FOM Hochschule für Oekonomie & Management

Studienzentrum München

Bachelor-Thesis

zur Erlangung des Grades eines

Bachelor of Science (B.Sc.)

über das Thema

Implementierung eines SLIP-Moduls für das Lehrbetriebssystem ULIX

von

Cliff Dornig

Erstgutachter	Dipl.-Math., Dipl.-Inf. Hans-Georg Eßer
Matrikelnummer	257211
Abgabedatum	2014-07-04

Inhalt

1 Verzeichnisse

1.1 Abbildungsverzeichnis

Abbildungsverzeichnis

1.2 Abkürzungsverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface
DLAB	Divisor Latch Access Bit
DNS	Domain Name Service
FIFO	First In First Out
GCC	GNU Compiler Collection
ICMP	Internet Control Message Protocol
IHL	Internet Header Length
IP	Internet Protokoll
IPv4	Internet Protokoll version 4
IPv6	Internet Protokoll version 6
OSI	Open Systems Interconnection
PIC	Programmable Interrupt Controler
SLIP	Serial Line Internet Protocol
TL	Total Length
TOS	Type Of Service
TTL	Time To Live
UART	Universal Asynchronous Receiver Transmitter
VM	Virtual Machine

1.3 Tabellenverzeichnis

Tabellenverzeichnis

1.4 Listingverzeichnis

Listings

2 Abstract

Heutige digitale Kommunikation wäre ohne Internet nicht möglich. Das Internet basiert auf den Netzwerkprotokollen der TCP/IP-Protokolle. Die systemnahe Programmierung der TCP/IP-Implementierungen heutiger Unix- und Linux-Systeme kann durch die Komplexität recht schwierig werden. Obwohl der Quellcode gut dokumentiert ist, stellt dies noch keine zusammenhängende Gesamtdokumentation dar. Oft wird Software immer noch nach dem Prinzip entwickelt, in der die Dokumentation innerhalb einzelner Codeabschnitte zu finden ist. Den einzelnen Dokumentationsabschnitten fehlt häufig die weiterführende Information zu Abhängigkeiten mit anderen Programmcodes. Um die Programmierung und die Dokumentation in nur einem Prozess innerhalb der Entwicklung zu vereinen, entwickelte Donald E. Knuth das Literate-Programming-System. Mittels der Literate-Programming-Methode wird eine Netzwerkkomponente auf Basis der seriellen Schnittstelle und den IP- und ICMP-Protokoll implementiert. Der Schwerpunkt der Thesis liegt beim Entwickeln von gut verständlichen Programmcodes für weitere Lehrzwecke. Dafür wurde das Lehrbetriebssystem Ulix-i386 als Basis für die Entwicklung des SLIP Modules ausgewählt.

3 Einleitung

Bisherige Betriebssystemlehrbücher wie das von Tanenbaum und Woodhull, beschäftigen sich sehr häufig mit den Basiskomponenten eines jeden Betriebssystems. In "Operating Systems - Design and Implementation" [TW97], sowie auch in "Betriebssysteme" [EKRSV05] werden die typischen Komponenten eines Betriebssystemes im Detail und mit Programmcode erklärt. Dazu zählen zum Beispiel der Bootprozess, Programme, Prozesse, Scheduler, IO Handler, Speicherhandling, Dateisysteme und weitere Bereiche. Über weitere Themen wie Netzwerk und andere Komponenten wie TCP/IP wird meistens nicht eingegangen und auch wenig Platz gelassen. Jedoch sind Computer heute mehr den je vernetzt. Das Verständnis zur Netzwerkprogrammierung auf Betriebssystemebene wird überflüssig durch die immer besser werdenden Hochsprachen wie Java, Go und andere. Selbst mit C und den entsprechenden API's ist die Programmierung einfacher Netzwerkprogramme schnell erledigt. Die Details zum Versenden / Empfangen und Verarbeiten der Daten und Protokolle auf Betriebssystemebene muss der Entwickler dabei nicht mehr wissen. Für die Entwicklung neuer Netzwerkprotokolle ist das Wissen über die Details im Betriebssystem jedoch von entscheidender Bedeutung. Hierzu gibt es meist sehr viele Manuals und auch RFC Beispielquellcodes, aber die Einarbeitung in die Details erfordert meist viel Zeit. Der dazu gehörige Quellcode und die Dokumentation in Form von Manuals der meisten "Open Source"-Betriebssysteme macht es Einsteigern sehr schwer, einen Überblick zu bekommen. Der TCP/IP Stack [FTS14], heutiger moderner Betriebssysteme ist mehr als Komplex aufgebaut und für Lehrzwecke eventuell nicht oder nicht mehr brauchbar. Die vorliegende Arbeit ist nicht nur ein Teil eines größeren Forschungsprojektes, sondern soll auch den Einblick in betriebssystemnahe Programmierung von Netzwerkprotokollen liefern. Ein weiterer Aspekt der Lehrbetriebssystementwicklung ist die Dokumentation. Tanenbaum beschreibt sein Betriebssystem Minix in seinem Buch sehr ausführlich. Die Beschreibung und der Programmcode selbst liegen dabei in getrennter Form vor. Dies macht das Verstehen des Quellcodes und das Lesen der dazugehörigen Dokumentation schwierig. Dieses Problem der Zusammenführung von Programmierung und Dokumentation hat auch schon Donald E. Knuth 1983 erkannt. Donald E. Knuth entwickelte deshalb die Literate-Programming-Methode [KNU83]. Mit dieser Programmiermethode ist es möglich, die Dokumentation und den Programmquelltext in einem Dokument zusammenzubringen. Mit der Literate-Programming-Methode soll ein Netzwerkmodul für das Lehrbetriebssystem Ulix-386 für Lehrzwecke detailliert dokumentiert und implementiert werden. Dabei liegt der Fokus auf dem Verständnis des Programmcodes und deren Zusammenhängen mit dem Betriebssystem.

3.1 Problemstellung

Mit dem Erfolg des Internets wurde die digitale Kommunikation weltweit möglich. Über Protokolle wie eMail, IRC und HTTP können über Programme Informationen schnell und einfach mit Millionenmenschen ausgetauscht werden. Die Datenübertragung der Netzwerkpakete im Internet funktioniert im wesentlichen nur dank der Erfindung der TCP/IP-Protokollfamilie. TCP/IP ist dabei eine Kurzschreibweise des gesamten Protokollstacks. Darunter fallen viele Protokolle wie zum Beispiel: IP, ICMP, TCP und UDP, welche im OSI Schichtmodel [ISO94] in den Ebenen drei bis vier benutzt werden. Die vorhandenen Unix/Linux Implementierungen der TCP/IP-Protokollfamilie [FTS14],[LIN14] beinhaltet sehr viele Protokolle und ist sehr komplex. Die dazugehörige Dokumentation des Quellcodes liegt meist als Manpage dem Betriebssystem bei, oder als Kommentar im Quellcode. Eine verständliche Zusammenfassung und Funktionsweise des Quellcodes selber ist nicht gegeben. Eine Einarbeitung in die Funktionsweise des Quellcodes erfordert deshalb auch einiges an Zeit. Andere TCP/IP-Implementierungen wie µIP [UIP14] oder nanoIP [MJRRH03] sind zwar sehr einfach aufgebaut aber es gibt auch hier wenig Dokumentation über die Funktionsweise des Quellcodes. Um überhaupt eine verständliche Zusammenfassung und Funktionsweise eines Programmes zu ermöglichen, gibt es die Literate-Programming-Methode [KNU83] von Donald E.Knuth. Diese ermöglicht die Kombination von Quellcode und Dokumentation in einem Schritt. Zudem wird der Quellcode in die Dokumentation fließend eingebettet. Der Vorteil dieser Methode liegt darin, dass nicht nur die Funktionsweise des Quellcodes erläutert wird, sondern es erleichtert die Pflege des Quellcodes und ermöglicht einen eventuell leichteren Einstieg. Die übliche Methode der Kommentierung im Quellcode gibt nicht die Ideen des Programmierers für den Algorithmus oder den Programmcode selbst wieder [PEP91]. Der Programmcode ist nur ein Stück Information. Für Änderungen am Programmcode, der nur mit Kommentaren hinterlegt ist, braucht es häufig tieferes Verständnis zum Algorithmus oder zum Quellcode. Literate-Programming ist sehr hilfreich im akademischen Umfeld [SPO98]. Studenten verstehen den Sachverhalt schneller mit Literate-Programming. Durch das Kombinieren von Dokumentation und Programmcode können ungewöhnliche formale Modelle erstellt und bearbeitet werden. Die vorliegende Arbeit ist Teil eines größeren Forschungsprojekts, in dem es darum geht herauszufinden, ob ein Betriebssysteme-Lehrbuch, das mit Literate-Programming den kompletten Quellcode eines Betriebssystemes vorstellt, bei Studenten das Verständnis der Betriebssysteme-Theorie verbessern kann. Es geht dabei um die Vermittlung der Grundlagen durch das Entwickeln von besonders gut verständlichem und gut dokumentiertem Code. Dazu soll ein Netzwerkmodul implementiert werden um die Kommunikation des Lehrbetriebssystemes Ulix-386 [EßE13] mit anderen Netzwerkbetriebssystemen zu ermöglichen. Dabei wird ein Teil der

Funktionalität des Unix/Linux typischen IP-Stacks dokumentiert und implementiert.

3.2 Forschungsmethodik

Mittels Online- und Literatur-Recherche wurde zuerst nach Begriffen, vorhandene Forschungen und Ergebnissen gesucht. Durch immer weitere Verfeinerung der Suchergebnisse und Suchbegriffe konnten die Ergebnisse eingegrenzt werden. Anhand der Tabelle 1 sind einige Suchbegriffe und die verwendeten Journale und Ergebnisse aufgeschlüsselt. Die Ergebnisse wurden untersucht nach vorhanden, Netzwerkprotokoll-Implementierungen. Als zweites wurden die Literaturergebnisse nach Literate-Programming untersucht. Die Zahl der Ergebnisse der Literate-Programming-Implementierungen sank auf weniger als 30. Zu Implementierungen von Netzwerkprotokollen, die mit der Literate-Programming-Methode erstellt wurden, konnten keine Suchergebnisse gefunden werden. Anhand des Design-Science-Ansatzes [BPH10] soll ein Artefakt geschaffen werden. welches in der Lehre der Betriebssystemtheorie verwendet werden kann. Dazu wurden vorhandene Implementierungen von Netzwerkprotokollen untersucht. Die vorliegende Arbeit wurde mit dem Programm "LaTeX" [LAT14] erstellt. Alle Abbildungen wurden mit dem Programm "dia" [DIA14] erstellt.

Journal	Suchbegriff	Suchart	Ergebnisse
Information & Management	Software AND Documentation	Volltext / Anschließende Einschränkung	66
Information Processing & Management	Software AND Documentation	Volltext / Anschließende Einschränkung	74
Formal Aspects of Computing	"Literate Programming" AND (NOWEB OR WEB)	Erweiterte Suche ohne anschließende Einschränkung	27
Journal of Computer Science & Technology	"Literate Programming" AND (NOWEB OR WEB)	Erweiterte Suche ohne anschließende Einschränkung	27
Computational Economics	"Literate Programming" AND (NOWEB OR WEB)	Erweiterte Suche ohne anschließende Einschränkung	27
Wirtschaftsinformatik	Software AND Documentation	Volltext	18
Science Direct	Software AND Documentation AND Network	Volltext	920
	"Literate Programming" AND Network	Volltext	741
	"Literate Programming" AND TCP/IP	Volltext	22

Tabelle 1: Literaturrecherche

3.3 Rahmen der Arbeit

Die vorliegende Arbeit beschäftigt sich rein mit SLIP, IP- und ICMP-Netzwerkprotokollen. Andere Protokolle wurden nicht betrachtet. Bei dem ICMP-Protokoll werden wiederum Teile der Spezifikation implementiert. Dazu zählen die Pakettypen von ICMP-Request und ICMP-Response. Alle anderen ICMP-Pakettypen werden nicht betrachtet und implementiert. Die IP-Pakete selbst sind so implementiert das diese zwischen zwei Maschinen versendet werden können. Weitere Spezifikationen des IP-Protokolls, wie z.B.: Routing oder Fragmentierung, wurden nicht betrachtet und implementiert. Auf dem vorhandenen TCP/IP-Stack von Linux und Unix wird auch nicht eingegangen. Für die Dokumentation des Quellcodes wurde Noweb genommen, auf Welches kurz eingegangen wird. Andere Literate-Programming-Software wurde nicht analysiert und verwendet.

3.4 Aufbau der Arbeit

Im ersten Teil der Arbeit wurden die Problemstellung und die Forschungsmethodik erläutert. Als nächstes werden ein paar Grundlagen zu Literate-Programming, IP und SLIP beschrieben. Der IP-Header wird analysiert und auf die Datenfelder angepasst, die für SLIP benötigt werden. Danach folgt die Beschreibung der Entwicklungsumgebung und die Implementierung der einzelnen Komponenten für das Betriebssystem Ulix-i386. Die einzelnen Programmbausteine und der dazugehörige Quellcode werden erklärt. Nachdem die Grundfunktionalität implementiert wurde, folgt der Test. Es wird die Netzwerkfunktionalität mit anderen Betriebssystemen und Ulix-386 getestet. Zum Schluss erfolgt eine Analyse und das Fazit. Es wurden zudem noch weiterführende mögliche Aufgaben für Studenten angehangen. Diese beschäftigen sich mit der Erweiterung der SLIP-Implementierung

4 Stand der Forschung

Für die Entwicklung von Software die auf Literate-Programming basiert, entwickelte D.E.Knuth das WEB-System [KNU83]. Das WEB-System unterstützt nur die Programmiersprache Pascal und für die Dokumentation das Textsatzsystem TEX. Aus der Weiterentwicklung von WEB entstand CWEB [KNU83]. CWEB unterstützt die Programmiersprachen C, C++ und Java. Weil CWEB nicht Programmiersprachen unabhängig ist, und als Ausgabeformate nur TEX unterstützt , entwickelte Norman Ramsey die Software "noweb" [NO14]. Noweb unterstützt LATEX, HTML und sogar troff als Ausgabeformat und ist flexibel einstellbar. Weitere Entwicklungszweige des WEB-Systems sind NuWEB [NU14] und FunnelWEB [FUN14]. Für spezielle Programmiersprachen wie R aus der Statistik gibt es eine eigene WEB Implementation namens: SWeave [SWE14]. Andere Softwareprojekte welche die Literate-Programming-Methode verwenden, sind zum Beispiele CDS[YUN91] und JABA[COC01]. Bei CDS handelt es sich um ein Entwicklungssystem ähnlich WEB. Mit dem Unterschied, dass CDS mit verschieden Sprachen und Schriften klar kommt. So unterstützt CDS chinesische Schriftzeichen genauso wie lateinische Schriftzeichen. Bei JABA hingegen handelt es sich um eine Entwicklungsumgebung mittels Hypertextsystem. Es soll die Literate-Programming-Methode mit der objektorientierten Programmierung verbinden. Dazu werden verschiedene Techniken in den grafischen Editor eingebaut, die es dem Entwickler erleichtern soll, wie z.B.: Fisheye-Ansichten oder Holophrasting [COC01]. JABA unterstützt sogar die automatische Änderung der Chunk-Struktur bei Veränderung eines Chunknames. Dies verhindert Fehler beim Tangle- oder Weave-Prozess. Was bisher alle Systeme noch nicht konnten war die Möglichkeit auch direkte Ausführungen über die Chunks zu ermöglichen. Dieses Problem wurde mit Lepton[THI01] angegangen. Das Ziel von Lepton ist, das die Ergebnisse vom Entwicklungsprozess immer gleich bleiben, bei jeder Ausführung. Den Code-Chunks können Parameter übergeben werden, wie zum Beispiel das Ausführen eines Programmes. Somit kann das Ergebnis eines Code-Chunks in der Dokumentation dynamisch erstellt werden. Bei "noweb" benötigt man die Schritte "tangle, compile, weave", bei Lepton hingegen nur einen einzigen Schritt.

Im Bereich der Betriebssystementwicklung und speziell der Netzwerkprogrammierung gibt es kaum bis gar keine vergleichbaren Ansätze der Implementierung mit Literate-Programming.

Mit dem Zitat aus der "Informatik Spektrum" soll etwas Licht in die Literate-Programming-Methode gebracht werden.

> "Noch 1984 sah Donald Knuth in seiner Arbeit "Literate Programming" Programmieren als Kunst und Programme als Stücke von

> Literatur an und schätzte die Informatik als Entwicklerin von Programmiersprachen und Programmen als Literaturwissenschaft ein, ein Ansatz, der sich nicht durchsetzte."

(Informatik Spektrum 2001, S. 92)

Im Bereich der Netzwerkentwicklung unter Unix/Linux gibt es schon zahlreiche TCP/IP-Implementierungen. Der bekannteste TCP/IP-Stack ist der von BSD [FTS14]. Allerdings enthält der Quellcode nur Kommentare jedoch keine genaue Beschreibung der Zusammenhänge des gesamtem TCP/IP-Stacks. Die Dokumentation liegt als Manpage dem System bei. Aber auch hier ist die Manpage keine vollständig zusammenhängende Dokumentation. Eine andere SLIP-Implementierung ist zum Beispiel die von Kutty, Jose, Rajan K, Antony, Linto Antony [KJKAA04]. Diese beschreibt sehr einfach die Struktur des Linux Kernels und wie ein SLIP Netzwerktreiber programmiert werden muss. Der Ulix-i386 Kernel in der Version 0.8 hat jedoch keine Netzwerktreiber-Struktur, womit sich diese SLIP-Implementierungen nicht eins zu eins anwenden lassen. Es müssten vorher erst einige Komponenten implementiert werden, damit der TCP/IP-Stack ins Ulix-i386 OS integriert werden kann. Eine reine TCP/IP- oder SLIP-Implementierung auf der reinen Userland-Ebene ist auch kaum Möglich. Denn für die Übertragung von Informationen von der seriellen Schnittstelle kann der Ulix-i386 Kernel nur über Interrupts arbeiten. Das Userland behandelt die Interrupts nicht und hat keinen Einfluss auf die Interrupts. Das Ulix-i386 OS müsste umgeschrieben werden, so dass es keine Trennung zwischen Kernel und Userland gibt. Andere TCP/IP-Stacks wie µIP (Micro IP) , lwIP und nanoIP wären mögliche Alternativen. Jedoch liegen die Schwerpunkte der drei Stacks auf der Ausführung auf ganz anderen Architekturen, als der Intel-x86-Architektur. So ist µIP ausgelegt für Microcontroller [UIP14] der 8 und 16bit Serie. Micro IP selbst ist eine sehr kleine TCP/IP-Implementierung. Wohingegen lwIP ausgelegt ist für Embedded-Systeme [LWIP14]. Die nanoIP Implementierung ist auch für Embedded-Systeme gemacht, allerdings mit einigen Einschränkungen. So gibt es keine Routing-Funktionen und auch keine IP-Adressen in nanoIP [MJRRH03]. Der IP-Header wurde in nanoIP dafür verändert. Das Ziel von nanoIP ist die Verwendung bei Geräten mit Sensoren im niedrig Preissegment. Über ein nanoIP-Gateway können die Geräte dann z.B. ins Internet eingespeist werden. Alle Geräte im nanoIP-Subnetz werden nur über die MAC-Adressen angesprochen.

5 Grundlagen

Alle modernen Betriebssysteme wie FreeBSD, Linux oder Windows sind netzwerkfähig. Mit Netzwerkfähig sind jene Protokolle gemeint, die es einem Computersystem ermöglichen mit anderen Computersystemen zu kommunizieren. Es muss dabei aber noch unterschieden werden, was mit Netzwerk gemeint ist. Am Besten lässt sich dies verdeutlichen mittels dem OSI-Modell [ISO94]. Ein Netzwerk besteht meist aus zwei oder mehr Computern, welche miteinander verbunden sind. Damit diese auch Daten austauschen können, müssen einige Bedingungen erfüllt werden. Dazu zählen z.B. die korrekte und fehlerfreie Datenübertragung, die Vermeidung von doppelten Übertragungen und auch das Interpretieren der Informationen. Das OSI-Model trennt und abstrahiert die verschiedenen Bedingungen und Anforderungen in einzelne Schichten. Jede Schicht übernimmt dabei eine oder mehrere Aufgaben die nur nach Regeln und Bedingungen funktionieren. Die einzelnen Schichten ermöglichen der jeweils nächst höheren Schicht auf die Informationen zuzugreifen. Somit ist es möglich das unterschiedliche Systeme mit einander kommunizieren können, solange sie das OSI-Model und die jeweiligen Protokolle benutzen. In Abbildung 1 ist das OSI-Modell auf der Linken Seite abgebildet. Auf der rechten Seite der Abbildung sind die Schichten und Protokolle für SLIP abgebildet.

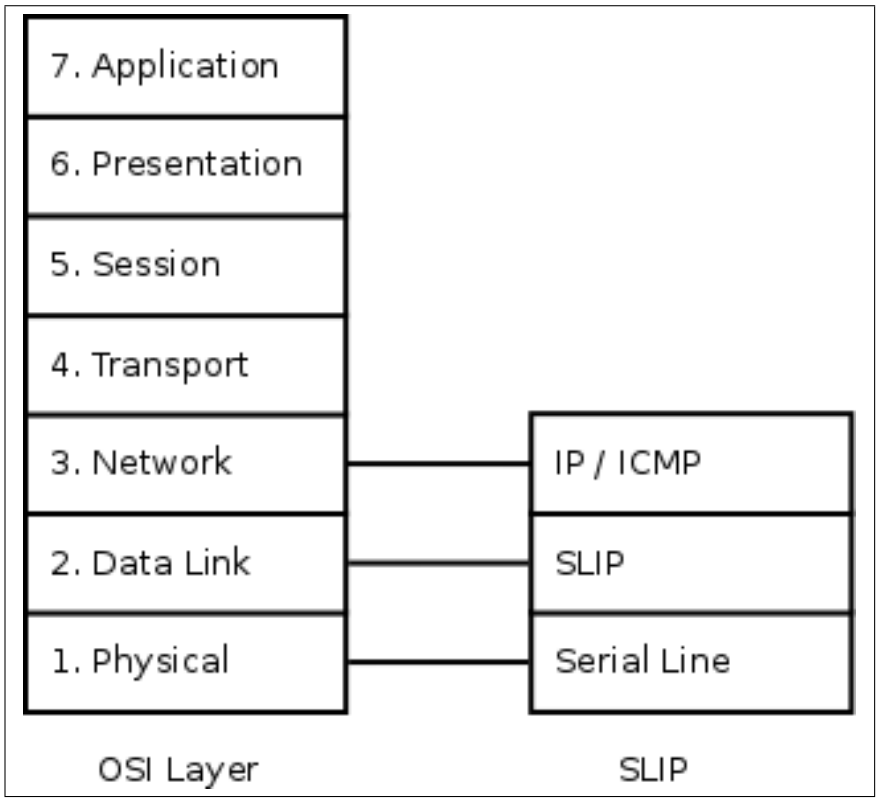

Abbildung 1: OSI-Modell nach ISO/IEC 7498-1:1994

Für die SLIP-Funktionalität werden die OSI-Schichten 1 bis 3 betrachtet und benötigt. Die OSI-Schichten 4 bis 7 werden nicht betrachtet und auch nicht implementiert.

Die sogenannte "Bitübtragungsschicht", also OSI-Schicht 1, stellt die Übertragung der einzelnen Informationen als kleinste Einheit (als Bit) sicher. Das Übertragungsmedium für die serielle Schnittstelle ist normalerweise ein 9 oder 24 poliges Kabel. Dies kann sowohl ein Modem als auch ein Null-Modem Kabel sein. Theoretisch wären auch andere Kabelarten möglich. Beim SLIP-Protokoll werden die Daten in der OSI-Schicht 2 als einzelne Bytes übertragen. In der OSI-Schicht 3 werden die Protokolle IP und ICMP behandelt. Das Protokoll IP stellt einen Standard für die Erstellung von Informationspaketen bereit. Jedes IP-Paket enthält Informationen vom Absender und Empfänger. Über das IP-Paket werden dann Pakete der höheren OSI-Schichten transportiert. Beispiele dafür sind die Protokolle TCP und UDP. Das Protokoll ICMP ist in der OSI-Schicht 3 definiert, und verwendet deshalb das IP-Protokoll für den Transport.

5.1 Noweb

Das Literate-Programming wurde von Donald E. Knuth entwickelt. Dabei handelt es sich um eine Art der Programmierung, bei der die gleichzeitige Dokumentation und Programmierung im Vordergrund stehen. Dafür entwickelte D.E.Knuth das WEB System, welches es ermöglicht die Programmierung und die Dokumentation in einer Datei zu handhaben. Diese WEB Datei dient als Ursprungsdatei für den Quellcode und der Dokumentation. Das WEB System besteht hauptsächlich aus zwei Programmen [KNU83]. Ein Programm dient zum Extrahieren des Quellcodes und ein weiteres Programm zum Erstellen der Dokumentation. In Abbildung 2 ist der Prozess dargestellt wie das WEB-System funktioniert. Das ursprüngliche WEB-System kann nur Pascal-Code extrahieren. Für die Programmiersprache C entwickelte Knuth das CWEB. Aus dem Umstand, dass man für jede Programmsprache einen extra Parser benötigen würde, entwickelte Norman Ramsey die Software "noweb". Mit "noweb" kann jede Programmiersprachensyntax extrahiert werden. Ein weiterer Umstand für die Entwicklung von "noweb" ist, dass die Dokumentationserstellung des Parsers mehrere Formate unterstützt. Mit WEB und CWEB ist als Format nur TEX möglich, wohingegen "noweb" die Formate HTML, TEX und LATEX unterstützten.

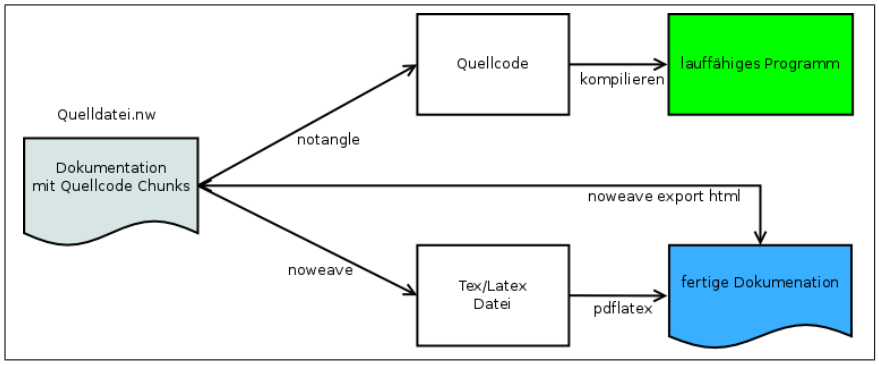

Abbildung 2: Funktionsweise von noweb

Ein kleines Beispiel soll die Vorzüge von Literate-Programming veranschaulichen.

Im Quelldokument wird der Programmcode als Chunk innerhalb der Dokumentation eingebunden. Dem "noweb" Parser für die Code Extrahierung aus dem Quelldokument ist es dabei egal wann und wo die Chunks verwendet werden. Der Parser fügt die Codefragmente der einzelnen Chunks wieder zusammen. Dadurch ist es möglich unterschiedliche Programmcodes hintereinander zu setzten. In dem folgenden Beispiel wird der Programmcode für die

Datei "beispiel.c" vollständig vom Parser wieder zusammengesetzt. Zwischen den einzelnen Chunks liegt die Dokumentation.

13a ⟨*Implementierung von einem Beispiel* 13a⟩≡
```
⟨beispiel.c: Variablen 13b⟩
⟨beispiel.c: Ausgabe 13c⟩
print();
```
Uses print 13c.

