Routing- und Protokollierungsdienste für Kleinstrechner

Bachelorarbeit

von

Yves Igor Jerschow

aus

Krefeld

vorgelegt am

Lehrstuhl für Rechnernetze
Prof. Dr. Martin Mauve
Heinrich-Heine-Universität Düsseldorf

August 2005

Betreuer:
Dipl.-Inf. Björn Scheuermann
Dipl. Wirtsch.-Inf. Christian Lochert

Danksagung

Mein besonderer Dank gilt Björn Scheuermann und Christian Lochert. Sie haben mich während der letzten drei Monate hervorragend betreut und halfen in zahlreichen Gesprächen, diese Bachelorarbeit zielstrebig voranzubringen. Vor allem möchte ich meinen beiden Betreuern für die kritischen Anregungen zur schriftlichen Arbeit danken. Sie motivierten mich, zahlreiche Abschnitte meiner Entwürfe zu überarbeiten, um eine objektive, gut strukturierte und sich auf das Wesentliche konzentrierende Endfassung vorlegen zu können. Björn Scheuermann und Christian Lochert hatten so gut wie immer für mich Zeit und und halfen auch bei zahlreichen Detailfragen weiter. Insbesondere wusste ich ihre Tipps zum Textsatzsystem LaTeX zu schätzen.

Ganz herzlich möchte ich mich sodann bei meiner Mutter bedanken. Sie gab mir während der gesamten Arbeitsphase Halt und stand stets mit Rat und Tat zur Seite.

Ein abschließender Dank geht an Prof. Dr. Martin Mauve, der mir den Anstoß dazu gab, die Bachelorarbeit an seinem Lehrstuhl zu verfassen, und sich über meine Fortschritte regelmäßig informierte.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Mobile Ad-Hoc-Netzwerke *(MANETs)* sind drahtlose Netzwerke aus frei beweglichen Endgeräten ohne jegliche Infrastruktur [32]. Die Funkreichweite der einzelnen Knoten ist stark begrenzt, daher werden Datenpakete in Kooperation mit den Nachbarknoten über mehrere Stationen zum eigentlichen Empfänger weitergeleitet *(Multihop-Kommunikation)*. Jeder Knoten agiert also gleichzeitig als Endsystem und als Router. Einsatzmöglichkeiten für solche selbstorganisierenden mobilen Systeme finden sich z. B. in der Fahrzeug-zu-Fahrzeug-Kommunikation, bei interaktiven mobilen Spielen oder in Sensornetzen (Abschnitt 2.1). Leider treten in MANETs häufig starke Überlasteffekte auf, die die gesamte Kommunikation leicht zum Erliegen bringen können. Das Projekt CXCC (Cooperative Crosslayer Congestion Control) [8] beschäftigt sich mit der Entwicklung neuer Überlastkontrollmechanismen, um Ad-Hoc-Netzwerke in Zukunft robuster zu machen, ihre geringe Bandbreite effizienter auszunutzen und Fairness zu sichern.

Für die praktische Evaluation der neuen Protokolle erweisen sich die gängigen Netzwerkgeräte wie Laptops oder PDAs als größtenteils ungeeignet, da sie nur den IEEE 802.11 WLAN-Standard [25] unterstützen und sich unterhalb der Vermittlungsschicht nicht programmieren lassen. Insbesondere die vom vorhandenen Medienzugriffsprotokoll ausgehenden Seiteneffekte wie Warteschleifen oder die automatische Neuübertragung beschädigter Pakete würden zeitkritische Messungen stark verzerren. Der Lehrstuhl für Rechnernetze hat daher zehn ESB-Sensorknoten *(Embedded Sensor Boards)* [52] angeschafft. Hierbei handelt es sich um Kleinstrechner mit einer primitiven Funkschnittstelle, die an der Freien Universität Berlin entwickelt worden sind. Sie lassen sich hardwarenah in C programmieren und die zugehörige Systemsoftware steht in Quellcodeform zur Verfügung. Sie sollten gute Rahmenbedingungen für die Erprobung von Überlast-

kontrolle in einem *statischen* Ad-Hoc-Netz bieten, denn die Abläufe auf den unteren Netzwerkschichten sind hier transparent und können bei Bedarf modifiziert werden.

1.2 Problemstellung und Beitrag

Im Rahmen dieser Bachelorarbeit sollen die praktischen Voraussetzungen geschaffen werden, um Überlastkontrollmechanismen auf den ESB-Sensorknoten testen zu können. Die Kernaufgabe besteht darin, statisches Routing für die Multihop-Kommunikation zu realisieren und für die Durchführung von Überlastexperimenten Protokollierungsdienste sowie weitere unterstützende Werkzeuge bereitzustellen.

Nach einigen Modifikationen an der vorhandenen Sicherungsschicht kann eine mit Zusatzfunktionen ausgestattete Vermittlungsschicht entstehen. Das Parsen und Verteilen der als Textdatei vorliegenden Routing-Tabellen erledigt dabei das für den PC entwickelte Tool `ESB-RTable`. Der realisierte Protokollierungsdienst erfasst auf jedem Knoten den zeitlichen Fluss von Datagrammen, anschließend werden die erstellten Protokolldaten mit einem weiteren Tool namens `ESB-RLog` auf den Computer übertragen und dort als Logdatei gespeichert. Sie kann im Textformat eingesehen und von einem Analysator weiterverarbeitet werden. Zusätzlich wird auf den ESB-Knoten auch ein Topologiedienst bereitgestellt. Das dritte Hilfswerkzeug `ESB-Dijkstra` berechnet dann auf dem PC aus den gesammelten Nachbarschaftsbeziehungen die *optimalen* Routing-Tabellen. Sie dienen als Eingabe für `ESB-RTable` und können bei Bedarf auch von Hand angepasst werden, um beispielweise einige suboptimale Routen einzubauen.

1.3 Struktur der Arbeit

In Kapitel 2 erfolgt zunächst die Vorstellung der verwendeten ESB-Sensorknoten, wobei nach einer kurzen Motivation für Sensornetze auf die zugrundeliegende Systemarchitektur eingegangen wird. Kapitel 3 widmet sich dann den praktischen Aspekten für die Arbeit mit den ESB-Knoten. Neben der Beschreibung der Entwicklungswerkzeuge und der ScatterWeb Systemsoftware werden die angewendeten Programmierparadigmen vorgestellt, die den stark limitierten Systemressourcen der Knoten Rechnung tragen. Kapitel 4 spricht die drei für den PC entwickelten Tools an und erläutert, wie der Datenaustausch mit den ESB-Knoten stattfindet. Dann konzentrieren sich die Kapitel 5, 6 und 7 auf die Realisierung der Routing-, Protokollierungs- und Topologiedienste im Zusammenspiel mit den PC-Tools. Zum Abschluss zieht Kapitel 8 ein Resümee des Erreichten.

Kapitel 2

Einstieg in die Welt der Sensorknoten

2.1 Drahtlose Sensornetzwerke

Der stetige Fortschritt auf dem Bereich der Halbleitertechnik, insbesondere die steigende Integrationsdichte [28], ermöglicht es seit einigen Jahren, selbst Kleinstgeräte *(Embedded Systems)* kostengünstig mit einer vollwertigen CPU und Speicher auszustatten. Simple Sensoren zur Erfassung von Umweltmessdaten (Temperatur, Feuchtigkeit, Konzentration bestimmter Gase, etc.) können daher leicht zu Kleinstrechnern mit einer Funkschnittstelle erweitert werden und formen dann je nach Einsatzgebiet ein statisches oder mobiles Ad-Hoc-Netzwerk. Als skalierbares und dezentrales System gestattet das Sensornetz die automatisierte und flächendeckende Beobachtung von verteilten, schwer zugänglichen oder bewegten Phänomenen. Es ist im einfachen Fall homogen, kann aber auch aus dedizierten Knoten bestehen, die entweder nur Messdaten aufnehmen oder ihren Dienst als Relais bzw. Datenspeicher verrichten.

Sensornetze sind Gegenstand zahlreicher aktueller Forschungsarbeiten [15]. Die untersuchten Bereiche reichen von Netzwerkprotokollen über Steuerungssysteme bis hin zu Sicherheitsmechanismen. Verschiedene Anwendungsfelder wurden bereits vorgeschlagen, so die Überwachung von Gebäuden und Einrichtungen, die Früherkennung von Waldbränden, das bedarfsorientierte Begießen von Ackerfeldern oder die automatische Erfassung und Kontrolle von Lagerbeständen [21].

Die Rechen- und Speicherkapazität von Sensorknoten ist auf das Mindeste begrenzt, denn neben der Berücksichtigung des Kostenaspekts muss mit den vorhandenen Energieressourcen (meist Batterien) sparsam umgegangen werden. Eine Wartung ist nicht vorgesehen, denn Sensorknoten werden in der Regel einmalig verteilt (z. B. von einem Flugzeug abgeworfen) und dann sich selbst überlassen.

2.2 Aufbau und Funktionsweise der ESB-Sensorknoten

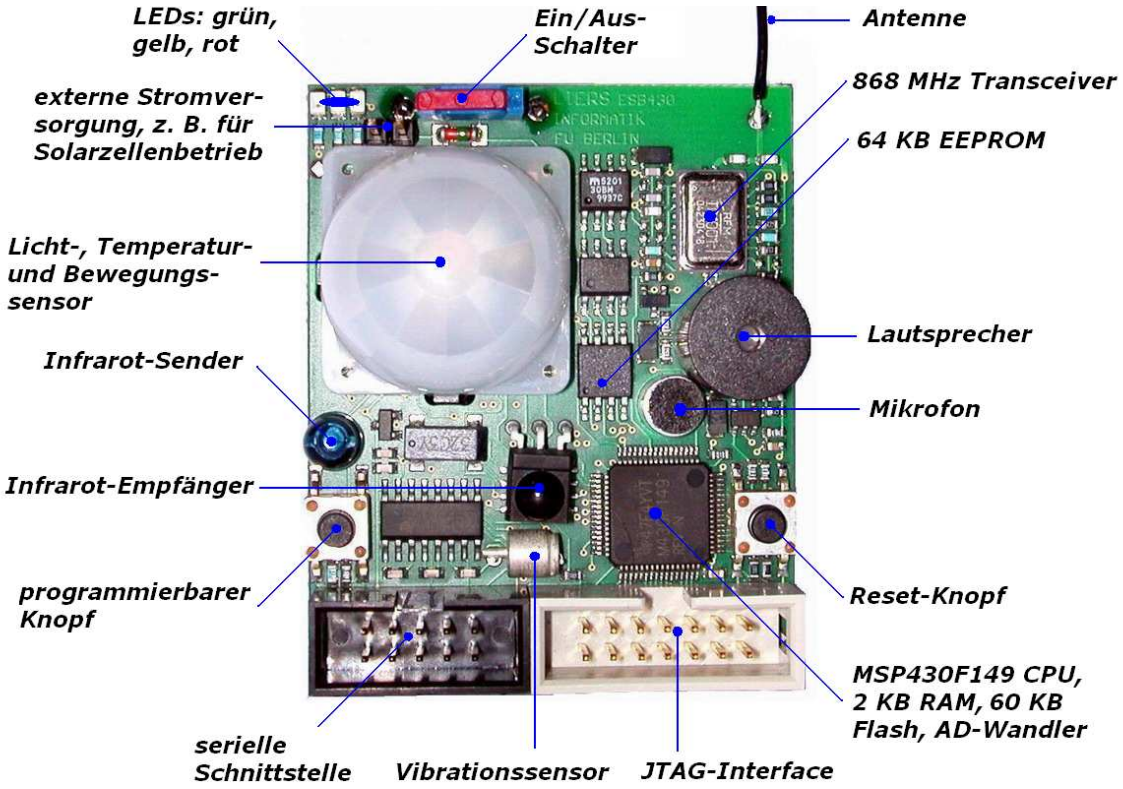


Abbildung 2.1: Ein ESB-Sensorknoten

2.2.1 Allgemeines

Der ESB-Sensorknoten (Abbildung 2.1) ist der Prototyp eines vollwertigen Überwachungssensors. Er soll einerseits das Potential der Sensornetze im kommerziellen Einsatz demonstrieren, andererseits aber auch eine Plattform für Forschung und Lehre auf dem Gebiet der Rechnernetze und der Telematik bieten. Der ESB-Knoten ist Teil der ScatterWeb-Familie von Sensorknoten und Kontrollmodulen [51], die an der Freien Universität Berlin entwickelt worden sind. Einen Überblick zu ScatterWeb im Zusammenhang mit energiebewusstem Routing in Sensornetzen gibt [49]. Zahlreiche Aspekete von Sensornetzen wurden bereits unter Verwendung von ESB-Sensorknoten untersucht, hier eine Auswahl der Arbeiten:

- integrierte Lösungen für Sensorknoten [10] – eine energieeffiziente MAC-Schicht, der RFC-konforme TCP/IP-Protokollstapel *uIP* [64] und das Mini-Betriebssystem

Contiki [7]
Zur Demonstration der erzielten Ergebnisse wird auf einem ESB-Sensorknoten ein kleiner Webserver betrieben [14].

