet eine Bewegung im Eilgang ($30\frac{mm}{s}$) zum Startpunkt statt. Dort findet keine Extrusion statt. Während des eigentlichen Drucks entspricht der



Objekt 1: Würfel	
Größe	10 × 10 × 10 mm
Druckdauer	9:59min
max. Schleppfehler	47µm (Achse α, ±2 Mikroschritte)

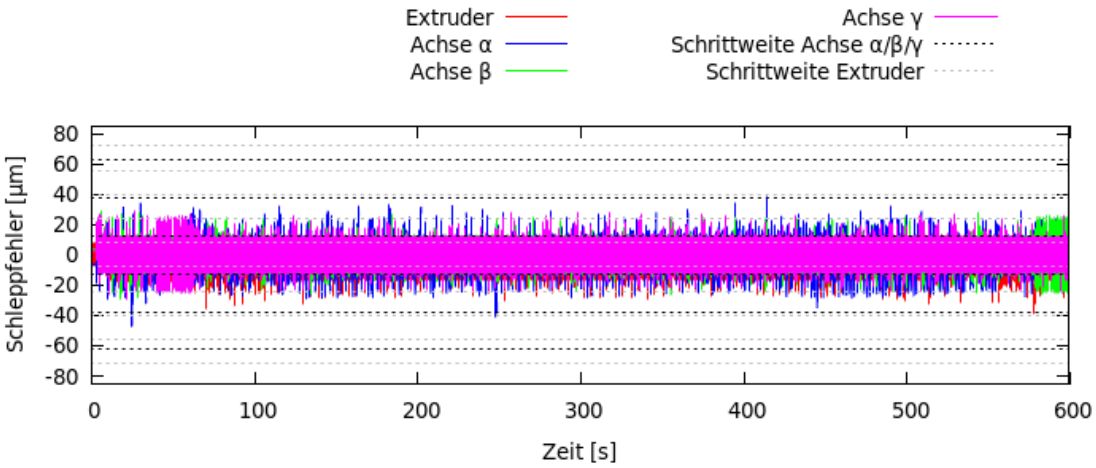


Abbildung 34: Fotografie, Eigenschaften und Schleppfehlerdiagramm zu Testobjekt 1 (Würfel)

Objekt 2: Wand	
Größe	20 × 20 × 10 mm
Druckdauer	6:49min
max. Schleppfehler	69µm (Achse γ, ±3 Mikroschritte)

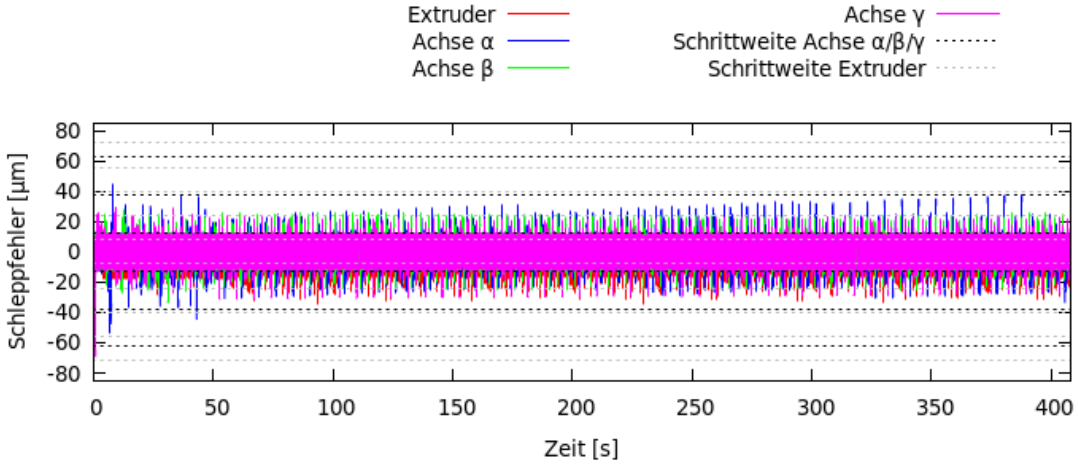


Abbildung 35: Fotografie, Eigenschaften und Schleppfehlerdiagramm zu Testobjekt 2 (Wand)

Schleppfehler höchstens 2 Mikroschritten (an insg. vier Stellen).