Dokumentation der Variablen

13b ⟨*beispiel.c: Variablen* 13b⟩≡ (13a)
```
int x;
int y;
```
Defines:
x, used in chunk 82a.
y, never used.

Dokumentation der Funktion print.

13c ⟨*beispiel.c: Ausgabe* 13c⟩≡ (13a)
```
void print(){
        printf("Ausgabe in Funktion print...\n");
}
```
Defines:
print, used in chunk 13a.

Die Extrahierung der einzelnen Chunks erfolgt durch das Programm "notangle". Der sogenannte root-Chunk hat im Beispiel den Namen "Implementierung von einem Beispiel". Dieser root-Chunk wird dem Programm "notangle" als Parameter übergeben. Das Programm "notangle" fängt beim root-Chunk an und setzt alle weiteren Chunks die inkludiert werden wiederum zusammen mit dem Befehl: "`notangle -L -R"Implementierung von einem Beispiel" Quelldatei.nw > beispiel.c`". Das Hauptaugenmerk der Literate-Programming-Systeme liegt vorrangig im Dokumentieren, erst danach kann programmiert werden.

5.2 UART

Für die Datenübertragung über die serielle Schnittstelle gibt es den Universal Asynchronous Receiver/Transmitter. Der UART stellt eine Schnittstelle für die digitale, serielle Übertragung bereit. Der UART unterstützt die seriellen Übertragungsstandards RS-232 und EIA-485 [NSC95]. Der RS232 ist ein Standard für die asynchrone Übertragung von Daten über eine "single signal"-Leitung [COO98]. Der UART stellt eine parallele Schnittstelle zur CPU bereit

und konvertiert von parallel auf seriell und umgekehrt [ASAAS04]. Für jede serielle Schnittstelle einer Maschine, ob virtuell oder real, ist jeweils ein UART zuständig. Jeder UART ist einzeln programmierbar und kann jederzeit vom System abgefragt werden. Dies ist notwendig um die verschiedenen Parameter für die serielle Verbindung einzustellen. Dazu zählen Parity Bit, Baudrate, Bit Zeichenfolge (von 5 bis 8 Bits) und die Start- und Stop-Bit-Einstellung. Der UART enthält auch einen programmierbaren Baudrate-Generator. Baud bezeichnet die Anzahl der Bits die pro Sekunden über die Leitung versendet werden. Dieser Baudrate-Generator ist notwendig um die korrekte Anzahl an Bits pro Sekunde zu gewährleisten. Der UART enthält dafür einen internen Taktgeber oder auch Clock genannt [NSC95]. Um die gewünschte Baudrate zu erhalten, wird die Taktfrequenz durch den entsprechenden Faktor geteilt. Beide Kommunikationspartner stellen dann die gleichen Baudraten ein. Der UART überträgt die Daten immer asynchron über die serielle Schnittstelle [ASAAS04]. Es gibt deshalb kein Taktsignal, welches übertragen wird, für die Synchronisierung mit der Gegenstelle. Stattdessen wird ein Start- und Stop-Bit für die Synchronisierung der beiden Kommunikationspartner verwendet. Programmieren lässt sich der UART über Register. Diese Register sind über Adressen konfigurierbar. Die Adressen der seriellen Ports sind beim PC häufig als COM1 bis COM4 bezeichnet und belegen die I/O Port Adressen 0x3F8, 0x2F8, 0x3E8 und 0x2E8 der Interrupt Ports 3 und 4. Die Schnittstellen COM1 und COM3 sowie COM2 und COM4 teilen sich jeweils einen Interrupt-Port am "Programmable Interrupt Controller". Unter Unix werden die seriellen Ports als Gerätedateien unter /dev als Namen ttyuN oder ttyN erzeugt (N ist dabei die Nummer des Ports). In Intel basierten x86 Systemen ist der UART ein fest eingebauter Chip auf dem Mainboard. Der bekannteste Baustein der UART Serie ist der PC16550D. Bei virtuellen Maschinen, wie z.B.: qemu, ist der UART über den Hypervisor [QEMU14] emuliert.

5.3 SLIP

Die Übertragung von IP Paketen über die serielle Schnittstelle ist im RFC 1055 unter dem Namen: "Serial Line Internet Protokoll" [RFC1055] definiert. SLIP ist sehr leicht zu implementieren. Das SLIP arbeitet auf der Sicherungsschicht im OSI-Schichtenmodel und braucht keine Netzwerkkarte. Das Protokoll braucht nur eine serielle Leitung. Bestehend aus Anschluss und Leitung. Die Datenübertragung der einzelnen Bytes des IP- Protokolls erfolgt durch den Versand und Empfang über die serielle Leitung. Alle Bytes werden unverändert übertragen. Jedes IP-Paket das über SLIP übertragen wird, beginnt mit einem Start-Byte und wird am Ende mit einem End-Byte abgeschlossen. Das Start- und das End-Byte ist jeweils definiert mit dem "END"-Zeichen (Dezimal 192). SLIP besitzt keine Fehlererkennung oder Kompressionsmöglichkeiten. Es gibt

eine CSLIP Variante, welche eine Kompression der Daten ermöglicht, und somit die Übertragungsraten erhöht. Weil es im IP-Protokoll vorkommen kann, dass das "END"-Zeichen übertragen wird welches als Information innerhalb des IP-Protokolls gebraucht wird, stellt SLIP vor dem Versand des Zeichens sicher, dass der Wert verändert wird. In solch einem Fall werden zwei Zeichen gesendet. Zuerst ein "ESC"-Zeichen (Dezimal 219) gefolgt von einem "ESC_END" (Dezimal 220). Bei vorkommen des "ESC"-Zeichens (Dezimal 219) wird dieses ersetzt durch Dezimal 219 und 220. Somit wird sicher gestellt das im IP-Protokoll selbst alle Zahlen von 0 bis 255 übertragen werden können ohne das die Übertragung durch ein "END"-Zeichen unterbrochen wird. Ein einfaches Beispiel wäre die Übertragung einer IP-Adresse wie zum Beispiel 192.168.10.1. In dem Fall würde ohne die Substitution der Zahl 192 die Übertragung beendet werden. Da SLIP kein offizieller Standard ist, gibt es auch keine eindeutig definierte maximale Paketlänge.

5.4 IP-Protokoll

Das Internet-Protokoll arbeitet innerhalb der Sicherungsschicht des OSI-Models und ist im RFC 791 definiert [RFC791]. Aktuell werden zwei IP-Versionen im Internet verwendet. Dazu zählen IP-Version 4 und IP-Version 6. Die IP-Version 6 ist im RFC 2460 definiert [RFC2460]. Die Abbildung 3 zeigt den IP-Header der Version 4. Die einzelnen Felder des IPv4-Protokolls sollen kurz erklärt werden.

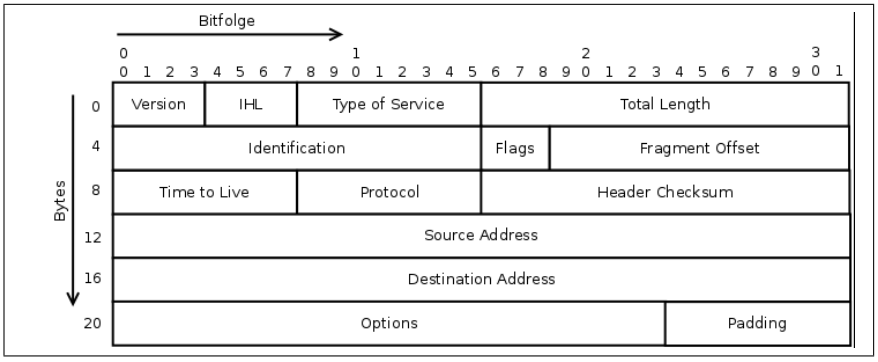

Abbildung 3: IPv4-Header nach RFC 791

Die IPv4-Protokoll-Header-Felder im Detail:

- Version: Die IP-Header-Version gibt an, welches Format verwendet werden soll. Für IPv4 wird einfach eine 4 eingetragen, für IPv6 ist es eine 6. Neben IPv4 und IPv6 gibt es noch weitere Formate, wie TP/IX oder

TUBA. Es können insgesamt bis zu 16 Formate definiert werden, weil das Versionsfeld 4 Bit lang ist.

- IHL: Das Feld "Internet Header Length" oder auch IHL ist 4 Bit lang und gibt an wie groß der IP-Header des verwendetet IP-Paketes ist. Der Wert für den IHL berechnet sich aus dem vielfachen der genutzten 32 Bit des IP-Headers. Je 32 Bit des IP-Headers werden um eins addiert zum IHL Wert. Die minimal Länge von IHL beträgt vier. Die Felder Options & Padding können weggelassen werden, diese sind optional.

- TOS: Mit dem Type of Service oder TOS kann dem IP-Paket eine Priorität gesetzt werden. Damit können IP-Pakete bei sehr hohem Paketaufkommen unterschiedlich behandelt werden. TOS ist im RFC 1349 [RFC1349] definiert.

- TL: Das Feld "Total Length" (TL) ist 16 Bit breit und gibt die Größe des gesamten IP-Pakets inklusive des IP-Headers in Byte an. Maximal kann ein IP-Paket 64 Kilobyte groß sein.

- ID: Mit dem Feld "Identification" wird dem Zielsystem, welches die Daten empfängt, mitgeteilt, ob die Daten fragmentiert sind. Dazu werden Sequenznummern verwendet, um die richtige Paketreihenfolge zu gewährleisten.

- Flags: Das Feld "Flags" ist 3 Bit breit und gibt an, ob das IP-Paket selbst fragmentiert wurde, oder ob weitere fragmentierte IP-Pakete folgen.

- Fragment Offset: Das Feld "Fragment-Offset" dient der Defragmentierung von fragmentierten IP-Paketen.

- Time to Live: Das Feld "Time To Live" enthält einen Zählerwert, der dazu dient die maximale Laufzeit eines IP-Paketes zu begrenzen. Dieser Wert wird vom Sender mit einer Zahl belegt. Meist wird der TTL Wert beim versenden des IP-Paketes auf 64 oder 128 gesetzt, je nach Betriebssystem. Jeder Router, der das IP-Paket weiterleitet, verringert diesen Wert um eins. Wenn der TTL-Wert 0 ist, so wird das IP-Paket vom jeweiligem System, welches das IP-Paket gerade erhalten hat, verworfen.

- Protocol: Das Feld "Protocol" gibt an, um welches Datenpaket es sich handelt, welches übertragen wird. Für die Übertragung eines ICMP-Paketes, wird der Wert von Protocol auf 1 gestellt. Im RFC 1700 sind die Nummern und Protokolle ausführlich definiert [RFC1700].

- Header Checksum: Das Feld "Header Checksum" dient zur Überprüfung des übertragenen IP-Headers. Das Feld enthält eine Prüfsumme, welche

vom Sender erzeugt und vom Empfänger überprüft wird. Die Prüfsumme wird meistens mit dem "Internet-Checksum"-verfahren [RFC1071] berechnet.

- Source Address: Das Feld "Source Address" enthält die IP-Adresse vom Absender. Die Source-Address wird als 32 Bit Dezimalzahlenwert hinterlegt.
- Destination Address: Das Feld "Destination Address" enthält die IP-Adresse vom Zielsystem und wird ebenfalls als 32 Bit Dezimalzahlenwert hinterlegt.
- Options: Das Feld "Options" enthält Sonderinformationen die nicht genauer betrachtet werden.
- Padding: Das Feld "Padding" dient als Lückenfüller, um die Größe des IP-Headers exakt an den IHL Wert anzupassen. Dies kann zum Beispiel der Fall sein, wenn das Feld "Options" verwendet wird.

5.5 IP-Header für SLIP

Für die SLIP-Umsetzung bedarf es jedoch nicht des gesamten IP-Headers. Einige Felder sind nicht notwendig und werden daher mit Nullen gefüllt. Diese nicht benötigten Felder des IP Headers sollen kurz erklärt werden.

- Version und IHL: Die Felder für Version und IP-Header Länge werden benötigt.
- TOS: Das Feld Type of Service wird nicht benötigt, weil keine Priorisierung der IP-Paketen zwischen zwei Rechnern notwendig ist. Eine Implementierung von TOS erfordert zudem einiges mehr an Logik und ist für den minimalen IP Header nicht notwendig.
- TL: Das Feld "Total Length" wird benötigt für die Berechnung der Paketlänge.
- Flags und Fragment Offset: Die Felder Identification, Flags und Fragment Offset werden ebenso wenig implementiert. Diese Felder dienen zur Fragmentierung von IP-Paketen, und werden nicht benötigt.
- TTL: Das TTL Feld ist auch nicht notwendig, weil die Pakete nur zwischen zwei Maschinen übertragen werden.

- Protocol: Das Feld "Protocol" ist notwendig und wird mit der Protokollnummer vom ICMP-Protokoll befüllt. Weitere Protokollnummern wie für UDP und TCP können in späteren Versionen implementiert werden.
- Header Checksum Das Feld "Header Checksum" wird benötigt.
- Source/Destination Address: Die Felder für die Quell und Ziel IP-Adressen werden benötigt.
- Options und Padding: Die Felder Options und Padding entfallen ebenfalls.

Alle benötigten Felder des IP-Headers sind in der Tabelle 4 nochmals aufgeschlüsselt.

IP Header Feld	**Notwendig für SLIP**
IP Version	Ja
IHL	Ja
Total Length	Ja
Protocol	Ja
Header Checksum	Ja
Source Address	Ja
Destination Address	Ja

Abbildung 4: Minimaler IP-Header für SLIP

5.6 ICMP

Das Internet-Control-Message-Protocol (ICMP) dient zur Übertragung einfacher Informationen zwischen zwei Computersystemen in einem Netzwerk. Die Übertragung des ICMP-Paketes wird über IP ermöglicht. Das ICMP-Paket besteht im wesentlichen aus einem einfachen Header gefolgt von mehreren Datenfeldern [RFC792]. Der Aufbau des ICMP-Headers ist in Abbildung 5 zu sehen. Die ersten 32 Bit des ICMP-Paketes enthalten den gesamten Header des Protokolls.

- Type: Das Feld Type ist 1 Byte breit, und gibt an um welchen ICMP Pakettyp es sich handelt
- Code: Das Feld Code ist ebenfalls 1 Byte breit, und enthält den Code Wert zum jeweiligen ICMP Pakettyp.
- Checksum: Das Checksum Feld ist 16 Bit breit und speichert die Prüfsumme des ICMP Paketes.

- Data: Direkt nach den ersten 32 Bit des ICMP-Headers folgen die Datenfelder. Die Felder Data können Informationen enthalten.

Zu jedem Type gibt es verschiedene Codewerte. Eine genaue Liste aller Type- und Code-werte kann anhand des RFC's genommen werden.

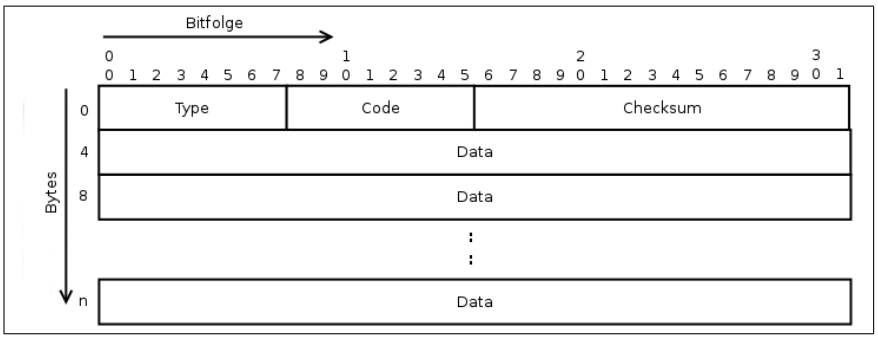

Abbildung 5: ICMP-Header nach RFC 792

6 Entwicklungsumgebung

Für die Implementierung des SLIP-Moduls wurde eine Entwicklungsumgebung eingerichtet. Die Entwicklungsumgebung soll kurz erklärt werden. Und es werden die Schritte zur Einrichtung der Entwicklungsumgebung aufzeigt.

6.1 Aufbau und Komponenten

Die Entwicklungsumgebung wurde auf einem Intel x86 64 Bit-System installiert. Als Betriebssystem wurde ein Unix ähnliches Betriebssystem namens FreeBSD [FBSD14] installiert.

Das Lehrbetriebssystem Ulix-i386 läuft in einer emulierten i386 Umgebung, welche mit dem Emulator Qemu [QEMU14] bereitgestellt wird. Der PC auf dem die Emulation läuft wird hier als Hostsystem bezeichnet. Die emulierte Ulix-386 Maschine wird als Gastsystem bezeichnet. Als Literate-Programming-System kommt "noweb" und LaTeX zum Einsatz. Für das Erzeugen des Ulix-i386 Betriebssystems aus dem Quellcode wird die GNU Compiler-Collection benötigt, sowie das Programm "make". Das erzeugen des ausführbaren Ulix-i386 Systems, erfolgt in mehreren Schritten. Dazu zählt das Extrahieren des Quellcodes aus der "noweb" Datei, das Kompilieren des Quellcodes und das Linken. Das Kompilieren und Linken übernimmt der C Compiler und Linker vom GCC. Die Extrahierung des Quellcodes erfolgt durch das Programm "notangle". Für die Funktionstests mit anderen Betriebssystem, wird eine weitere Maschine benötigt. Für die Übertragung von IP-Paketen über SLIP, müssen beide Systeme SLIP unterstützen. Das installierte Betriebssystem (FreeBSD Version 9) auf dem Hostsystem unterstützt kein SLIP mehr. Deshalb wurde eine zweite VM aufgesetzt mit OpenBSD [OBSD14]. Diese OpenBSD-VM wurde mittels VirtualBox [VBOX14] realisiert. Das Ulix-i386 System wird dann durch einen Tunnel vom Hostsystem mit der OpenBSD-VM über die seriellen Schnittstellen verbunden. Für das Tunneln der seriellen Schnittstellen zwischen der Ulix-i386 und der OpenBSD-VM wird das Program "socat" benötigt. Zum auswerten der übertragenen IP-Pakete zwischen der Ulix-i386 und der OpenBSD-VM wird das Programm "wireshark" genutzt.

6.2 Hostsystem

Die Entwicklung, Tests und Ausführung des Ulix-i386 können auf einem einzigen Computer laufen. Das ist möglich, weil die i386 Umgebung für Ulix mit der heutigen Intel x86 Architektur emuliert oder virtualisiert werden kann. In der Arbeit wurde die Emulation gewählt, weil diese am einfachsten zu nutzen

ist, und keine Anpassungen des Gastes benötigt. Der C Compiler, welcher in der FreeBSD Distribution des Hostsystems enthalten ist, ist die "gcc" Version 4.2.1. Weil das Ulix-i386 System sich mit dieser "gcc" Version nicht kompilieren lässt, wurde eine neuere "gcc" Version 4.6 installiert. Um zusätzliche Programme unter FreeBSD zu installieren gibt es mehrere Möglichkeiten. Unter FreeBSD können Programme über das Port-System [FPP14] kompiliert und installiert werden, oder es werden fertige, binäre Pakete installiert.

Mit dem Befehl "pkg install gcc46" wird der GCC mit der Version 4.6 als Binärpaket installiert, inklusive aller noch nicht vorhandenen Abhängigkeiten.

Für die Emulation von Ulix-i386 und der OpenBSD-VM wurden noch Qemu und Virtualbox installiert. Die Literate-Programming-Umgebung mit "noweb" erfolgt durch die Installation von teTeX-base. Für die Tests und Auswertung der Ergebnisse wird Wireshark als grafisches Netzwerkanalyseprogramm installiert.

Mit dem Befehl "pkg install qemu socat tetex-base virtualbox wireshark" werden die benötigten Programme installiert.

6.3 Übersetzung der noweb Quelldatei

Zum erstellen eines lauffähigen Ulix-i386 Systems, muss der Quellcode aus der "noweb"-Quelldatei extrahiert und anschließend kompiliert werden. Die nachfolgenden Schritte erklären die Extrahierung des C-Quellcodes aus der "noweb" Quelldatei und das anschließende Erstellen des Ulix-i386 mit der SLIP-Modul Implementierung.

6.3.1 Extrahieren des C Quellcodes

Der C-Quellcode und die Dokumentation des Quellcodes aus der Arbeit befindet sich in der "noweb" Quelldatei. Aus dieser "noweb"-Datei wird der gesamte C-Quellcode extrahiert.

```
notangle -L -Rslip-mod.c bachelorarbeit.nw > slip-mod.c
notangle -L -Rslip-mod.h bachelorarbeit.nw > slip-mod.h
notangle -L -Ripv4-tools.c bachelorarbeit.nw > ipv4-tools.c
notangle -L -Ripv4-tools.h bachelorarbeit.nw > ipv4-tools.h
```

Listing 1: Extrahieren des Quellcodes

In Listing 1 werden beide Teile des Quellcodes aus der Implementierung extrahiert. Einmal das Kernelmodul und zum anderen die Userland-Funktionen. Der

Paramter "-R" vom Programm "notangle" wird dem zu extrahierende Chunk übergeben. Der Parameter "-L" fügt für jeden Chunkabschnitt, der extrahiert wird, im Quellcode eine Kommentarzeile ein, mit der Beschriftung der Zeile aus der "noweb"-Datei.

Bevor der extrahierte Quellcode kompiliert werden kann, müssen ein paar kleine Änderungen in der "sh.c" vorgenommen werden. Die "sh.c" ist die implementierte Shell vom Ulix-i386 System. Die Shell dient als Kommandointerpreter. Mit der Shell können Befehle an das Ulix-i386 System gesendet werden, und Textausgaben auf der Kommandozeile angezeigt werden.

22a ⟨*sh.c: Einfügen der ipv4-tools.h (Zeile 2)* 22a⟩≡
```
#include "ipv4-tools.h"
```

Die `ipv4-tools.h` muss eingebunden werden in die "sh.c", damit die Funktionen `get_hostname`, `set_hostname`, `ipd` und `cmd_ping` verfügbar sind. Die `SHELL_COMMANDS` Variable in der "sh.c" wird erweitert mit den neu verfügbaren Kommandos.

22b ⟨*sh.c: Erweiterung der* `SHELL_COMMANDS` *(Zeile 3)* 22b⟩≡
```
#define SHELL_COMMANDS "help, ps, fork, ls, cat,
                head, cp, diff, sh, hexdump, kill, loop,
                test, brk, cd, ln, rm, pwd, touch, read,
                edit, exit, hostname, ping, ipd"
```
Defines:
`SHELL_COMMANDS`, never used.
Uses `hostname` 47c and `ipd` 65.

In der "sh.c" fehlen noch die Aufrufe der einzelnen Funktionen bei Eingabe der Shell-Befehle. Dazu sind drei weitere Verzweigungen in der `run_command` Funktion innerhalb der "sh.c" notwendig.

22c ⟨*sh.c: Erweiterung der* `run_command` *Funktion (Zeile 790)* 22c⟩≡
```
    else if ( strcmp ((char*)cmd, "hostname") ) {
          if(argc > 1){
                  set_hostname(argv[1]);
          }
      char arr[MAXLEN_HOSTNAME];
      get_hostname(&arr);
      printf ("Hostname: %s\n", arr);
    }
    else if ( strcmp ((char*)cmd, "ping") ) {
          cmd_ping(argv[1]);
    }
    else if ( strcmp ((char*)cmd, "ipd") ) {
          ipd();
```

```
}
```

Uses cmd_ping 77c, get_hostname 56, hostname 47c, ipd 65, MAXLEN_HOSTNAME 47c, and set_hostname 57b.

Eine Anpassung an der "ulix.c" ist zusätzlich zur "sh.c" notwendig. Die "ulix.c" ist der Kernel vom Ulix-i386-System. Dieser muss angepasst werden, damit bei eintreffen eines Interrupts, durch den UART der seriellen Schnittstelle 2, eine Funktion im SLIP Modul aufgerufen wird. Der Kernel ruft im Normalfall bei eintreffen eines Interrupts am seriellen Port 2 die Funktion `serial_hard_disk_handler` auf. Dieser Aufruft der Funktion muss angepasst werden auf die "slip-mod" Funktion `slip_isr`.

23 ⟨*ulix.c: Anpassung des IRQ Handlers für den COM2 Port (Zeile 8190)* 23⟩≡

```
irq_install_handler (IRQ_COM1, serial_hard_disk_handler);
extern void slip_isr();
irq_install_handler (IRQ_COM2, slip_isr);
```

Uses IRQ_COM1 26b, IRQ_COM2 26b, and slip_isr 40b.

6.3.2 Erzeugung des Ulix-i386-OS-Code

Um aus dem extrahierten Quellcode ein lauffähiges Programm zu erzeugen, muss der Quellcode kompiliert und anschließend noch gelinkt werden. Zuerst wird die "ipv4-tools.c" wie in Listing 2 kompiliert und dann wie in Listing 3 gelinkt.

```
gcc46 -m32 -nostdlib -ffreestanding -fforce-addr -fomit-frame-
   pointer -fno-function-cse -nostartfiles -momit-leaf-frame-
   pointer -c ipv4-tools.c
```

Listing 2: Kompilieren der Userland Programme

Nach erfolgreicher Ausführung des "gcc" Aufrufes, hat der Compiler eine Objekt-Datei names "ipv4-tools.o" erzeugt. In dieser Datei befindet sich der Objekt-Code, welcher jetzt noch mit der "ulixlib" und der "sh" verlinkt werden muss.

```
ld -m elf_i386_fbsd -static -s --pie -T process.ld --entry 0 -o sh
    sh.o ulixlib.o ipv4-tools.o
```

Listing 3: Linken von ipv4-tools

Als nächstes wird das SLIP-Kernelmodul kompiliert und danach der "ulix" Kernel. Die Abfolge ist in Listing 4 beschrieben.

```
gcc46 -O0 -m32 -fno-stack-protector -std=c99 -g -O0 -nostdlib -
   nostdinc -fno-builtin -I./include -c -o slip-mod.o slip-mod.c
gcc46 -O0 -m32 -fno-stack-protector -std=c99 -g -O0 -nostdlib -
   nostdinc -fno-builtin -I./include -c -o ulix.o -aux-info ulix.
   aux ulix.c
```

Listing 4: Kompilieren des Kernelmoduls

Nach dem erzeugen der Objekt Dateien vom "slip-mod" und "ulix" Kernels, werden diese nur noch verlinkt. Der linker Befehl mit allen notwendigen Parametern ist in Listing 5 beschrieben.

```
ld -m elf_i386_fbsd -T ulix.ld -o ulix.bin *.o
```

Listing 5: Linken von slip-mod in den Ulix-Kernel

6.3.3 Erzeugung der Dokumentation

Um aus der "noweb"-Quelldatei eine Dokumentation in Form einer PDF Datei zu erhalten sind zwei Schritte notwendig. Zuerst wird mit dem Programm "noweave" die "noweb"-Quelldatei konvertiert in eine LaTeX Datei. Aus der erzeugten LaTeX Datei kann nun das PDF erzeugt werden. In Listing 6 ist die Befehlsabfolge beschrieben.

```
noweave -autodefs c -index -delay bachelorarbeit.nw >
   bachelorarbeit.tex
pdflatex bachelorarbeit.tex
makeindex bachelorarbeit.nlo -s ./nomencl.ist -o bachelorarbeit.
   nls -t bachelorarbeit.nlg
pdflatex bachelorarbeit.tex
pdflatex bachelorarbeit.tex
```

Listing 6: Erzeugen der Dokumentation

Für die Erzeugung des Abkürzungsverzeichnisses wird die "nomencl.ist" Datei benötigt. Diese muss im selben Verzeichnis liegen wie die LaTeX Datei "bachelorarbeit.tex". Für die Formatierung des eingebetteten Quellcodes aus der "noweb"-Quelldatei, wird die "noweb.sty" noch benötigt. Auch diese Datei muss im selben Verzeichnis wie die "bachelorarbeit.tex" liegen. Die doppelte Ausführung von "pdflatex" ist notwendig und liegt an der Verarbeitungsweise von LaTeX. In der Dokumentation von LaTeX [LAT14] kann dies nachgelesen werden. Nach erfolgreicher Ausführung liegt die Dokumentation als fertiges PDF vor. Das PDF besteht aus der Dokumentation und dem Quellcode.