- ein IP-basiertes Sensornetz als Einbruchmeldesystem [11]
- Aufnahme von Bildern und deren stromsparender Versand im Sensornetz [30]
- experimentelle Untersuchung der Lebenszeit von Sensorknoten; Einsatz von Solarzellen und eines GoldCap-Kondensators [48]
- Routing unter Berücksichtigung der verfügbaren Solarenergie [65]
- Eine Sicherheitsarchitektur für mobile Sensornetze [50]

Die relevanten technischen Spezifikationen des ESB-Sensorknotens sind in Tabelle 2.1 zusammengefasst.

2.2.2 MSP430 Systemarchitektur

Als Vertreter der RISC-Familie hat der für Embedded Systeme konzipierte MSP430-Prozessor einen einfachen Befehlssatz, der im Wesentlichen aus Vergleichs- und Transferbefehlen sowie arithmetischen und logischen Instruktionen besteht. Sie können alle in höchstens fünf Systemtakten ausgeführt werden. Die ALU ebenso wie die beiden Busse zur Speicheradressierung und zum Datentransfer haben eine 16-Bit Architektur. Die Adressierung erfolgt byteweise, wobei die meisten Befehle in zwei Varianten vorliegen, denn der Operand kann ein Byte als auch ein Wort sein. Genau wie auf x86er-Systemen gilt die *Little Endian Konvention*. Beim Datenaustausch zwischen PC und Sensorknoten wird sich diese Eigenschaft als sehr nützlich erweisen, da keine Konvertierung von Integern erforderlich ist. Zwölf Arbeitsregister (R4 – R15) stehen zur Verfügung und vier weitere Register (R0 – R3) erfüllen spezielle Funktionen, z. B. enthält das Register R0 den Programmzähler und R1 den Stack-Pointer.

Der im ESB-Sensorknoten verbaute MSP430F149 Mikrocontroller hat bereits einige Peripheriegeräte integriert. Dies sind der Hardware-Multiplizierer (kein Bestandteil der ALU), der Analog-Digital-Wandler mit einer Auflösung von 12 Bit sowie zwei serielle Kommunikationsschnittstellen (USART – *Universal Synchronous/Asynchronous Receive/Transmit*). Im ESB-Knoten wird ein USART unmittelbar als serielle Schnittstelle verwendet, während der andere an den Transceiver gekoppelt ist.

Flash-Speicher, RAM, Bootloader, Peripherieregister und Interrupt-Vektoren bilden zusammen einen kontinuierlichen 64 KB Adressraum (Abbildung 2.2). So kann auf den

Prozessor	16-Bit RISC CPU MSP430F149 von Texas Instruments, Taktfrequenz variabel bis zu 8 MHz [37]
Speicher	• 2 KB RAM und 60 KB Flash-Memory (bis zu 10^5 Schreibzyklen) in der CPU integriert [37] • 64 KB EEPROM (bis zu 10^6 Schreibzyklen) [12]
Funkschnittstelle	• TR1001 868,25 MHz Transceiver [63] Datenrate bis zu 115,2 Kbps, Reichweite im Freien bis zu 100 m • 8,6 cm Drahtantenne[1]
externe Schnittstellen	• serieller Anschluss RS-232 mit bis zu 115,2 Kbps: 10-polige Standard-Pfostenleiste sowie eine Buchse *(seitlich, nicht abgebildet)* zum Anschließen eines Mobiltelefons für SMS-Kommunikation • Programmier- und Debuginterface zum Anschließen eines JTAG-Adapters, z. B. in Parallelportausführung [29]
Sensoren	registrieren Lichteinfall, Temperatur, Vibration, Bewegung, Geräusche [53]
sonstige Ausstattung	Infrarot-Sender und -Empfänger[2], drei LEDs, Lautsprecher [53]
Knöpfe & Schalter	Ein/Aus-Schalter, Reset-Knopf, frei programmierbarer Knopf
Stromversorgung	• Batteriefach *(hinten, nicht abgebildet)* für drei Batterien oder Akkus der Größe AA-Mignon • Anschluss für eine externe Stromquelle wie z. B. Solarzellen *Intern wird die Spannung auf 3V heruntergeregelt [13].*

[1] Aus der Nachrichtentechnik ist bekannt, dass an einer Antenne dann die maximale Spannung induziert wird, wenn ihre Länge ein Viertel der verwendeten Wellenlänge beträgt [1]. Zwischen der Wellengeschwindigkeit c, der Wellenlänge λ und der Frequenz f besteht der einfache Zusammenhang $c = \lambda f$. Für die optimale Antennenlänge α ergibt sich daher

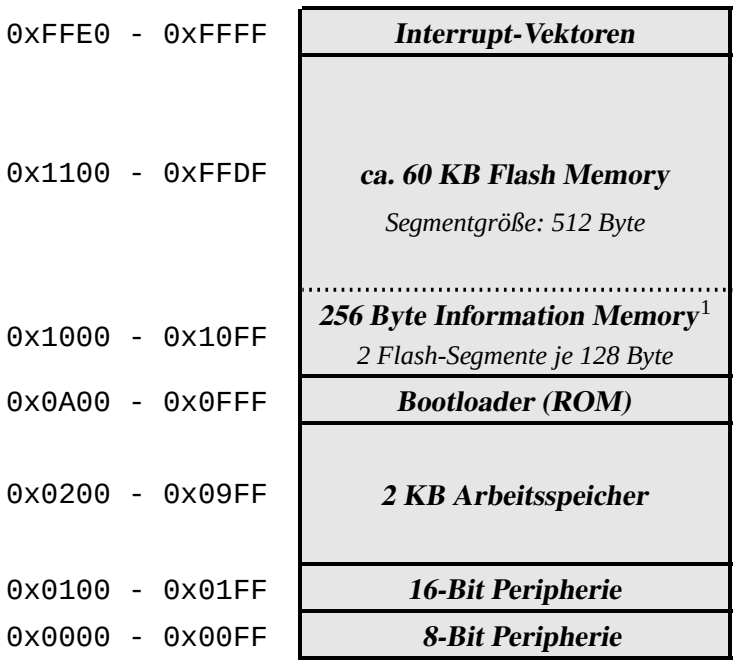
$$\alpha = \frac{\lambda}{4} = \frac{1}{4} \cdot \frac{c}{f} = \frac{1}{4} \cdot \frac{3 \cdot 10^8 \frac{m}{s}}{868{,}25 \cdot 10^8 \frac{1}{s}} = 8{,}64\ cm$$

wobei für die Wellengeschwindigkeit die Lichtgeschwindigkeit eingesetzt worden ist.

[2] Ermöglicht es beispielsweise, den Sensorknoten so zu programmieren, dass er sich mit einer Fernbedienung nach dem RC5-Standard [46] steuern lässt.

Tabelle 2.1: Technische Spezifikationen

gesamten Speicher und insbesondere die Peripheriegeräte aus einer Hochsprache wie C heraus mittels Zeigern direkt zugegriffen werden. Man bezeichnet dieses Konzept als *memory mapped I/O* [35]. Im ESB-Sensorknoten sind die Knöpfe, LEDs, Sensoren und das EEPROM als zusätzliche Peripheriegeräte angebunden. Über die Register des Analog-Digital-Wandlers ist die am Mikrofon und Funkempfänger induzierte Spannung sowie die Spannung der Batterien direkt verfügbar. Der System- und Anwendungscode wird im Flash-Speicher beginnend mit den unteren Segmenten abgelegt und auch von dort ausgeführt. Die oberen, frei gebliebenen Flash-Segmente können zur Laufzeit byteweise



[1]Gehört zum Flash-Speicher enthält aber keinen Maschinencode, sondern ist für die Speicherung von Konfigurationseinstellungen gedacht. Die Information Memory bleibt beim regulären Löschen des Flash-Speichers unangetastet.

Abbildung 2.2: MPS430F149: 64 KB Adressraum

beschrieben werden, falls vorher eine segmentweise Löschung erfolgt. Der EEPROM-Speicher ist hingegen direkt wiederbeschreibbar, jedoch muss der Zugriff auf ihn über einen im RAM befindlichen Puffer erfolgen, da der EEPROM als Zusatzspeicher nicht Teil des gemeinsamen Adressraums ist. Im Arbeitsspeicher werden in den unteren Segmenten nur die globalen Variablen abgelegt (kein Code!), an die sich falls vorhanden der Heap (dynamisch zugewiesener Speicher) anschließt, während der Stack im obersten Segment beginnt und nach unten hin wächst. Der Bootloader ist eine vom Hersteller in ROM eingebrannte Steuersoftware zum Flashen von Speicherblöcken über die serielle Schnittstelle eines PCs. Von ihm wird aber kein Gebrauch gemacht, sofern man zum Programmieren das JTAG-Interface verwendet.

Der CPU-Takt kann in einem großen Intervall von einigen KHz bis zu 8 MHz frei eingestellt werden und man hat die Wahl zwischen mehreren Taktquellen – es stehen zwei unterschiedlich schnelle Quarzoszillatoren und ein DCO *(Digitally Controlled Oscillator)* zur Verfügung.

Timer ermöglichen mittels Interrupts die Ausführung von periodischen Prozessen, so zum Beispiel das Senden von Daten über die serielle Schnittstelle mit einer festen Baudrate oder das Betreiben einer softwaregesteuerten Uhr. Der sogenannte *Watchdog* ist

ein ebenfalls durch Timer realisierter Zähler, welcher bei Überlauf (z. B. nach einer Sekunde) einen Reset des Knotens durchführt. Der Watchdog dient als eine Schutzvorrichtung. Er wird von der Systemsoftware regelmäßig reinitialisiert, sodass es zum besagten Zählerüberlauf nur im Falle einer Endlosschleife in einem fehlerhaften Codeabschnitt kommt.