Testobjekt 3: EPSG-Logo Nach den zwei eher simplen Objekten soll nun ein filigraneres Objekt betrachtet werden: Das Logo der Ethernet POWERLINK Standardization Group. Besonders die in diesem Objekt vorkommenden kleinen Schriftzeichen lassen eine gute Bewertung der Gesamtleistung zu, da dem menschlichen Auge Unregelmäßigkeiten in dem Schriftbild sofort auffallen würden. Das Objekt ist darüber hinaus anspruchsvoll, da viele Lücken zu überbrücken sind, an denen Retract stattfindet. Das äußert sich in einem maximalen Schleppfehler von vier Mikroschritten an der Extruder-Achse. Die größte Abweichung an den vertikalen Achsen beträgt drei Schritte. Kleine sichtbare Fehler im Objekt sind vor allem prozesstechnischer Natur und kommen so auch in kommerziell erhältlichen 3D-Druckern vor. So war während des Drucks die Haftung am Druckbett nicht immer ausreichend, wodurch an den betreffenden Stellen der Kunststoff etwas verklumpt ist. An einigen Stellen sind auch feine Kunststofffäden zu sehen, die beim Überspringen von Lücken entstehen. Ursache ist vermutlich eine nicht optimale Düsentemperatur.

Fazit Zu Beginn der Arbeit an dem Drucker war nicht erwartet worden, dass mit der relativ großen Zykluszeit und der ursprünglich nicht dafür vorgesehenen Software so

Objekt 3: EPSG-Logo	
Größe	40 × 116 × 0,3 mm
Druckdauer	12:00min
max. Schleppfehler	61µm/80µm (Extruder/Achse α, $\pm 4/\pm 3$ Mikroschritte)

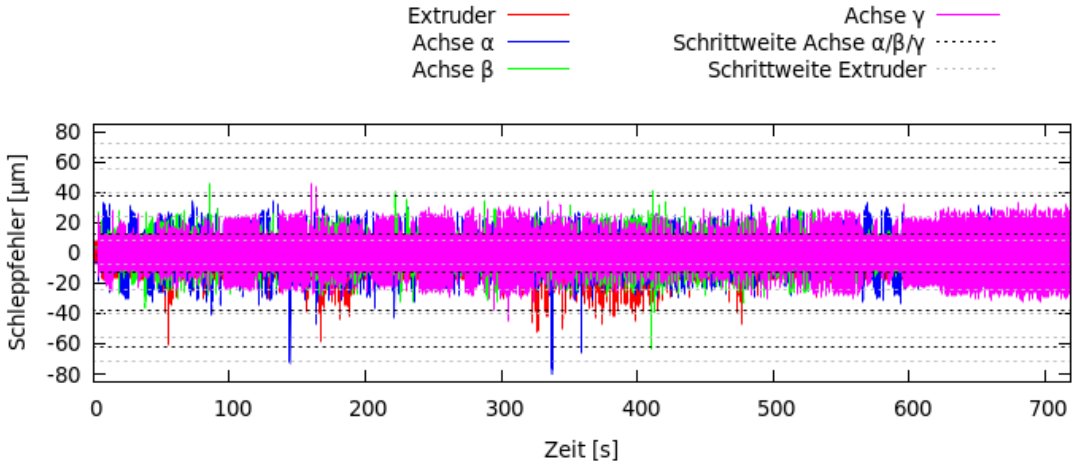


Abbildung 36: Fotografie, Eigenschaften und Schleppfehlerdiagramm zu Testobjekt 3 (EPSG-Logo)

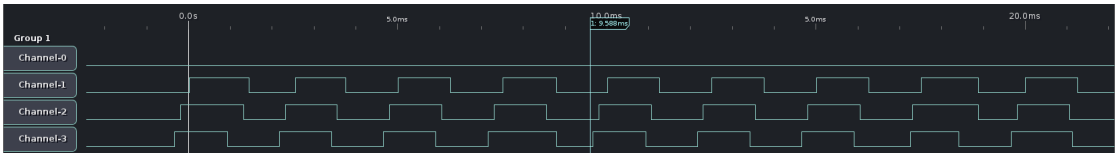


Abbildung 37: Signal am Schritteingang des Motortreibers; aufgenommen an drei CNs ohne Timer-Rücksetzung

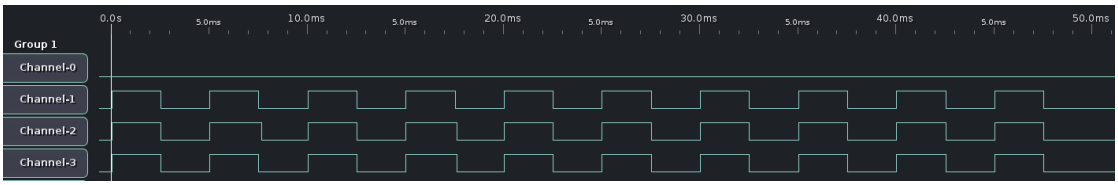


Abbildung 38: Signal am Schritteingang des Motortreibers, aufgenommen an drei CNs mit Timer-Rücksetzung

gute Ergebnisse erreicht werden können. Die Objekte sind optisch nahezu fehlerfrei und die verfahrensbedingten Abweichungen sind so klein, dass sie vor prozesstechnischen Problemen und Abweichungen in den Hintergrund treten bzw. von ihnen überdeckt werden.