7 Implementierung

Das SLIP-Modul für das Ulix-i386-OS enthält zwei Komponenten. Zum einen das Kernelmodul, bestehend aus der "slip-mod.h" und der "slip-mod.c". Und zum Anderen aus dem Userland-Komponenten bestehend aus "ipv4-tools.h" und "ipv4-tools.c". Die Header-Dateien "slip-mod.h" und "ipv4-tools.h" enthalten die Definitionen und Typdeklarationen. Die Funktionen befinden sich in den Dateien "slip-mod.c" und "ipv4-tools.c".

Das SLIP-Kernelmodul hat mehrere Aufgaben. Zum Einen werden die IP-Paket-Informationen empfangen und versendet. Und zum Anderen werden die Pakete in einem Puffer für die weitere Verarbeitung im Userland zwischen gespeichert. Das Kernelmodul wurde sehr klein gehalten, um nur die grundlegenden Funktionen zu gewährleisten. Die "ipv4-tools.c" enthält die ganze Logik und Algorithmen für die Verarbeitung der IP- und ICMP-Pakete. Für die Erstellung eines gültigen IP- oder ICMP-Paket müssen einige Funktionen in der "ipv4-tools.c" implementiert werden. Dazu zählt zum Beispiel die Erzeugung der richtigen Informationen zu den IP-Header Feldern mit der richtigen Byte-Order-Reihenfolge. Für die IP-Adressen muss eine Konvertierung von String auf Dezimal und umgekehrt implementiert werden. Die Prüfsumme der IP- und ICMP-Header werden berechnet und die IP-Felder TTL und IHL werden gesetzt.

Die meisten Funktionen enthalten keine Fehlerprüfungen für die übergebenen Parameterwerte. Eine Fehlerprüfung wurde bewusst weggelassen, weil diese den Quellcode weiter aufgebläht und die Übersichtlichkeit verringert hätte.

7.1 SLIP-Kernelmodul

Die Initialisierung des Kernelmoduls erfolgt über die Funktion `initialize_module`. Diese wird vom Ulix-Kernel beim Starten des Systems aufgerufen. In der Version 0.8 von Ulix-i386 passiert dies in der Funktion `main`. Zuerst soll ein kurzer Überblick über die Reihenfolge der Implementierungen des Kernelmoduls etwas mehr Übersicht schaffen. Die nachfolgenden Funktionen der "slip-mod.c" sind notwendig, damit die Übertragung der Informationen aus den Highlevel-Funktionen des Userlands überhaupt funktionieren.

7.1.1 Aufbau der slip-mod.h

In der "slip-mod.h" werden Makros, Variablen die CPU Register-Struktur und alle notwendigen Funktionen deklariert und in die "slip-mod.c" eingebunden.

Somit sind alle Funktionen, Variablen und Makros definiert, und müssen in der "slip-mod.c" nicht nochmal definiert werden.

26a ⟨*slip-mod.h* 26a⟩≡

```
#ifndef __SLIP_MOD__
#define __SLIP_MOD__
⟨SLIP Kernelmodul-Definitionen 26b⟩
⟨SLIP Kernelmodul-Variablen 29a⟩
⟨CPU Registerstruktur 35b⟩
⟨SLIP Kernelmodul-Funktionendeklarationen 28⟩
#endif
```

Defines:
__SLIP_MOD__, never used.

Die Definitionen der Funktionen aus der Datei "slip-mod.h" im Detail.

26b ⟨*SLIP* Kernelmodul-Definitionen 26b⟩≡ (26a)

```
#define NULL ((void*)0)
#define u_char unsigned char
#define u_short unsigned short
#define u_int unsigned int
#define IO_PIC1                0x20
#define IO_PIC2                0xA0
#define COM1                  0x3f8
#define COM2                  0x2f8
static int uart;
static int uart2;
#define IRQ_COM1                 4
#define IRQ_COM2                 3
u_short int USED_COM_PORT;
#define MAX_IPV4_MBUF           40
#define MAX_IPV4_MBUF_SIZE       8
#define SYS_recvmsg             27
#define SYS_sendmsg             28
#define SYS_mbufsize            29
#define SYS_delmbuf             30
#define SYS_gettick             31
#define DEBUG
#define END                    0300  /* Dezimal 192 */
#define ESC                    0333  /* Dezimal 219 */
#define ESC_END                0334  /* Dezimal 220 */
#define ESC_ESC                0335  /* Dezimal 221 */
```

Defines:
COM1, used in chunks 30b, 33, and 34.
COM2, used in chunks 30, 33, and 34.
DEBUG, used in chunks 38b, 41b, 58–60, 67b, 76a, 82c, and 83.
END, used in chunks 37b, 40c, and 42c.

ESC, used in chunks 37b, 40c, and 43a.
ESC_END, used in chunks 37b and 40c.
ESC_ESC, used in chunks 37b and 40c.
IO_PIC1, used in chunk 34.
IO_PIC2, used in chunk 34.
IRQ_COM1, used in chunk 23.
IRQ_COM2, used in chunks 23 and 40a.
MAX_IPV4_MBUF, used in chunks 29a, 38b, 41b, and 43b.
MAX_IPV4_MBUF_SIZE, used in chunks 29a, 39a, and 42a.
NULL, used in chunk 30b.
SYS_delmbuf, used in chunks 36a and 76b.
SYS_gettick, used in chunks 36a, 76a, 81b, and 82b.
SYS_mbufsize, used in chunks 36a, 67a, and 79a.
SYS_recvmsg, used in chunks 36a, 68, 69b, 71a, 72a, 74a, and 75a.
SYS_sendmsg, used in chunks 36a, 72b, and 81b.
u_char, used in chunks 28, 29a, 37b, 44, 46, 52c, 61a, 67a, 69a, 71b, 73, 74, 79a, and 81b.
u_int, used in chunks 28, 35b, 44, 46, 49a, 61, 63a, and 72b.
u_short, used in chunks 28–30, 44, 46, 49a, 52, 55a, 63c, 64, 67a, 71b, 73, and 79a.
uart, used in chunks 33–35.
uart2, used in chunk 33.

Zu erst wird das Makro __SLIP_MOD__ definiert falls es noch nicht definiert wurde. Dies verhindert ein nochmaliges Laden der Headerdatei beim Kompilieren. Dadurch kann die "slip-mod.h" mehrmals inkludiert werden, ohne dass der Compiler meckert. Um den Schreibaufwand beim weiteren Programmieren zu optimieren und auch gleich die Datentypen richtig fest zu legen, werden ein paar eigene Datentypen definiert. Diese neuen Datentypen stellen nur eine kurze Schreibweise der ursprünglichen Datentypen dar. Das Makro NULL gibt es bisher noch nicht in der Ulix-i386 Version 0.8.. Es folgen die UART und Interrupt-Adressen, die notwendig sind, um den UART zu programmieren. Der IO_PIC1 gibt die Adresse vom Master-Programmable-Interrupt-Controler (kurz PIC) an. Die IO Addresse vom Slave-PIC ist im IO_PIC2 gespeichert. Die Makros COM1 und COM2 definieren die Basisadressen für die seriellen Schnittstellen des UART. Die Makros uart und uart2 legen fest, welcher serielle Port verfügbar ist oder nicht. Sollte der serielle Port bereit und verfügbar sein, so wird der Wert auf 1 gesetzt. Mit IRQ_COM1, IRQ_COM2 werden die Interruptnummern definiert auf denen die seriellen Schnittstellen verbunden sind am Master PIC. Das Makro IRQ_SLAVE definiert die Interruptnummer vom Slave-PIC. Der Slave-PIC ist am Master-PIC mit Interrupt-Schnittstelle 2 verbunden. Der USED_COM_PORT gibt an, welche serielle Schnittstelle das SLIP Modul zum übertragen der Daten nutzen soll. Der IPv4-Buffer besteht aus einem zweidimensionalen Array namens ipv4_mbuf. Der erste Feldindex gibt das IP-Paket an, der zweite das einzelne Byte vom IP-Paket. Für die Interaktion mit dem Userland werden die Syscallnummern definiert. Die Syscalls des Ulix-i386 Version 0.8 sind in der "ulixlib.c" definiert. Die Nummern 27 bis 31 wurde frei gewählt und überschneiden sich mit keinem vorhandenen Syscall.

Um eine erweiterte Ausgabe auf der Konsole zu erhalten, gibt es das DEBUG Makro. Einige Makros für das Übertragen von speziellen Zeichen werden noch definiert. Makro END definiert das Byte für den Start und das Ende einer SLIP Übertragung. Das Makro ESC wird vollständigkeitshalber definiert. Es kommt aber nicht mehr zur Anwendung. Makro ESC_END und ESC_ESC sind Ersetzungsbytes für das Vorkommen einzelner ESC oder END Bytes während der Übertragung.

Es folgen die definierten Funktionen des Kernelmodules.

28 ⟨*SLIP* Kernelmodul-Funktionendeklarationen 28⟩≡ (26a)

```
extern u_char inportb (u_short);
extern void outportb (u_short, u_char);
extern insert_syscall (int , void*);
extern u_int system_ticks;
void initialize_module();
void slip_init(u_short int COM);
int check_uart();
int uart_putc(int c);
int uart_getc();
void syscall_sendmsg(struct regs *r);
void syscall_recvmsg();
void syscall_delmbuf();
void syscall_gettick();
void syscall_mbufindex(struct regs *r);
int slip_send(u_char *msg, int len);
void slip_recv(char *msg, int len);
short delmbuf(short index);
unsigned short get_sizeof_mbuf();
void slip_isr();
```

Uses c 41a 59a 93, check_uart 33d, delmbuf 38b, get_sizeof_mbuf 39a, initialize_module 30a, slip_init 30b, slip_isr 40b, slip_recv 40c, slip_send 37b, syscall_delmbuf 38a, syscall_gettick 39b, syscall_mbufindex 39a, syscall_recvmsg 36b, syscall_sendmsg 37a, u_char 26b 47c, u_int 26b 47c, u_short 26b 47c, uart_getc 35a, and uart_putc 34.

Für die Übermittlung der Informationen aus dem UART können die im Ulix-i386 vorhandenen Funktionen `inportb` und `outportb` verwendet werden. Die `inportb` Funktion gibt die aktuellen Daten der jeweiligen IO Adresse aus dem UART zurück. Die `outportb` Funktion wird gebraucht um Daten an den UART zu senden. Der UART wiederum versendet die Daten über die serielle Schnittstelle. Um die Funktionen `inportb` und `outportb` vom Ulix-i386 Kernel zu nutzen, müssen diese Funktionen als extern definiert werden. Eine wichtige Funktion für den Betrieb des SLIP-Moduls ist die `insert_syscall`, welche dazu dient, eine Syscallnummer mit einer Funktion zu verknüpfen. Als Parameter wird die Syscallnummer und die Speicheradresse der Funktion

übergeben. Für die Zeitmessung vom Versenden und Empfangen von ICMP-Paketen wird die system_ticks-Variable aus dem Kernel gebraucht.

Es werden die Initialisierungsfunktionen deklariert initialize_module und slip_init, die Funktionen für den UART zugriff check_uart, uart_putc, uart_getc und die Syscallfunktionen syscall_sendmsg, syscall_recvmsg, syscall_delmbuf, syscall_gettick und syscall_mbufindex. Jede dieser Syscallfunktionen ruft selbst wiederum eine weitere Funktionen auf. Diese heißen slip_send, slip_recv, delmbuf und get_sizeof_mbuf. Die slip_isr Funktion wird aufgerufen sobald ein Interrupt am IRQ Port 4 anliegt.

29a ⟨*SLIP* Kernelmodul-Variablen 29a⟩≡ (26a)

```
u_char ipv4_mbuf[MAX_IPV4_MBUF_SIZE][MAX_IPV4_MBUF];
u_short receive;
u_short ipv4_mbuf_byte;
u_short ipv4_mbuf_index;
u_char esc_receive;
```

Uses MAX_IPV4_MBUF 26b, MAX_IPV4_MBUF_SIZE 26b, u_char 26b 47c, and u_short 26b 47c.

Der IP-Buffer wird initialisiert mit maximal 8 IP-Paketen mit der Variable MAX_IPV4_MBUF_SIZE, und jeweils 40 Bytes mit der Variable MAX_IPV4_MBUF. Das reicht aus um einfache ICMP-Nachrichten mit kurzen Daten zu empfangen. Die Variable receive dient zum zählen der empfangenen Bytes. Diese Variable wird verwendet um zu ermitteln ob ein übertragenes END-Zeichen zu einem neuen IP-Paket gehört oder das IP-Paket fertig übertragen wurde. Die Variable ipv4_mbuf_byte ist der Array-Index vom aktuellem IP-Paket. Die Variable ipv4_mbuf_index gibt die aktuelle Schreibposition des Bytes vom IP-Paket an. Die Variable esc_receive wird gesetzt, wenn ein ESC Zeichen empfangen wird.

7.1.2 Aufbau der slip-mod.c

In der ”slip-mod.c” werden die Funktionen implementiert die für den Empfang und den Versand von IP- oder ICMP-Pakten notwendig sind. Als Erstes wird die Lizenz als Kommentar eingebunden und anschließend die ”slip-mod.h”. Danach folgen die einzelnen Implementierungen der Funktionen.

29b ⟨*slip-mod.c* 29b⟩≡

```
⟨license 93⟩

#include "slip-mod.h"
```

⟨*Implementierung von* initialize_module 30a⟩
⟨*Implementierung von* slip_init 30b⟩
⟨*Implementierung von* check_uart 33d⟩
⟨*Implementierung von* slip_isr 40b⟩
⟨*Implementierung von* slip_send 37b⟩
⟨*Implementierung von* slip_recv 40c⟩
⟨*Implementierung von* uart_putc 34⟩
⟨*Implementierung von* uart_getc 35a⟩
⟨*Implementierung von* syscall_recvmsg 36b⟩
⟨*Implementierung von* syscall_sendmsg 37a⟩
⟨*Implementierung von* syscall_mbufsize 39a⟩
⟨*Implementierung von* syscall_delmbuf 38a⟩
⟨*Implementierung von* syscall_gettick 39b⟩
⟨*Implementierung von delmbuf* 38b⟩

7.1.3 Initialisierung der seriellen Schnittstelle

Der Ulix-Kernel ruft nach dem Programmstart die Funktion initialize_module auf, welche als extern definiert ist. Somit wird die Funktion initialize_module des SLIP-Modul aufgerufen. Bei der Initialisierung durch initialize_module werden einige weitere Funktionen aufgerufen. Zuerst wird der UART und die IRQ Maskierung aktiviert innerhalb der slip_init Funktion. Die Funktion slip_init wird dem zu benutzenden COM-Port übergeben. In diesem Fall wird der COM2 verwendet. Der COM1 dient als Debugausgabe des Kernels auf der seriellen Konsole.

30a ⟨*Implementierung von* initialize_module 30a⟩≡ (29b)

```
void initialize_module () {
  slip_init(COM2);
  ⟨initialize_module: Variablen 33c⟩
  ⟨initialize_module: Syscalls 36a⟩
  return;
}
```

Defines:
initialize_module, used in chunk 28.
Uses COM2 26b and slip_init 30b.

In der slip_init wird der zu benutzende COM-Port festgelegt. Dafür dient die USED_COM_PORT Variable. Danach folgt die Konfiguration des UART über mehrere Schritte.

30b ⟨*Implementierung von* slip_init 30b⟩≡ (29b)

```
void slip_init(u_short int COM) {
  char *COM_NAME;
```

```
    switch (COM) {
      case COM1:
        COM_NAME = "COM1";
        USED_COM_PORT = COM1;
        break;
      case COM2:
        COM_NAME = "COM2";
        USED_COM_PORT = COM2;
        break;
      default:
        COM_NAME = "unknown";
        USED_COM_PORT = NULL;
        break;
    }
    ⟨slip_init: Aktivieren des UART 31⟩
    ⟨slip_init: Setzten des DLAB 32a⟩
    ⟨slip_init: Einstellen der Baudrate 32b⟩
    ⟨slip_init: UART Interrupt Versand aktivieren 32c⟩
    ⟨slip_init: Bitrate setzen 32d⟩
    ⟨slip_init: Data Terminal Ready 32e⟩
    ⟨slip_init: UART Rückmeldung zur CPU 32f⟩
    ⟨slip_init: Prüfen des UART Ports 32g⟩
    ⟨slip_init: Speichern des Port Status 33a⟩
    ⟨slip_init: UART Konfigurationsabschluss 33b⟩
  }
```

Defines:
slip_init, used in chunks 28 and 30a.
Uses COM1 26b, COM2 26b, NULL 26b 47c, and u_short 26b 47c.

7.1.4 Konfiguration des UART

Die Konfiguration des UART erfolgt über einfache Wertezuweisung der UART Register. Die Reihenfolge der Initialisierung wurde vom xv6 [XV14] Betriebssystem weitgehend übernommen.

Die zu ändernden Werte, werden als Parameter an die Funktion `outportb` übergeben.

Die Basisadresse von COM addiert mit 2 stellt den Zugriff zum First-In-First-Out-Buffer-Control-Register [NSC95] sicher. Mit dem Wert 0 wird der FIFO vom UART aktiviert.

31 ⟨slip_init: *Aktivieren des UART* 31⟩≡ (30b)

```
    outportb(COM+2, 0);
```

Über das Offset +3 mit dem Hexadezimal-Wert 0x80 wird das Divisor-Latch-Access-Bit (kurz DLAB) im Line-Control-Register gesetzt. Dies ist notwendig um die Baudrate einzustellen.

32a ⟨slip_init: *Setzten des DLAB* 32a⟩≡ (30b)
```
outportb(COM+3, 0x80);
```

Die Einstellung der Baudrate erfolgt über das Divisor-Latch-Byte. Bei dem Divisor-Latch-Byte handelt es sich um den Konfigurationswert der Baudrate, die verwendet werden soll. Der Wert errechnet sich aus der Formel $DivisorLatchByteValue = \frac{115200}{Baudrate}$. Eine Liste aller Werte für die Baudraten ist im PC16550D [NSC95] beschrieben.

32b ⟨slip_init: *Einstellen der Baudrate* 32b⟩≡ (30b)
```
outportb(COM+0, 115200/9600);
```

Sobald Daten am UART empfangen werden, soll ein Interrupt ausgelöst werden. Dazu wird im Interrupt-Enable-Register der Interrupt aktiviert.

32c ⟨slip_init: *UART Interrupt Versand aktivieren* 32c⟩≡ (30b)
```
outportb(COM+1, 0);
```

Die Daten-Bitrate wird auf 8 Bit im Line-Control-Register eingestellt.

32d ⟨slip_init: *Bitrate setzen* 32d⟩≡ (30b)
```
outportb(COM+3, 0x03);
```

Im Modem Control Register wird nun das Data Terminal Ready auf 0 gesetzt.

32e ⟨slip_init: *Data Terminal Ready* 32e⟩≡ (30b)
```
outportb(COM+4, 0);
```

Im Interrupt-Enable-Register wird der Enable-Transmitter-Holding Register-Empty-Interrupt gesetzt. Damit gibt der UART der CPU eine Rückmeldung nachdem die Daten vom UART gesendet wurden.

32f ⟨slip_init: *UART Rückmeldung zur CPU* 32f⟩≡ (30b)
```
outportb(COM+1, 0x01);
```

Es muss auch geprüft werden, ob der serielle Port überhaupt verfügbar ist. Dazu wird das Line-Status-Register abgefragt. Sollte der Wert 0xFF ergeben, so ist der Port nicht verfügbar.

32g ⟨slip_init: *Prüfen des UART Ports* 32g⟩≡ (30b)
```
if(inportb(COM+5) == 0xFF) {
  printf("SLIP: Serial Port could not enabled, \
no IRQ free\n");
  return;
}
```

Wenn der serielle Port-COM verfügbar ist, so wird die jeweilige uart Variable auf 1 gesetzt. Die Variable uart ist für COM1 und uart2 für COM2.

33a ⟨slip_init: *Speichern des Port Status* 33a⟩≡ (30b)

```
switch (COM) {
  case COM1:
    uart = 1;
    break;
  case COM2:
    uart2 = 1;
    break;
}
```

Uses COM1 26b, COM2 26b, uart 26b, and uart2 26b.

Als Letztes wird der Receive-Buffer eingelesen, jedoch ohne die Informationen zu verarbeiten. Ebenfalls wird das Interrupt-Identification-Register abgefragt. Wenn alles erfolgreich Initialisiert wurde, wird eine Ausgabe auf der Konsole erzeugt.

33b ⟨slip_init: *UART Konfigurationsabschluss* 33b⟩≡ (30b)

```
inportb(COM+0);
inportb(COM+2);
printf("SLIP: Serial console port %s initialized\n",
        COM_NAME);
```

Ein paar Variablen werden in der Funktion initialize_module mit Startwerten initialisiert.

33c ⟨initialize_module: *Variablen* 33c⟩≡ (30a)

```
receive = -1;
ipv4_mbuf_byte = 0;
ipv4_mbuf_index = 0;
esc_receive = 0;
```

7.1.5 Senden von Bytes über den UART

Mit der check_uart Funktion wird geprüft, ob der genutzte COM-Port überhaupt aktiviert und bereit ist. Dazu wird die uart Variable je COM-Port ausgewertet.

33d ⟨*Implementierung von* check_uart 33d⟩≡ (29b)

```
int check_uart() {
  switch (USED_COM_PORT) {
    case COM1:
      if(uart)
```

```
        return 0;
      break;
      case COM2:
        if(uart2)
          return 0;
      break;
      default:
      break;
        }
    return -1;
  }
```

Defines:
  check_uart, used in chunks 28, 34, and 35a.
Uses COM1 26b, COM2 26b, uart 26b, and uart2 26b.

Das Versenden der einzelnen Bytes über die serielle Schnittstelle erfolgt über die uart_putc Funktion. Der Übergabeparameter c ist das jeweilige Byte welches versendet werden soll. Vor dem Versand wird geprüft, ob der uart schon aktiviert wurde. Falls das nicht der Fall ist, wird ein Hinweis ausgegeben. Wenn der Port aktiviert ist, wird die PIC_DATA_PORT Variable gesetzt. Je nach COM-Port ist diese entweder IO_PIC1 oder IO_PIC2. Die Variable PIC_DATA_PORT enthält die IO Adresse des jeweiligen COM Ports. Die for Schleife erzeugt eine kleine Verzögerung sobald das Line-Status-Register vom UART nicht bereit sein sollte, oder der Ports nicht aktiv ist. Dies kann der Fall sein, wenn der UART gerade Daten versendet.

34 ⟨*Implementierung von* uart_putc 34⟩≡ (29b)

```
int uart_putc (int c) {
  int i, PIC_DATA_PORT;
  if (check_uart() == -1) {
    printf("SLIP: Could not get stream from uart\n");
  }

  if(USED_COM_PORT == COM1) {
    PIC_DATA_PORT = IO_PIC1;
  }
  if(USED_COM_PORT == COM2) {
    PIC_DATA_PORT = IO_PIC2;
  }

  for(i = 0; i < 128 && !(inportb(USED_COM_PORT+5) &
          PIC_DATA_PORT); i++)
    microdelay(10);
  outportb(USED_COM_PORT+0, c);
}
```

Defines:
uart_putc, used in chunks 28 and 37b.
Uses c 41a 59a 93, check_uart 33d, COM1 26b, COM2 26b, i 41a 59a 63b 67a 69a 74b 79a, IO_PIC1 26b, IO_PIC2 26b, and uart 26b.

7.1.6 Empfang von Bytes aus dem UART-Puffer

Das auslesen der einzelnen Bytes vom UART Puffer geschieht in der Funktion uart_getc. Zuerst wird der uart geprüft ob der COM-Port aktiv und bereit ist. Wenn der Port aktiv ist, wird noch geprüft ob der Port gesperrt ist. Dazu wird das Line-Status-Register abgefragt und falls der Port aktiv und nicht gesperrt ist, wird das Byte aus dem Puffer direkt ausgelesen und direkt zurückgegeben.

35a ⟨*Implementierung von* uart_getc 35a⟩≡ (29b)
```
int uart_getc () {
  if (check_uart() == -1) {
    printf("SLIP: Could not get stream from uart\n");
    return -1;
  }
  if(!(inportb(USED_COM_PORT+5) & 0x01))
    return -1;
  return inportb(USED_COM_PORT+0);
}
```
Defines:
uart_getc, used in chunks 28 and 40c.
Uses check_uart 33d and uart 26b.

7.1.7 Syscalls

Der Syscall-Handler vom Ulix-Kernel erwartet einen korrekten Datentyp um die Informationen aus den CPU-Registern auszuwerten. Dieser Datentyp ist in Form einer Struktur "regs" umgesetzt worden. Die Struktur wurde vom Ulix-System übernommen und implementiert.

35b ⟨*CPU Registerstruktur* 35b⟩≡ (26a)
```
struct regs {
  u_int gs, fs, es, ds;
  u_int edi, esi, ebp, esp, ebx, edx, ecx, eax;
  u_int int_no, err_code;
  u_int eip, cs, eflags, useresp, ss;
};
```
Uses u_int 26b 47c.

Jeder Syscall wird beim Laden des SLIP-Moduls mit einer Nummer und einer Funktion verknüpft. Die jeweilige Syscallnummer ist in der "slip-mod.h" schon deklariert.

36a ⟨initialize_module: *Syscalls* 36a⟩≡ (30a)

```
insert_syscall(SYS_recvmsg, syscall_recvmsg);
insert_syscall(SYS_sendmsg, syscall_sendmsg);
insert_syscall(SYS_mbufsize, syscall_mbufindex);
insert_syscall(SYS_delmbuf, syscall_delmbuf);
insert_syscall(SYS_gettick, syscall_gettick);
```

Uses SYS_delmbuf 26b 47c, SYS_gettick 26b 47c, SYS_mbufsize 26b 47c, SYS_recvmsg 26b 47c, SYS_sendmsg 26b 47c, syscall_delmbuf 38a, syscall_gettick 39b, syscall_mbufindex 39a, syscall_recvmsg 36b, and syscall_sendmsg 37a.

Die Funktion syscall_recvmsg holt die Daten aus dem ipv4_mbuf Array. Dazu kopiert syscall_recvmsg das angeforderte IP-Paket aus dem ipv4_mbuf in ein Array, welches der Funktion übergeben wird.

36b ⟨*Implementierung von* syscall_recvmsg 36b⟩≡ (29b)

```
void syscall_recvmsg(struct regs *r){
  if(r->ecx == ipv4_mbuf_index){
    r->eax = -1;
    return;
  } else {
    memcpy(r->ebx, &ipv4_mbuf[r->ecx][0], r->edx);
        r->eax = 0;
    return;
  }
}
```

Defines:
syscall_recvmsg, used in chunks 28 and 36a.
Uses memcpy 81b.

Im CPU-Register ebx ist die Speicheradresse des zu befüllenden Arrays gespeichert. Das CPU-Register ecx gibt den Index für das IP-Paket vomipv4_mbuf an. Das CPU-Register edx enthält die Anzahl der zu kopierenden Bytes des IP-Pakets. Der ipv4_mbuf_index enthält den aktuellen Index des IP-Pakets vom ipv4_mbuf Arrays, welches aktuell vom SLIP-Kernelmodul genutzt wird um ankommende Daten vom UART zu speichern. Auf diesem Index darf erst wieder gelesen werden, sobald das IP-Paket fertig im ipv4_mbuf Array gespeichert wurde. Sollte der Index, aus der Anfrage vom Userland, mit dem aktuell im Kernel gesperrten Index übereinstimmen, so wird der Syscall abgebrochen. Es werden damit Fehler beim Auswerten der IP-Daten verhindert, falls das jeweilige IP-Paket noch nicht fertig gespeichert wurde.