Der MSP430 Mikrocontroller unterstützt mehrere Stromsparmodi mit interruptgesteuerter Reaktivierung. Er ist somit zum energieeffizienten Betreiben von Sensorknoten geradezu prädestiniert. Der hierbei übliche Verarbeitungsablauf, wie er sich auch in der ScatterWeb Systemsoftware (Abschnitt 3.2) wiederfindet, ist in Abbildung 2.3 dargestellt.

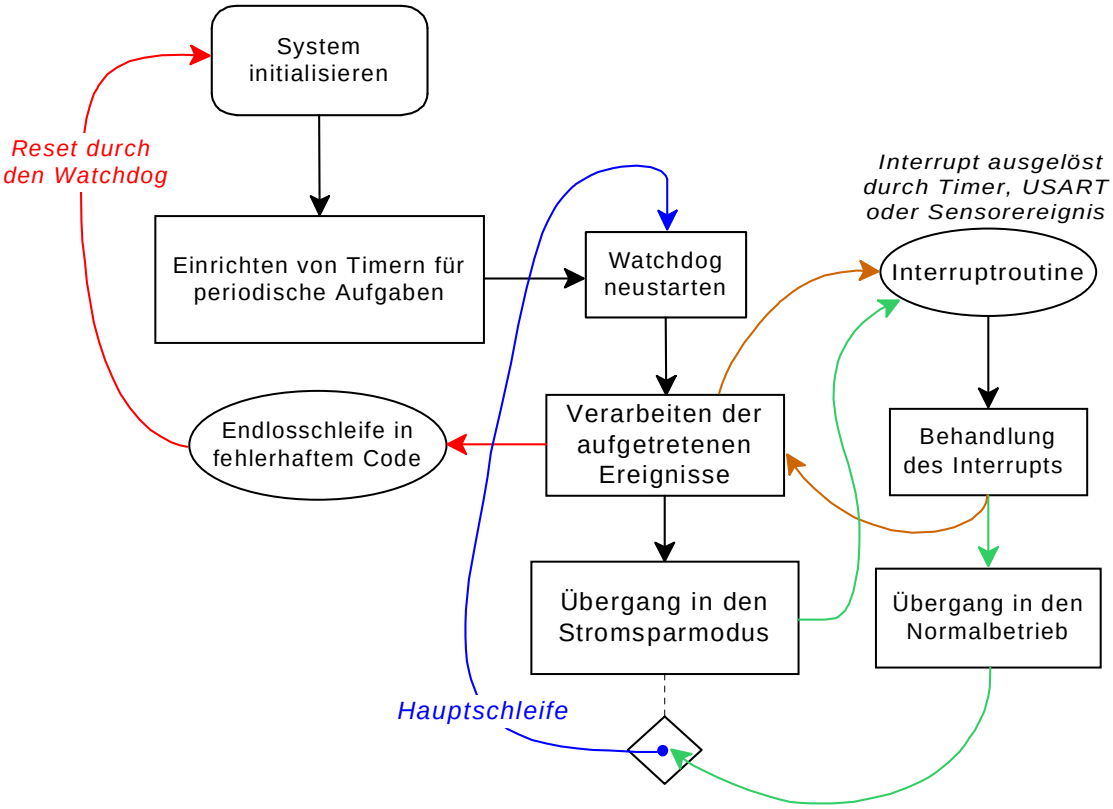


Abbildung 2.3: Verarbeitungsablauf in einem Sensorknoten

Dieser Unterabschnitt sollte am Beispiel des ESB-Sensorknotens einen groben Überblick über die Architektur eines mit dem MSP430 Mikrocontroller betriebenen Embedded Systems geben. Für Details empfiehlt es sich, das Datenblatt [37] und die technische Referenz [36] von Texas Instruments zu konsultieren, ein leicht verständlicher Leitfaden zur MSP430 Prozessorfamilie ist außerdem das Buch [42].

Kapitel 3

ESB-Sensorknoten in der Praxis

Nach der Vorstellung der Entwicklungswerkzeuge werden die ScatterWeb-Systemsoftware und die angewendeten Programmierparadigmen beschrieben.

3.1 Programmieren, die ersten Schritte

Für den MSP430 Mikrocontroller existieren einige kommerzielle C Compiler mit integrierter Entwicklungsumgebung [24] [23] [26] sowie eine Portierung des Open Source GNU C Compilers unter dem Namen *mspgcc* [38]. Da die ScatterWeb Systemsoftware für den *mspgcc* Compiler geschrieben wurde und kostenfreie Software insbesondere für akademische Zwecke gut geeignet ist, fiel die Wahl des zu verwendenden Compilers entsprechend auf *mspgcc*. Zumal stehen der *mspgcc* Compiler und die zugehörige Toolbox sowohl für Windows- als auch Linux-Betriebssysteme zur Verfügung [38] [39].

Zum automatisierten Kompilieren, Linken und Flashen wird ein Makefile [34] verwendet. Hierbei übersetzt `msp430-gcc` die Quelltexte und bindet die entstandenen Objektdateien zu einer Binärdatei zusammen, welche darauf vom Flasher `msp430-jtag` [40] mit Hilfe des am Parallelport angeschlossenen JTAG-Adapters auf den ESB-Knoten übertragen wird. Für die Fehlersuche ist in der *mspgcc* Toolsammlung der Kommandozeilendebugger `msp430-gdb` enthalten, es empfiehlt sich jedoch, zum Debuggen das separat erhältliche graphische gdb-Frontend *Insight* [27] einzusetzen. In beiden Fällen muss vor Beginn einer Debug-Sitzung der im Hintergrund arbeitende JTAG-Proxy `msp430-gdbproxy` aufgerufen werden. Ein Editor mit Projektverwaltung und der Möglichkeit, externe Programme aufzurufen, kann die fehlende Entwicklungsumgebung ersetzen. Weiterführende Informationen und Installationshinweise geben das *mspgcc* Handbuch mit der zugehörigen FAQ [59] sowie der Artikel [41].

3.2 Die ScatterWeb Software

3.2.1 Überblick

Für die ESB-Sensorknoten stellen die ScatterWeb-Entwickler ein Paket von C-Quelltexten zur Verfügung [52], es beinhaltet die ScatterWeb Systemsoftware (auch *Firmware* genannt), einige Testanwendungen (z. B. einen SMS-Dienst) sowie ein Skelett zum Erstellen eigener Applikationen.

Die Firmware ist eine Art Mini-Betriebssystem: Sie initialisiert nach dem Einschalten sämtliche Module, realisiert die Interruptbehandlung, betreibt den Transceiver, eine Terminalschnittstelle sowie eine softwaregesteuerte Uhr und stellt Routinen für den Zugriff auf die Hardware (Sensoren, EEPROM, Timer, USARTs) zur Verfügung. Insbesondere ist in der Firmware die *Sicherungs-* und *Bitübertragungsschicht* des Funkmoduls implementiert. Diese unteren Schichten bildeten die Grundlage für die Realisation von Routing- und Protokollierungsdiensten.

Die *Anwendungssoftware* nutzt die von der *Firmware* bereitgestellten Systemfunktionen, um den Knoten für eine konkrete Aufgabe einzusetzen. Sie enthält einige als *Callbacks* registrierte *Handler-Funktionen*, um auf Sensormeldungen, abgelaufene Timer und über Funk empfangene Datenpakete reagieren zu können. Diese Handler-Funktionen werden von der Firmware aufgerufen, sobald das jeweilige Ereignis eingetreten ist. Die Trennung zwischen Systemsoftware und Anwendung ist rein logischer Natur, denn es existieren keine Virtualisierungs- oder Zugriffsschutzkonzepte – der gesamte Code wird auf einer Ebene ausgeführt.

Die im Rahmen dieser Arbeit entwickelten Dienste wurden in Form von *Anwendungs*modulen implementiert und bauen auf dem bereitgestellten Applikationsskelett auf. An zahlreichen Stellen musste aber auch der vorhandene Firmwarecode modifiziert oder um neue Funktionen erweitert werden. Insbesondere erwiesen sich einige Eingriffe in die Sicherungsschicht des Funkmoduls als erforderlich. Die entsprechenden Stellen sind mit markanten Kommentaren gekennzeichnet. Eine mit Annotationen versehene Übersicht sämtlicher vorhandener und neu hinzugekommener Quelltextdateien befindet sich im Anhang C.4.

Die ScatterWeb Firmware wird kontinuierlich weiterentwickelt. Die im April 2005 erschienene Version 2.0 wies im Vergleich zur Vorgängerversion 1.0 eine grundlegend überarbeitete sowie im Funktionsumfang stark erweiterte API auf. Es wurde daher beschlossen, die ESB-Sensorknoten mit der neusten Firmware zu betreiben. Die Quelltexte von Version 2.0 waren jedoch sehr spärlich kommentiert und eine Dokumentation exis-

tierte nur für die besagte Vorgängerversion. Beim Durcharbeiten des Firmwarequellcodes und Ausprobieren der bereitgestellten Funktionen wurden außerdem mehrere kritische Fehler sowie eine Reihe kleinerer Ungereimtheiten beseitigt. Das Gros der Probleme betraf gerade die Funkschnittstelle, ein ordnungsgemäßer Versand von Datenpaketen zwischen zwei Knoten war anfangs nicht möglich[1].

3.2.2 Taktgeber, Uhr und Interrupts

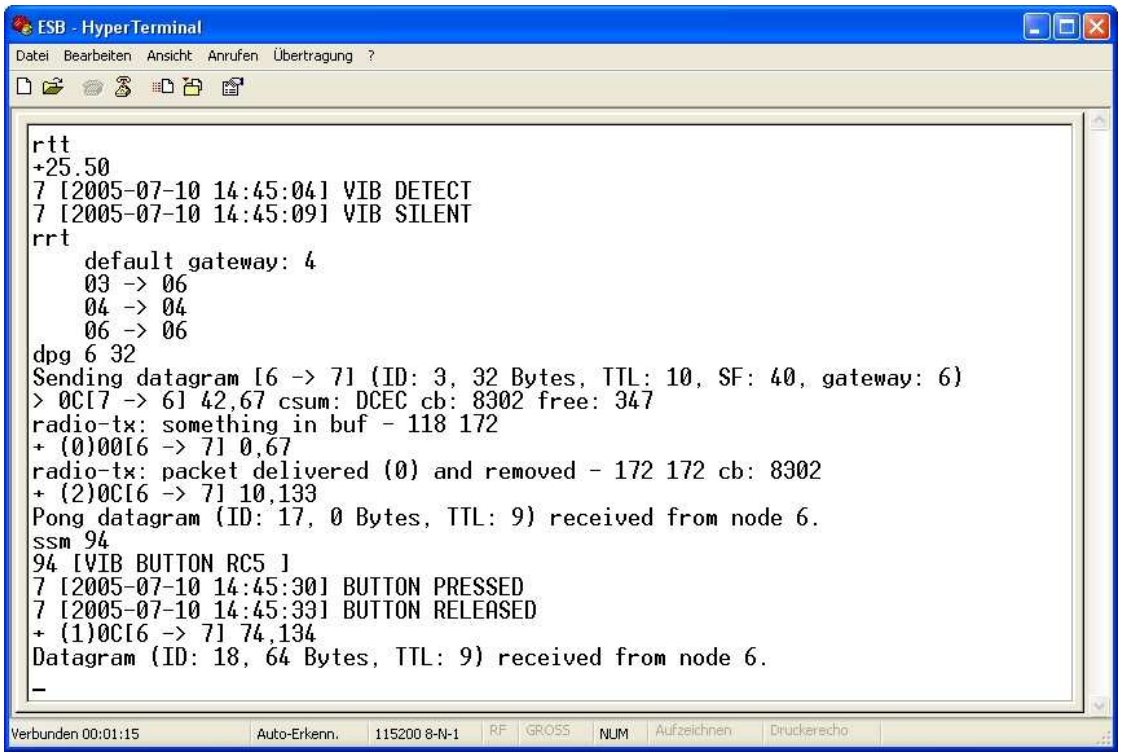
Die Firmware taktet die MSP430-CPU im Regelbetrieb mit 4,506 MHz, Taktgeber ist der DCO. Durch Ableiten dieses DCO-Takts werden die serielle Schnittstelle und der Transceiver mit 115,2 Kbps respektive 27,6 Kbps interruptgesteuert betrieben. Ein mit 32.768 Hz schwingender Quarzkristall löst 1024 Mal in der Sekunde einen Timerinterrupt aus. Dieser wird benutzt, um die in Software realisierte Uhr zu takten und auf eventuell ablaufende Timer (z. B. Temperatur periodisch überprüfen) zu reagieren. Die Millisekunden der Uhr sind Timerticks und erst nach Ablauf von 1024 Pseudo-Millisekunden bricht eine neue Sekunde an. Eine weitere Interruptquelle bilden die Sensoren, insgesamt gilt leicht vereinfachend der in Abbildung 2.3 dargestellte Verarbeitungsablauf. Es sei hier angemerkt, dass die Interruptroutinen Ereignisse lediglich registrieren und vorverarbeiten, die eigentliche Reaktion auf diese Ereignisse findet dann in den zuständigen Handler-Funktionen statt. Sie werden beim nächsten Durchlauf der Hauptschleife aufgerufen.