Die weitere Verbesserung des Druckers sollte sich deswegen auf den Druckprozess konzentrieren. Wesentliche Verbesserungsmöglichkeiten, die thematisch in anderen Arbeiten besser aufgehoben sind, ergeben sich z.B. durch eine akkuratere Kalibrierung, eine stabilere Temperaturregelung, bessere Kühlung von Druckkopf und Druckobjekt und eine größere Versteifung des Rahmens. Auch an den Slic3r-Parametern sind noch Detailoptimierungen möglich.

7.7.3. Schrittmotor-Synchronität

Die Auswertung des Schleppfehlers eignet sich hervorragend, um zu zeigen, dass die Diskrepanz zwischen Bewegungsbefehl und Ausführung über einen längeren Zeitraum nahezu vernachlässigbar gering ist. Aufgrund der niedrigen Samplingrate der Positionswerte von $\frac{1}{\text{Zyklus}}$ bleibt sie jedoch eine Bewertung des Bewegungsverhaltens *innerhalb* eines Zyklus schuldig. Insbesondere zeigt die Schleppfehleranalyse nicht, ob die durch EPL gegebenen Synchronisationsmechanismen korrekt funktionieren.

Deswegen wird nun das Ausgangssignal, das zum Schrittmotortreiber führt, mittels eines Logikanalysators gemessen. Bei jeder steigenden Taktflanke dieses Signals führt der Motor einen *Micro Step* aus (vgl. Abschnitt 7.2). Solange der Motor aufgrund von Trägheit und anderen äußeren physikalischen Einflüssen keine Schritte verliert, korreliert das Signal also direkt mit der ausgeführten Bewegung.

Abbildung 38 zeigt das Signal, das an den Schrittmotortreibern von den drei CNs anliegt, die im 3D-Drucker für die Ansteuerung der drei vertikalen Achsen zuständig sind. Die Sollgeschwindigkeit wird bei allen Knoten im gleichen Zyklus von 0 auf 200 Schritte

pro Sekunde erhöht. Die Zielposition ist jeweils auf 10 Schritte relativ zur vorherigen Position gesetzt. Wie man sieht, beginnen alle Knoten gleichzeitig nach dem Empfang von `SoC` mit der Taktgenerierung. Hierfür ist wichtig, dass der Timer in jedem Zyklus zurückgesetzt und neu gestartet wird. Andernfalls können die Signale außer Phase geraten, wie es in Abb. 37 dargestellt ist. An der Zeitskala lässt sich ablesen, dass auch die Geschwindigkeit mit 200 Schritten pro Sekunde korrekt ist und wie geplant exakt zehn Schritte ausgeführt werden.

8. Zusammenfassung und Ausblick

Die zentrale Frage dieser Arbeit, ob es möglich und sinnvoll ist, Echtzeit-Ethernet mit günstiger Allzweck-Hardware und einem Open-Source-RTOS zu betreiben, kann mit einem klaren Ja beantwortet werden. Die Kombination aus *Ethernet POWERLINK* als IE-Technologie, *NuttX* als RTOS und einem *ARM Cortex-M*-Mikrocontroller als Hardware-Plattform erweist sich nicht nur als besonders geeignet, um alle gestellten Anforderungen zu erfüllen, sondern verspricht zusätzlich auch Interoperabilität mit bestehender Unix/Linux-Software.

Die Reaktions- und Zykluszeiten sind wesentlich kleiner, als das zu Beginn aufgrund des Leistungsunterschieds zu einem Vergleichs-PC erwartet wurde. Sie sind mit Antwortzeiten von deutlich unter einer Millisekunde so klein, dass damit auch besonders anspruchsvolle Anwendungen, wie synchrone Motoransteuerungen, möglich sind. Die Arbeit beweist das anschaulich mithilfe eines Linear-Delta-3D-Druckers, dessen vier Antriebsmotoren von autarken Modulen, bestehend aus der oben genannten Technologiekombination, gesteuert werden. Auch hier sind die Ergebnisse besser als erwartet. Der maximale Schleppfehler beträgt nur wenige Hundertstel Millimeter und mit bloßem Auge sind praktisch keine Abweichungen vom Modell zu erkennen.

Beitrag dieser Arbeit sind nicht nur die Erörterung und der Nachweis der Umsetzbarkeit, sondern auch mehrere als Open-Source veröffentlichte Softwareentwicklungen. Dazu zählen eine Portierung des openPOWERLINK-Softwarestacks auf NuttX, ein NuttX-Treiber zum gepufferten Ethernet-Paketempfang, ein Tandem aus Machinekit-Plugin und openPOWERLINK-basiertem Managing Node als 3D-Drucker-Firmware sowie ein Analysetool für Ethernet POWERLINK-Netzwerkverkehr.