Die Funktion `syscall_sendmsg` überträgt die Daten vom Userland zum Kernel, welche dann vom UART an die serielle Schnittstelle versendet werden.

37a ⟨*Implementierung von* `syscall_sendmsg` 37a⟩≡ (29b)

```
void syscall_sendmsg(struct regs *r){
  r->eax = slip_send ((char*) r->ebx, (int) r->ecx);
  return;
}
```

Defines:
`syscall_sendmsg`, used in chunks 28 and 36a.
Uses `slip_send` 37b.

Das CPU-Register `ebx` enthält den Zeiger auf das Array, welches Übertragen werden soll. Das CPU-Register `ecx` enthält die Anzahl der zu übertragenden Datenfelder.

Die Funktion `slip_send` überträgt die übergebenen Daten vom Userland an den UART.

37b ⟨*Implementierung von* `slip_send` 37b⟩≡ (29b)

```
int slip_send(u_char *msg, int len){
  uart_putc(END);
  while(len--){
    switch (*msg) {
      case END:
        uart_putc(ESC);
        uart_putc(ESC_END);
      break;
      case ESC:
        uart_putc(ESC);
        uart_putc(ESC_ESC);
      break;
      default:
        uart_putc(*msg);
      break;
    }
    msg++;
  }
  uart_putc(END);
  return 0;
}
```

Defines:
`slip_send`, used in chunks 28 and 37a.
Uses `END` 26b, `ESC` 26b, `ESC_END` 26b, `ESC_ESC` 26b, `u_char` 26b 47c, and `uart_putc` 34.

Die Funktion `uart_putc` sendet die einzelnen Bytes an den UART. Die Funktion `slip_send` wurde aus dem RFC 1055 übernommen und leicht angepasst.

Jedes SLIP-Datenpaket beginnt mit dem END-Zeichen gefolgt von den Daten. Der Datenstrom endet auch wieder mit einem END-Zeichen. Das Makros ESC hat die Dezimalnummer 219. Diese Werte liegen außerhalb des ASCII Codebereiches für Buchstaben. Jedoch können die Werte 192 und 219 als IP-Adressnummern verwendet werden. Deshalb muss bei der Übertragung geprüft werden, ob die Werte als Anfang- oder End-Zeichen gültig sind. Sollte es vorkommen, dass Daten übertragen werden sollen die ein END- oder ESC-Zeichen enthalten, so müssen diese ersetzte werden. Für das Ersetzten eines einzelnen END oder ESC Zeichen, werden einfach zwei Zeichen versendet. Das END-Zeichen wird ersetzt durch das senden von ESC und ESC_END. Und ESC wird ersetzt durch das Senden von ESC und ESC_ESC. Am Ende der Übertragung wird ein END Zeichen geschickt um die SLIP-Übertragung zu beenden.

Der Syscall syscall_delmbuf hat die Aufgabe das jeweilige Feld des IP-Buffers ipv4_mbuf zu löschen.

38a ⟨*Implementierung von* syscall_delmbuf 38a⟩≡ (29b)

```
void syscall_delmbuf(struct regs *r){
  r->eax = delmbuf(r->ebx);
  return;
}
```

Defines:
syscall_delmbuf, used in chunks 28 and 36a.
Uses delmbuf 38b.

Das CPU-Register ebx hält die Indexnummer des zu löschend IP-Paketes. Diese Nummer wird an die Funktion delmbuf übergeben.

38b ⟨*Implementierung von delmbuf* 38b⟩≡ (29b)

```
short delmbuf(short index){
  if (index == ipv4_mbuf_index){
#ifdef DEBUG
    printf("Index not deleted: %d\n", index);
#endif
    return -1;
  }
  int i;
  for(i=0;i<MAX_IPV4_MBUF;i++){
    ipv4_mbuf[index][i] = 0;
  }
#ifdef DEBUG
  printf("Index deleted: %d\n", index);
#endif
  return 0;
}
```

Defines:

delmbuf, used in chunks 28 and 38a.
Uses DEBUG 26b 47c, i 41a 59a 63b 67a 69a 74b 79a, and MAX_IPV4_MBUF 26b.

Die Funktion delmbuf prüft auch wieder ob der Array-Index der Variable index mit der ipv4_mbuf_index übereinstimmt. Ist dies der Fall, wird der Syscall abgebrochen. Ansonsten wird das IP-Paket mit der Indexnummer durch ein einfaches überschreiben des ipv4_mbuf[index] mit 0 gelöscht.

Der nachfolgende Syscall syscall_mbufindex gibt dem Userland die maximale Anzahl an IP-Paketen an, die dass ipv4_mbuf Arrays speichern kann.

39a ⟨*Implementierung von* syscall_mbufsize 39a⟩≡ (29b)

```
void syscall_mbufindex(struct regs *r){
  r->eax = get_sizeof_mbuf();
  return;
}
unsigned short get_sizeof_mbuf(){
  return MAX_IPV4_MBUF_SIZE;
}
```

Defines:
get_sizeof_mbuf, used in chunk 28.
syscall_mbufindex, used in chunks 28 and 36a.
Uses MAX_IPV4_MBUF_SIZE 26b.

Der Syscall syscall_gettick gibt die aktuelle Kernel-Systemtickvariable zurück. Dies dient zur Zeitmessung beim Versenden und Empfangen von ICMP-Paketen.

39b ⟨*Implementierung von* syscall_gettick 39b⟩≡ (29b)

```
void syscall_gettick(struct regs *r){
  r->eax = system_ticks;
  return;
}
```

Defines:
syscall_gettick, used in chunks 28 and 36a.

7.1.8 Interrupts

Bei jedem Empfang von Daten am UART, wird ein Interrupt ausgelöst. Der Interrupt wird vom Interrupt-Handler entgegengenommen und dieser ruft dann die Funktion slip_isr auf, falls der Interrupt vom COM2 kommt. Die Interrupt-Service-Routine ist registriert mit der Funktion

`irq_install_handler` von der "ulix.c" Im Ulix-i386 Kernel sind dazu in der Funktion `main` folgende zwei Zeilen eingefügt:

40a ⟨*ulix.c IRQ Install Routine* 40a⟩≡
```
extern void slip_isr();
irq_install_handler (IRQ_COM2, slip_isr);
```
Uses IRQ_COM2 26b and slip_isr 40b.

Die `slip_isr` macht nichts weiter, als die Funktion `slip_recv` mit den Parametern des aktuellen IP-Paketindex `ipv4_mbuf_index` und dem aktuellen IP-Paket Datenindex `ipv4_mbuf_byte` aufzurufen. Der dritte Parameter gibt die Anzahl der empfangenden Bytes an.

40b ⟨*Implementierung von* `slip_isr` 40b⟩≡ (29b)
```
void slip_isr(){
  slip_recv(&ipv4_mbuf[ipv4_mbuf_index][ipv4_mbuf_byte]
            ,1);
}
```
Defines:
slip_isr, used in chunks 23, 28, and 40a.
Uses slip_recv 40c.

7.1.9 Speichern von eingehenden IP-Daten

Der eigentlich SLIP-Empfangsalgorithmus befindet sich in der Funktion `slip_recv` und entspricht dem RFC 1055. Die Funktion `slip_recv` hat zwei Parameter, zum Ersten den Zeiger `msg` auf dem aktuellen `ipv4_mbuf` Index und als Zweites die Länge `len` der einzulesenden Daten. Nach der Variablendeklaration beginnt die `while`-Schleife anhand der Länge `len` zu iterieren. Die `while` Schleife läuft solange bis die Anzahl der einzulesenden Bytes erreicht wurde. In der Variable `c` wird das aktuell vorliegende Byte vom UART Puffer geholt und zwischengespeichert. Dann folgt die Überprüfung des empfangenen Bytes.

40c ⟨*Implementierung von* `slip_recv` 40c⟩≡ (29b)
```
void slip_recv(char *msg, int len){
  ⟨slip_recv: Variablendeklaration 41a⟩
  while(len!=0){
    len--;
    c = uart_getc();
    receive++;
    switch(c) {
      case END:
        ⟨slip_recv: END-Zeichen verarbeiten 41b⟩
      break;
```

```
        case ESC:
          ⟨slip_recv: ESC-Zeichen verarbeiten 42b⟩
        break;
        case ESC_END:
          ⟨slip_recv: ESC_END-Zeichen verarbeiten 42c⟩
        case ESC_ESC:
          ⟨slip_recv: ESC_ESC-Zeichen verarbeiten 43a⟩
        default:
          ⟨slip_recv: Alle anderen Zeichen verarbeiten 43b⟩
        break;
      }
    }
  }
```

Defines:
slip_recv, used in chunks 28 and 40b.
Uses c 41a 59a 93, END 26b, ESC 26b, ESC_END 26b, ESC_ESC 26b, and uart_getc 35a.

Die Variable c speichert das aktuelle Zeichen, welches am UART Port anliegt. Variable i und j sind Hilfsvariablen.

41a ⟨slip_recv: *Variablendeklaration* 41a⟩≡ (40c)

```
unsigned char c;
int i,j;
```

Defines:
c, used in chunks 28, 34, 40c, 42, 43, and 58.
i, used in chunks 34, 38b, 41b, 43b, 58–61, 63, 65, 67–69, 75b, 77c, 82a, and 83.
j, used in chunks 41b, 63, and 65.

Beim Eintreffen eines END-Zeichens wird geprüft, ob receive schon gesetzt ist. Falls receive noch nicht gesetzt ist, wird es gesetzt. Ansonsten handelt es sich um das letzte END und somit ist das gesamte IP-Paket übertragen worden. In dem Fall, dass das letzte END-Zeichen übertragen wurde, wird nach den Daten-Bytes noch der aktuelle Systemtick gespeichert.

41b ⟨slip_recv: *END-Zeichen verarbeiten* 41b⟩≡ (40c)

```
if(receive == 0){
  receive=1;
} else {

  for(i=24,j=0;i>=0;i-=8,j++){
    ipv4_mbuf[ipv4_mbuf_index][ipv4_mbuf_byte+j] =
              (system_ticks>>i & 0xFF);
  }

#ifdef DEBUG
  printf("\nINDEX: %d :: ipv4_mbuf: ", ipv4_mbuf_index);
```

```
    for(i=0;i<MAX_IPV4_MBUF;i++){
      printf(" %d",ipv4_mbuf[ipv4_mbuf_index][i]);
    }
    printf("\n");
  #endif

  ⟨slip_recv: Bei END Zeichen, zurücksetzten der Variablen 42a⟩
  }
```

Uses DEBUG 26b 47c, i 41a 59a 63b 67a 69a 74b 79a, j 41a 63b 69a, and MAX_IPV4_MBUF 26b.

Der Systemtick selbst, ist eine Integer-Variable. Um das Integer nicht aufwändig in char zu konvertieren, werden einfach nur die Bits davon gespeichert. Das Bitshift der `for` Schleife erledigt diese Aufgabe. Dazu wird viermal ein Bitshift nach rechts um 8 Bits vorgenommen und mittels logischen "und" auf `0xFF` nur die letzten 8 Bits herausgefiltert.

Nach der DEBUG-Ausgabe folgt die Zurücksetzung einiger Variablen. Der IP-Paketindex wird erhöht, und bei erreichen der maximalen Array-Länge wieder auf Anfang gestellt.

42a ⟨slip_recv: *Bei END Zeichen, zurücksetzten der Variablen* 42a⟩≡ (41b)

```
ipv4_mbuf_byte=0;
ipv4_mbuf_index++;
receive = -1;
esc_receive = 0;
if(ipv4_mbuf_index >= MAX_IPV4_MBUF_SIZE) {
  ipv4_mbuf_index = 0;
}
```

Uses MAX_IPV4_MBUF_SIZE 26b.

Bei dem Empfang eines ESC-Zeichen wird dies vermerkt.

42b ⟨slip_recv: *ESC-Zeichen verarbeiten* 42b⟩≡ (40c)

```
esc_receive = 1;
*msg = c;
```

Uses c 41a 59a 93.

Bei dem Empfang eines ESC_END-Zeichen wird geprüft, ob vorher schon einmal ein ESC-Zeichen übertragen wurde, falls dies der Fall ist, wird das eingelesene Zeichen als END-Zeichen überschrieben und der ESC-Zeichen-Vermerkt wieder gelöscht.

42c ⟨slip_recv: ESC_END-*Zeichen verarbeiten* 42c⟩≡ (40c)

```
if(esc_receive == 1){
```

```
    c = END;
    esc_receive = 0;
  }
```

Uses c 41a 59a 93 and END 26b.

Bei dem Empfang eines ESC_ESC-Zeichen, wird auch wieder überprüft, ob das vorherige Zeichen ein ESC-Zeichen war. Ist dies der Fall, wird das empfangene Zeichen als ESC ersetzt, und der ESC-Zeichen-Vermerkt gelöscht.

43a ⟨slip_recv: ESC_ESC-*Zeichen verarbeiten* 43a⟩≡ (40c)

```
  if(esc_receive == 1){
    c = ESC;
    esc_receive = 0;
  }
```

Uses c 41a 59a 93 and ESC 26b.

Bei jeden anderen empfangenen Zeichen als END, ESC, ESC_END oder ESC_ESC, wird die default Verzweigung vom switch aufgerufen. Sollte zuvor ein ESC Zeichen empfangen worden sein, wird der ESC-Zeichen-Vermerkt gelöscht. Es folgt noch die Prüfung ob das Ende vom Array ipv4_mbuf erreicht wurde. Falls das IP-Array voll ist, wird das Paket im Array ipv4_mbuf mit einer Ausgabe verworfen. Ansonsten wird der IP-Paketindex erhöht und die Schreibposition für das nächste empfangene Zeichen auf Anfang gestellt. Nach der Prüfung auf das Ende des Arrays ipv4_mbuf, wird das Byte aus dem UART Puffer gespeichert. Die Variable ipv4_mbuf_byte wird auf das nächste freie Feld im Array ipv4_mbuf des aktuellen IP-Paket Indexes ipv4_mbuf_index erhöht.

43b ⟨slip_recv: *Alle anderen Zeichen verarbeiten* 43b⟩≡ (40c)

```
  if(esc_receive == 1){
    esc_receive = 0;
  }
  if(receive == MAX_IPV4_MBUF){
    printf("\nPacket to large for Buffer. Drop packet");
    for(i=0;i<=MAX_IPV4_MBUF; i++){
      ipv4_mbuf[ipv4_mbuf_index][i] = 0;
    }
    ipv4_mbuf_byte = 0;
    receive = -1;
    break;
  }
  if(receive >= 1){
    *msg = c;
    ipv4_mbuf_byte++;
  }
```

Uses c 41a 59a 93, i 41a 59a 63b 67a 69a 74b 79a, and MAX_IPV4_MBUF 26b.

7.2 IPv4/ICMP-Header

Die Header der Protokolle IP und ICMP werden definiert. Für beide Protokolle werden Strukturen verwendet. Die benötigten Definitionen befinden sind in der Datei "ipv4-tools.h"

7.2.1 Aufbau des IP-Headers

Das IPv4-Paket wird als einfache Struktur deklariert und entspricht dem RFC 791 [RFC791]. Die einzelnen IPv4-Header-Felder sind wie folgt konfiguriert:

44 ⟨*IPv4 Paketstruktur* 44⟩≡ (47a)

```
typedef struct ip_header {
 #ifdef LITTLE_ENDIAN
  u_char    HEADER_LENGTH:4,
            VERSION:4;
 #endif
 #ifdef BIG_ENDIAN
  u_char    VERSION:4,
            HEADER_LENGTH:4;
 #endif
  u_char    TOS;
  u_short   TOTAL_LENGTH;
  u_short   IDENTIFICATION;
 #ifdef LITTLE_ENDIAN
  u_short   FLAGS:3,
            FRAGMENT_OFFSET:13;
 #endif
 #ifdef BIG_ENDIAN
  u_short   FRAGMENT_OFFSET:13,
            FLAGS:3;
 #endif
  u_char    TIME_TO_LIVE;
  u_char    PROTOCOL;
  u_short   HEADER_CHECKSUM;
  u_int     SOURCE_IPADDRESS;
  u_int     DESTINATION_IPADDRESS;
  u_int     OPTIONS_AND_PADDING;
} __attribute__((packed)) IP;
```

Defines:
__attribute__, never used.
Uses BIG_ENDIAN 48, LITTLE_ENDIAN 48, u_char 26b 47c, u_int 26b 47c, and u_short 26b 47c.

Je nach definierten Endiantyp, wird die Anordnung der IP-Header-Felder "HEADER_LENGTH" und "VERSION" sowie "FLAGS" und "FRAGMENT_OFFSET" angepasst. Dies ist wichtig für den Versand und Empfang von Daten über das Netzwerk. Die Festlegung des Endian-Typs ist im Kapitel "Aufbau der ipv4-tools.h" beschrieben. Das Feld "VERSION" enthält die IP-Version. Im Feld "HEADER_LENGTH" ist die Länge des IP-Headers als Double-Word-Wert angegeben. Das Feld "TOS" ist für den Wert von Type of Service. Der Wert von Total Length des IP-Pakets wird im Feld "TOTAL_LENGTH" gespeichert. Der Identification Wert wird in das Feld "IDENTIFICATION" geschrieben. Die Felder "FLAGS" und "FRAGMENT_OFFSET" enthalten die Informationen zur Fragmentierung von IP-Paketen. Das Feld "TIME_TO_LIVE" enthält den Wert von Time-To-Live, sowie das Feld "PROTOCOL" die Protokoll Nummer des IP-Pakets. Im Feld "HEADER_CHECKSUM" ist die Prüfsumme des IP-Headers enthalten. Die Felder "SOURCE_IPADDRESS" und "DESTINATION_IPADDRESS" enthalten die IP-Adressen von Absender und Empfänger. Die IP-Header-Felder Option und Padding sind in der Struktur als ein Feld "OPTIONS_AND_PADDING" vorhanden. Wichtig ist, dass die Struktur zusammenhängend ohne sogenanntes Padding (Leerstellen) im System gespeichert wird. Dazu wird das Attribut `((packed))` hinter die Struktur geschrieben, damit der C Compiler die Struktur ohne padding erzeugt.

7.2.2 Aufbau des ICMP-Headers

Der Aufbau des ICMP-Paket ist sehr einfach gehalten und entspricht dem RFC 792 [RFC792]. Die einzelnen ICMP Pakettypen und Kodierungen werden durch Makros definiert. Die Makros ICMP_REQ_TYPE und ICMP_REQ_CODE werden zusammen verwendet um ein ICMP-Request-Paket zu erzeugen. Dass Antwortpaket auf ein ICMP-Request-Paket wird mit den Makros ICMP_REPLY_TYPE und ICMP_REPLY_CODE erzeugt.

45 ⟨*ICMP Paketstruktur* 45⟩≡ (47a) 46▷

```
#define ICMP_REQ_TYPE   8
#define ICMP_REQ_CODE   0
#define ICMP_REPLY_TYPE 0
#define ICMP_REPLY_CODE 0
```

Defines:
ICMP_REPLY_CODE, used in chunks 73 and 77c.
ICMP_REPLY_TYPE, used in chunks 73 and 77c.
ICMP_REQ_CODE, used in chunks 65 and 80.
ICMP_REQ_TYPE, used in chunks 65 and 80.

Die ICMP-Header-Struktur besteht aus nur sechs Datenfeldern und muss auch als zusammenhängendes Datenfeld im Programm gespeichert werden.

46 ⟨*ICMP Paketstruktur* 45⟩+≡ (47a) ◁45

```
typedef struct icmp_header {
  u_char  TYPE;
  u_char  CODE;
  u_short CHECKSUM;
  u_short ID;
  u_short SEQ;
  u_int   DATA;
} __attribute__((packed)) ICMP;
```

Defines:
__attribute__, never used.
Uses u_char 26b 47c, u_int 26b 47c, and u_short 26b 47c.

Die ICMP-Header-Felder "TYPE" und "CODE" bestimmen den ICMP-Pakettyp. Diese Felder werden, je nach Pakettyp, mit den oben definierten Makros befüllt. Das Feld "CHECKSUM" enthält die Prüfsumme des ICMP-Headers vom erzeugten Paket. Die Felder "ID" und "SEQ" stellen die Identifikations- und Sequenz-Nummer dar, und werden mit fortlaufenden Nummern verwendet, um bei Request und Responce festzustellen welches Antwortpaket zu welcher Anfrage gehört. Das Feld "DATA" ist flexibel gehalten und kann mehr als 32 Bit groß sein. In der Implementierung wird es auf 32 Bit begrenzt.

7.3 IPv4-Userlandfunktionen

Ein kurzer Überlick der Implementierungen der Userlandfunktionen. Nachdem der IP- und ICMP-Protokollheader definiert wurden, folgen die Definitionen und Deklarationen von Makros, Variablen und Funktionen in der "ipv4-tools.h". Dann folgt die Übersicht aller Funktionen innerhalb der "ipv4-tools.c". Zuerst werden alle "lowlevel" Hilfsfunktionen zur Verarbeitungen der Informationen für die IP- und ICMP-Header implementiert. Danach folgen die "Highlevel" Funktionen zur Bearbeitung der IP-Pakete. Die "Highlevel" Funktionen werden zusammen mit dem IP-Daemon Beschrieben um den Zusammenhang besser darzustellen. Zum Schluss folgt dann die Implementierung des "ping" Programmes.

7.3.1 Aufbau der ipv4-tools.h

Die Headerdatei "ipv4-tools.h" enthält alle Variablen, Markos und Funktionsdefinitionen welche, in der "ipv4-tools.c" verwendet werden. Nach der Preprozessorüberprüfung von __IPV4_TOOLS__ werden die Makros definiert. Danach

folgen die Strukturen für das IP- und ICMP-Paket und zum Schluss werden alle Funktionen deklariert.

47a ⟨*ipv4-tools.h* 47a⟩≡
```
#ifndef __IPV4_TOOLS__
#define __IPV4_TOOLS__
⟨Userland Definitionen 47c⟩
⟨Endian Definitionen 48⟩
⟨IP Protokollnummern 47b⟩
⟨IPv4 Paketstruktur 44⟩
⟨ICMP Paketstruktur 45⟩
⟨Deklarationen der Userland-Funktionen 49a⟩
#endif
```
Defines:
__IPV4_TOOLS__, never used.

Die IP- und ICMP-Protokollnummern werden als Makros definiert. Es wird die IP-Version 4 verwendet und ICMP mit der Protokollnummer 1 definiert.

47b ⟨*IP Protokollnummern* 47b⟩≡ (47a)
```
#define IP_VERSION       4
#define IP_PROTOCOL_ICMP 1
```
Defines:
IP_PROTOCOL_ICMP, used in chunks 65, 77c, and 79b.
IP_VERSION, used in chunk 79b.

Es folgen die weiteren Definitionen, welche nachfolgend erklärt werden.

47c ⟨*Userland Definitionen* 47c⟩≡ (47a)
```
#define u_char unsigned char
#define u_short unsigned short
#define u_int unsigned int
#define size_t int
#define NULL ((void*)0)
#define EMPTY           ""
#define O_RDONLY         0
#define SYS_recvmsg             27
#define SYS_sendmsg             28
#define SYS_mbufsize    29
#define SYS_delmbuf     30
#define SYS_gettick             31
#define DEBUG
#define FILE_HOSTS      "/etc/hosts"
#define FILE_HOSTNAME   "/etc/myhostname"
#define MAXLEN_HOSTNAME 64
char hostname[MAXLEN_HOSTNAME];
```

Defines:
DEBUG, used in chunks 38b, 41b, 58–60, 67b, 76a, 82c, and 83.
EMPTY, used in chunks 56 and 77c.
FILE_HOSTNAME, used in chunk 57a.
FILE_HOSTS, used in chunk 58.
hostname, used in chunks 22, 49a, 56, 57, 65, 77a, and 79b.
MAXLEN_HOSTNAME, used in chunks 22c and 57.
NULL, used in chunk 30b.
O_RDONLY, used in chunks 57a and 58.
size_t, used in chunk 51a.
SYS_delmbuf, used in chunks 36a and 76b.
SYS_gettick, used in chunks 36a, 76a, 81b, and 82b.
SYS_mbufsize, used in chunks 36a, 67a, and 79a.
SYS_recvmsg, used in chunks 36a, 68, 69b, 71a, 72a, 74a, and 75a.
SYS_sendmsg, used in chunks 36a, 72b, and 81b.
u_char, used in chunks 28, 29a, 37b, 44, 46, 52c, 61a, 67a, 69a, 71b, 73, 74, 79a, and 81b.
u_int, used in chunks 28, 35b, 44, 46, 49a, 61, 63a, and 72b.
u_short, used in chunks 28–30, 44, 46, 49a, 52, 55a, 63c, 64, 67a, 71b, 73, and 79a.

Wie im Kernelmodul, werden im Userland auch einige Makros gebraucht und gleich definiert. Das Makro EMPTY dient zur Prüfung auf leere Strings. Makro O_RDONLY definiert den Hexadezimalwert für das Öffnen einer Datei im readonly Modus. Die Syscallnummern vom Kernelmodul müssen auch im Userland identisch sein. Deshalb werden diese nochmals definiert. FILE_HOST enthält den fixen Pfad der Hostsdatei. Makro FILE_HOSTNAME gibt die Datei an, worin der Hostname des Systems gespeichert ist. Die maximale Länge des Hostnames wird auf 64 Zeichen Begrenzt.

Um die systemspezifische Endian-Einstellung zu setzten, wird das ARCH Makro gesetzt. Das Ulix-i386 läuft auf einer Intel i386 Architektur, welche Little-Endian verwendet. Die Endianess bestimmt die Reihenfolge der Bytes beim Schreiben und Lesen im Computersystem.

48 ⟨*Endian Definitionen* 48⟩≡ (47a)

```
#define ARCH_INTEL      1
#define ARCH_SPARC          2
#define ARCH            ARCH_INTEL

#if ARCH == ARCH_INTEL
#define             LITTLE_ENDIAN
#else
#define                BIG_ENDIAN
#endif
```

Defines:
ARCH, never used.
ARCH_INTEL, never used.
ARCH_SPARC, never used.
BIG_ENDIAN, used in chunk 44.

LITTLE_ENDIAN, used in chunk 44.

Es folgen die Definitionen der Funktionen in der Datei "ipv4-tools.h".

49a ⟨*Deklarationen der* Userland-Funktionen 49a⟩≡ (47a)
⟨*Einbinden von Ulixlib Funktion* 51a⟩

```
u_short calc_ip_hl(IP *ptr);
u_short calc_ip_tl(int len_ip_hl, int proto_hl);
u_short checksum (u_short count, u_short * ptr);
void check_checksum(IP *ip_packet, ICMP *icmp_packet);
int check_ipaddr(char *ipaddr);
void get_hostname(char *cHostname);
void set_hostname(char *hostname);
void get_ip_by_name(char *cIPaddr, char *search);
void get_name_by_ip(char * , char *);
void inet_iton(u_int number, char *ptr);
u_int inet_atoi(char * cIPAddr);
u_int bswap32(u_int value);
u_short bswap16(u_short value);
```

⟨*IP- und ICMP-Paketfunktionen* 49b⟩

Uses check_checksum 64, check_ipaddr 54, get_hostname 56, get_ip_by_name 58, hostname 47c, icmp_packet 67a, inet_iton 61b, ip_packet 67a 69a 79b, set_hostname 57b, u_int 26b 47c, and u_short 26b 47c.