3.2.3 Steuerung des Sensorknotens mit Terminalkommandos

Die Bedienung des ESB-Sensorknotens erfolgt interaktiv durch Terminalkommandos, hierzu wird der Knoten an die serielle Schnittstelle des PCs angeschlossen und ein Terminalprogramm aufgerufen (siehe Abbildung 3.1). Im Terminalfenster gibt der Sensorknoten auch laufend Statusmeldungen zu aktuellen Ereignissen (z. B. Sensoraktivität, Empfang von Datenpaketen über Funk). Die meisten Betriebssysteme liefern bereits ein für Modemwählverbindungen gedachtes Terminalprogramm mit, so steht unter Windows *HyperTerminal* und unter Linux *minicom* zu Verfügung. Für die Kommunikation mit dem Sensorknoten muss die serielle Schnittstelle folgendermaßen konfiguriert werden:

- 115.200 Bps Übertragungsrate
- 8 Datenbits
- keine Parität
- 1 Stoppbit
- keine Flusskontrolle
- lokales Echo an.

[1] Mitte Mai 2005 erschien Version 2.1 der ScatterWeb-Firmware, in der die meisten funktionsrelevanten Bugs behoben wurden, außerdem wiesen zahlreiche Codeabschnitte deutlich ausführlichere Kommentare auf. Für ein Update bestand jedoch kein Bedarf.



Temperaturabfrage, Vibrationsmeldung, Ausgabe der Routing-Tabelle, Senden eines Ping-Datagramms, Setzen der Sensoreinstellungen, Reaktion auf einen Knopfdruck, Empfang eines Datagramms

Abbildung 3.1: Ein ESB-Sensorknoten im Terminalbetrieb

Terminalkommandos bestehen aus drei Buchstaben und haben manchmal Parameter, die Eingabe wird mit dem `Line Feed (LF)` Zeichen[2] abgeschlossen. Neben der Konfiguration und Diagnose der ESB-Knoten dienen sie der Durchführung von Experimenten sowie der Erprobung von neuer Funktionalität. Einige Dutzend Terminalkommandos sind bereits Bestandteil der ScatterWeb-Software und zahlreiche weitere wurden im Rahmen dieser Arbeit programmiert. Eine Übersicht der gebräuchlichsten Befehle befindet sich im Anhang A.

3.3 Paradigmen der Embedded-Programmierung

Obwohl sich durch die Verfügbarkeit eines C Compilers das Programmieren der ESB-Sensorknoten nicht wesentlich vom Schreiben kleiner Programme für den PC unterscheidet, waren bei der Entwicklung einige Designaspekte zu beachten. Insbesondere musste den stark limitierten Systemressourcen Rechnung getragen werden.

[2] Sollte – falls nicht bereits geschehen – mit der Return-Taste verknüpft werden.

Die knappste Ressource ist der Arbeitsspeicher, es stehen lediglich 2048 Byte an RAM zur Verfügung und man sollte sich stets bewusst sein, wieviel Speicher durch was belegt wird und wieviel noch frei bleibt. Beim Compilieren des Projekts wird im Makefile zum Schluss das Tool `msp430-size` aufgerufen, es zeigt sowohl die Größe des Programmcodes *(text)* als auch den durch globale Variablen in Anspruch genommenen Arbeitsspeicher *(bss)* an. Zur Laufzeit kann außerdem mit dem Terminalkommando `mem` die aktuelle Speicherbelegung ermittelt werden. Der größte Teil des Arbeitsspeichers wird bereits durch die in der Systemsoftware definierten *globalen* Puffer und Zustandsvariablen belegt. Für die Benutzung von zum Compilierungszeitpunkt fest reservierten globalen Puffern gibt es einen guten Grund: Die Allokierung von dynamischem Speicher ist nämlich zwar prinzipiell möglich, praktisch aber nicht empfehlenswert. Dynamischer Speicher würde auf einem MSP430-System auf Dauer zur starken Speicherfragmentierung führen, die mangels MMU *(Memory Management Unit)* nicht behoben werden könnte; zudem bestünde die Gefahr, dass Stack und Heap unbemerkt kollidieren. Auch das Debuggen wäre durch die Benutzung von dynamischem Speicher erheblich erschwert. Daher wurde auf die Vewendung von `malloc` vollständig verzichtet.

Wenn globale Variablen benötigt wurden, dann galt es stets, den passenden Datentyp auszuwählen. Während auf PCs selbst boolesche Werte oft durch einen Integer dargestellt werden, sollte man auf Embedded Systemen den verfügbaren Speicher möglichst sparsam ausnutzen. Konkret heißt das, dass für viele Zwecke der Datentyp `UINT8`[3] vollkommen ausreicht. Man kann darin gut den Index eines Arrays, einen Zähler oder bis zu acht boolesche Werte (Flags) ablegen. Für Zahlen über 255 wurde dann auf den Datentyp UINT16 zurückgegriffen, ansonsten kam im Zusammenhang mit Puffern und Rohdaten häufig der Zeiger UINT8* zum Einsatz.

Variablen vom Typ `UINT8` oder `UINT8`-Arrays ungerader Länge können sich sowohl an einer geraden als auch ungeraden Speicheradresse befinden, während Variablen der anderen Datentypen (2, 4 oder 8 Byte groß) an geraden Adressen beginnen müssen. Im Zusammenhang mit Puffern verdiente dieser Umstand beim Casten von UINT8* Zeigern auf UINT16* Zeiger besondere Beachtung.

Die Größe des Programmcodes erwies sich nicht als weiter kritisch, da die zur Verfügung stehenden 60 KB Flash-Speicher mehr als ausreichend sind, um neben der bereitgestellten Firmware (ca. 24 KB an Code) zahlreiche Anwendungsmodule und neue

[3] Der *mspgcc* Compiler kennt die elementaren Datentypen `char` (8-Bit), `int` (16-Bit), `long` (32-Bit) und `long long` (64-Bit) sowie `float` (32-Bit) für Fließpunktzahlen. Zeiger sind grundsätzlich 16 Bit lang. Die UINT-Bezeichner wurden von den ScatterWeb-Entwicklern eingeführt, da sie handlicher als beispielsweise `unsigned char` sind und die Übersichtlichkeit verbessern.

Systemfunktionen unterzubringen. Auf den Aufruf von C-Bibliotheksfunktionen wurde jedoch verzichtet, da der große Funktionsumfang von Routinen wie `sprintf` zu einer unnötigen Aufblähung der Codegröße führt. Stattdessen wurden die in der ScatterWeb-Firmware implementierten Stringroutinen benutzt und kleine Suchalgorithmen ließen sich ohne größeren Aufwand neu programmieren.

Multiplikation, Division und Modulorechnung sind auf einem MSP430-System teuere Operationen. Man beachte, dass sie keine elementaren Instruktionen sind, sondern vom Hardware-Multiplizierer bzw. in einer Unterprozedur ausgeführt werden. In performancerelevanten Routinen wurden sie durch effiziente Schiebe- und Bitoperationen ersetzt, falls der rechte Operand eine Zweierpotenz war. Beim Rechnen mit Byteanzahlen konnte dies durch geeignet dimensionierten Puffer und Strukturen erreicht werden.

Der GNU C Compiler unterstützt eine Reihe von nützlichen Sprachkonstrukten [19], die nicht Teil des ANSI C Standards [22] sind; zudem ist der neue ISO C99 Standard [5] nahezu vollständig umgesetzt worden. Diese Erweiterungen wurden im Makefile nachträglich aktiviert, denn die ScatterWeb-Entwickler haben von dieser Möglichkeit keinen Gebrauch gemacht. Man kann nun beispielsweise wie in C++ Deklarationen und Anweisungen mischen oder Strukturen definieren, die ein Array unbestimmter Größe enthalten *(flexible array members)*. Insbesondere steht zur Optimierung kleiner Funktionen das Schlüsselwort `inline` zur Verfügung. Es erspart den Overhead eines Prozeduraufrufs (Hin- und Rücksprung, Parameterübergabe) durch Einfügen des kompletten Funktionscodes an all den Stellen, wo diese Funktion verwendet wird. Einen ähnlichen Effekt bewirkt die Benutzung von Makros. Sie finden an zahlreichen Stellen in der vorhandenen ScatterWeb-Firmware Anwendung, setzen aber die Lesbarkeit des Codes herab und sind fehleranfälliger als Funktionen. Das Ausnutzen der verfügbaren Erweiterungen sollte insgesamt zu besser strukturiertem und leichter zu wartendem Quellcode geführt haben.

Bei Verwendung des Debuggers war es meist nicht möglich, einen Codeabschnitt Schritt für Schritt auszuführen, da zwischenzeitlich Timerinterrupts stattfinden und eine Verzweigung in die Interruptroutine erfolgt oder der Sensorknoten vom Watchdog resettet wird. Man konnte sich damit begnügen, an einer verdächtigen Stelle einen Haltepunkt zu setzen und dort angekommen die Variablenbelegung zu inspizieren, gegebenenfalls musste dieses Verfahren durch Neustarten der Debugsitzung wiederholt werden. Beim Aufspüren von Speicherproblemen half auch das im Makefile aufgerufene Tool `msp430-objdump`, es erstellt eine Übersicht der Speicheradressen sämtlicher Variablen (RAM) und Funktionen (Flash).

Kapitel 4

Hilfsprogramme für den PC

4.1 Überblick

Für den praktischen Einsatz der auf den ESB-Sensorknoten implementierten Dienste wurden die drei PC-Hilfsprogramme `ESB-RTable`, `ESB-RLog` und `ESB-Dijkstra` entwickelt. Sie tauschen mit einem am Computer angeschlossenen ESB-Knoten Daten aus, formatieren diese und werten sie aus. `ESB-RTable` ist dazu da, um die in einer Textdatei vorgefertigten Routing-Tabellen zu parsen und auf die ESB-Knoten zu übertragen; sogar eine automatisierte Verteilung im gesamten Netz ist möglich. Umgekehrt lädt `ESB-RLog` die in einem Knoten zum Datagrammverkehr erstellten Protokolldaten auf den PC, wo sie sowohl im Binärformat als auch in einer menschenlesbaren Logdatei gespeichert werden. Das dritte Tool `ESB-Dijkstra` transferiert die vom Topologiedienst gesammelten und in einem Hauptknoten gespeicherten Nachbartabellen ebenfalls auf den Rechner, um daraus mit Hilfe des eingebauten Routing-Algorithmus die optimalen Routing-Tabellen für alle Knoten zu berechnen.