Die Arbeit an dem Projekt lieferte einige Erkenntnisse, die bei der Verwendung von Industrial Ethernet auf Mikrocontrollern von Bedeutung sind und künftige Entwicklungen erheblich erleichtern können. So hat etwa die Speicheranbindung im Mikrocontroller, der in dieser Leistungsklasse gewöhnlich keine Caching-Architektur aufweist, bei einer IE-Anwendung mit ihren naturgemäß hohen Datenraten bedeutenden Einfluss auf die Leistungsfähigkeit der Implementierung. Es lohnt sich deshalb, schon bei der Auswahl des Mikrocontrollers auf die Verfügbarkeit von schnell angebundenem Arbeits- und Programmspeicher in ausreichender Menge zu achten. Ein interessantes Problem bei der Umsetzung des 3D-Druckers ist die Anbindung des Maschinensteuerungsprogramms Machinekit an den zyklisch arbeitenden Feldbus Ethernet POWERLINK. Machinekit ist für direkt angebundene Motoren ausgelegt ist und erwartet damit sofortige Rückmeldung der aktuellen Position. Für die Interpolation der Zielpositionen und für die Schleppfehlerberechnung sind deswegen besondere Synchronisationsmaßnahmen nötig, um unvollständige und fehlerhafte Bewegungen zu vermeiden. Die hierfür erarbeiteten Konzepte sind nicht nur für den vorliegenden Anwendungsfall von Bedeutung, sondern auch allgemein für die Umstellung einer Anwendung, die bisher auf Direktansteuerung basiert, auf ein IE-System.

Für die weitere Verbesserung der Leistungsfähigkeit ergeben sich mehrere Möglichkeiten. Zunächst ist zu erwarten, dass die Ausführungsgeschwindigkeit weiter erhöht werden kann, wenn Teile des Programmcodes in den internen Arbeitsspeicher des Mi-

krocontrollers verlagert werden. Hier müssen passende Kompromisse bei der Verteilung dieser kappen Ressource gefunden und experimentell bestätigt werden. Eine allgemeine Verkürzung der Zykluszeit ist auch mit dem sogenannten *PRes-Chaining* zu erwarten, das in der Ethernet POWERLINK-Erweiterung *EPSG DS302-C* standardisiert ist. Weitere Entwicklungsarbeit lohnt sich außerdem aufseiten der Machinekit-Anbindung. Hier wäre als Ersatz für die externe Synchronisation der Bewegungssteuerung eine Implementierung interessant, die die Feldbuslatenz nativ berücksichtigt und damit auch eine dynamische Regelung erlaubt. Der systematisch induzierte Schleppfehler könnte damit auf einen Wert nahe Null reduziert werden.

Auch ohne größere Änderungen am vorgestellten System ergeben sich noch Themen für weitere Arbeiten. So wurde bisher nicht näher auf den bereits voll funktionsfähigen Ethernet-Seitenkanal (*Virtual Ethernet*) eingegangen, dabei ist hier eine Vielzahl von Anwendungsfällen denkbar. Interessant ist hierbei nicht nur die Kompromissfindung bezüglich der Zuteilung von Ressourcen, sondern auch, wie verschiedene Anwendungsprotokolle auf den verzögerten und zyklenorientierten Paketversand reagieren. Letztlich kommt auch eine Realisierung des EPL Managing Nodes auf Basis eines Mikrocontrollers und NuttX infrage, denn nicht zuletzt hat diese Arbeit gezeigt, dass die Kombination aus Mikrocontroller und RTOS trotz geringer Rechenleistung bei einer IE-Anwendung durchaus Vorteile gegenüber einem leistungsfähigen PC ausspielen kann.

A. Anhang

A.1. Schnittstellenzuordnung auf den Controlled Nodes

	CN1/Achse α & Heizbett	**CN2/Achse β & Lüfter**	**CN3/Achse γ**	**CN4/Extruder**
Motor	Antrieb α	Antrieb β	Antrieb γ	Filament-Feeder
PWM1	Heizbett	Objektlüfter	–	Heizstab
PWM2	–	–	–	*Hot End*-Kühler
SPI1	–	–	–	Temperaturfühler
Analog-in 1	Thermistor am Heizbett	–	–	–
Analog-in 2	–	–	–	–
I/O 1	Endschalter α	Endschalter β	Endschalter γ	–

A.2. Linux-Netzwerkkonfiguration

In Linux hat eine Vielzahl von Programmen und Diensten Zugriff auf angeschlossene Netzwerkgeräte. Auch ohne Verbindung zum Internet und ohne explizite Anfragen sorgen diese für regen Netzwerkverkehr, wie sich leicht anhand eines Netzwerkmitschnitts nachvollziehen lässt. Beispiele hierfür sind *ICMPv6*, *mDNS*, oder auch das *Dropbox LAN Sync Discovery Protocol*.