Die Berechnung der IP-Headerlänge wird in der Funktion `calc_ip_hl` implementiert. Für die Berechnung der IP-Paket-Gesamtlänge dient die Funktion `calc_ip_tl`. Die Checksummen-Berechnung für das IP- oder ICMP-Paket ist in der Funktion `checksum` implementiert. Das Prüfen der Checksumme von empfangenen Paketen erledigt die Funktion `check_checksum`. Für die Überprüfung einer gültigen IP-Adresse dient die Funktion `check_ipaddr`. Zur Ermittlung und Einstellung des Hostnames sind die Funktionen `get_hostname` und `set_hostname` da. Die Funktionen `get_ip_by_name` und `get_name_by_ip` ermitteln die IP bzw. den Hostname anhand der gesuchten IP oder Hostnamen. Die Umwandlung der IP-Adresse als String in eine Zahl vom Datentyp Integer und umgekehrt, wird über die Funktionen `inet_iton` und `inet_atoi` erledigt. Für die Bearbeitung und Konvertierung der Variablen braucht es noch ein paar Hilfsmethoden. Die Funktionen `bswap32` und `bswap16` ändern die Byteorder des übergebenen Datentypen.

Es folgen weitere Funktionen zur Bearbeitung und Manipulation von IP- und ICMP-Paketen.

49b ⟨*IP- und ICMP-Paketfunktionen* 49b⟩≡ (49a)

```
int cmd_ping(char *);
short get_icmp_packet(short index, IP *ip_packet,
                      ICMP *icmp_packet, char *timearray);
void find_packet(short mbuf_size, char *search_ip,
                 short search_proto, char *ptr);
short find_icmp_packet(short index, short type, short code,
                       short id, short seq, IP *ip_packet,
                       ICMP *icmp_packet, char *timearray);
void icmp_response(IP *ip_packet, ICMP *icmp_packet);
void remove_unused_packets(short index);
short remove_packet(short index);
int ipd();
```

Defines:
find_icmp_packet, used in chunks 65 and 77c.
find_packet, used in chunks 65 and 77c.
get_icmp_packet, used in chunk 70a.
Uses cmd_ping 77c, icmp_packet 67a, icmp_response 72b, ip_packet 67a 69a 79b, ipd 65, remove_packet 76b, and remove_unused_packets 74a.

Der Shellbefehl "ping" ist in der Funktion `cmd_ping` realisiert. Die Funktion `get_icmp_packet` kopiert ein ICMP-Paket aus dem Kernel-Buffer anhand eines übergebenen Indexwertes. Die Funktion `find_packet` sucht anhand der übergebenen IP-Adresse und des Protokolls im Kernel nach IP-Paketen die dem Suchmuster entsprechen. Ist dies der Fall, wird das übergebene Array mit der Protokollnummer des gefundenen Paketes gespeichert. Die Funktion `find_icmp_packet` sucht anhand der übergebenen ICMP-Protokollfelder `type`, `code`, `id` und `seq` (Sequence), sowie der IP-Adresse, im Kernel nach dem passenden Paket. Wird ein Paket gefunden, welches zu den Suchmustern passt, werden das IP-Paket, das ICMP-Paket und der Systemtick vom Speicherzeitpunkt in die einzelnen Strukturen `ip_packet`, `icmp_packet` und in das Array `timearray` gespeichert. Die Funktion `icmp_response` erstellt und sendet ein ICMP-Antwortpaket. Die Funktion `remove_unused_packets` löscht Pakete aus dem Kernel-Buffer welche entweder zu lange gespeichert und nicht gelsöcht wurden, oder wenn nicht verwendetet Pakete noch gespeichert sind. Die Funktion `remove_packet` löscht ein Paket, anhand des Indexwertes, direkt aus dem Kernel-Buffer. Der IP-Daemon, welcher auch für die Beantwortung von ICMP-Request-Paketen zuständig ist, ist in der Funktion `ipd` implementiert.

Die Funktion `printf` wird benötigt für die Ausgabe von Text auf der Shell-Kommandozeile. Die Funktion `strlen` und `strncpy` sind für die Bearbeitung von Strings und `open`, `read` und `close` werden für die Dateibearbeitung benötigt. Die 6 Funktionen (`printf`, `strlen`, `strncpy`, `open`, `read`, `close`) stammen von der "ulixlib.c" und werden deshalb als extern eingebunden.

51a ⟨*Einbinden von Ulixlib Funktion* 51a⟩≡ (49a)

```
extern int printf(const char *format, ...);
extern int strlen(const char *str);
extern void *strncpy (void *dest, const void *src,
                      size_t count);
extern int open(const char *path, int oflag, ...);
extern int read(int fildes, void *buf, size_t nbyte);
extern int close(int fildes);
```

Uses buf 59a and size_t 47c.

7.3.2 Aufbau der ipv4-tools.c

In der "ipv4-tools.c" wird als erstes die Lizenz als Kommentar eingefügt. Dann folgt die Einbindung der "ipv4-tools.h", mit allen Makros, Variablen und Funktionsdefinitionen. Danach sind die einzelnen Funktionen implementiert. Die einzelnen Funktionen werden nachfolgend im Detail erklärt.

51b ⟨*ipv4-tools.c* 51b⟩≡

```
⟨license 93⟩

#include "ipv4-tools.h"

⟨Implementierung von calc_ip_hl 52a⟩
⟨Implementierung von calc_ip_tl 52b⟩
⟨Implementierung von get_hostname 56⟩
⟨Implementierung von get_ip_by_name 58⟩
⟨Implementierung von set_hostname 57b⟩
⟨Implementierung von inet_atoi 61a⟩
⟨Implementierung von inet_iton 61b⟩
⟨Implementierung von bswap32 63a⟩
⟨Implementierung von checksum 52c⟩
⟨Implementierung von bswap16 63c⟩
⟨Implementierung von check_checksum 64⟩
⟨Implementierung von check_ipaddr 54⟩
⟨Implementierung von IP Daemon 65⟩
⟨Implementierung von find_packet 68⟩
⟨Implementierung von find_icmp_packet 70a⟩
⟨Implementierung von get_icmp_packet 71a⟩
⟨Implementierung von icmp_response 72b⟩
⟨Implementierung von remove_unused_packets 74a⟩
⟨Implementierung von remove_packet 76b⟩
⟨Implementierung eines ping Programms 77c⟩
```

7.3.3 Berechnung der IP-Headerlänge

Die Berechnung der IP-Headerlänge ist wichtig, um zu ermitteln wie groß der IP-Header ist und um zu ermitteln wann das nächste Protokoll anfängt. Für die Berechnung der IP-Headerlänge wird die Struktur von IP benötigt. Dazu wird die Speicheradresse der Struktur an den Zeiger `ptr` als Parameter übergeben. Weil die Implementierung auf Padding und andere Optionsfelder verzichtet, wird ein fixer Wert von fünf zurückgegeben. Falls jedoch die IP-Felder von `OPTIONS_AND_PADDING` gefüllt sind, wird die Größe berechnet. Weil `sizeof` immer nur die Bytes angibt, und die IP-Header-Länge ein komplettes Feld von 32 Bit angibt, wird die Größe der Struktur durch vier geteilt.

52a ⟨*Implementierung von* `calc_ip_hl` 52a⟩≡ (51b)

```
u_short calc_ip_hl(IP *ptr){
        if (ptr->OPTIONS_AND_PADDING == 0){
                return 5;
        } else {
                return (sizeof(*ptr)/4);
        }
}
```

Uses `u_short` 26b 47c.

7.3.4 Berechnung der IP-Paketgesamtlänge

Die Berechnung der Total Length ist in der Funktion `calc_ip_tl` implementiert. Der minimale IP-Header ist 20 Bytes und ein IP-Paket kann maximal bis zu 64 KB groß sein. Die Berechnung ist sehr einfach. Es wird die Länge des IP-Headers inklusive der Länge des zu übertragenen Protokolls addiert.

52b ⟨*Implementierung von* `calc_ip_tl` 52b⟩≡ (51b)

```
u_short calc_ip_tl(int len_ip_hl, int proto_hl){
        return ((len_ip_hl * 4) + proto_hl);
}
```

Uses `u_short` 26b 47c.

7.3.5 Berechnung der Prüfsumme

Die Prüfsumme wird benötigt für die Prüfung eines gültigen Protokollheaders. Die `checksum` Berechnung entspricht dem RFC 1071 und ist wie folgt aufgebaut.

52c ⟨*Implementierung von checksum* 52c⟩≡ (51b)

```
u_short checksum (u_short count, u_short * ptr) {
```

```
    int sum = 0;
    while (count > 1) {
      sum += *ptr++;
      count-=2;
    }
    if( count > 0 ) {
      sum += * (u_char *) ptr;
    }
    while (sum >> 16){
      sum = (sum & 0xFFFF) + (sum >> 16);
    }
    return (unsigned short) ~sum;
  }
```

Uses u_char 26b 47c and u_short 26b 47c.

Es wird der Funktion `checksum` die größe des zu prüfenden Feldes an Variable `count` übergeben, und das zu prüfende Array selbst als Speicheradresse an den Zeiger `*ptr`. Die Variable `sum` ist der errechnete Rückgabewert der Funktion. Der Zeiger `*ptr` zeigt auf ein unsigned short, weil somit die Größe der Felder beim Einlesen auf 16 Bit eingeschränkt wird. Die Struktur auf die `ptr` zeigt kann damit in 16 Bit Blöcke iteriert werden. Es ist auch wichtig, dass die Strukturen vom IP- und ICMP-Paket keine Füllfelder (Padding) besitzen. Das wurde bei der Deklaration schon verhindert durch das `packed attribute` welches der Compiler nutzt, um keine Leer oder Füllfelder zwischen den einzelnen Daten zu schreiben.

Die Prüfsumme selbst ist eine einfache bitweise Berechnung mit Übertrag. Das Array `*ptr` wird iteriert, und jedes Feld zusammen addiert in die Variable `sum`. Bei einem 32 Bit Datentyp, werden jeweils die beiden 16 Bit Felder miteinander addiert. Dies geschieht solange, bis alle Werte addiert wurden. Wichtig ist, dass der Datentyp, von der Summe (`sum`), einen größeren Wertebereich besitzt als die iterieren Werte (`ptr`) selbst. Andernfalls verliert man den Übertrag. Deshalb ist die Variable "`sum`" ein integer Datentyp, und die Felder der Struktur ein short Datentyp. Danach wird der Übertrag noch addiert durch ein Bitshift um 16 Bits nach rechts. Das Ganze geschieht zweimal. Im Prinzip würde ein einfaches Bitshift auch ausreichen. Aber der Code ist somit theoretisch auch 64 Bit fähig, sofern der integer Datentyp auch 64 Bit groß ist. Zum Schluss wird das Ergebnis noch negiert und zurückgegeben.

7.3.6 Validierung einer IP-Adresse

Beim Einlesen und verarbeiten einer IP-Adresse, muss sichergestellt werden, das die IP-Adresse auch gültig ist. Die Überprüfung einer gültigen IP-Adresse

ist mit der Funktion check_ipaddr implementiert. Die Funktion wird bei der Ermittlung der IP-Adresse aus der Hostsdatei benötigt. Eine IPv4-Adresse besteht aus 4 Nummern mit den Werten von 0 bis 255, die durch 3 Punkte getrennt werden. Die Länge einer IP-Adresse beträgt minimale 7 Zeichen und maximal 16 Zeichen. Eine Beispiel-IP-Adresse wäre die "127.0.0.1".

Die Funktion check_ipaddr gibt nur dann den Wert 0 zurück, wenn die IP-Adresse gültig ist. Innerhalb der do-while Schleife wird jedes einzelne Zeichen der übergebenen IP-Adresse geprüft. Gültig sind nur Zeichen der Zahlen von 0 bis 9 sowie das "." Zeichen und das Array abschließende "0" Zeichen. Bei jeden anderen Zeichen bricht die Funktion ab, und es wird ein negativer Wert zurückgegeben. Als erstes wird die Anzahl der schon verarbeiteten Zeichen geprüft. Sollten es mehr als 16 Zeichen (einschließlich dem "0" Zeichen) sein, so kann es sich nicht um eine gültige IPv4-Adresse handeln. In dem Fall bricht die Funktion ab. Bei vorkommen eines "." Zeichens oder von Zahlen zwischen 0 bis 9 erfolgen weitere Verarbeitungen. Bei einem "0" Zeichen wird abschließend geprüft ob vorher die Punktzeichen ermittelt wurden, und die Anzahl der verarbeiteten Zeichen stimmen.

54 ⟨*Implementierung von* check_ipaddr 54⟩≡ (51b)

```
int check_ipaddr(char *ipaddr){
  ⟨check_ipaddr: Variablendeklarationen 55a⟩
  do{
        ccount++;
    if(ccount > 16) {
      return -1;
    }
    if(*ipaddr == '.'){
      ⟨check_ipaddr: Punktzeichen Verarbeitung 55b⟩
    } else if((*ipaddr >= 48) && (*ipaddr <= 57)){
      ⟨check_ipaddr: Zahlen Verarbeitung 55c⟩
    } else if (*ipaddr == '\0'){
      if((pcount == 3) && (ccount > 6)){
        return 0;
      }
      break;
    } else {
          return -2;
        }
        lastchar = *ipaddr;
  } while (*ipaddr++);
  if((pcount == 3) && (ccount > 6)){
    return 0;
  }
  return -1;
}
```

Defines:
check_ipaddr, used in chunks 49a, 59b, and 77c.
Uses lastchar 55a.

Die Variable pcount zählt alle vorkommenden Punkte in dem IP-Adressstring. Der ccount zählt alle vorkommenden Zeichen in der IP-Adresse. Die Variable num berechnet den Zahlenwert für jede Stelle in der IP-Adresse. Mit der Variable lastchar wird das vorhergehende Zeichen in der IP-Adresse gespeichert.

55a ⟨check_ipaddr: *Variablendeklarationen* 55a⟩≡ (54)
```
u_short pcount = 0;
u_short ccount = 0;
u_short num = 0;
char lastchar;
```
Defines:
lastchar, used in chunks 54 and 55b.
Uses u_short 26b 47c.

Bei vorkommen des Punktzeichens, wird der pcount Wert erhöht und geprüft. Wurden mehr als 3 Punkte ermittelt und ist das vorherige ebenfalls ein Punktzeichen gewesen, so bricht die Funktion ab.

55b ⟨check_ipaddr: *Punktzeichen Verarbeitung* 55b⟩≡ (54)
```
pcount++;
if((pcount > 3) || (lastchar == *ipaddr)){
  return -1;
}
num = 0;
```
Uses lastchar 55a.

Wenn das Zeichen in der IP-Adresse eine Zahl ist, so muss diese zum Gesamtwert der Zahlenstelle berechnet werden. Die Variable num wird dazu mit 10 multipliziert und dann mit dem Zahlenwert des Zeichens aus *ipaddr addiert. Wenn der Wert von num größer als 255 ergibt, wird die Funktion abgebrochen.

55c ⟨check_ipaddr: *Zahlen Verarbeitung* 55c⟩≡ (54)
```
num = (num*10) + (*ipaddr - '0');
if(num > 255){
  return -1;
}
```

7.3.7 Ermittlung des eigenen Hostnames

Unter Unix, Linux und sogar Windows enthalten das Programm "hostname", um den eigenen Hostnamen des laufenden Hostsystems zu erhalten. Mit diesem Programm wird der aktuelle Hostname ausgelesen und auf der Textkonsole ausgegeben. Dieser Ablauf des Programmes "hostname" wurde ähnlich implementiert. Unter Ulix reicht es die Datei "/etc/myhostname" auszuwerten. Der Dateiname selbst ist im Makro `FILE_HOSTNAME` definiert. Diese Datei wird beim Start vom SLIP-Modul eingelesen und als globale Variable `hostname` gespeichert. Die Funktion `get_hostname` erwartet eine Speicheradresse eines Arrays. Dieses Array, auf das der Zeiger `*cHostname` zeigt, wird befüllt mit dem Hostname aus der Datei die im Array `FILE_HOSTNAME` gespeichert ist. Als erstes wird geprüft, ob die Variable `hostname` schon gesetzt ist, und falls ja wird dem Zeiger `*cHostname` der aktuelle Wert aus `hostname` übertragen und die Funktion beendet. Zum Vergleichen ob der Hostname schon gesetzt ist nicht, wird die Variablen `hostname` und das Makro `EMPTY` verglichen.

Die Implementierung der `get_hostname` Funktion sieht wie folgt aus.

56 ⟨*Implementierung von* `get_hostname` 56⟩≡ (51b)

```
void get_hostname(char *cHostname){
  int len = 0;
  int fd;

  if (strcmp(&hostname, EMPTY) == 0) {
    strncpy(cHostname, &hostname, sizeof(hostname));
    return;
  }

  ⟨get_hostname: Hostname Datei einlesen 57a⟩

  if (strcmp(&hostname, EMPTY) == 1) {
    strncpy(&hostname, "localhost", 9);
  }
  strncpy(cHostname, &hostname, len);
}
```

Defines:
`get_hostname`, used in chunks 22c and 49a.
Uses `EMPTY` 47c and `hostname` 47c.

Wenn `hostname` noch nicht gesetzt ist, folgt das Einlesen der Hostsdatei. Falls die Datei nicht geöffnet werden kann, so soll das Programm einen Fehler anzeigen. Wenn die Datei zum Lesen geöffnet werden kann, so wird diese eingelesen. Anschließend wird das Array `hostname` nach dem letzten Zeichen noch terminiert.

57a ⟨get_hostname: *Hostname Datei einlesen* 57a⟩≡ (56)

```
fd = open(FILE_HOSTNAME, O_RDONLY);
if (fd == -1){
  printf("Failed to open File: %s\n", FILE_HOSTNAME);
} else {
  len = read(fd, &hostname, MAXLEN_HOSTNAME);
  close(fd);
  hostname[len-1] = '\0';
}
```

Uses FILE_HOSTNAME 47c, hostname 47c, MAXLEN_HOSTNAME 47c, and O_RDONLY 47c.

7.3.8 Ändern des Hostnames zur Laufzeit

Um den Hostname eines laufenden Ulix-Systems auch zu verändern, muss nur die Variable hostname geändert werden. Dies wird durch die Funktion set_hostname erreicht. Dieser Funktion wird einfach nur der neue Hostname übergeben. Als Parameter erwartet die Funktion set_hostname eine Speicheradresse zu einem Array.

57b ⟨*Implementierung von* set_hostname 57b⟩≡ (51b)

```
void set_hostname(char *cHostname){
  strncpy(&hostname, cHostname, MAXLEN_HOSTNAME);
}
```

Defines:
set_hostname, used in chunks 22c and 49a.
Uses hostname 47c and MAXLEN_HOSTNAME 47c.

7.3.9 Auflösen der IP vom Hostname

Die Auflösung eines Hostnames zu der jeweiligen IP-Adresse wird standardmäßig über DNS gelöst. Der eigene Hostname des Systems könnte zwar auch über einen DNS aufgelöst werden, dies ist aber nicht möglich, weil es aktuell noch keine Implementation eines DNS für Ulix gibt. Im Normalfall wird der eigene Hostname über die lokale Hostsdatei gefunden. Der Aufbau der lokalen Hostsdatei ist wie folgt: Jede Zeile enthält die IP-Adresse und einen oder mehrere Namen. Die Werte sind dabei mit Leerzeichen (sogenannten Whitespaces) getrennt.

Das einlesen und auswerten der Datei wurde wie folgt implementiert: Der Funktion get_ip_by_name wird das Array übergeben in welcher die IP-Adresse gespeichert werden soll. Als zweiter Parameter wird der gesuchte Hostname übergeben. Es wird zuerst die FILE_HOSTS im Lese-Modus geöffnet. Sollte dies fehlschlagen wird abgebrochen und eine Fehlermeldung angezeigt. Danach

folgt das Einlesen der IP-Adresse und des Hostnamens. Sollte die IP-Adresse des gesuchten Hostnames nicht gefunden werden, so wird die IP-Adresse auf 127.0.0.1 gesetzt.

58 ⟨*Implementierung von* get_ip_by_name 58⟩≡ (51b)

```
void get_ip_by_name(char *ipaddr, char *search){
  ⟨get_ip_by_name: Variablendeklarationen 59a⟩

#ifdef DEBUG
  printf("\nSEARCH: IP for Hostname [%s]\n", search);
#endif

  int fd = open(FILE_HOSTS, O_RDONLY);
  if (fd == -1){
    printf("Failed to open File: %s\n", FILE_HOSTS);
  } else {
    while (read(fd, &c, 1) != 0){
      buf[i] = c;
      if(((c == 32) || (c == 9)) && (wc == 0)){
                ⟨get_ip_by_name: Prüfen der IP Adresse 59b⟩
      }
      if (c == 10) {
                ⟨get_ip_by_name: Auslesen des Hostnames 60⟩
      }
      i++;
    }
    close(fd);
  }
  if(found != 1){
    strncpy(ipaddr, "127.0.0.1", 10);
#ifdef DEBUG
   printf("IP not found for [%s], set IP to %s\n",
           search, ipaddr);
#endif
  }
}
```

Defines:
get_ip_by_name, used in chunks 49a, 65, and 79b.
Uses buf 59a, c 41a 59a 93, DEBUG 26b 47c, FILE_HOSTS 47c, found 59a, i 41a 59a 63b 67a 69a 74b 79a, O_RDONLY 47c, and wc 59a.

Die Variable c ist das aktuell eingelesene Zeichen. Mit i wird die Position im Datenfeld gespeichert. Das Hilfsarray buf dient zum Zwischenspeichern der eingelesen Daten. Im Array tmp_ipaddr werden die IP-Adressen aus der Hostsdatei zwischengespeichert. Die wc Variable ist der Wortzähler und wird

gesetzt, wenn eine IP-Adresse eingelesen wurde und ein Hostname oder Mehrere folgen.

59a ⟨get_ip_by_name: *Variablendeklarationen* 59a⟩≡ (58)

```
char c;
int i = 0;
int wc = 0;
int found;
short size_ipaddr = 16;
char buf[1024];
char tmp_ipaddr[size_ipaddr];
```

Defines:
buf, used in chunks 51a, 58–60, 81b, and 83.
c, used in chunks 28, 34, 40c, 42, 43, and 58.
found, used in chunks 58 and 60.
i, used in chunks 34, 38b, 41b, 43b, 58–61, 63, 65, 67–69, 75b, 77c, 82a, and 83.
size_ipaddr, used in chunks 59b and 60.
tmp_ipaddr, used in chunks 59b and 60.
wc, used in chunks 58–60.

Falls nach der IP-Adresse ein Space- (Dezimal 32), Tabulator- (Dezimal 9) oder Newline- (Dezimal 10) Zeichen folgt, dann wird die eingelesene IP-Adresse geprüft. Dies erledigt die Funktion `check_ipaddr`. Wenn die Funktion `check_ipaddr` 0 zurück gibt, dann ist die IP-Adresse gültig und die IP-Adresse wird zwischengespeichert.

59b ⟨get_ip_by_name: *Prüfen der IP Adresse* 59b⟩≡ (58)

```
buf[i] = '\0';
if(check_ipaddr(&buf) == 0){
  strncpy(&tmp_ipaddr, &buf, size_ipaddr);
} else {
#ifdef DEBUG
  printf("String:%s is not a valid ipv4 address\n", buf);
#endif
}
i = -1;
wc = 1;
```

Uses buf 59a, check_ipaddr 54, DEBUG 26b 47c, i 41a 59a 63b 67a 69a 74b 79a, size_ipaddr 59a, tmp_ipaddr 59a, and wc 59a.

Nachdem die IP-Adresse eingelesen wurde, wird beim ersten weiteren "Space"-Zeichen abgebrochen und das Array `buf` mit dem gesuchten Hostname verglichen. Wenn der Eintrag aus der Hostsdatei mit dem gesuchten Namen übereinstimmt, wird der Hostname in `ipaddr` kopiert. Die IP-Adresse zum gesuchten Hostname wurde damit gefunden. Die Prüfung auf weitere Hostname-Einträge nach dem ersten gefundenen Namen wurde nicht implementiert. Es ist nicht

möglich eine Kombination wie: "192.2.3.1 hostname hostname.domain.de hostname.local" zu verwenden.

60 ⟨get_ip_by_name: *Auslesen des Hostnames* 60⟩≡ (58)

```
buf[i] = '\0';
if (wc == 1) {
  if (strcmp(&buf, search) == 1){
    strncpy(ipaddr, &tmp_ipaddr, size_ipaddr);
#ifdef DEBUG
   printf("IP found for [%s]: %s\n", search, ipaddr);
#endif
    found = 1;
    break;
  }
}
wc = 0;
i = -1;
```

Uses buf 59a, DEBUG 26b 47c, found 59a, i 41a 59a 63b 67a 69a 74b 79a, size_ipaddr 59a, tmp_ipaddr 59a, and wc 59a.

7.3.10 Konvertierung der IP-Adresse von String zu Dezimal

Der Datentyp für die Übertragung der Quell- und Ziel- IP-Adressen im IP-Protokoll ist ein Integer. Bei der Erstellung eines IP-Paketes werden aber die IP-Adressen als String angegeben. Deshalb muss der IP-Adress-String (bsp.: 192.34.56.32) umgewandelt werden in einen Dezimalzahlenwert. Als Parameter wird die IP-Adresse als String der Funktion `inet_atoi` übergeben. Der Rückgabewert der Funktion ist die IP-Adresse als Dezimalzahl. Es wird ein Array verwendet um jedes Oktet einzeln zu separieren und zu konvertieren (`oktet[]`). Ein Oktet ist der Dezimalzahlwert der Stelle in der IP-Adresse. Eine IP-Adresse hat immer 4 Oktets. Die Schleife durchläuft den String und trennt jedes Oktet bei vorliegen des "." Separators. Bei allen anderen Zeichen, des iterierten Strings, werden die char Datentypen zu einen Integer konvertiert. Um den richtigen Dezimalzahlenwert eines Oktets zu berechnen, muss eine Multiplikation mit 10, des vorhanden Wertes zum aktuellen Wert addiert werden. Danach werden die Zahlenwerte in den Oktets, durch Bitshift nach links, erhöht. Bei dem Beispiel mit der IP-Adresse 192.34.56.32 bedeutet dies, dass der Zahlenwert "192" im Array `oktet[0]` steckt, der Zahlenwert "34" steckt in `oktet[1]` usw. Um den integer Datentypwert zu berechnen, kann nicht einfach eine Addition von "192+32+56+32" gemacht werden. Jedes Oktet muss zuerst an der richtigen Position im Bitfeld des integer Datentyps liegen. Für das Oktet 1 (`oktet[0]`) mit dem Wert "192" wird daher ein Bitshift um 24 nach links vorgenommen. Jedes weitere Oktet wird um 8 Bits weniger

des vorherigen Oktets nach links verschoben. Danach folgt die Addition der neuen Werte, welche in der Variable `rvalue` gespeichert und zurückgegeben wird.

61a ⟨*Implementierung von* `inet_atoi` 61a⟩≡ (51b)

```
u_int inet_atoi(char *ipaddr){
  u_int rvalue = 0;
  u_char oktet[3];
  u_char count = 0;
  oktet[count] = 0;
  while(*ipaddr != '\0'){
    if(*ipaddr == '.') {
      count++;
      oktet[count] = 0;
      *ipaddr++;
    }
    oktet[count] = oktet[count]*10 + (*ipaddr-'0');
    *ipaddr++;
  }
  rvalue = (oktet[0]<<24) + (oktet[1]<<16) +
           (oktet[2]<<8) + oktet[3];
  return rvalue;
}
```

Uses `rvalue` 63b 71b, `u_char` 26b 47c, and `u_int` 26b 47c.

7.3.11 Konvertierung der IP-Adresse von Dezimal zu String

Die Rückkonvertierung der IP-Adresse als Dezimalzahl zu einem String geschieht in der Funktion `inet_iton`. Als Übergabeparameter wird die IP-Adresse übergeben und auch das Array worin der String gespeichert werden soll. Die Umwandlung der IP-Adresse als Dezimalzahl erfolgt in zwei Einzelschritten. Als Erstes wird die Dezimalzahl aufgeteilt in die einzelnen Oktets. Als Zweites folgt die Konvertierung der einzelnen Oktets in den Datentyp "char".