Bei den drei Hilfsprogrammen handelt es sich um englischsprachige Kommandozeilentools, die in der Programmiersprache C++ geschrieben wurden. Eine grafische Benutzeroberfläche *(GUI)* würde bei den eng umgrenzten Aufgaben der Tools kaum Bedienungsvorteile bringen und der Verzicht auf betriebssystemspezifische GUI-Elemente ermöglicht es, die ESB-Sensorknoten in den später anstehenden Überlastexperimenten plattformunabhängig einzusetzen. Dieses Ziel ist bereits durch die Wahl des *mspgcc* Compilers anvisiert worden. Konkret sind die ESB-Tools auf allen aktuellen Versionen der Betriebssysteme Windows und Linux lauffähig. Werden sie ohne Parameter aufgerufen, so bekommt man eine kurze Beschreibung mit den Aufrufoptionen angezeigt.

Die Implementierung der ESB-Tools fand auf einer Windows-Plattform unter der Entwicklungsumgebung *Microsoft Visual C++ 6.0* [66] statt. Sie ermöglicht die unmittel-

bare Erzeugung einer ausführbaren Datei aus den zu einem Projekt zusammengefassten Quelltexten und bietet einen funktionsreichen grafischen Debugger. Für den Einsatz unter Linux musste dann noch ein Makefile [34] erstellt werden, welches die Abhängigkeiten zwischen den Quelltextdateien erfasst sowie Compiler und Linker aufruft. Hierbei kam der im Lieferumfang sämtlicher Linux-Distributionen enthaltene C++ Compiler `g++` aus der GNU Compiler-Familie *GCC* [18] zum Einsatz.

Neben der C Standard Bibliothek [43] wurde auch von der C++ Standard Template Library *(STL)* [4] sowie den beiden SGI-Erweiterungen `slist` [56] und `hash_map` [55] Gebrauch gemacht. Diese sind unter Linux in der Bibliothek des GNU C++ Compilers in aller Regel enthalten, und für Visual C++ sowie andere C++ Compiler besteht die Möglichkeit, die komplette STL inklusive der Erweiterungen kostenlos von SGI zu beziehen [57]. Dort ist auch eine Dokumentation sämtlicher STL-Klassen verfügbar, insbesondere enthält sie Aussagen zur Laufzeitkomplexität von zahlreichen Klassenmethoden. Diese Dokumentation half dabei, eine algorithmisch effiziente Implementierung zu realisieren.

4.2 Kommunikation über die serielle Schnittstelle

Die Kommunikation zwischen den Tools und den ESB-Sensorknoten erfolgt genau wie die Steuerung mit Terminalkommandos über die serielle Schnittstelle. Da der Zugriff auf die serielle Schnittstelle im Sprachumfang von C++ nicht enthalten ist (im Gegensatz zum Dateizugriff), sondern über vom Betriebssystemkernel bereitgestellte Routinen erfolgt, müssen die plattformabhängigen API-Aufrufe für jedes Betriebssystem einzeln implementiert werden. Üblicherweise kapselt man sie in einer Klasse, die nach außen eine einheitliche Schnittstelle zum Öffnen und Konfigurieren des seriellen Ports, zum Setzen von Timeouts und zum Senden und Empfangen von Daten bereitstellt.

Eine Anleitung zum Programmieren der seriellen Schnittstelle und zum Implementieren einer solchen Klasse für die Betriebssysteme Windows und Linux gibt [33]. Zum Einsatz kam die Klasse `rlSerial` [45], sie ist Teil eines größeren Open-Source-Projekts [44]. `rlSerial` unterstützt neben den beiden Plattformen Windows und Linux auch OpenVMS. Es waren jedoch einige kleine Modifikationen an dieser Klasse erforderlich, u. a. zeigte das Setzen von Timeouts unter Linux anfangs keine Wirkung.

Im ESB-Knoten ist der Versand von Binärdaten über den seriellen Port direkt möglich, deren Empfang würde aber mit der auf neue Kommandos lauschenden Terminalschnittstelle kollidieren. Daher musste die Firmware um einen binären Empfangsmodus erweitert werden, in welchem die Terminalschnittstelle ausgeschaltet ist.

Kapitel 5

Routingdienste

Für das Funkmodul stellt die Firmware eine Bitübertragungs- und Sicherungsschicht zur Verfügung. Darauf aufbauend entstand für die Multihop-Kommunikation eine Vermittlungsschicht mit statischem Routing. Nach einer kurzen Beschreibung der unteren Schichten erfolgt eine ausführliche Darstellung der Vermittlungsschicht. Anschließend wird das Hilfsprogramm `ESB-RTable` präsentiert. Im Mittelpunkt stehen dabei die zum Einsatz kommenden Techniken, um die in einer Textdatei gespeicherten Routing-Tabellen zu parsen und im ESB-Netz automatisiert zu verteilen.

5.1 Die Bitübertragungs- und die Sicherungsschicht

Versand und Empfang von Daten erfolgen byteweise durch ausgelöste Interrupts. Die tx- und rx-Interruptroutinen sind intern wie ein endlicher Automat aufgebaut, in Abhängigkeit vom aktuellen Zustand führen sie Aktionen und Zustandsübergänge durch. Als Basisbandverfahren kommt die *Manchester-Codierung* zum Einsatz. Durch die dabei zwingende Transition in der Intervallmitte wird jedes Bit durch zwei Bits kodiert (`0` → `01` und `1` → `10`), was entsprechend zu einer Halbierung der effektiven Datenrate führt. Es ist nur ein Halbduplexbetrieb möglich. Das dabei verwendete Medienzugriffsverfahren ist *CSMA/CA*.

Neben dem *Medienzugriff* und der *Rahmenbildung* bietet die Sicherungsschicht eine *Fehlererkennung* (CRC-16 Prüfsumme) und garantiert eine *zuverlässige Übertragung*. Dies wird durch Versenden von Bestätigungen (ACKs) in Verbindung mit Übertragungswiederholungen erreicht. Das verwendete Paketformat ist Abbildung 5.1 zu entnehmen.

Im Paketheader sind *Ziel* und *Quelle* die IDs der jeweiligen Knoten. Der *Pakettyp* bestimmt, wie ein Paket vom Empfänger behandelt werden soll, es kann z. B. ein ACK, eine Sensormeldung, ein Ping oder ein verkapseltes Datagramm (siehe nächsten Abschnitt)

Ziel	*Quelle*	*Pakettyp*	*Sequenz-nummer*	*Datenlänge*	*Nutzdaten* (n Byte)	*CRC-16*
1 2	3 4	5	6	7 8	9 10 ... 8+n	8+n+1 8+n+2

Abbildung 5.1: Paketformat auf der Sicherungsschicht

sein. Anhand der *Sequenznummer* können durch Übertragungswiederholungen entstandene Duplikate erkannt und ignoriert werden. Die Größe der Nutzdaten ist durch die Dimensionierung des Empfangspuffers auf etwa 260 Byte begrenzt. Dieser ist exklusiv belegt, solange das empfangene Paket nicht von sämtlichen Handlerroutinen in der Firmware und Anwendung behandelt wurde. Erst nach Freigabe des Empfangspuffers werden neue Pakete angenommen. Der deutlich größere Sendepuffer ist hingegen als Ringpuffer angelegt und kann mehrere ausgehende Pakete in Warteschlange stellen.

Der Paketheader hat eine gerade Größe und damit bei Hintereinanderreihung von mehreren Paketen im Ringpuffer die 16 Bit langen Headereinträge an geraden Speicheradressen beginnen, muss die Datenlänge zwingender Weise auch gerade sein. Die ScatterWeb-Entwickler erreichten dies durch eine etwas radikale Methode: Bei allen Paketen mit ungerader Datenlänge fügte die Senderoutine `Net_send()` an die Nutzdaten ein zufälliges Byte an und erhöhte den Headereintrag *Datenlänge* um eins. Dieser Vorgang war irreversibel, denn der Empfänger konnte nicht mehr feststellen, ob die übertragenen Nutzdaten bereits eine gerade Länge hatten oder auf eine gerade Länge erweitert wurden. Insbesondere für den Betrieb des Topologiedienstes (Kapitel 7) musste die Möglichkeit bestehen, auch Nutzdaten ungerader Länge unverfälscht übermitteln zu können. Mit Hilfe eines im Headereintrag *Pakettyp* untergebrachten Flags zur Kennzeichnung ungerader Nutzdaten wurde dieses Problem aber behoben.

5.2 Entwurf einer Vermittlungsschicht

Zum Transport werden Datagramme in Paketen der Sicherungsschicht verkapselt, der hierfür speziell eingeführte Pakettyp ist die Konstante `ROUTING_PACKET`. Ein empfangenes Paket dieses Typs wird nach seiner Auslieferung an die Anwendung von der Funktion `RoutingHandler()` behandelt, ihr Gegenstück ist die Senderoutine `SendDatagram()`.

Abbildung 5.2 zeigt das gewählte Datagrammformat. In Anlehnung an den Header auf der Sicherungsschicht sind die Einträge *Ziel*, *Quelle* und *Datenlänge* 16 Bit lang, die Adressierung erfolgt analog. Jedes Datagramm bekommt eine eindeutige 16-Bit ID zuge-

Ziel	*Quelle*	*ID*	*Datenlänge*	*Hop Counter*	*Service Flags*	*Nutzdaten* (*n Byte*)
1 2	3 4	5 6	7 8	9	10	11 12 ... 10+n

Abbildung 5.2: Datagrammformat auf der Vermittlungsschicht

wiesen; diese kann beim Aufgeben des Datagramms manuell zugeteilt oder automatisch vergeben werden. Im Gegensatz zur 8 Bit kurzen Sequenznummer auf der Sicherungsschicht hat die ID den Zweck, bei der im nächsten Kapitel vorgestellten Protokollierung des Datagrammverkehrs eine eindeutige Identifikation der Datagramme über einen längeren Zeitraum zu ermöglichen. Der *Hop Counter*[1] ist genau wie bei IP-Datagrammen dazu da, die Anzahl der zwischen Sender und Empfänger zu passierenden Stationen zu begrenzen, damit Datagramme aufgrund von Routing-Schleifen nicht ewig im Netzwerk kreisen. Jede Zwischenstation dekrementiert den Hop Counter um eins und bei Erreichen des Werts null wird das Datagramm verworfen. Der initiale Wert kann entweder für jedes Datagramm individuell vergeben werden oder es wird ein Default-Wert von zehn verwendet, der für kleine Netze im Allgemeinen ausreicht.