Es ist umständlich, alle relevanten Dienste so zu konfigurieren, dass sie auf der für Ethernet POWERLINK vorgesehenen Netzwerkschnittstelle keine Pakete versenden. Eine bessere Lösung ist die Erzeugung eines Netzwerk-Namensraums, in dem sich neben dem Netzwerkgerät nur der EPL-MN/CN und ggf. eine Mitschnittssoftware befindet. Folgende Befehle realisieren das:

```
#!/bin/bash
# device name as optional first argument or eth0
sudo ip address flush dev ${1:-eth0}
sudo ip -6 address flush dev ${1:-eth0}
sudo ip netns add epln s
sudo ip link set ${1:-eth0} netns epln s
sudo ip netns exec epln s ip link set dev ${1:-eth1} up
```

Zur Nutzung des virtuellen Ethernetkanals bietet es sich außerdem an, eine Route anzulegen, die Pakete ins EPL-Subnetz über den MN im Netzwerk-Namensraum leitet:

```
#!/bin/bash
sudo ip netns exec epln s ip route add 192.168.100/24 via 192.168.100.240
```

A.3. Pseudocode zum gepufferten Paketempfang

globale Variablen:
    buffer[buffersize] // *Paketpuffer*
    wo, ro // *Schreib− und Leseposition im Paketpuffer*
    size[maxnumpackets] // *Paletgrößen*
    wi, ri // *Schreib− und Leseindex im Paketgrößen−Feld*

// Ringpuffer befüllen. Argumente: Puffer des Ethernet−Treibers, Paketgröße
Funktion fill(device_buffer, len):
    wo_bak = wo // *Speichere Ursprungszustand*
    wi_bak = wi //
    if wo < ro and wo+len >= ro
        Fehler: Pufferüberlauf
    if wo+len >= buffersize
        wo = 0 // *Paket passt nicht ans Ende. Springe zum Anfang.*
        if len >= ro
            wo = wo_bak // *Ursprungszustand wiederherstellen*
            Fehler: Pufferüberlauf
    if wi>=maxnumpackets
        if ri == 0
            wo = wo_bak // *Ursprungszustand wiederherstellen*
            Fehler: Überlauf des Paketgrößen−Feldes
        wi = 0
    if wi+1 == ri
        wo = wo_bak // *Ursprungszustand wiederherstellen*
        wi = wi_bak //
        Fehler: Überlauf des Paketgrößen−Feldes
    memcopy(buffer+wo, device_buffer, len)
    wo += len
    size[wi] = len
    wi += 1
    sem_post() // *signalisiere neu angekommenes Paket*

// Paket aus Ringpuffer entnehmen. Argumente: Anwendungspuffer und Puffergröße
Funktion take(user_buffer, len):
    sem_wait() // *warte auf Paket*
    if ri >= maxnumpackets
        ri = 0
    if ro+size[ri] >= buffersize
        ro = 0 // *Paket hätte nicht ans Ende gepasst. Springe zum Anfang.*
    if size[ri]>len
        Fehler: Puffer zu klein
    memcopy(user_buffer, buffer+ro, size[ri])

A.4. Begriffserklärungen zu 3D-Druck

Extrusion Beim *Extrudieren* wird Kunststofffilament durch die heiße Düse gepresst. Der dickflüssige Kunststoff erhärtet dann auf dem Druckobjekt oder auf der Druckplatte und bildet eine neue Schicht.

Homing Mit *Homing* wird der Prozess bezeichnet, bei dem einer logischen Motorposition eine reale Raumkoordinate zugeordnet wird. Meist geschieht das durch Anfahren eines Referenz- oder Endschalters, dessen Position bereits bekannt ist.

Retract Bei Bewegungen des Druckkopfes, in denen keine Extrusion stattfinden soll, z.B. um eine Lücke zu überspringen, wird das Kunststofffilament ruckartig aus der heißen Düse zurückgezogen. Diese Maßnahme wird *Retract* genannt (von engl. *to retract sth. – etw. zurückziehen*) und vermeidet, dass allein aufgrund der Schwerkraft weiteres Material austritt und sich ungewollte Kunststofffäden bilden.

Literatur

[1] Industrial Ethernet Book Issue 69/42. The world market for Industrial Ethernet components, 2015.

[2] ARMmbed. Bambino-210E. `https://developer.mbed.org/platforms/Micromint-Bambino-210E/`. [Online; zugegriffen am 11.09.2016].

[3] ARMmbed. mbed LPC1768. `https://developer.mbed.org/platforms/mbed-LPC1768/`. [Online; zugegriffen am 24.01.2017].