Die Variable `pos` bestimmt den Index zum Schreiben des Zeichens in dem Array `ipaddr`, und wird fortlaufend beim Schreiben in das Array erhöht. Die Variable `oktet` speichert den Zahlenwert der IP-Adresse pro Oktet. Die `for` Schleife iteriert 4 mal und extrahiert, mittels einem Bitshift um 8 nach rechts, den jeweiligen Dezimalwert des Oktets durch ein anschließendes Modulo der Variable `number`. Danach folgt die Umwandlung des Zahlenwertes von `Oktet` in einen String.

61b ⟨*Implementierung von* `inet_iton` 61b⟩≡ (51b)

```
void inet_iton(u_int number, char *ipaddr){
```

```
    int i;
    int pos = 0;
    u_int oktet;
    for(i=3;i>=0;i--){
      oktet = (number >> (8*i)) % 256;

      ⟨inet_iton: Umwandlung des Oktets von Dezimal zu String 62⟩

          if(i != 0){
            ipaddr[pos++] = '.';
          }
    }
    ipaddr[pos] = '\0';
  }
```

Defines:
inet_iton, used in chunks 49a and 68.
Uses i 41a 59a 63b 67a 69a 74b 79a and u_int 26b 47c.

Die Umwandlung des Zahlenwertes in einen String wird durch die Division und Restwertberechnung des Nummernwertes mit anschließender Addition des '0' Zeichens erreicht. Wenn der Zahlenwert einen dreistelligen Zahlenwert hat, so wird die erste Nummer des dreistelligen Wertes extrahiert und in das Array `ipaddr` mit der Position am Anfang geschrieben. Das Selbe passiert noch einmal für zweistellige und einstellige Zahlenwerte.

62 ⟨inet_iton: *Umwandlung des Oktets von Dezimal zu String* 62⟩≡ (61b)

```
  if(oktet > 99) {
    ipaddr[pos++] = (oktet / 100) + '0';
  }
  if(oktet > 9) {
    ipaddr[pos++] = ((oktet / 10) % 10) + '0';
  }
  ipaddr[pos++] = (oktet % 10) + '0';
```

7.3.12 Ändern der Byteorder

Jedes Paket, welches über das Netzwerk gesendet wird, unterliegt einer Byteorder. Diese Byteorder bestimmt die Reihenfolge der übertragenen Bytes und wird als "network byte order" [RFC1700] bezeichnet. Die "network byte order" entspricht dem Big-Endian Format. Die Byteorder der Daten muss eingehalten werden, damit die Informationen am Zielsystem richtig interpretiert werden. Zum Tragen kommt diese Reihenfolge bei allen Datenfeldern die mehr als 1 Byte belegen. Einige Felder des IP/ICMP-Headers belegen mehr als 1 Byte.

Deshalb müssen alle Felder, die mehr als 1 Byte groß sind, vor dem Versenden und auch nach dem Empfang der Änderung der Byteorder unterzogen werden.

Eine Änderung der Byteorder für einen 32 Bit Integer erledigt die Funktion `bswap32`. Das Prinzip besteht darauf, das durch einfaches Bitshift und einer Bitweisen "oder" Berechnung die Bytes vertauscht werden. Der Algorithmus selbst funktioniert wie folgt. Jeweils 8 Bit aus dem Array `value` werden solange verschoben und vertauscht, bis die Reihenfolge der Werte im Array umgekehrt wurde.

63a ⟨*Implementierung von bswap32* 63a⟩≡ (51b)

```
u_int bswap32(u_int value){
  ⟨bswap32: Variablendeklarationen 63b⟩
  for(j=24;j>=0;j-=8){
    tmp = ((value>>i)& 0xFF);
    rvalue = rvalue | tmp<<j;
    i+=8;
  }
  return rvalue;
}
```

Uses i 41a 59a 63b 67a 69a 74b 79a, j 41a 63b 69a, rvalue 63b 71b, tmp 63b, and u_int 26b 47c.

Die Variable `rvalue` ist der Rückgabewert, `i` und `j` sind Laufvariablen.

63b ⟨*bswap32: Variablendeklarationen* 63b⟩≡ (63a)

```
int rvalue = 0;
int tmp = 0;
int i = 0;
int j;
```

Defines:
i, used in chunks 34, 38b, 41b, 43b, 58–61, 63, 65, 67–69, 75b, 77c, 82a, and 83.
j, used in chunks 41b, 63, and 65.
rvalue, used in chunks 61a, 63, and 70–72.
tmp, used in chunks 63, 72b, and 73.

Das Selbe wird auch nochmal für 16 Bit gebraucht. Es reicht dazu den Startwert von `j` anzupassen.

63c ⟨*Implementierung von bswap16* 63c⟩≡ (51b)

```
u_short bswap16(u_short value){
  int rvalue;
  int tmp;
  int i,j;
  rvalue = 0;
  tmp = 0;
  i = 0;
```

```
    for(j=8;j>=0;j-=8){
      tmp = ((value>>i)& 0xFF);
      rvalue = rvalue ^ tmp<<j;
      i+=8;
    }
    return rvalue;
  }
```

Uses i 41a 59a 63b 67a 69a 74b 79a, j 41a 63b 69a, rvalue 63b 71b, tmp 63b, and u_short 26b 47c.

7.3.13 Überprüfung der Paketprüfsummen

Um eingehende Netzwerkpakete auf die Gültigkeit zu Prüfen, muss die Prüfsumme ausgewertet werden. Dazu wird die Prüfsumme des eingehenden Paketes einfach nochmals berechnet.

Der Funktion check_checksum wird die Speicheradresse der IP-Paketstruktur und der ICMP-Paketstruktur übergeben. Die Originalwerte werden temporär gespeichert. Die Werte der Prüfsummen vom IP- und ICMP-Header werden genullt. Dann folgt die Änderung der Byteorder für das Total-Length Feld vom IP-Header, um die ICMP-Paketlänge zu berechnen.

Die Länge von IP und ICMP kann jetzt der Funktion checksum übergeben werden. Sollten die Werte der berechneten und der temporär gespeicherten Prüfsummen nicht übereinstimmen, so wird eine Meldung ausgegeben. Zum Schluss werden die ursprünglichen Prüfsummenwerte zurückgeschrieben.

64 ⟨*Implementierung von* check_checksum 64⟩≡ (51b)

```
void check_checksum(IP *ip_packet, ICMP *icmp_packet){
  u_short ip_chksum_orig;
  u_short ip_chksum;
  u_short icmp_chksum_orig;
  u_short icmp_chksum;

  ip_chksum_orig = ip_packet->HEADER_CHECKSUM;
  icmp_chksum_orig = icmp_packet->CHECKSUM;

  ip_packet->HEADER_CHECKSUM = 0;
  icmp_packet->CHECKSUM = 0;
  ip_packet->TOTAL_LENGTH = bswap16(
                            ip_packet->TOTAL_LENGTH);
  int icmplen = ip_packet->TOTAL_LENGTH -
                (ip_packet->HEADER_LENGTH*4);
  ip_packet->TOTAL_LENGTH = bswap16(
                            ip_packet->TOTAL_LENGTH);
```

```
    ip_chksum = checksum((ip_packet->HEADER_LENGTH*4),
                         ip_packet);
    icmp_chksum = checksum(icmplen, icmp_packet);
    if(ip_chksum != ip_chksum_orig){
      printf("IP Checksum is wrong: %d, should: %u\n",
                  ip_chksum_orig, ip_chksum);
    };
    if(icmp_chksum != icmp_chksum_orig){
      printf("ICMP Checksum is wrong: %d, should: %u\n",
              icmp_chksum_orig, icmp_chksum);
    };
    ip_packet->HEADER_CHECKSUM = ip_chksum_orig;
    icmp_packet->CHECKSUM = icmp_chksum_orig;
  }
```

Defines:
check_checksum, used in chunks 49a and 82a.
Uses icmp_packet 67a, icmplen 73, ip_packet 67a 69a 79b, and u_short 26b 47c.

7.4 Der IP-Daemon

Für die Beantwortung von ICMP-Requestpaketen die an das Ulix-System gesendet werden, braucht es eine Möglichkeit auf diese Pakete automatisch zu antworten. Diese Aufgabe übernimmt der IP-Daemon. Die Hauptaufgabe des Daemons ist das Beantworten von ICMP-Requestpaketen und das Löschen von veralteten und nicht brauchbaren Paketen aus dem `ipv4_mbuf` Array. Zuerst wird der Prozess geforkt (⟨*ipd: Process Fork* 66⟩). Danach folgt die ⟨*ipd: Variablendeklarationen* 67a⟩. Bevor allerdings die Endlosschleife startet, wird der eigene Hostname aufgelöst, um dies später nicht jedes mal beim Iterieren neu aufzulösen. Bei jedem Durchlauf der `while` Schleife, wird das `packets` Array mit den IP-Protokollnummern der IP-Pakete aus dem Array `ipv4_mbuf` vom Kernel befüllt. Dann folgt die Iteration des Arrays `packets`. Bei jeder ICMP-Protokollnummer, im Array `packets`, wird das IP/ICMP Paket aus dem Array `ipv4_mbuf` geholt. Sollte im Array `ipv4_mbuf` ein gesuchtes ICMP-Requestpaket vorhanden sein, so wird ein ICMP-Responsepaket erzeugt und versendet (⟨*ipd: ICMP Antwortpaket senden* 67b⟩). Zum Schluss folgt das Löschen von nicht verwendetet Paketen aus dem Kernel. Nicht verwendetet Pakete sind z.B.: Pakete vom Protokoll IPv6. Die Implementierung des `ipd` sieht wie folgt aus:

65 ⟨*Implementierung von IP Daemon* 65⟩≡ (51b)

```
  int ipd(){
    ⟨ipd: Process Fork 66⟩
    ⟨ipd: Variablendeklarationen 67a⟩
    get_ip_by_name(&my_ip, &hostname);
```

```
    int j =0;
    while(1){
      find_packet(mbuf_size, &my_ip, IP_PROTOCOL_ICMP,
                  &packets);

      for(i=0;i<mbuf_size;i++){
        if(packets[i] == IP_PROTOCOL_ICMP) {
          ret = find_icmp_packet(i, ICMP_REQ_TYPE,
                                 ICMP_REQ_CODE, -1, -1,
                                 &ip_packet, &icmp_packet,
                                 &timearray);
          if(ret == 1){
                    ⟨ipd: ICMP Antwortpaket senden 67b⟩
          }
        }
        remove_unused_packets(i);
      }
    }
  }
```

Defines:
ipd, used in chunks 22, 49b, and 77a.
Uses find_icmp_packet 49b 70a, find_packet 49b 68, get_ip_by_name 58, hostname 47c, i 41a 59a 63b 67a 69a 74b 79a, icmp_packet 67a, ICMP_REQ_CODE 45, ICMP_REQ_TYPE 45, ip_packet 67a 69a 79b, IP_PROTOCOL_ICMP 47b, j 41a 63b 69a, remove_unused_packets 74a, and ret 67a 69a 74b 79a.

Um einen Daemonprozess zu erzeugen sind ein paar Schritte notwendig. Zuerst muss der Prozess geforkt werden. Weil die zur Programmierung vorliegende Ulix Version 0.8 noch keinen Binary-Loader wie den ELF Loader hat, sind alle Programme als Funktionen in der Shell eingebaut. Deshalb wird die aktuell laufende Shell geforkt. Auch eine Übernahme eines Vaterlosen-Prozesses vom Kernel ist nicht möglich. Eine Abhilfe bringt das zurückgeben des Vaterprozesses, ohne ein Exit/Kill auf die Prozess-ID zu machen. Die Prozess ID bleibt damit erhalten und der Prompt kehrt zur Shell Eingabe zurück. Der Prozess des Sohnes läuft im Hintergrund weiter. Beim Sohn oder auch Daemon müssen dann nur noch das aktuelle Arbeitsverzeichnis und der Name des Prozesses geändert werden. Das Arbeitsverzeichnis des Prozesses wird auf das Wurzelverzeichnis "/" geändert. Der Name des Prozesses wird geändert auf IPD.

66 ⟨*ipd: Process Fork 66*⟩≡ (65)

```
  int pid = fork();
  if (pid != 0) {
    return 0;
```

```
}
chdir("/");
setpsname ("IPD");
```

Defines:
pid, never used.

Die notwendigen Variablen der Funktion ipd im Überblick. Es wird die IP- und ICMP-Struktur benötigt um die Pakete aus dem ipv4_mbuf zu speichern. Für die aktuelle eigene IP-Adresse vom Host wird ein Array my_ip mit 16 Zeichen benötigt. Die Größe es Array ipv4_mbuf wird in der Variable mbuf_size über den Syscall SYS_mbufsize abgefragt und gespeichert. Ein Array packets wird angelegt und sicherheitshalber mit Nullen befüllt. Das Array timearray hat die Aufgabe die vier Bytes des Systemticks vom empfangenen IPv4-Paketes zwischenzuspeichern.

67a ⟨*ipd: Variablendeklarationen* 67a⟩≡ (65)

```
IP ip_packet;
ICMP icmp_packet;
u_char my_ip[15];
u_short mbuf_size;
short i = 0;
short ret;
mbuf_size = syscall1(SYS_mbufsize);
u_char packets[mbuf_size];
memset(&packets, 0, mbuf_size);
u_char timearray[4];
```

Defines:
i, used in chunks 34, 38b, 41b, 43b, 58–61, 63, 65, 67–69, 75b, 77c, 82a, and 83.
icmp_packet, used in chunks 49, 64, 65, 67b, and 70–73.
ip_packet, used in chunks 49, 64, 65, 67b, 68, 70–73, 77c, and 80–83.
ret, used in chunks 65, 68, 69b, 72b, 74a, 75a, 77c, 81b, and 82b.
Uses SYS_mbufsize 26b 47c, u_char 26b 47c, and u_short 26b 47c.

Für den Versand des ICMP-Antwortpakets werden drei Aktionen ausgeführt. Zuerst wird das ICMP-Requestpaket aus dem ipv4_mbuf wieder gelöscht. Dies kann gemacht werden, weil das Requestpaket nicht mehr gebraucht wird. Die IP-Protokollnummer aus dem Array packets wird wieder zurückgesetzt und es folgt der Versand des ICMP-Antwortpaketes.

67b ⟨*ipd: ICMP Antwortpaket senden* 67b⟩≡ (65)

```
#ifdef DEBUG
        printf("Found req, send response, i: %d\n", i);
#endif
remove_packet(i);
packets[i] = 0;
```

```
icmp_response(&ip_packet, &icmp_packet);
```

Uses DEBUG 26b 47c, i 41a 59a 63b 67a 69a 74b 79a, icmp_packet 67a, icmp_response 72b, ip_packet 67a 69a 79b, and remove_packet 76b.

Die erste Funktion die innerhalb der Endlosschleife vom IP-Daemon `ipd` aufgerufen wird ist die Funktion `find_packet`. Die Funktion `find_packet` hat die Aufgabe alle Protokollnummern der im Kernel gespeicherten IP-Pakete zu finden und in das Array `packets` zu schreiben. Um nicht alle Pakettypen zu ermitteln, werden die Suchparameter übergeben um die Ergebnisse zu filtern. Die Funktion `find_packet` ist wie folgt implementiert.

68 ⟨*Implementierung von* find_packet 68⟩≡ (51b)

```
void find_packet(short mbuf_size, char *search_ip,
                 short search_proto, char *ptr){
 ⟨find_packet: Variablendeklarationen 69a⟩
 for(i=0;i<mbuf_size;i++){
  ret = syscall4(SYS_recvmsg, &buffer, i, 10);
  if(ret == -1){
   *ptr = 0;
  } else {
   if(buffer[9] == search_proto) {
    ⟨find_packet: IP Buffersize ermitteln 69b⟩
    memcpy(&ip_packet, &buffer[0], buffer_size);
    ip_packet.DESTINATION_IPADDRESS =
              bswap32(ip_packet.DESTINATION_IPADDRESS);
    inet_iton(ip_packet.DESTINATION_IPADDRESS, dst_ip);
    if((strcmp(search_ip, dst_ip) == 1) &&
       (search_proto == ip_packet.PROTOCOL)) {
     *ptr = ip_packet.PROTOCOL;
    }
   }
  }
  ptr++;
 }
}
```

Defines:
find_packet, used in chunks 65 and 77c.
Uses buffer_size 71b 74b 81b, dst_ip 69a, i 41a 59a 63b 67a 69a 74b 79a, inet_iton 61b, ip_packet 67a 69a 79b, memcpy 81b, ret 67a 69a 74b 79a, and SYS_recvmsg 26b 47c.

Die `for`-Schleife von `find_packet` iteriert den gesamten Kernel-Buffer. Sollte der Zugriff auf den Kernel-Buffer nicht funktionieren, wird die IP-Protokollnummer auf 0 gestellt. Dieser Fall kann eintreten, wenn der aktuell Index `ipv4_mbuf_index` verwendet wird. Dann wird der Zugriff auf

das Paket verhindert. Danach folgt die Prüfung ob das gesuchte Protokoll `search_proto` mit dem IP-Paket übereinstimmt. Ist dies der Fall, wird der `buffer` in das `ip_packet` kopiert. Es folgt die Byteorder-Änderung und das Umwandeln der Dezimalzahl in einen String durch `inet_iton`. Jetzt kann die gesuchte IP-Adresse `search_ip` mit der IP-Adresse aus dem Paket verglichen werden. Stimmen die Werte überein, dann wird die IP-Protokollnummer in das Array `packets` gespeichert. Am Schluss jeder Iteration des Indexes, wird der `packets` Array-Zeiger auf das nächste Feld verschoben.

Die eigene Host-IP wird in `dst_ip` gespeichert. Die Variable `buffer_size` wird zur Berechnung der Gesamtlänge des IP-Paketes genutzt. Im Array `buffer` wird das IP-Paket aus dem Kernel geladen und gespeichert. Die `IP`-Struktur `ip_packet` wird noch benötigt. Die Variablen `ret`, `i` und `j` sind lauf Variablen.

69a ⟨`find_packet`: *Variablendeklarationen* 69a⟩≡ (68)

```
char *dst_ip[15];
u_char buffer_size;
u_char buffer[255];
IP ip_packet;
short ret;
int i,j;
```

Defines:
`dst_ip`, used in chunk 68.
`i`, used in chunks 34, 38b, 41b, 43b, 58–61, 63, 65, 67–69, 75b, 77c, 82a, and 83.
`ip_packet`, used in chunks 49, 64, 65, 67b, 68, 70–73, 77c, and 80–83.
`j`, used in chunks 41b, 63, and 65.
`ret`, used in chunks 65, 68, 69b, 72b, 74a, 75a, 77c, 81b, and 82b.
Uses `buffer_size` 71b 74b 81b and `u_char` 26b 47c.

Die Größe zum Speichern des IP-Paketes aus dem Kernel `ipv4_mbuf` muss zuerst berechnet werden, um nicht zu viele oder zu wenige Informationen zu erhalten. Um die Größe des IP-Paketes zu ermitteln, wird die IHL aus dem Paket extrahiert. Danach enthält `buffer_size` die genaue Größe des IP-Paketes und kann jetzt das gesamte IP-Paket aus dem Kernel kopieren.

69b ⟨`find_packet`: *IP Buffersize ermitteln* 69b⟩≡ (68)

```
buffer_size = ((buffer[0] & 0xF) * 4);
ret = syscall4(SYS_recvmsg, &buffer, i, buffer_size);
if(ret == -1){
  break;
}
```

Uses `buffer_size` 71b 74b 81b, `i` 41a 59a 63b 67a 69a 74b 79a, `ret` 67a 69a 74b 79a, and `SYS_recvmsg` 26b 47c.

Nachdem das Array `packets` alle aktuellen IP-Protokollnummern des Kernel-Buffers kennt, müssen die Pakete noch analysiert werden. Dazu wird nach

ICMP-Requestpaketen gesucht, die als Ziel-IP-Adresse die eigene IP-Adresse des Hosts haben. Die Suchparameter ICMP `type`, ICMP `code`, ICMP `id` und ICMP `seq` sind notwendig, um das richtige ICMP-Responsepakete. Im `rvalue` wird gespeichert, ob der Kopiervorgang erfolgreich war oder nicht. Sollte die Funktion `get_icmp_packet` nicht erfolgreich sein, wird dies gleich wieder an die Funktion `ipd` zurückgegeben. Bei Erfolg wird das ICMP-Paket auf die Suchkriterien geprüft (⟨`find_icmp_packet`*: Prüfen des ICMP Paketes* 70b⟩).

70a ⟨*Implementierung von* `find_icmp_packet` 70a⟩≡ (51b)

```
short find_icmp_packet(short index, short type, short code,
                       short id, short seq, IP *ip_packet,
                       ICMP *icmp_packet, char *timearray){
  short rvalue = 0;
  rvalue = get_icmp_packet(index, ip_packet, icmp_packet,
                           timearray);
  if(rvalue == -1){
    return rvalue;
  }
  ⟨find_icmp_packet: Prüfen des ICMP Paketes 70b⟩
  return rvalue;
}
```

Defines:
`find_icmp_packet`, used in chunks 65 and 77c.
Uses `get_icmp_packet` 49b 71a, `icmp_packet` 67a, `ip_packet` 67a 69a 79b, and `rvalue` 63b 71b.

Die Prüfung des ICMP-Paketes ist relativ einfach. Es muss nur geprüft werden, ob die 4 ICMP-Felder (ID, Sequenznummer, Code, Type) übereinstimmen mit den Suchparametern. Der `ipd` sucht nur nach ICMP-Requestpaketen die den Type `ICMP_REQ_TYPE` und Code `ICMP_REQ_CODE` haben.

70b ⟨`find_icmp_packet`*: Prüfen des ICMP Paketes* 70b⟩≡ (70a)

```
if((type == icmp_packet->TYPE) &&
  (code == icmp_packet->CODE)){
  if((id == -1) && (seq == -1)) {
    rvalue = 1;
  } else {
    if((id == icmp_packet->ID) &&
      (seq == icmp_packet->SEQ)){
      rvalue = 1;
    }
  }
}
```

Uses `icmp_packet` 67a and `rvalue` 63b 71b.

Innerhalb der Funktion find_icmp_packet wird die Funktion get_icmp_packet aufgerufen. Die Funktion get_icmp_packet hat die Aufgabe das IP/ICMP-Paket aus dem Kernel zu kopieren. Dafür ist der Index des IP-Pakets von Array ipv4_mbuf notwendig. Dieser index wurde der find_icmp_packet schon übergeben und wird einfach weitergereicht. Ebenso weitergereicht werden die IP- und ICMP-Strukturzeiger und das Array timearray für den Systemtick . Nach der Variablendeklaration werden zuerst 4 Bytes aus dem Array ipv4_mbuf geholt. Danach folgt wieder die Ermittlung der Größe des Paketes ⟨get_icmp_packet: *ICMP Buffersize ermitteln* 72a⟩. Nun müssen nur noch das IP-, ICMP-Paket in die Strukturen ip_packet und icmp_packet kopiert werden.

71a ⟨*Implementierung von* get_icmp_packet 71a⟩≡ (51b)

```
short get_icmp_packet(short index, IP *ip_packet,
                      ICMP *icmp_packet, char *timearray){
  ⟨get_icmp_packet: Variablendeklarationen 71b⟩
  rvalue = syscall4(SYS_recvmsg, &buffer, index, 4);
  if(rvalue == -1){
    return rvalue;
  }
  ⟨get_icmp_packet: ICMP Buffersize ermitteln 72a⟩
  memcpy(ip_packet, &buffer[0], iplen);
  memcpy(icmp_packet, &buffer[iplen], icmplen);
  memcpy(timearray, &buffer[buffer_size], 4);
  return rvalue;
}
```

Defines:
get_icmp_packet, used in chunk 70a.
Uses buffer_size 71b 74b 81b, icmp_packet 67a, icmplen 73, ip_packet 67a 69a 79b, iplen 73, memcpy 81b, rvalue 63b 71b, and SYS_recvmsg 26b 47c.

Zu den Variablen der Funktion get_icmp_packet im Überblick. Der Wert von IP-Header-Length wird in die Variable iplen und der ICMP-Header-Length in Variable icmplen gespeichert. Die Variable buffer_size speichert die Gesamtgröße des IP-Pakets. Im Array buffer selbst wird wieder das IP/ICMP-Paket zwischen gespeichert.

71b ⟨get_icmp_packet: *Variablendeklarationen* 71b⟩≡ (71a)

```
int rvalue = 0;
u_char iplen;
u_short icmplen;
int buffer_size;
u_char buffer[255];
```

Defines:
buffer_size, used in chunks 68, 69, 71–73, 75a, and 83.
rvalue, used in chunks 61a, 63, and 70–72.

Uses icmplen 73, iplen 73, u_char 26b 47c, and u_short 26b 47c.

Es folgt die Berechnung der Paketgrößen. Zuerst wird die Gesamtlänge des IP-Paketes ermitteln. Die Länge des ICMP-Paketes ergibt sich aus der IP-Total-Length minus der IP-Header-Length. Sobald die Länge errechnet wurde, wird das gesamte Paket kopiert.

72a ⟨get_icmp_packet: *ICMP Buffersize ermitteln* 72a⟩≡ (71a)

```
buffer_size = buffer[2];
buffer_size = (buffer_size<<8) + buffer[3];
iplen = ((buffer[0] & 0xF) * 4);
icmplen = (buffer_size - iplen);

rvalue = syscall4(SYS_recvmsg, &buffer, index,
                  (buffer_size + 4));
if(rvalue == -1){
  return rvalue;
}
```

Uses buffer_size 71b 74b 81b, icmplen 73, iplen 73, rvalue 63b 71b, and SYS_recvmsg 26b 47c.

Es folgt die Versendung des ICMP-Antwortpaketes. Die Funktion icmp_response erstellt anhand des ICMP-Requestpaketes ein Antwortpaket. Dazu werden die IP ip_packet und ICMP icmp_packet Strukturen übergeben. Beide Strukturen enthalten schon die Daten der gefundenen IP/ICMP-Pakete. Vor der Erstellung des Antwortpaketes wird die ursprüngliche Quell-IP-Adresse noch temporär zwischengespeichert. Nach der ⟨icmp_response: *Erstellen des Antwortpaketes* 73⟩ wird der Versand des Paketes eingeleitet. Dazu werden die Strukturen ip_packet und icmp_packet über das Buffer Array kopiert und versendet.

72b ⟨*Implementierung von* icmp_response 72b⟩≡ (51b)

```
void icmp_response(IP *ip_packet, ICMP *icmp_packet){
  u_int tmp;
  tmp = ip_packet->SOURCE_IPADDRESS;
  int ret = 0;
  ⟨icmp_response: Erstellen des Antwortpaketes 73⟩
  memcpy(&buffer, ip_packet, iplen);
  memcpy(&buffer[iplen], icmp_packet, icmplen);
  ret = syscall3(SYS_sendmsg, &buffer, buffer_size);
}
```

Defines:
icmp_response, used in chunks 49b and 67b.

Uses buffer_size 71b 74b 81b, icmp_packet 67a, icmplen 73, ip_packet 67a 69a 79b, iplen 73, memcpy 81b, ret 67a 69a 74b 79a, SYS_sendmsg 26b 47c, tmp 63b, and u_int 26b 47c.

Bei dem Erzeugen des Antwortpaketes muss im Prinzip nur die Absender- und die Empfänger-IP getauscht werden und die Felder ICMP-Code und Type umgeändert werden. Zusätzlich muss aber noch die Prüfsumme neu errechnet werden. Bei dem Feld `ip_packet->TOTAL_LENGTH` muss die Byteorder vor und nach dem Berechnen der Prüfsumme getauscht werden. Alle anderen Felder die größer als ein Byte sind, werden nicht der Byteorderänderung unterzogen, da diese schon in der "network byte order" vorliegen.