Inspiriert vom selten benutzten *Type Of Service* Eintrag im IPv4-Header [47] wurden in den Datagrammheader zusätzlich die *Service Flags* aufgenommen, um unterschiedliche Dienstmerkmale zu realisieren. Sie sollen primär dazu dienen, die Chancen für die Zustellbarkeit eines wichtigen Datagramms zu erhöhen. Bei Überlastexperimenten würden sie die Untersuchung von komplexeren Konstellationen ermöglichen. Die für ein Datagramm aktivierten Optionen werden von allen Knoten auf der Route zwingend angewendet, denn das ESB-Netz ist experimentell und die Gefahr des Missbrauchs muss im Gegensatz zu einem IP-Netzwerk wie dem Internet nicht berücksichtigt werden. Konkret sind in den Service Flags die folgenden drei Dienstmerkmale enthalten:

- *hohe Priorität:* Datagramme mit hoher Priorität werden nicht im gewöhnlichen Ringpuffer in Warteschlange gestellt, sondern in einem kleinen, speziell erstellten Prioritätsringpuffer abgelegt. Erst nachdem dieser priorisierte Sendepuffer leer geworden ist, kann der Versand der restlichen im normalen Ringpuffer wartenden Datagramme erfolgen.
- *Kontrolle der maximalen Anzahl von Übertragungswiederholungen:* Auf der Sicherungsschicht wird ein nicht bestätigtes Paket standardmäßig bis zu 14 Mal erneut übertragen, man kann jedoch zur Steigerung der Zustellungschancen einen

[1] Wird manchmal auch als TTL *(Time To Live)* bezeichnet.

doppelt oder gar dreimal so hohen Wert einfordern. Auch die Halbierung des Default-Wertes für Neuübertragungen ist möglich.

- *Signalstärke:* Die Stärke des Funksignals und damit seine Reichweite kann in der Firmware variabel eingestellt werden; diese Möglichkeit wird nun für jedes einzelne Datagramm verfügbar gemacht. Es sind sieben Stufen – von *sehr schlecht* bis *hervorragend* – definiert worden, die gewählte Signalstufe kommt dann bei jedem Knoten-zu-Knoten-Transport des Datagramms zur Anwendung.

Für die Realisation dieser Dienstmerkmale mussten insbesondere die Sicherungsschichtroutinen `Net_send()` und `Net_txHandler()` erweitert werden.

Wenn am ESB-Knoten die gesetzte Systemmeldungsstufe das Anzeigen von Netzwerkereignissen beinhaltet (Terminalkommandos `rlo/rlb` und `slo/slb`), so werden sämtliche Aktivitäten auf der Sicherungs- und Vermittlungsschicht im Terminalfenster gemeldet. Zum Testen der Verbindung sind für beide Schichten kleine Pingtools implementiert worden, die das Senden eines Ping-Pakets / Datagramms benutzerdefinierter Größe erlauben, welches im Erfolgsfall dann als Pong-Paket / Datagramm zurückkommt. Der Aufruf erfolgt mit den Terminalkommandos `pps` (Sicherungsschicht) und `dpg` (Vermittlungsschicht), letzteres unterstützt optional auch das Setzen der Service Flags. In Abbildung 3.1 wurde mit dem Befehl `dpg 6 32` von Knoten 7 an Knoten 6 ein Ping-Datagramm mit 32 Byte Nutzdaten geschickt.

5.3 Die statische Routingtabelle

In jedem Knoten auf der Route zwischen Quelle und Ziel wird das Datagramm von der Funktion `RoutingHandler()` dem angekommenen Sicherungsschichtpaket entnommen und durch Aufruf der `SendDatagram()` Routine[2] in einem neuen Paket an die nächste Zwischenstation weitergeschickt. Um ermitteln zu können, welcher Nachbarknoten in Richtung des Ziels liegt und somit das passende Gateway darstellt, muss für jeden Knoten eine Routing-Tabelle gepflegt werden.

Bei statischem Routing werden die Routing-Tabellen entweder manuell angelegt oder von einem Routing-Algorithmus (siehe Kapitel 7) berechnet und dann so lange benutzt, wie sich die Topologie des Netzes nicht ändert. In einem statischen Ad-Hoc-Netz wäre

[2] `SendDatagram()` bekommt einen Zeiger auf den fertigen Datagrammheader sowie auf die zugehörigen Nutzdaten übergeben und kann daher sowohl zum Weiterleiten von erhaltenen als auch zum Aufgeben von neu gebauten Datagrammen verwendet werden.

eine Neuberechnung der Routing-Tabellen nur im Falle eines Knotenausfalls oder einer bewussten Umstrukturierung des Netzes erforderlich.

Die in den ESB-Knoten verwendeten Routing-Tabellen sollten so einfach wie möglich aufgebaut sein, um die CPU nicht zu belasten und einen verzögerungsfreien Versand von Datagrammen zu garantieren. Für jeden Zielknoten wird ein separater Tabelleneintrag angelegt, zusätzlich ist die Verwendung eines *Default-Gateways* möglich. Auf dieses wird zurückgegriffen, sofern in der Routing-Tabelle zu einem Ziel kein expliziter Eintrag existiert. Die Möglichkeit zur Aggregation von Adressen ist bei einer manuellen Erstellung der Routing-Tabellen trotzdem gegeben, bloß werden Regeln mit aggregierten Zielen von `ESB-RTable` (Abschnitt 5.4) intern in einzelne Tabelleneinträge zerlegt. Konkret ist die Routing-Tabelle ein nach den Zielknoten sortiertes Array aus `RoutingTableEntry` – Strukturen:

```
typedef struct
{
    UINT16 destination;   // ID des Zielknotens
    UINT16 gateway;       // ID des zu verwendeten Gateways

} RoutingTableEntry;

//Zeiger, kann aber mittels [] wie ein Array benutzt werden
RoutingTableEntry* RoutingTable;
```

Falls ein Default-Gateway existiert, so ist der zugehörige Tabelleneintrag das erste Element des `RoutingTable` – Arrays und `destination` ist auf *null* gesetzt. Alle direkten Nachbarn eines Knotens sind in der Routing-Tabelle ebenfalls gelistet, bei den betreffenden Einträgen sind `destination` und `gateway` entsprechend identisch.

Unter Ausnutzung der von `ESB-RTable` durchgeführten Sortierung der Routing-Tabelle wird das passende Gateway mittels *binärer Suche* in $O(log\ n)$ gefunden. Die Implementierung der binären Suche erfolgte iterativ und die Effizienz dieses Algorithmus konnte im Vergleich zu den in der Literatur [20] [54] [61] vorgestellten Versionen leicht gesteigert werden:

```
    while (LeftBorder <= RightBorder)
    {
       Middle = (LeftBorder + RightBorder) >> 1; // Division durch 2

       if (destination > RoutingTable[Middle].destination)
          LeftBorder = Middle + 1;
```

```
        else if (destination < RoutingTable[Middle].destination)
            RightBorder = Middle - 1;
        else
            return RoutingTable[Middle].gateway;
    }
```

Anstatt in der Schleifenbedingung oder zu Beginn des Schleifenrumpfs einen Test auf Gleichheit durchzuführen, wird die Gleichheit des gesuchten mit dem gerade inspizierten Schlüssel gefolgert, sofern beide Vergleiche auf *größer* und *kleiner* `false` zurückliefern. Da die Gleichheit bei einer erfolgreichen Suche nur im letzten Schleifendurchlauf gegeben ist, reicht hier in allen vorherigen Durchläufen mit einer Wahrscheinlichkeit von $\frac{1}{2}$ schon der erste Vergleich aus und man spart dann die sonst zwingende zweite Vergleichsoperation ein. Im Mittel benötigt die binäre Suche genau wie im Worst Case logarithmisch viele Iterationen [61]. Daher bringt das Überspringen des zweiten Vergleichs in durchschnittlich der Hälfte aller Schleifendurchläufe[3] eine mittlere Ersparnis in der Größenordnung $\Theta(log\ n)$.

Zum effizienten Design der Routing-Tabelle gehörte auch die Wahl eines geeigneten Speicherorts. Prinzipiell kamen hierfür alle drei vorhandenen Speicher (Flash, RAM und EEPROM) in Frage. Das Ablegen der Routing-Tabelle im RAM wäre in einem kleinen ESB-Netz mit wenigen Dutzend Knoten noch vertretbar, stellte aber insgesamt eine schlecht skalierbare Lösung dar. Im EEPROM könnte auf die Routing-Tabelle hingegen nicht direkt zugegriffen werden, die binäre Suche müsste die zu inspizierenden Tabelleneinträge aus dem EEPROM einzeln auslesen, was zu Verzögerungen führen würde. Daher erfolgt die Unterbringung der Routing-Tabelle in den oberen Segmenten des Flash-Speichers, welche keinen Programmcode[4] mehr enthalten. Dies ermöglicht einen schnellen aber auch direkten Zugriff auf die Tabelleneinträge als Array-Elemente mittels eines Zeigers, wie im Codeausschnitt zur binären Suche bereits zu sehen war.

Es wurden spezielle Routinen bereitgestellt, um zur Laufzeit den noch freien Flash-Speicher löschen und wieder beschreiben zu können. Bevor `ESB-RTable` die fertige Routing-Tabelle über die serielle Schnittstelle transferiert, kündigt das Tool mittels eines internen Terminalkommandos die Größe der Tabelle an und schaltet damit im ESB-Knoten den hierzu eingerichteten binären Empfangsmodus ein. Es wird sodann berechnet, an welcher Adresse im Flash-Speicher das Schreiben der blockweise übermittelten

[3] Sofern die gesuchten Schlüssel in der Menge der gespeicherten Schlüssel gleichverteilt sind.

[4] Dieser belegte nach Fertigstellung der Arbeit weniger als 40 KB, sodass über 20 KB an Flash-Speicher für die Routing-Tabelle und kommende Programmerweiterungen übrig bleiben.

Routing-Tabelle beginnen muss, damit sie gerade zwei Byte vor Ende des letzten nutzbaren Flash-Segments an Adresse `0xFDFD` aufhört. Die beiden letzten Segmentbytes (`0xFDFE` und `0xFDFF`) sind nämlich dazu da, um die eben ermittelte Anfangsadresse der Routing-Tabelle ebenfalls dauerhaft verfügbar zu machen:

```
RoutingTable = *((RoutingTableEntry* *)0xFDFE);
```

Die Anzahl der Tabelleneinträge ist dann implizit durch den Zeiger `RoutingTable`, die Größe der Struktur `RoutingTableEntry` (4 Byte) und das feste Ende der Routing-Tabelle an Adresse `0xFDFD` gegeben:

```
UINT16 NumberOfRoutingEntries = (0xFDFE - (UINT16)RoutingTable) >> 2;
```

Das soeben vorgestellte Verfahren zum Speichern der Routing-Tabelle verzichtet gänzlich auf globale, dauerhaft Arbeitsspeicher belegende Variablen und gestattet der Routine `SendDatagrams()` trotzdem einen effizienten Zugriff auf die Tabelleneinträge. Unabhängig vom Programmcode kann mit `ESB-RTable` jederzeit eine neue Routing-Tabelle von nahezu beliebiger Größe geladen werden. Sie lässt sich dann mit dem Terminalkommando `rrt` anzeigen. Wird der ESB-Knoten jedoch neu programmiert, so löscht das Flash-Tool `msp430-jtag` vorher den gesamten Flash-Speicher durch Setzen aller Bytes auf den Wert `0xFF` und eine Neuübertragung der Routing-Tabelle wird fällig[5]. Eine nicht vorhandene Tabelle erkennt der Knoten daran, dass der Zeiger `RoutingTable` in Folge der Löschoperation auf die ungültige Adresse `0xFFFF` verweist.