[4] ARMmbed. mbed OS. `https://www.mbed.com/en/platform/mbed-os/`. [Online; zugegriffen am 27.10.2016].

[5] Electronic Industries Association et al. *Interchangeable Variable Block Data Format for Positioning, Contouring, and Contouring/Positioning Numerically Controlled Machines*. Electronic Industries Association, 1979.

[6] P. Brooks. Ethernet/IP-industrial protocol. In *Emerging Technologies and Factory Automation, 2001. Proceedings. 2001 8th IEEE International Conference on*, volume 2, pages 505–514 vol.2, Oct 2001.

[7] Bundesministerium für Bildung und Forschung. Zukunftsprojekt Industrie 4.0. `https://www.bmbf.de/de/zukunftsprojekt-industrie-4-0-848.html`. [Online; zugegriffen am 05.01.2017].

[8] David R. Butenhof. *Programming with POSIX threads*. Addison-Wesley professional computing series. Addison-Wesley, Boston, Mass., 19. print. edition, 2007.

[9] Microsoft Hardware Dev Center. NDIS Intermediate Drivers. `https://msdn.microsoft.com/en-us/windows/hardware/drivers/network/ndis-intermediate-drivers2`. [Online; zugegriffen am 23.01.2017].

[10] ChibiOS free embedded RTOS. `http://www.chibios.org`. [Online; zugegriffen am 27.10.2016].

[11] ChibiOS Licensing. `http://www.chibios.org/dokuwiki/doku.php?id=chibios:licensing:start`. [Online; zugegriffen am 27.10.2016].

[12] CAN in Automation e.V. `https://www.can-cia.org/`. [Online; zugegriffen am 08.02.2017].

[13] H. Buttner D. Jansen. Real-time Ethernet: the EtherCAT solution. *Computing and Control Engineering*, 15:16–21(5), February 2004.

[14] David A. Wheeler. SLOCCount User's Guide, 2004.

[15] D. Jeff Dionne, Michael Durrant, and Arcturus Networks Inc. uClinux Embedded Linux/Microcontroller Project. `http://www.uclinux.org`. [Online; zugegriffen am 27.10.2016].

[16] Linux Kernel Dokumentation. Universal TUN/TAP device driver. https://www.kernel.org/doc/Documentation/networking/tuntap.txt. [Online; zugegriffen am 27.01.2017].

[17] dumpcap - Dump network traffic. https://www.wireshark.org/docs/man-pages/dumpcap.html. [Online; zugegriffen am 28.10.2016].

[18] Embedded Microprocessor Benchmark Consortium. CoreMark. http://www.eembc.org/coremark/index.php. [Online; zugegriffen am 28.10.2016].

[19] Wikipedia: The Free Encyclopedia. Comparison of real-time operating systems. https://en.wikipedia.org/wiki/Comparison_of_real-time_operating_systems. [Online; zugegriffen am 23.12.2016].

[20] Institute of Electrical and Electronics Engineers. IEEE Standard 802.3u. *IEEE Standard 802.3u*, 1995.

[21] Institute of Electrical and Electronics Engineers. IEEE Std 802.3-2012 (Revision to IEEE Std 802.3-2008), Dec 2012.

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

[23] K. Erwinski, M. Paprocki, L. M. Grzesiak, K. Karwowski, and A. Wawrzak. Application of Ethernet Powerlink for Communication in a Linux RTAI Open CNC system. *IEEE Transactions on Industrial Electronics*, 60(2):628–636, Feb 2013.

[24] EtherCAT Technology Group. EtherCAT for Embedded Systems. https://www.ethercat.org/download/documents/ETG_BrochureEmbedded_EN.pdf. [Online; zugegriffen am 13.11.2016].

[25] EtherCAT Technology Group. EtherCAT Slave Controller Overview. https://www.ethercat.org/download/documents/ESC_Overview.pdf. [Online; zugegriffen am 13.11.2016].

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

[27] Raspberry Pi Foundation. Raspberry PI 3 Model B. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/. [Online; zugegriffen am 24.01.2017].

[28] Wireshark Foundation. Wireshark. https://www.wireshark.org/. [Online; zugegriffen am 28.10.2016].

[29] Free Software Foundation. The GNU General Public License v3.0. https://www.gnu.org/licenses/gpl-3.0.en.html, 2007. [Online; zugegriffen am 16.02.2017].

[30] FreeRTOS - Market leading RTOS for embedded systems with Internet of Things extensions. http://www.freertos.org/. [Online; zugegriffen am 27.10.2016].

[31] Official FreeRTOS Ports. http://www.freertos.org/RTOS_ports.html. [Online; zugegriffen am 16.02.2017].

[32] FreeRTOS+TCP License. http://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_TCP/FreeRTOS_Plus_TCP_License.html. [Online; zugegriffen am 24.02.2017].