73 ⟨`icmp_response`: *Erstellen des Antwortpaketes* 73⟩≡ (72b)

```
ip_packet->SOURCE_IPADDRESS =
            ip_packet->DESTINATION_IPADDRESS;
ip_packet->DESTINATION_IPADDRESS = tmp;

ip_packet->TOTAL_LENGTH = bswap16(
                          ip_packet->TOTAL_LENGTH);
u_short buffer_size = ip_packet->TOTAL_LENGTH;
u_char buffer[buffer_size];

icmp_packet->TYPE = ICMP_REPLY_TYPE;
icmp_packet->CODE = ICMP_REPLY_CODE;

ip_packet->HEADER_CHECKSUM = 0;
icmp_packet->CHECKSUM = 0;
int iplen = (ip_packet->HEADER_LENGTH * 4);
int icmplen = (ip_packet->TOTAL_LENGTH - iplen);
ip_packet->TOTAL_LENGTH = bswap16(ip_packet->TOTAL_LENGTH);
ip_packet->HEADER_CHECKSUM = checksum(iplen, ip_packet);
icmp_packet->CHECKSUM = checksum(icmplen, icmp_packet);
```

Defines:
`icmplen`, used in chunks 64, 71, and 72.
`iplen`, used in chunks 71 and 72.
Uses `buffer_size` 71b 74b 81b, `icmp_packet` 67a, `ICMP_REPLY_CODE` 45, `ICMP_REPLY_TYPE` 45, `ip_packet` 67a 69a 79b, `tmp` 63b, `u_char` 26b 47c, and `u_short` 26b 47c.

Nach dem Versand des ICMP-Antwortpakets erfolgt die Löschung von nicht mehr benötigten oder ungültigen IP-Paketen aus dem Kernel-Buffer. Diese Aufgabe hat die Funktion `remove_unused_packets`. Es werden zum Einen alle Pakete gelöscht die älter als 5 Sekunden sind. Dieser Zeitwert wird mit der `max_ip_timeout` Variable initialisiert. Innerhalb dieser Zeit sollte der jeweilige Prozess der ein Paket abfragt, dieses auch gelöscht haben. Zum Zweiten werden Pakete gelöscht die nicht dem IPv4-Protokoll entsprechen.

Als Erstes wird die ⟨`remove_unused_packets`: *Variablendeklarationen* 74b⟩ aufgerufen. Die IP-Version wird in der Variable `iptype` gespeichert und dann

überprüft. Sollte keine IP-Version vorliegen, wird abgebrochen und die Funktion beendet. In dem Fall geht das Programm davon aus das, dass kein Paket im Speicher vorliegt. Bei IP-Version 4 muss zuerst wieder das gesamte Paket geladen werden um den Wert des Systemticks zu erhalten. Danach kann der Timeout errechnet werden, und falls dieser erreicht oder überschritten wird, wird das IP-Paket gelöscht. Es gibt noch keine Zeitberechnungsfunktion wie `ftime` oder ähnliche in der vorliegen Ulix Version 0.8. Das Timeout kann aber mittels Systemticks berechnet werden.

74a ⟨*Implementierung von* `remove_unused_packets` 74a⟩≡ (51b)

```
void remove_unused_packets(short index){
  ⟨remove_unused_packets: Variablendeklarationen 74b⟩
  ret = syscall4(SYS_recvmsg, &buffer, index, 4);
  if(ret == -1){
    return;
  }
  u_char iptype = (buffer[0]>>4);
  switch (iptype){
    case 0:
     break;
    case 4:
     ⟨remove_unused_packets: IP-Packetssize ermitteln 75a⟩
     ⟨remove_unused_packets: Systemtick zurück berechnen 75b⟩
     ⟨remove_unused_packets: IP-Paket timeout überprüfen 76a⟩
     break;
    default:
     remove_packet(index);
     break;
  }
}
```

Defines:
`remove_unused_packets`, used in chunks 49b and 65.
Uses `remove_packet` 76b, `ret` 67a 69a 74b 79a, `SYS_recvmsg` 26b 47c, and `u_char` 26b 47c.

Zu den Variablen der Funktion `remove_unused_packets`. Das Timeout wird mit der `max_ip_timeout` auf die gewünschte Anzahl der Systemticks gestellt. Die Variable `packet_time` speichert den Systemtick des IP Pakets zur Ankunftszeit und ist im `ipv4_mbuf` Array für jedes Paket gespeichert. Der aktuelle Systemtick wird in `measure_time` gespeichert. Die Total-Length der IP-Pakete wird in `iptl` gespeichert. Die Variablen `buffer` und `buffer_size` werden zum kopieren der IP-Pakete genutzt.

74b ⟨`remove_unused_packets`: *Variablendeklarationen* 74b⟩≡ (74a)

```
int max_ip_timeout = 1000;              // ~ 5 Sekunden
int packet_time;
```

```
int measure_time;
u_char iptl;
int buffer_size;
u_char buffer[255];
short ret;
int i;
```

Defines:
buffer_size, used in chunks 68, 69, 71–73, 75a, and 83.
i, used in chunks 34, 38b, 41b, 43b, 58–61, 63, 65, 67–69, 75b, 77c, 82a, and 83.
max_ip_timeout, used in chunk 76a.
measure_time, used in chunks 76a and 82b.
packet_time, used in chunks 75–77 and 82a.
ret, used in chunks 65, 68, 69b, 72b, 74a, 75a, 77c, 81b, and 82b.
Uses u_char 26b 47c.

Um den Systemtick des IP-Paketes zu erhalten, muss das Programm wissen, an welcher Stelle im ipv4_mbuf Array sich die Systemticks befinden. Der Systemtick beim Empfang eines Pakets wird in der Funktion slip_recv direkt nach dem IP-Paket gespeichert und belegt 4 Bytes. Deshalb wird die buffer_size noch zusätzlich um 4 Bytes erweitert zum Wert von Total-Length. Die Berechnung der IP-Paketgröße erfolgt wieder über die Total-Length.

75a ⟨remove_unused_packets: *IP-Packetssize ermitteln* 75a⟩≡ (74a)

```
iptl = buffer[2];
iptl<<8;
iptl += buffer[3];
buffer_size = iptl + 4;
ret = syscall4(SYS_recvmsg, &buffer, index, buffer_size);
if(ret == -1){
  break;
}
```

Uses buffer_size 71b 74b 81b, ret 67a 69a 74b 79a, and SYS_recvmsg 26b 47c.

Die Berechnung des Systemticks vom IP-Paket ist recht einfach. Der Systemtick wurde mittels Bitshift in das ipv4_mbuf Array geschrieben. Jetzt muss dieser Vorgang nur umgekehrt werden.

75b ⟨remove_unused_packets: *Systemtick zurück berechnen* 75b⟩≡ (74a)

```
packet_time = 0;
for(i=0;i<4;i++){
  packet_time = packet_time<<8;
  packet_time += buffer[iptl+i];
}
```

Uses i 41a 59a 63b 67a 69a 74b 79a and packet_time 74b 79a.

Zur Prüfung des Timeouts muss der aktuelle Systemtick vom Kernel bezogen werden. Danach kann über eine Subtraktion der Messzeit und Paketzeit, verglichen werden ob die Differenz größer ist als der maximale Timeoutwert. Ist dies nicht der Fall, wird die Funktion beendet. Sollte die Paketzeit 0 sein, so wird ebenfalls abgebrochen. In diesem Fall könnte die Speicherung des Systemticks im Kernel noch nicht ganz vollzogen sein.

76a ⟨remove_unused_packets: *IP-Paket timeout überprüfen* 76a⟩≡ (74a)

```
measure_time = syscall1(SYS_gettick);
if((measure_time-packet_time) > max_ip_timeout){
  if(packet_time == 0){
    break;
  }
#ifdef DEBUG
  printf("\nMessung: %d, Paket:%d, Diff:%d \n",
        measure_time, packet_time,
        (measure_time-packet_time) );
#endif
  remove_packet(index);
}
```

Uses DEBUG 26b 47c, max_ip_timeout 74b, measure_time 74b 79a, packet_time 74b 79a, remove_packet 76b, and SYS_gettick 26b 47c.

Das Entfernen eines IP-Pakets aus dem Kernel wird per Syscall erreicht. Dazu muss nur der aktuelle Indexwert des IP-Pakets dem Syscall SYS_delmbuf übergeben werden.

76b ⟨*Implementierung von* remove_packet 76b⟩≡ (51b)

```
short remove_packet(short index){
  return syscall2(SYS_delmbuf,index);
}
```

Defines:
remove_packet, used in chunks 49b, 67b, 74a, 76a, and 82a.
Uses SYS_delmbuf 26b 47c.

7.5 Das Ping-Programm

Das Ping-Programm dient im wesentlichen dem Test, ob ein Computer oder ein System unter der IP-Adresse erreichbar ist. Für das Ping-Programm werden zwei verschiedene ICMP-Pakettypen gebraucht. Ein ICMP-Request- und ein ICMP-Response-Paket. Das ICMP-Request-Paket wird vom Sender erstellt und an das Zielsystem versendet. Das ICMP-Response wird wiederum

vom Empfänger erzeugt und an den Sender zurückgesendet, sobald ein ICMP-Request vorliegt. Das Ping-Programm wird als einfaches Kommando in die Ulix-Shell `sh.c` als Builtin-Kommando hinzugefügt.

77a ⟨*Ulix sh.c Zeile 11: Shell Kommandos* 77a⟩≡

```
#define SHELL_COMMANDS "help, ps, fork, ls, cat, head, cp,
                        diff, sh, hexdump, kill, loop,
                        test, brk, cd, ln, rm, pwd, touch,
                        read, edit, exit, hostname, ipd,
                        ping"
```

Defines:
SHELL_COMMANDS, never used.
Uses hostname 47c and ipd 65.

Damit der Aufruf der Funktion `cmd_ping` aus der Shell klappt, muss die `switch` Fallunterscheidung der Variable `cmd` in der Funktion `run_command` noch um die folgenden drei Codezeilen erweitert werden:

77b ⟨*Ulix sh.c Zeile 693: Shell Kommandos* 77b⟩≡

```
    else if ( strcmp ((char*)cmd, "ping") ) {
          cmd_ping(argv[1]);
    }
```

Uses cmd_ping 77c.

Die nachfolgende Funktion `cmd_ping` nimmt die IP-Adresse des Empfängersystems als Parameter `destination_ip` entgegen. Sollte die IP-Adresse leer sein, wird die Funktion mit einem Fehlerhinweis auf der Konsole beendet. Auch bei einer ungültigen IP-Adresse bricht die Funktion ab. Danach folgen die Variablendeklaration und die Erstellung der IP- und ICMP-Strukturen. Sobald die Strukturen erstellt und befüllt sind, wird das ICMP-Paket versendet. Nach dem Versand des Paketes über den Kernel, wird in der `while` Schleife das ICMP-Responsepaket abgefragt, welches zum versendeten ICMP-Request passt. Und falls ein ICMP-Responsepaket vorliegt, wird die Zeitdifferenz von Versand und Empfang angezeigt. Falls kein Paket innerhalb des eingestellten Timeoutwertes vorliegt, soll eine Meldung auf der Textkonsole angezeigt werden.

77c ⟨*Implementierung eines ping Programms* 77c⟩≡ (51b)

```
int cmd_ping(char *destination_ip){
  if(strcmp(&destination_ip, EMPTY) == 1){
    printf("Please usage: ping [ipv4 address]\n");
    return -1;
  }
  if(check_ipaddr(destination_ip) != 0){
    printf("%s is not a valid IPv4 address\n",
```

```
            destination_ip);
      return -1;
    }
    ⟨cmd_ping: Variablendeklaration 79a⟩
    ⟨cmd_ping: IP Paketstruktur erstellen 79b⟩
    ⟨cmd_ping: ICMP Paketstruktur erstellen 80⟩
    ⟨cmd_ping: Debug Ausgabe Paketstruktur 82c⟩
    ⟨cmd_ping: IP/ICMP Prüfsummen berechnen 81a⟩
    ⟨cmd_ping: ICMP Paketversenden 81b⟩
    ⟨cmd_ping: Debug Ausgabe nach dem Versand 83⟩

    printf("\nPING %s: 1 data byte at tick %d: \n",
           destination_ip, my_time);
    ret = 0;
    while(ret == 0){
     find_packet(mbuf_size, &src_ip_addr, IP_PROTOCOL_ICMP,
                 &packets);
      for(i=0;i<mbuf_size;i++){
       if(packets[i] == IP_PROTOCOL_ICMP){
        ret = find_icmp_packet(i, ICMP_REPLY_TYPE,
                               ICMP_REPLY_CODE, icmp.ID,
                               icmp.SEQ, &ip_packet, &icmp,
                               &timearray);
        if(ret == 1){
         ⟨cmd_ping: ICMP Antwortpaket prüfen 82a⟩
         printf("1 byte from %s: seq=%d ttl=%d ticks=%d\n",
                destination_ip, icmp.SEQ,
                ip_packet.TIME_TO_LIVE,
                (packet_time-my_time));
         return 0;
        }
       }
      }

          ⟨cmd_ping: Timeout 82b⟩
    }
    return 0;
  }
```

Defines:
cmd_ping, used in chunks 22c, 49b, and 77b.
Uses check_ipaddr 54, EMPTY 47c, find_icmp_packet 49b 70a, find_packet 49b 68, i 41a 59a 63b 67a 69a 74b 79a, icmp 80, ICMP_REPLY_CODE 45, ICMP_REPLY_TYPE 45, ip_packet 67a 69a 79b, IP_PROTOCOL_ICMP 47b, my_time 79a, packet_time 74b 79a, ret 67a 69a 74b 79a, and src_ip_addr 79a.

Die `while`-Schleife sucht bei jedem Durchlauf die Pakete vom Kernel nach passenden ICMP-Response-Paketen ab. Sollte ein ICMP-Paket gefunden werden, wird das Paket aus dem Kernel kopiert. Danach folgen die Auswertung des ICMP Paketes und die Ausgabe des Zeitunterschiedes von ICMP-Request und ICMP-Response auf der Konsole.

Die Variablen `packet_time` und `measure_time` werden zur Zeitmessung genutzt. Der ICMP `timeout` wird initialisiert mit dem Wert 3 und definiert damit 3 Sekunden. Die Variable `mbuf_size` speichert die Größe des Arrays `ipv4_mbuf` aus dem Kernel. Das Array `packets` speichert die Protokoll-ID vom IP-Paket. Das `timearray` enthält den Systemtick vom IP-Paket aus dem Array `ipv4_mbuf`. In die Variable `my_time` wird der aktuelle Systemtick zur Zeitmessung gespeichert. Die eigene IP-Adresse vom System wird in das Array `src_ip_addr` gespeichert.

79a ⟨cmd_ping: *Variablendeklaration* 79a⟩≡ (77c)

```
int packet_time, measure_time, my_time, i, x;
int timeout = 3;
int ret = 0;
char src_ip_addr[16];
u_short mbuf_size;
mbuf_size = syscall1(SYS_mbufsize);
u_char packets[mbuf_size];
u_char timearray[4];
memset(&packets, 0, mbuf_size);
```

Defines:
i, used in chunks 34, 38b, 41b, 43b, 58–61, 63, 65, 67–69, 75b, 77c, 82a, and 83.
measure_time, used in chunks 76a and 82b.
my_time, used in chunks 77c, 81b, and 82b.
packet_time, used in chunks 75–77 and 82a.
ret, used in chunks 65, 68, 69b, 72b, 74a, 75a, 77c, 81b, and 82b.
src_ip_addr, used in chunks 77c, 79b, and 82c.
timeout, used in chunk 82b.
x, used in chunk 82a.
Uses SYS_mbufsize 26b 47c, u_char 26b 47c, and u_short 26b 47c.

Zum Erstellen des IP-Pakets für den ICMP-Request wird die Struktur `IP` genutzt. Dann werden die IP-Headerfelder befüllt. Die IP-Adresse vom Absender wird anhand des eigenen Hostnamens ermittelt und in `src_ip_addr` gespeichert.

79b ⟨cmd_ping: *IP Paketstruktur erstellen* 79b⟩≡ (77c)

```
IP ip_packet;
ip_packet.VERSION = IP_VERSION;
ip_packet.TOS = 0;
ip_packet.IDENTIFICATION = 0;
ip_packet.FLAGS = 0;
```

```
ip_packet.FRAGMENT_OFFSET = 0;
ip_packet.TIME_TO_LIVE = 1;
ip_packet.PROTOCOL = IP_PROTOCOL_ICMP;
ip_packet.HEADER_CHECKSUM = 0;
ip_packet.OPTIONS_AND_PADDING = 0;
get_ip_by_name (&src_ip_addr, &hostname);
ip_packet.SOURCE_IPADDRESS = inet_atoi(&src_ip_addr);
ip_packet.DESTINATION_IPADDRESS = inet_atoi(destination_ip);
ip_packet.HEADER_LENGTH = calc_ip_hl(&ip_packet);
ip_packet.VERSION = IP_VERSION;
ip_packet.TOS = 0;
ip_packet.IDENTIFICATION = 0;
ip_packet.FLAGS = 0;
ip_packet.FRAGMENT_OFFSET = 0;
ip_packet.TIME_TO_LIVE = 64;
ip_packet.PROTOCOL = IP_PROTOCOL_ICMP;
ip_packet.HEADER_CHECKSUM = 0;
ip_packet.OPTIONS_AND_PADDING = 0;
```

Defines:
ip_packet, used in chunks 49, 64, 65, 67b, 68, 70–73, 77c, and 80–83.
Uses get_ip_by_name 58, hostname 47c, IP_PROTOCOL_ICMP 47b, IP_VERSION 47b, and src_ip_addr 79a.

Es folgt die Erstellung des ICMP-Requestpakets. Das ICMP-Paket wird deklariert als `icmp` von der Struktur `ICMP`. Die ICMP `TYPE` und `CODE` Felder werden gesetzt mit den Informationen für ein ICMP-Requestpaketes. Die Variable `ID` und Sequenznummer `SEQ` werden fix vergeben. Da nur ein Paket versendet wird, ist die Erhöhung der Sequenznummer nicht notwendig. Die Information im ICMP-Paket selbst, besteht nur aus einem Buchstaben.

80 ⟨cmd_ping: *ICMP Paketstruktur erstellen* 80⟩≡ (77c)

```
ICMP icmp;
icmp.TYPE = ICMP_REQ_TYPE;
icmp.CODE = ICMP_REQ_CODE;
icmp.CHECKSUM = 0;
icmp.ID = 1;            // 12297 ID: 0x3009
icmp.SEQ = 100;
icmp.DATA = 65;
ip_packet.TOTAL_LENGTH = calc_ip_tl(ip_packet.HEADER_LENGTH,
                                    sizeof(icmp));
```

Defines:
icmp, used in chunks 77c and 81–83.
Uses ICMP_REQ_CODE 45, ICMP_REQ_TYPE 45, and ip_packet 67a 69a 79b.

Nachdem alle Felder der IP- und ICMP-Header befüllt sind, werden nachfolgend die Prüfsummen für die Header berechnet. Dann folgt die Änderung der Byteorder von Quell- und Ziel-IP-Adresse, sowie der Total-Length.

81a ⟨cmd_ping: *IP/ICMP Prüfsummen berechnen* 81a⟩≡ (77c)

```
ip_packet.SOURCE_IPADDRESS =
          bswap32(ip_packet.SOURCE_IPADDRESS);
ip_packet.DESTINATION_IPADDRESS =
          bswap32(ip_packet.DESTINATION_IPADDRESS);
ip_packet.TOTAL_LENGTH =
          bswap16(ip_packet.TOTAL_LENGTH);
ip_packet.HEADER_CHECKSUM = checksum(sizeof(ip_packet),
                                     &ip_packet);
icmp.CHECKSUM = checksum(sizeof(icmp), &icmp);
```

Uses icmp 80 and ip_packet 67a 69a 79b.

Nachdem das IP/ICMP-Paket erstellt wurde, wird das fertige ICMP-Requestpaket versendet. Falls beim Versenden des Pakets über den Syscall SYS_sendmsg ein Fehler auftritt, so wird eine Fehlermeldung auf der Konsole ausgegeben. Nach dem erfolgenden Versand wird nur noch er aktuelle Systemtick des Kernels für die später folgende Messung gespeichert.

81b ⟨cmd_ping: *ICMP Paketversenden* 81b⟩≡ (77c)

```
short buffer_size = (ip_packet.HEADER_LENGTH*4) +
                    sizeof(icmp);
u_char buf[buffer_size];
memcpy(&buf, &ip_packet, (ip_packet.HEADER_LENGTH*4));
memcpy(&buf[(ip_packet.HEADER_LENGTH*4)], &icmp,
            sizeof(icmp));

ret = syscall3(SYS_sendmsg, &buf, buffer_size);
if (ret != 0) {
  printf("Failed to send packet over SLIP");
  return -1;
}
my_time = syscall1(SYS_gettick);
```

Defines:
buffer_size, used in chunks 68, 69, 71–73, 75a, and 83.
memcpy, used in chunks 36b, 68, 71a, and 72b.
Uses buf 59a, icmp 80, ip_packet 67a 69a 79b, my_time 79a, ret 67a 69a 74b 79a, SYS_gettick 26b 47c, SYS_sendmsg 26b 47c, and u_char 26b 47c.

Es folgt der Ablauf für den Fall, dass ein ICMP-Responsepaket empfangen wurde. Der Systemtick des ICMP-Response-Paketes wird zurück berechnet und gespeichert. Sollte der Systemtickwert vom Paket 0 ergeben, so wird sofort

abgebrochen. Dann wurde vermutlich der Systemtick nicht richtig gespeichert. Ansonsten wird das empfangene Paket auf Übertragungsfehler geprüft und danach wieder aus dem Kernel-Buffer gelöscht. Die Funktion wird nach der Anzeige der Zeitdifferenz erfolgreich beendet.

82a ⟨cmd_ping: *ICMP Antwortpaket prüfen* 82a⟩≡ (77c)

```
 packet_time = 0;
 for(x=0;x<4;x++){
  packet_time = packet_time<<8;
  packet_time += timearray[x];
 }
 if(packet_time == 0){
  break;
 }
check_checksum(&ip_packet, &icmp);
remove_packet(i);
packets[i] = 0;
```

Uses check_checksum 64, i 41a 59a 63b 67a 69a 74b 79a, icmp 80, ip_packet 67a 69a 79b, packet_time 74b 79a, remove_packet 76b, and x 13b 79a.

Der Timeout beim Warten auf das ICMP-Responsepaket muss eingehalten werden. Dazu wird die Zeitmessung über die Differenz der Systemticks genutzt. Es soll auch eine Meldung auf der Textkonsole angezeigt werden, falls der Timeout eintritt.

82b ⟨cmd_ping: *Timeout* 82b⟩≡ (77c)

```
measure_time = syscall1(SYS_gettick);
if(((measure_time-my_time)/100) > timeout){
  printf("Timeout reached\n");
  printf("1 packets transmitted, 0 packets received, \
100.0%% packet loss\n");
  ret = 1;
}
```

Uses measure_time 74b 79a, my_time 79a, ret 67a 69a 74b 79a, SYS_gettick 26b 47c, and timeout 79a.

Wenn das Makro-DEBUG deklariert ist, so werden einige Informationen zu den IP- und ICMP-Header ausgegeben.

82c ⟨cmd_ping: *Debug Ausgabe Paketstruktur* 82c⟩≡ (77c)

```
#ifdef DEBUG
  printf("IP Version: %d\n", ip_packet.VERSION);
  printf("IP Header Laenge: %d\n", ip_packet.HEADER_LENGTH);
  printf("src_ip_addr: %s\n", src_ip_addr);
  printf("dst_ip_addr: %s\n", destination_ip);
```

```
    printf("IP Source: %u\n", ip_packet.SOURCE_IPADDRESS);
    printf("IP Dest: %u\n", ip_packet.DESTINATION_IPADDRESS);
    printf("OPT / PADDING: %u\n",
          ip_packet.OPTIONS_AND_PADDING);
    printf("IP Total Length: %d\n", ip_packet.TOTAL_LENGTH);
    printf("Buffersize IP (inc. pad): %d\n",
          sizeof(ip_packet));
    printf("ICMP Type: %u\n",icmp.TYPE);
    printf("ICMP Code: %u\n",icmp.CODE);
    printf("ICMP ID: %u\n",icmp.ID);
    printf("ICMP Seq: %u\n",icmp.SEQ);
    printf("ICMP Data: %u\n",icmp.DATA);
    printf("Buffersize ICMP: %d\n", sizeof(icmp));
  #endif
```

Uses DEBUG 26b 47c, icmp 80, ip_packet 67a 69a 79b, and src_ip_addr 79a.

Nach dem Versand des ICMP-Request-Paketes können die Bytes des Paketes angezeigt werden, falls das Makro-DEBUG deklariert ist.

83 ⟨cmd_ping: *Debug Ausgabe nach dem Versand* 83⟩≡ (77c)

```
  #ifdef DEBUG
    printf("IP ChkSum: %u\n",ip_packet.HEADER_CHECKSUM);
    printf("ICMP ChkSum: %u\n",icmp.CHECKSUM);
    printf("\nSend Buffer: ");
    for (i=0;i<buffer_size;i++){
      printf("%d ", buf[i]);
    }
    printf("\n");
  #endif
```

Uses buf 59a, buffer_size 71b 74b 81b, DEBUG 26b 47c, i 41a 59a 63b 67a 69a 74b 79a, icmp 80, and ip_packet 67a 69a 79b.

8 Tests

Um zu prüfen ob, das implementierte SLIP-Modul funktioniert, werden mehrere ICMP-"ping"-Tests erstellt. Es soll geprüft werden, ob die IP- und ICMP-Pakete richtig erstellt werden, und bei dem Zielsystem ankommen. Das Zielsystem wird die Pakete ablehnen, wenn die Pakete fehlerhaft sind. Sollten die IP/ICMP-Prüfsummen zum Beispiel nicht stimmen, wird das Zielsystem die Pakete verwerfen, ohne ein Antwort-Paket zu generieren. Es werden insgesamt zwei Tests gemacht. Der erste Test soll einen Ping von einem Ulix-System zu einem anderen Betriebssystem senden. Als Ergebnis wird erwartet, dass der ICMP-Request vom Zielsystem beantwortet wird und das SLIP-Modul in Ulix das Antwortpaket empfängt und auswertet. Beim Zweiten Test wird ein Ping von einem anderen Betriebssystem an Ulix gesendet. Damit wird der IP-Daemon vom SLIP-Modul getestet. Sobald der ICMP-Request beim Ulix-System ankommt, soll der IP-Daemon das Request-Paket analysieren und ein Response-Paket erzeugen. Das Response-Paket soll nur dann erzeugt werden, wenn das Request-Paket an die richtige IP-Adresse versendet wurde. Das Ergebnis bei zweiten Test soll sein, dass das Response-Paket vom Sender richtig empfangen wurde und dass das "ping"-Programm den erfolgreichen Empfang anzeigt. Zur Überprüfung der übertragenen Datenpakete werden die Programme "wireshark" [WIR14] und "tcpdump" verwendet. Als Test-IP-Adressen werden 192.168.0.1 für die Ulix-VM und 192.168.0.2 für die zweite VM genommen.