5.4 Parsen von Routing-Tabellen mit ESB-RTable

`ESB-Dijkstra` kann in Verbindung mit dem Topologiedienst (Kapitel 7) die optimalen Routing-Tabellen für alle Knoten automatisch erstellen. Manchmal kann aber auch eine manuelle Konfiguration der Tabellen erforderlich sein. Eine bequeme Methode, um Routing-Tabellen von Hand anzulegen, ist das Schreiben einer Routing-Konfigurationsdatei, die für jeden Knoten die entsprechende Tabelle enthält. Das gewählte Format sollte, wie allgemein in Konfigurationsdateien üblich, einen flexiblen Umgang mit Leerzeichen, Tabulatoren sowie leeren Zeilen erlauben. Die syntaktische Korrektheit der Konfiguration muss zudem überprüfbar sein. Anstatt für jedes Format einen neuen Parser zu programmieren, greift man bei professionellem Anwendungsdesign auf einen Parser-Generator zurück. Dies ist ein Hilfsprogramm, das als Eingabe eine Beschreibung der zu

[5] Dies könnte auch vom Makefile automatisch erledigt werden.

erkennenden Texte sowie die Spezifikation der beim Erkennen auszuführenden Aktionen erwartet und als Ausgabe den Quellcode des zugehörigen Parsers liefert.

Bevor auf die Details des Parsens eingegangen wird, soll kurz das für die Routing-Konfigurationsdatei gewählte Format vorgestellt werden:

```
node 6
{
    7 -> 7;
    8 -> 8;
    9 -> 9;
    2 -> 7;
    4 - 6 -> 7;
    1, 3 -> 8;
    0 -> 9;      // Knoten 9 ist das Default-Gateway.
}
node 7
{
    ...
}
```

Die Knoten 7, 8 und 9 sind von Knoten 6 aus direkt adressierbar. Um ein Datagramm an Knoten 2 zu schicken, leitet man es zunächst an Knoten 7 weiter. Die Sensoren mit den IDs 4, 5 und 6 sind ebenfalls über Knoten 7 zu erreichen und Knoten 8 ist das richtige Gateway zum Weiterleiten von Datagrammen an die Sensoren 1 und 3. Wie bereits erwähnt, ist hier von der Möglichkeit, Zieladressen in den Tabelleneinträgen zu aggregieren, Gebrauch gemacht worden. Neben der Angabe eines Adressintervalls ist auch eine Aufzählung von Zielknoten mit gleichem Gateway vorgesehen. Wie am Eintrag zum Default-Gateway zu sehen ist, können in der Konfigurationsdatei Kommentare im Stile von C++ gesetzt werden. Die Verwendung von Leerzeichen, Tabulatoren und Zeilenumbrüchen ist in beliebiger Variation möglich. Entscheidend alleine ist, dass jede Routing-Tabelle von einem `Node x {...}` Block umschlossen wird und die einzelnen Einträge mit einem Semikolon enden.

Zum Erzeugen eines C/C++ Parsers wurden die Tools *Flex* [16] und *Bison* [2] verwendet. Dies sind verbesserte und frei verfügbare Versionen der Compilerbau-Werkzeuge *lex* und *yacc* [31].

Der von *Flex* erzeugte Code ist für die *lexikalische Analyse* des zu parsenden Textes zuständig. Dabei werden einzelne Zeichenketten – auch *Token* genannt – im Text als solche identifiziert und einem bestimmten Typ zugeordnet, anschließend erfolgt die Extraktion ihres *semantischen Werts*. Für die Beschreibung der zu erkennenden Token be-

nutzt man *reguläre Ausdrücke*, und der semantische Wert der Token wird mittels C/C++ Anweisungen aus der String-Variablen `yytext` extrahiert:

```
NUM   [0-9]+       // Definition einer Nummer

//Token-Regel für eine Aufzählung: zwei Nummern mit einem
//Bindestrich und beliebig vielen Leerzeichen dazwischen
{NUM}" "*-" "*{NUM}
{
    yylval.num_range[0] = atoi(yytext);
    yylval.num_range[1] = atoi(strchr(yytext, '-')+1);
    return NUMBER_RANGE;  // Rückgabe des erkannten Typs
}
```

Das Erkennen von Zusammenhängen zwischen den Token gehört einer höheren Analyseebene an, nämlich der *syntaktischen Analyse*. Diese Aufgabe nimmt der von *Bison* erzeugte Quellcode wahr. Hierzu greift er auf die *Flex*-Hauptroutine `yylex()` zurück, um das jeweils nächste Token im Text zu identifizieren. `yylex()` liefert dabei, wie im Beispiel zu sehen, den Typ des erkannten Tokens zurück und speichert dessen semantischen Wert in der globalen Variablen `yylval`.

Genau wie *Flex* benötigt auch *Bison* eine Beschreibungsdatei, die eine *kontextfreie Grammatik* in Backus-Naur-Form und C/C++ Code zum Neuberechnen von semantischen Werten bei Regelanwendungen enthält. Die *Terminalsymbole* sind dabei die im Rahmen der lexikalischen Analyse definierten Tokenklassen, beispielsweise werden alle Adressintervalle durch das Terminalsymbol `NUMBER_RANGE` repräsentiert. Die frei wählbaren *Nichtterminalsymbole* dienen hingegen dazu, um die erkannten Terminalsymbole zu einem komplexeren Objekt zu aggregieren. Später können dann auch mehrere Nichtterminale rekursiv durch ein einziges Nichtterminal ersetzt werden. Anstatt anhand der Grammatikregeln den zu parsenden Text aus dem Startsymbol abzuleiten, wird versucht, den Text *bottom up* auf das Startsymbol zurückzuführen und somit zu zeigen, dass er korrekt aufgebaut ist. Nebenbei werden die einzelnen Textobjekte (hier die Routing-Einträge) identifiziert und für die spätere Bearbeitung gespeichert. Den semantischen Wert der Terminale liefert `yylex()` und der semantische Wert von Nichterminalen wird durch die den Grammatikregeln beigestellten C/C++ Anweisungen berechnet:

```
routing_rule:        NUMBER '-' '>' NUMBER ';'
{
    $$ = new vector<RoutingTableEntry>;
    RoutingTableEntry entry;
```

```
    entry.destination = $1;
    entry.gateway = $4;
    $$->push_back(entry);  // Einfügen in den leeren Vektor
}
```

Während das Terminal `NUMBER` alle möglichen Nummern umfasst, sind `'-'`, `'>'` und `';'` keine Tokenklasssen, sondern repräsentieren unmittelbar die zugehörigen ASCII-Zeichen. Die obige Regel besagt, dass eine Zeichenfolge von der Form „*Nummer1 -> Nummer2;*“ auf das Nichtterminal `routing_rule` reduziert werden kann. Der semantische Wert der linken Regelseite (stets ein Nichtterminal) wird in der Variablen `$$` gespeichert, während auf den semantischen Wert des x-ten Symbols auf der rechten Regelseite mit `$x` zugegriffen werden kann. Das Nichtterminal `routing_rule` hat als semantischen Wert einen STL-Vektor[6] aus `RoutingTableEntry` – Strukturen (bereits vorgestellt), denn in den beiden anderen Regeln zu `routing_rule` produziert eine Routing-Direktive beim Auflösen der Adressaggregation mehrere Tabelleneinträge:

```
routing_rule:       NUMBER_SEQUENCE '-' '>' NUMBER ';'
routing_rule:       NUMBER_RANGE '-' '>' NUMBER ';'
```

Nachdem gezeigt wurde, wie mehrere Token zu einem Nichtterminal zusammengefasst werden, soll nun demonstriert werden, wie `routing_rule` – Nichtterminale *rekursiv* auf ein einziges Nichtterminal `routing_rules` reduziert werden können:

```
routing_rules:     routing_rule
{
    //Übernahme des semantischen Werts: Vektor-Zeiger kopieren
    $$ = $1;
}
                |  routing_rules routing_rule
{
    $$ = $1;    // den zu routing_rules gehörenden Vektor übernehmen
    for (unsigned int i=0; i < $2->size(); i++)
       //und die Einträge aus dem routing_rule Vektor hinzufügen
       $$->push_back( (*$2)[i] );
    delete $2;  // Vektor zu routing_rule löschen
}
```

Nach der Aggregation aller Routing-Regeln zu `routing_rules` kann durch Hinzunehmen der Begrenzungsklammern und des `NODE` – Terminalsymbols[7] die komplette Routing-Tabelle mit dem Nichtterminal `routing_table` identifiziert werden:

[6] Eigentlich ein Zeiger auf den Vektor.

[7] Repräsentiert Zeichenketten der Form `"node x"`. Semantischer Wert ist die ID des Tabellenbesitzers.

```
routing_table:      NODE '{' routing_rules '}'
{
  /*Falls es sich hierbei um die Routing-Tabelle für den angeschlosse-
  nen Knoten handelt, so stelle sie dem Hauptprogramm über die globale
  Variable   vector<RoutingTableEntry>* RTable   zur Verfügung.*/
}
```

Die STL-Datenstruktur `vector` garantiert, dass n Elemente in beliebigen Häppchen in linearer Zeit eingefügt werden können [58]. Daher eignet sie sich gut, um die beim Parsen enstehenden Tabelleneinträge aufzunehmen, deren Anzahl ja von vorneherein gänzlich unbekannt ist. Beim Reduzieren von `routing_rule` – Nichtterminalen auf das Nichtterminal `routing_rules` muss jeder Tabelleneintrag (mit Ausnahme des allerersten) genau einmal zusätzlich kopiert werden. Durch die Verwendung eines Zeigers auf die Vektorstruktur wird der semantische Wert des ersten Nichtterminals auf der rechten Regelseite in konstanter Zeit übernommen und am Ende erfolgt die Rückgabe der fertigen Routing-Tabelle an das Hauptprogramm ebenso effizient. Da die Vektorelemente intern in einem gewöhnlichen C-Array abgelegt sind [60], kann das Hauptprogramm dann mittels `&(*RTable)[0]` auf den zugehörigen Speicherblock direkt zugreifen und die Routing-Tabelle an den angeschlossenen ESB-Knoten binär übertragen. Mit der CRC-16 Prüfsumme wird anschließend noch verifiziert, ob die vom ESB-Knoten geflashte Routing-Tabelle mit der übertragenen tatsächlich übereinstimmt.

5.5 Automatische Verteilung von Routing-Tabellen

In der bisher vorgestellten Variante muss jeder ESB-Knoten an einen Rechner angeschlossen werden, damit `ESB-RTable` dessen Routing-Tabelle aus der Konfigurationsdatei extrahiert und überträgt. Bei einer größeren Anzahl von Knoten ist diese Vorgehensweise mit einem hohen Arbeitsaufwand verbunden. Daher wurde ein Verfahren entwickelt, um das Verteilen der Routing-Tabellen genau wie das Sammeln der Nachbartabellen im Topologiendiest (Kapitel 7) *über Funk* durchzuführen. Die Verteilung erfolgt in Paketen der Sicherungsschicht von einem am PC angeschlossenen ESB-Knoten, dem *Masterknoten*. Neben dem Masterknoten selbst werden alle Ziele in dessen Routing-Tabelle mit ihren eigenen Tabellen versorgt, diese müssen in der Routing-Konfigurationsdatei dementsprechend vorhanden sein. Die Verteilungspfade zu den einzelnen Knoten werden mit Hilfe der restlichen Routing-Tabellen automatisch berechnet. Sind aber die Routing-Tabellen bewusst nicht optimal gewählt, so kann der vollständige Pfad zu einem Ziel auch manuell vorgegeben werden:

```
node 6   // Masterknoten
{
    7 -> 7;
    2-4 -> 7; // Pfad wird aus den anderen Routing-Tabellen ermittelt
    8 -> 7;   [6 -> 7 -> 4 -> 8] // manuelle Pfadvorgabe für Knoten 8
}
```

Zum Erkennen von Routing-Tabellen mit Pfaden existiert in der *Bison*-Grammatik eine zusätzliche Regel und der C/C++ Code für die anderen Grammatikregeln weicht daher in Wirklichkeit von der im letzten Abschnitt vorgestellten Version minimal ab. In den eckigen Klammern können wie üblich beliebig viele Leerzeichen benutzt werden.