[33] Sari Germanos, Wolfgang Seiss, and Elias Ahmed. Synchronizing Mechatronic Systems in Real Time Using FPGAs and Industrial Ethernet Communications.

[34] GitHub. https://github.com/. [Online; zugegriffen am 27.01.2017].

[35] Bernecker + Rainer Industrie Elektronik GmbH. POWERLINK. https://www.br-automation.com/en/technologies/powerlink/. [Online; zugegriffen am 14.11.2016].

[36] EtherCAT Technology Group. EtherCAT - Der Ethernet Feldbus. https://www.ethercat.org/download/documents/EtherCAT_Introduction_DE.pdf, October 2012. [Online; Zugegriffen am 23.02.2017].

[37] Ethernet POWERLINK Standardization Group. Interface Cards. http://www.ethernet-powerlink.org/en/products/interface-cards/. [Online; zugegriffen am 21.11.2016].

[38] Ethernet POWERLINK Standardization Group. Technical documents. http://www.ethernet-powerlink.org/de/downloads/technical-documents/. [Online; zugegriffen am 16.02.2017].

[39] Ethernet POWERLINK Standardization Group. Industrial Ethernet Facts - 3rd Edition (de), 2016.

[40] Vector Informatik GmbH Hans-Werner Schaal. Ethernet und IP im Kraftfahrzeug. *Elektronik automotive 4.2012*, April 2012.

[41] Meinrad Happacher. Powerlink mit Leistungssprung: Die Gigabit-Marke geknackt. http://www.computer-automation.de/feldebene/vernetzung/artikel/73882/, 2008. [Online; zugegriffen am 16.02.2017].

[42] IGS e.V. SERCOS-III - Innovation by Combining SERCOS interface and Ethernet. http://www.sercos.com/literature/pdf/sercos3_whitepaper.pdf, 2004. [Online; zugegriffen am 13.11.2016].

[43] CAN in Automation e.V. CANopen profiles. https://www.can-cia.org/can-knowledge/canopen/canopen-profiles/. [Online; zugegriffen am 16.02.2017].

[44] CAN in Automation e.V. CiA DSP-302 V3.2.1: Framework for CANopen Managers and Programmable CANopen Devices.

[45] CAN in Automation e.V. Technical documents. https://www.can-cia.org/can-knowledge/canopen/canopen-profiles/. [Online; zugegriffen am 16.02.2017].

[46] 3D Systems Inc. Stereolithography interface specification, 1988.

[47] Motorola Inc. SPI Block Guide V03.06, February 2003.

[48] Silicon Laboratories Inc. Which ARM Cortex Core Is Right for Your Application: A, R or M?, 2014.

[49] Till Jaeger and Axel Metzger. *Open Source Software : rechtliche Rahmenbedingungen der freien Software*. C.H. Beck, München, 4. auflage edition, 2016.

[50] Kconfig Language Introduction. https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt. [Online; zugegriffen am 23.01.2017].

[51] Thomas Keh. Powerlink Analyzer. https://github.com/thk1/powerlink_analyzer, 2016. [Github; zugegriffen am 28.10.2016].

[52] Fritz Klocke. Fertigungsverfahren 5 : Gießen, Pulvermetallurgie, Additive Manufacturing, 2015.

[53] LinuxCNC. http://linuxcnc.org/. [Online; zugegriffen am 03.11.2016].

[54] Pete Loshin. *Essential ethernet standards: RFCs and protocols made practical*. John Wiley & Sons, Inc., 1999.

[55] ARM Ltd. Cortex-M4 Processor. https://www.arm.com/products/processors/cortex-m/cortex-m4-processor.php. [Online; zugegriffen am 16.02.2017].

[56] Machinekit. http://www.machinekit.io/. [Online; zugegriffen am 03.11.2016].

[57] Machinekit Hardware Abstraction Layer. http://www.machinekit.io/docs/hal/intro/. [Online; zugegriffen am 04.11.2016].

[58] Machinekit Mission Statement. http://www.machinekit.io/about/. [Online; zugegriffen am 04.11.2016].

[59] Machinekit User Introduction. http://www.machinekit.io/docs/common/user_intro/. [Online; zugegriffen am 08.01.2017].

[60] NuttX Real-Time Operating System . http://nuttx.org/. [Online; zugegriffen am 18.09.2016].

[61] NXP Semiconductors. LPC43xx/LPC43Sxx ARM Cortex-M4/M0 multi-core microcontroller User Manual Rev 2.1. http://www.nxp.com/documents/user_manual/UM10503.pdf, December 2015. [Online; zugegriffen am 27.09.2016].

[62] openCONFIGURATOR - An Open Source POWERLINK network configuration toolkit. https://sourceforge.net/projects/openconf/. [Online; zugegriffen am 04.11.2016].