8.1 Ping Test aus der Ulix VM

Für den Ping-Test aus einem Ulix-System zu einem anderen System werden zwei Maschinen verwendet. Die erste Maschine ist das Ulix-System selbst. Die zweite Maschine ist ein weiteres Unix-System. In diesem Fall ein OpenBSD welches in einer VirtualBox VM läuft. Für die virtuelle serielle Leitung zwischen den beiden virtuellen Maschinen, dienen zwei Socket-Dateien welche, mit dem Programm "socat" verbunden werden. Zuerst wird das Ulix-System gestartet.

```
qemu-system-i386 -m 64 -fda ulixboot.img -fdb minixdata.img -
   serial file:/dev/null \
                -serial unix:/tmp/serial_machine1,server,nowait &
```

Listing 7: Start der Ulix VM

Die Einrichtung einer zweiten VM mittels VirtualBox wird nicht beschrieben. Es wird auf die Dokumentation von VirtualBox [VBOX14] für die Einrich-

tung einer VM verwiesen. Zum Installieren und Konfigurieren des OpenBSD Betriebssystems wird auch hier auf die Dokumentation [OBSD14] verwiesen. Die VirtualBox VM muss vor dem Start noch auf die Benutzung einer virtuellen seriellen Schnittstelle eingestellt werden. Dazu wird der VM der virtuelle serielle Port 1 aktiviert. Der Portmodus steht auf Host-Pipe, welcher als "uartmode1 server" definiert wird. Die Host-Pipe Datei muss angegeben werden und ist in dem Beispiel als Datei "`/tmp/serial_machine2`" definiert. Durch den Portmodus "Host-Pipe" wird auf dem Host eine Socket-Datei erzeugt, welche von beiden virtuellen Maschinen benutzt werden kann. Der Aufruf des Befehls für die Einstellung der seriellen Schnittstelle von der zweiten VM erfolgt über den Befehl: "`VBoxManage modifyvm machine2 --uartmode1 server /tmp/serial_machine2`".

Zum Starten der VirtualBox-VM reicht das folgende Kommando "VBoxSDL –startvm machine2" Die VM startet dann in einer grafischen Konsole. Es ist auch möglich, die VM ohne grafische Ausgabe und ohne jegliche Konsole zu starten.

Nachdem beide VMs laufen, müssen nur noch die beiden Unix-Socket-Dateien verbunden werden. Dazu wird das "socat" Programm mit dem Befehl "`socat unix-connect:/tmp/serial_machine1 unix:/tmp/serial_machine2`" genutzt.

In der OpenBSD VM wird die serielle Schnittstelle mit dem "`sl0`"-Interface verbunden. Danach wird die IP-Adresse der OpenBSD VM und die IP-Adresse der Ulix-VM auf dem Interface-"`sl0`" konfiguriert. Für die Einrichtung des Interfaces sind Root-Rechte notwendig.

```
ifconfig sl0 192.168.0.2 192.168.0.1 -link2
slattach tty00
```

Listing 8: Einstellen des SLIP Interfaces auf der OpenBSD-VM

In der Ulix-VM wird die eigene IP-Adresse angepasst. Der Hostname wird auf "`machine1`" gesetzt. Die IP vom Hostname "`machine1`" ist die 192.168.0.1, welche wiederum in der Datei "/etc/hosts" definiert ist. Danach folgt der Ping von der Ulix-VM zur OpenBSD-VM.

```
hostname machine1
ping 192.168.0.2
```

Listing 9: Test-Ping von Ulix

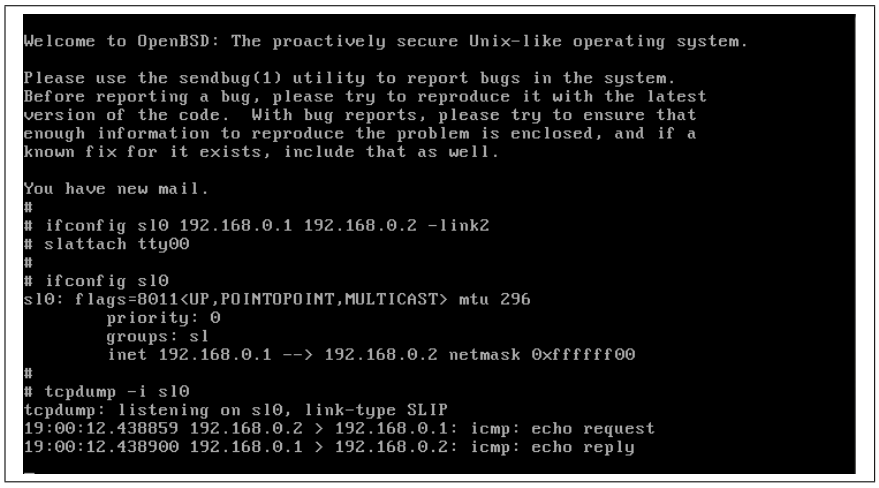

Abbildung 6: tcpdump des Ping-Tests auf der OpenBSD-VM

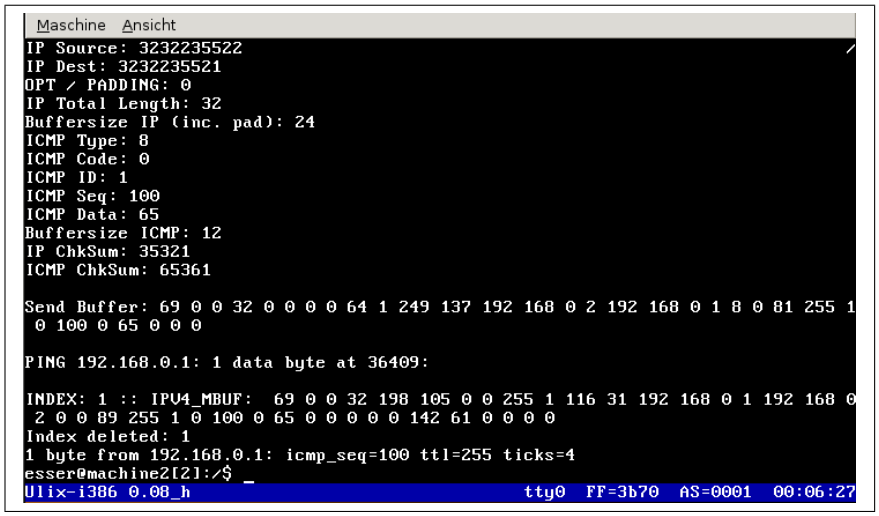

Abbildung 7: Ulix-Konsolenausgabe des Ping-Tests

Der Ping von der Ulix-VM an die OpenBSD-VM zeigt, dass das ICMP-Requestpaket übertragen und ein Response-Paket empfangen wurde. Auf der OpenBSD-VM wurde ein tcpdump gestartet, der auf dem Interface "sl0" lauscht. Wie in Abbildung 6 zu sehen ist, wurde auch hier ein ICMP-Request-Paket empfangen und danach das ICMP-Response-Paket an die Ulix-VM gesendet.

8.2 IP-Daemon Test

Für den IP-Daemon-Test können die selben VMs genutzt werden wie im ersten Test. Diesmal wird der Ping vom OpenBSD-System aus versendet. Als Ergebnis sollte die Ulix-VM das ICMP-Request-Paket erhalten und im Puffer speichern. Der IP-Daemon sollte daraufhin das ICMP-Request-Paket im Puffer finden und dann ein ICMP-Response-Paket versenden.

Für den Test muss in der Ulix-VM der IP-Daemon gestartet werden. Dies erfolgt über ein einfaches Kommando mittels "`ipd`". Der IPD wird die IP-Adresse vom Hostname festellen (Abbildung 8).

Um den Netzwerkverkehr im Nachhinein auswerten zu können, wird in der OpenBSD-VM ein Netzwerk-Trace aktiviert. Das Starten des Netzwerk-Traces erfolgt mit dem Befehl "`tcpdump -i sl0 -w /tmp/trace.pcap &`".

Beim Ping von der OpenBSD-VM muss die Paketegröße angepasst werden. Der Puffer in der Ulix-VM ist auf 40 Bytes eingestellt und, ein normales ICMP-Paket wird mit 56 Bytes versendet. Auch die Anzahl der zu testenden ICMP-Pakete wird auf 1 gestellt. Der Ping erfolgt mit dem Befehl "`ping -s 1 -c 1 192.168.0.1`"

Das ICMP-Response-Paket vom Ulix-System sollte innerhalb weniger Millisekunden erstellt und an die OpenBSD-VM gesendet werden. Nach dem Ping kann der Trace mit "`kill %1`" wieder abgeschalten werden. Die Analyse erfolgt durch das grafische Programm "wireshark" auf dem Hostsystem. Das Tracefile wird dazu aus der VM kopiert und anschließend in Wireshark geöffnet.

In Abbildung 11 kann man den Netzwerk-Trace im Wireshark sehen. Beide Pakete wurden erfolgreich übertragen.

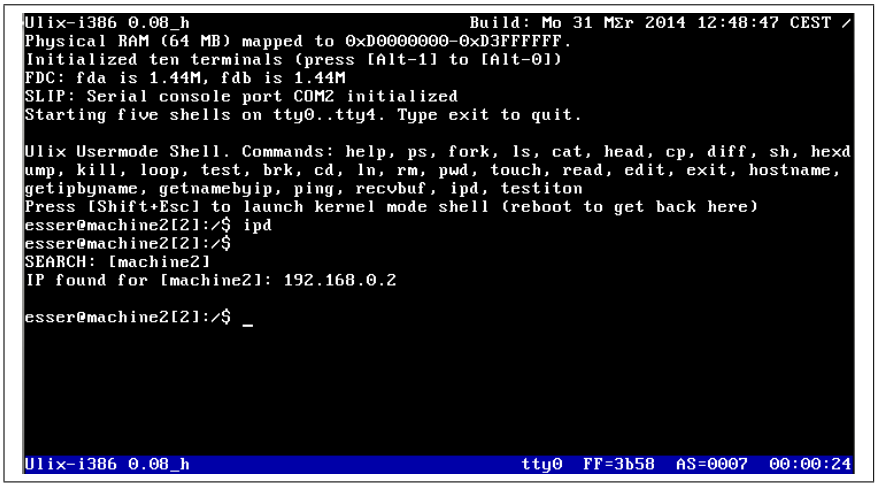

Abbildung 8: Konsolenausgabe beim Starten des IP-Daemons auf der Ulix-VM

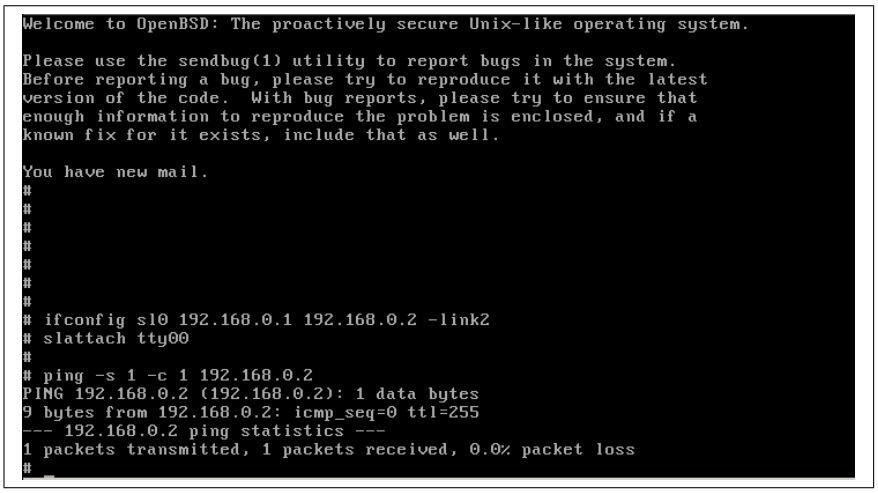

Abbildung 9: Konsolenausgabe des Ping-Tests in der OpenBSD-VM

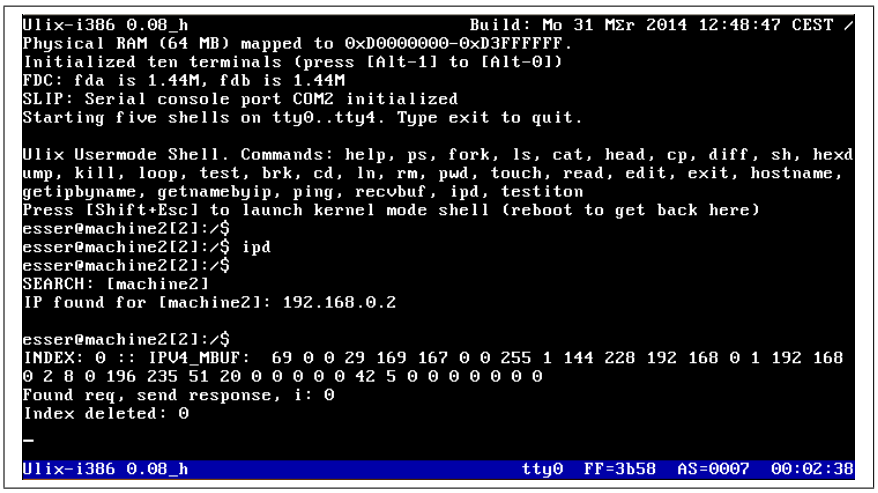

Abbildung 10: Konsolenausgabe des empfangenen ICMP-Paketes in der Ulix-VM

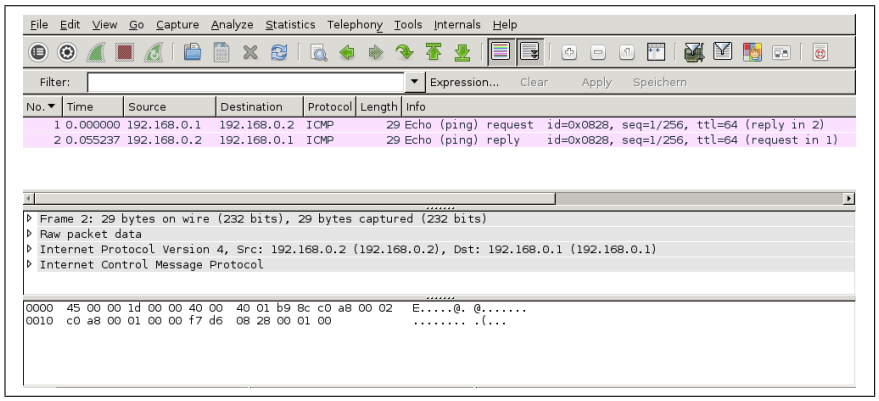

Abbildung 11: Anzeige des Paketmitschnitt in Wireshark

9 Fazit

Das SLIP-Protokoll wurde erfolgreich implementiert und getestet. Es zeigt sich, dass die Implementierung des IP-Protokolls einiges an Programmieraufwand bedarf. Von der Berechnung der Checksum bis hin zur Umwandlung der IP-Adressen von String auf Dezimal und umgekehrt. Zudem müssen die IP- und ICMP-Informationen bei Empfang ausgewertet werden. Das Empfangen und Senden von Datenpaketen über die serielle Schnittstelle ist mit wenig Programmieraufwand möglich. Mit der Literate-Programming-Methode ist die Implementierung des Programmcodes übersichtlich und dokumentiert. Durch die Beschreibung jeder einzelnen Funktion wird das Lesen des Quellcodes verständlicher. Die Zusammenhänge und die Motivation des Programmierers für jeden Programmabschnitt kann so nachvollzogen werden. Bei der Entwicklung des SLIP-Moduls zeigte sich, dass die Entwicklung von Software mit Literate-Programming am Anfang etwas Einarbeitung bedarf. Es kommt leicht vor, dass man in die eigenen gewohnten Muster der Entwicklungsmethoden zurück verfällt.

9.1 Kritik

Diese SLIP-Implementierung unterstützt nicht alle Features die im SLIP-Standard vorgesehen sind, denn der Fokus der Arbeit liegt auf einer verständlichen Erklärung der Prinzipien und nicht auf Vollständigkeit. Auch vom ICMP-Protokoll werden aus demselben Grund nur zwei Pakettypen unterstützt. Es fehlen ebenfalls viele Funktionen die das IP-Protokoll bietet. Weiterhin enthält die Implementierung keine Optimierungen für Performance und Sicherheit. Es werden übergebene Argumente in den Funktionen nicht immer vollständig geprüft. Einige Abschnitte der Arbeit konnten auch nicht vollständig mit der Literate-Programming-Methode umgesetzt werden.

9.2 Ausblick und Forschungsbedarf

Ob sich mit der Literate-Programming-Methode größere Softwareprojekte im Team bei Unternehmen umsetzten lassen, muss erst noch erforscht werden. Es ist denkbar, dass größere Softwareprojekte sich mit Literate-Programming umsetzten lassen. Software wie CDS und Jaba verwenden schon die Methoden der Literate-Programming. Auch die im Moment in Unternehmen verbreitete Agile-Softwareentwicklung [VMSG14] könnte durch die Literate-Programming-Methode verbessert oder erweitert werden. In der wissenschaftlichen Lehre der Softwareentwicklung stellen die Agilen-Methoden einen neuen Bereich dar [RISA09]. Die Prozessmethoden (wie z.B.: Scrum) haben das Ziel die Zusammenarbeit in Teams effektiver und effizienter zu gestalten. Die Anzahl der Wissenschaftlichenarbeiten zur Agile-Softwareentwicklung sind seit der Veröffentlichung (2001) des Agilen-Software-Manifestes weiter steigend [DNBM12]. Es ist denkbar, dass ein Prozessmodel entwickelt wird, welches die Möglichkeiten der Literate-Programming-Methode mit den Möglichkeiten der Agilen-Methoden kombiniert. Ob sich solch ein Model dann auch in der wissenschaftlichen Lehre durchsetzt, müsste noch erforscht werden.

9.3 Mögliche Aufgaben für Studenten

1. Bauen Sie die Funktionen compress_slip() und uncompress_slip() ein. Alle versendeten und empfangenen Bytes, die über die serielle Schnittstelle gehen, sollen nach dem RFC 1144 komprimiert werden. Wie müssen die IP-Headerinformationen komprimiert werden ? Und wie lassen sich die komprimierten Informationen wieder zusammensetzten ?

2. Bauen Sie die Funktion sanity_check() ein, welche alle übergebenen Parameter in den SLIP-Funktionen überprüft. Wie sollten sich die Funktionen verhalten, wenn unsinnige Argumente erkannt werden ?

3. Bauen Sie eine Möglichkeit für ein IP-Routing. Testen Sie das IP-Routing mit drei Ulix-Machinen. Wie lässt sich die Routingtabelle verwalten ? Welche IP-Headerfelder müssen verwendet werden um ein IP-Routing zu ermöglichen ?

4. Implementieren Sie das Programm-"netstat" und werten Sie nur die übertragenen Bytes aller IP-Pakete aus. Wie können die Informationen der IP-Pakete gesammelt und gemessen werden ?

5. Implementieren Sie das UDP-Protokoll und testen Sie es. Was sind die Eigenschaften des Protokolls ?

6. Implementieren Sie das TCP-Protokoll und testen Sie es. Speichern Sie die TCP-Verbindungen in einer Zustandstabelle. Wie lassen sich die TCP-Verbindungen umsetzten, und die Zustände der TCP-Verbindungen anzeigen ?

10 Lizenz

Als Lizenz für das SLIP-Modul wurde die BSD 4 Clause `licence` gewählt.

93 ⟨*license* 93⟩≡ (29b 51b)

```
/*
Copyright (c) C.Dornig

The Regents of the University of California.
All rights reserved.

Redistribution and use in source and binary forms, with
or without modification, are permitted provided that the
following conditions are met:

1. Redistributions of source code must retain the above
copyright notice, this list of conditions and
the following disclaimer.
2. Redistributions in binary form must reproduce the above
copyright notice, this list of conditions and the following
disclaimer in the documentation and/or other materials
provided with the distribution.
3. All advertising materials mentioning features or use
of this software must display the following
acknowledgement: "This product includes software developed
by the University of California, Berkeley and
its contributors."
4. Neither the name of the University nor the names
of its contributors may be used to endorse or promote
products derived from this software without specific prior
written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING,
BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS
BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE
GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

```
*/
```

Defines:
AND, never used.
and, never used.
BUSINESS, never used.
c, used in chunks 28, 34, 40c, 42, 43, and 58.
IF, never used.
IN, never used.
LIABLE, never used.
NOT, never used.
OR, never used.
Regents, never used.
rights, never used.
SOFTWARE, never used.
THEORY, never used.

11 Literatur

Literatur

[AGI14] Agile Manifesto - Manifesto for agile software development, URL: http://www.agilemanifesto.org/iso/de/, Zugriff: Mai 2014

[ASAAS04] Liakot Ali, Roslina Sidek, Ishak Aris, Alauddin Mohd. Ali, Bambang Sunaryo Suparjo, *Design of a micro-UART for SoC application* Computers and Electrical Engineering, Volume 30, 2004, S. 257-268

[BPH10] Richard Baskerville, Jan Pries-Heje, *Erklärende Designtheorie* Wirtschaftsinformatik, 5, 2010, S. 259-271

[COC01] Andy Cockburn, *Supporting tailorable program visualisation through literate programming and fisheye views* Information and Software Technology, 43, 2001, S. 745-758

[COO98] Barry M.Cook, *IEEE 1355 data-strobe links: ATM speed at RS232 cost* Microprocessors and Microsystems, 21, 1998, S. 421-428

[DIA14] Dia - GTK+ based diagram creation program for GNU/Linux, URL: https://wiki.gnome.org/Apps/Dia, Zugriff: März 2014

[DNBM12] Torgeir Dingsøyr, Sridhar Nerur, VenuGopal Balijepally, Nils Brede Moe, *A decade of agile methodologies: Towards explaining agile software development* The Journal of Systems and Software, 85, 2012, S. 1213 - 1221

[EßE13] Hans-Georg Eßer, *Design, Implementation, and Evaluation of the ULIX-i386 Teaching Operating System* University of Erlangen, 2013, S. 1.438

[EKRSV05] Erich Ehses, Lutz Köhler, Petra Riemer, Horst Stenzel, Frank Victor, *Betriebssysteme - Ein Lehrbuch mit Übungen zur Systemprogrammierung in UNIX/Linux* Pearson Studium, 2005

[FBSD14] FreeBSD, URL: www.freebsd.org, Zugriff: März 2014

[FPP14] FreeBSD - Packages and Ports, URL: http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/ports.html, Zugriff: Apr. 2014

[FTS14] FreeBSD - Quellcode TCP/IP Stack, URL: http://svnweb.freebsd.org/base/head/sys/netinet/, Zugriff: Apr. 2014

[FUN14] FunnelWEB, URL: http://www.ross.net/funnelweb/, Zugriff: April 2014

[GCC14] , GNU Compiler Collection URL: http://gcc.gnu.org/, Zugriff: März 2014

[ISO94] ISO/IEC, *Open System Interconnect - Basis Reference Model* ISO/IEC 7498:1:1994(E), 1994

[KJKAA04] Ajan Daniel Kutty, Bestin Jose, Ciju Rajan K, Daise Antony, Linto Antony, *Serial Line IP Implementation for Linux Kernel TCP/IP Stack* URL: www.cse.iitb.ac.in/~bestin/pdfs/slip.pdf, Zugriff: März 2014

[KNU83] Donald E. Knuth, *Literate Programming* Computers Journal, 1983, S. 1-15

[LAT14] TEX, LATEX, URL: www.latex-project.org, Zugriff: März 2014

[LIN14] Linux Kernel, URL: http://www.kernel.org, Zugriff: März 2014

[LWIP14] lwIP - A Lightweight TCP/IP stack, URL: http://savannah.nongnu.org/projects/lwip/, Zugriff: März 2014

[MJRRH03] P. Mähönen, J. Riihijärvi, O. Raivio, Pertti Huuskonen, *NanoIP: The Zen of Embedded Networking* URL: IEEEInternationalConferenceonCommunications,2003,Volume2,S.1218-1222

[NO14] Noweb - A Simple, Extensible Tool for Literate Programming, URL: http://www.cs.tufts.edu/~nr/noweb/, Zugriff: April 2014

[NSC95] National Semiconductor Corporation, *PC16550D Universal Asynchronous Receiver Transmitter with FIFOs* RRD-B30M75 Printed in USA

[NU14] The nuweb system for Literate Programming, URL: http://nuweb.sourceforge.net/, Zugriff: April 2014

[OBSD14] OpenBSD, URL: www.openbsd.org, Zugriff: Mai 2014

[PEP91] Peter Pepper, *Literate Program Derivation: A Case Study* Lecture Notes in Computer Science, Volume 544, 1991, S. 101-124

[QEMU14] , QEMU - generic and open source machine emulator and virtualizer URL: http://wiki.qemu.org/Main_Page, Zugriff: April 2014

[RISA09] David F. Rico, Hasan H. Sayani, *Use of agile methods in software engineering education* Agile Conference, 2009. AGILE 09

[RFC791] Internet Protocol URL: http://www.rfc-editor.org/rfc/rfc791.txt, Zugriff: März 2014

[RFC792] INTERNET CONTROL MESSAGE PROTOCOL URL: http://www.rfc-editor.org/rfc/rfc792.txt, Zugriff: März 2014

[RFC1055] A NONSTANDARD FOR TRANSMISSION OF IP DATAGRAMS OVER SERIAL LINES: SLIP URL: http://www.rfc-editor.org/rfc/rfc1055.txt, Zugriff: März 2014

[RFC1071] Computing the Internet Checksum URL: http://www.rfc-editor.org/rfc/rfc1071.txt, Zugriff: März 2014

[RFC1349] Type of Service in the Internet Protocol Suite URL: http://www.rfc-editor.org/rfc/rfc1349.txt, Zugriff: März 2014

[RFC1700] Assigned Numbers URL: http://www.rfc-editor.org/rfc/rfc1700.txt, Zugriff: März 2014

[RFC2460] Internet Protocol, Version 6 (IPv6) URL: http://www.rfc-editor.org/rfc/rfc2460.txt, Zugriff: März 2014

[SK01] Britta Schinzel, Karin Kleinn, *Quo vadis, Informatik?* Informatik Spektrum, Volume 24, 2001, S. 91-97

[SPO98] Robert Spotnitz, *Literate programming* Computers Chem. Engng, Volume 22, 1998, S. 1745-1747

[SWE14] Sweave Homepage, URL: http://www.stat.uni-muenchen.de/~leisch/Sweave/, Zugriff: April 2014

[THI01] Sebastien Li-Thiao-Te, *Literate Program Execution for Reproducible Research and Executable Papers* Procedia Computer Science, 9, 2012, S. 439-448

[TW97] Andrew S. Tanenbaum, Albert S. Woodhull, *Operating Systems - Design and Implementation* Prentice-Hall Internation, Inc, 1997

[UIP14] µIP - Micro IP, URL: http://www.sics.se/~adam/uip/, Zugriff: März 2014

[VBOX14] VirtualBox, URL: www.virtualbox.org, Zugriff: April 2014

[VMSG14] Raoul Vallon, Michael Müller-Wernhart, Wolfgang Schramm, Thomas Grechenig, *Kombination von Agil und Lean in der Softwareentwicklung* Informatik Spektrum, Volume 37, 2014, S. 28 - 35

[XV14] vx6 - a simple Unix-like teaching operating system, URL: http://pdos.csail.mit.edu/6.828/2012/xv6.html, Zugriff: April 2014

[YUN91] Donald E. Knuth, *Literate Programming System CDS* Journal of Computer Science & Technologie, Volume 6, Nr. 3, 1991, S. 263-270

[WIR14] Wireshark, URL: www.wireshark.org, Zugriff: April 2014

Versicherung (eidesstattlich)

Hiermit versichere ich, dass die vorliegende Arbeit von mir selbstständig und ohne unerlaubte Hilfe angefertigt worden ist, insbesondere dass ich alle Stellen, die wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen sind, durch Zitate als solche gekennzeichnet habe. Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Weiterhin erkläre ich, dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen hat. Ich erkläre mich damit einverstanden, dass die Arbeit der Öffentlichkeit zugänglich gemacht wird. Ich erkläre mich damit einverstanden, dass die Digitalversion dieser Arbeit zwecks Plagiatsprüfung auf die Server externer Anbieter hoch geladen werden darf. Die Plagiatsprüfung stellt keine Zurverfügungstellung für die Öffentlichkeit dar.

München, 04.07.2014
(Ort, Datum) (Eigenhändige Unterschrift)