[63] openPOWERLINK Software Architecture. http://openpowerlink.sourceforge.net/doc/2.5/2.5.0/page_software-architecture.html. [Online; zugegriffen am 18.11.2016].

[64] François Pierrot, C Reynaud, and Alain Fournier. DELTA: a simple and efficient parallel robot. *Robotica*, 8(02):105–109, 1990.

[65] port GmbH. POWERLINK Enhanced Ethernet MAC. http://www.port.de/en/products/ethernet-powerlink/hardwareip-cores/powerlink-enhanced-ethernet-mac.html. [Online; zugegriffen am 18.11.2016].

[66] port GmbH. POWERLINK Library. http://www.port.de/de/products/ethernet-powerlink/software/powerlink-library.html. [Online; zugegriffen am 18.11.2016].

[67] port GmbH Torsten Gedenk. Use cases and advantages of the new XML device descriptions for CANopen devices. *IEEE International Conference on Communications 2008*, 2008.

[68] POSIX.1-2008. The Open Group Base Specifications. Also published as IEEE Std 1003.1-2008, July 2008.

[69] Prof. Dr.-Ing. Jürgen Schwager, Labor für Prozessdatenverarbeitung, Hochschule Reutlingen. Informationsportal für Echtzeit-Ethernet in der Industrieautomation. http://www.pdv.reutlingen-university.de/rte/. [Online; zugegriffen am 05.01.2017].

[70] Alessandro Ranellucci. Slic3r G-code generator for 3D printers. http://slic3r.org/. [Online; zugegriffen am 03.11.2016].

[71] RepRap Wiki. Kossel. http://reprap.org/wiki/Kossel. [Online; zugegriffen am 27.10.2016].

[72] RepRap Wiki. http://reprap.org/. [Online; zugegriffen am 27.01.2017].

[73] Viktor Schiffer, DJ Vangompel, and R Voss. The common industrial protocol (CIP) and the family of CIP networks. *Milwaukee, Wisconsin, USA, ODVA*, 2006.

[74] Semicast. Opportunities for ARM in Embedded Processing - 2015 Edition, 2016.

[75] Softing Industrial Automation GmbH. Ethernet POWERLINK Device for Altera FPGA. http://industrial.softing.com/en/products/software/protocol-stacks-for-fpgas/ethernet-powerlink/ethernet-powerlink-controlled-node-slave-for-altera-fpga.html. [Online; zugegriffen am 21.11.2016].

[76] Charles E. Spurgeon and Joann Zimmerman. *Ethernet : the definitive guide; [designing and managing local area networks]*. OReilly, Beijing, 2. ed. edition, 2014.

[77] W Richard Stevens and Stephen A Rago. *Advanced programming in the UNIX environment*. Addison-Wesley, 2013.

[78] Tcpdump & Pcap. `http://www.tcpdump.org/`. [Online; zugegriffen am 23.01.2017].

[79] Microsoft TechNet. Network Driver Interface Specification. `https://technet.microsoft.com/en-us/library/cc958797.aspx`. [Online; zugegriffen am 23.01.2017].

[80] Wolfgang Wallner and Josef Baumgartner. openPOWERLINK in Linux Userspace: Implementation and Performance Evaluation of the Real-Time Ethernet Protocol Stack in Linux Userspace. *Bernecker+ Rainer Industrie-Elektronik Ges. mbH, Austria*, 2011.

[81] Wolfgang Wallner, Dietmar Brucknerr, and Josef Baumgartner. openPOWERLINK 2.0: A split kernel/user space implementation of the real-time Ethernet protocol POWERLINK. *Bernecker + Rainer Industrie-Elektronik Ges. mbH, Austria*, 2013.

[82] David A. Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`. [Online; zugegriffen am 18.11.2016].

[83] NuttX Wiki. Delayed ACK and TCP Performance. `http://nuttx.org/doku.php?id=wiki:networking:delayedacks`. [Online; zugegriffen am 26.01.2017].

[84] Real-Time Linux Wiki. RT Preempt Howto. `https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO`. [Online; zugegriffen am 24.01.2017].

[85] RepRap Wiki. List of Firmwares. `http://reprap.org/wiki/Firmware`. [Online; zugegriffen am 02.01.2017].

[86] W. Grega Wojciech Modzelewski. Introduction to EtherNet/IP Technology.

[87] Heinz Wörn. *Echtzeitsysteme: Grundlagen, Funktionsweisen, Anwendungen*. Springer-Verlag, 2006.

[88] Joseph Yiu. *The definitive guide to ARM Cortex-M3- and Cortex-M4 processors*. Elsevier, Amsterdam, 3. ed. edition, 2014.

[89] Hubert Zimmermann. OSI reference model–The ISO model of architecture for open systems interconnection. *IEEE Transactions on communications*, 28(4):425–432, 1980.