Im Verteilungsmodus (Aufruf mit Parameter `-d`) überprüft `ESB-RTable` zunächst, ob für alle Ziele in der Routing-Tabelle des Masterknotens auch Routing-Tabellen vorhanden sind und berechnet – falls nicht bereits vorgegeben – die Verteilungspfade. Darauf wird ein aus mehreren *Subpackages* bestehendes *Verteilungspackage* aufgebaut und in den EEPROM-Speicher des Masterknotens übertragen. Jedes *Subpackage* enthält den kompletten Pfad zu einem Zielknoten und sämtliche Routing-Tabellen für die Knoten entlang dieses Pfades, sofern nicht einige Tabellen bereits in einem früher zusammengesetzen Subpackage gespeichert worden sind. Zum Aufbauen der Subpackages werden die Zielknoten nach ihrer Pfadlänge *absteigend* sortiert und dann in dieser Reihenfolge berücksichtigt. Die ersten Subpackages sollen so möglichst viele Knoten mit Routing-Tabellen versorgen, damit nur für wenige Zielknoten ein separates Subpackage erforderlich wird. Die Zahl der benötigten Subpackages ist damit in der Regel kleiner als die Anzahl der Ziele, was die Menge der zu versendenden Sicherungsschichtpakete minimiert und die Verteilung beschleunigt. Wenn ein Subpackage größer als 64 Byte ist, wird es auf mehrere Pakete aufgeteilt.

Der Masterknoten empfängt über die serielle Schnittstelle neben dem eigentlichen Verteilungspackage auch seine eigene Routing-Tabelle. Nachdem er sie geflasht hat, schickt er die Subpackages Stück für Stück an den nächsten Knoten auf dem jeweiligen Pfad. Alle anderen Knoten verfahren bei Erhalt eines Subpackages folgendermaßen: Ist das Subpackage auf mehrere Pakete aufgeteilt, dann wird es zuerst blockweise im EEPROM-Speicher abgelegt, ansonsten kann es direkt im RAM inspiziert werden. Wenn das Subpackage eine Routing-Tabelle für den betreffenden Knoten enthält, so wird sie geflasht und dem Subpackage entnommen. Anderenfalls schickt der Knoten das Subpackage nach dem obligatorischen Entfernen der eigenen ID aus dem Pfad direkt an den nächsten Knoten auf der Verteilungsroute weiter. Der letzte Knoten auf der Route verwirft schließlich nach Entnahme seiner Routing-Tabelle das leer gewordene Subpackage.

Kapitel 6

Protokollierungsmechanismen

Zentraler Bestandteil von Überlastexperimenten ist eine detaillierte und genaue Protokollierung des erzeugten Datagrammverkehrs. Dabei speisen die Knoten während einer gewissen Zeitspanne periodisch Datagramme in das Netzwerk ein[1], und die gesammelten Protokolldaten werden anschließend automatisiert ausgewertet.

6.1 Strategien der Protokollierung

Für die Speicherung von Protokolldaten ist der im Normalbetrieb leere, 64 KB große EEPROM-Speicher geradezu prädestiniert. Das gewählte Protokollformat zeigt Tabelle 6.1. In jedem Protokolleintrag sollten einerseits möglichst viele Informationen untergebracht werden, andererseits war es wünschenswert, dass der vorhandene Speicherplatz auch für einen längeren Protokollierungszeitraum ausreicht. Der von den einzelnen Protokollpunkten zu belegende Speicher wurde daher bitgenau kalkuliert. So stehen für die als Adresse dienende Knoten-ID anstatt der üblichen 16 nur 4 Bit zur Verfügung, was eine der Praxis entsprechende Beschränkung des ESB-Netzes auf 15 Knoten bedeutet. Bei vier Adressfeldern macht die damit pro Protokolleintrag erzielte Ersparnis ganze 6 Byte aus. Weiterhin wurden im Zeitstempel für die Sekunden nur 12 Bit reserviert, während die softwarebetriebene Uhr des ESB-Knotens die Sekunden in einem 32-Bit Integer speichert und damit neben der Uhrzeit auch das Datum darstellen kann. Der Zeitstempel erfasst die seit dem Start des Überlastexperiments verstrichene Zeit und reicht für einen einstündigen Protokollbetrieb. Der Beginn eines Überlastexperiments wird mit der Synchronisation der Uhren (siehe nächsten Abschnitt) eingeläutet.

Ein einzelner Protokolleintrag belegt insgesamt nur 8 Byte. In der C-Implementierung

[1] Den Datagrammverkehr erzeugt die Routine `ProduceCongestion()`. Sie ist für das konkrete Experiment anzupassen und wird mit den Terminalkommandos `gtr` und `str` initiiert bzw. gestoppt.

Protokollpunkt	***Bitanzahl***	***Bemerkung***
Datagramm-ID	16	wurde in den Datagramm-Header speziell für die eindeutige Identifikation bei der Protokollierung aufgenommen
Richtung	1	empfangenes oder versendetes Datagramm
Zeitstempel	22	10 Bit für die Millisekunden *(1024 ms $\widehat{=}$ 1 s)* und 12 Bit für die Sekunden
Ziel	4	Adressat des Datagramms
Quelle	4	ursprünglicher Absender des Datagramms
MAC-Empfänger	4	Knoten, an den das in einem Sicherungsschichtpaket verkapselte Datagramm gerade weitergeleitet wird
MAC-Sender	4	Knoten, der das Sicherungsschichtpaket mit dem Datagramm abgeschickt hat
Hop Counter	4	bei eingehenden Datagrammen bereits um eins dekrementiert
Neuübertragungen	4	Anzahl der Übertragungswiederholungen auf der Sicherungsschicht, nur für ausgehende Datagramme gültig
zugestellt	1	null, falls das Datagramm trotz Übertragungswiederholungen an den nächsten Hop nicht zugestellt werden konnte, nur für ausgehende Datagramme gültig

Tabelle 6.1: Das Protokollformat

wird er durch eine Struktur mit Bitfeldern beschrieben. Bitfelder ermöglichen es dem Programmierer, auf einzelne Bits oder Bitgruppen genauso bequem zuzugreifen wie auf gewöhnliche Strukturelemente. Um die intern stattfindenden Schiebe- und Maskierungsoperationen kümmert sich der Compiler. Insgesamt finden im EEPROM-Speicher über 8000 Protokolleinträge Platz.

Es sind drei Protokollierungsoptionen definiert worden: *eingehende*, *ausgehende* oder *fremde (Promiscuous Mode)* Datagramme erfassen. Diese drei Optionen lassen sich mit dem Terminalkommando `srl` in beliebiger Kombination setzen und können mit `rrl` ausgelesen werden. Eine vierte von den beiden Terminalkommandos unterstützte Option gestattet es, für Datagramme die auf der Sicherungsschicht realisierte zuverlässige Übertragung (ACKs in Verbindung mit Übertragungswiederholungen) abzuschalten. Dann werden Sicherungsschichtpakete, die Datagramme transportieren, nur einmalig verschickt und vom Empfänger nicht bestätigt. Regelmäßige Übertragungswiederholungen z. B. im Zusammenhang mit einem ausgefallenenen oder nicht erreichbaren Knoten kosten beim Weiterleiten unnötig viel Zeit und können bei Überlastexperimenten in bestimmten Konstellationen unerwünscht sein. Der Protokollpunkt *Neuübertragungen* hat in dem Fall den Wert null und *zugestellt* wird auf eins gesetzt.

Genau wie das Routing erfolgt das Schreiben der Protokolleinträge in der Anwendung. Da der Zeitstempel aber den Moment erfassen sollte, in dem das in einem Sicherungs-

schichtpaket verkapselte Datagramm physikalisch angekommen bzw. vollständig verschickt worden ist[2], wird die zugehörige Zeit bereits in der rx- bzw. tx-Interruptroutine in einer globalen Variablen erfasst. Die Protokollierungsroutinen berechnen dann, wie viele Sekunden und Millisekunden zwischen dem Beginn des Experiments und dem gemessenen Sende- oder Empfangszeitpunkt verstrichen sind.

Der Protokollierungsbetrieb soll den Empfang und Versand von Datagrammen so wenig wie möglich behindern oder verzögern. Daher werden die Protokolleinträge zunächst in einem globalen Puffer zwischengespeichert und erst anschließend ins EEPROM geschrieben, sodass es nur hin und wieder zu kurzen Verzögerungen kommt. Die optimale Dimensionierung des Protokollpuffers mit 128 Byte (16 Protokollierungseinträge) ergibt sich aus der Messreihe im Anhang B.1. Beim Leeren des Protokollpuffers wird der ESB-Knoten aber nicht 10 ms lang vollständig blockiert. Denn während der EEPROM-Speicher beschrieben wird, sind die Interrupts nach wie vor aktiv. Daher können die rx- und tx-Interruptroutinen in der Zwischenzeit ein neues Paket empfangen oder versenden, sofern sie sich im richtigen Zustand befinden. Konkret bedeutet dies, dass beim Schreiben ins EEPROM der Empfangspuffer geleert und das zuletzt gesendete Paket aus dem Senderingpuffer entfernt sein muss. Diese Bedingungen sind beim Erstellen eines Protokolleintrags in der Anwendung offenbar nicht erfüllt, es wird ja gerade das zuletzt gesendete oder empfangene Paket inspiziert. Darum schreiben die Protokollierungsroutinen ausschließlich in den Protokollpuffer. Der Transfer eines vollen Protokollpuffers in den EEPROM-Speicher erfolgt dann separat in der Hauptschleife der Firmware nach Aufruf der beiden Handler-Funktionen für den Versand und Empfang von Paketen.

6.2 Zeitsynchronisation

Erst die Summe der einzelnen in den Knoten erstellten Protokolle ergibt ein vollständiges Bild des gesamten Datagrammverkehrs. Von großer Bedeutung für die Analyse der Protokolle ist ein möglichst synchroner Zeitstempel. Für ein Datagramm sollten im Idealfall die Zeitstempel in den Protokollen des MAC-Senders und MAC-Empfängers übereinstimmen. Dies setzt natürlich voraus, dass die Uhren der ESB-Knoten synchron laufen. `ESB-RTable` stellt beim Übertragen der Routing-Tabelle auf Wunsch auch die Knotenuhr ein, jedoch lediglich mit einer Auflösung von einer Sekunde. Daher wurde folgendes Vefahren für die millisekundengenaue Zeitsynchronisation entwickelt:

[2] Wenn ACKs mit Übertragungswiederholungen eingeschaltet sind, ist der Zeitpunkt des letzten erfolgreichen oder erfolglosen Sendeversuchs maßgeblich.

1. Ein zum *Masterknoten* ernannter ESB-Knoten hat nach der Einstellung der Uhr durch `ESB-RTable` die Referenzzeit.
2. Die mit ihm zu synchronisierenden Knoten werden alle in seine Nähe gebracht. Mit dem Terminalbefehl `syn` wird vom Masterknoten ein Synchronisationspaket mit der Referenzzeit als Broadcast versendet.
3. Die anderen Knoten addieren zur empfangenen Referenzzeit die während der Übertragung des Synchronisationspakets verstrichene Zeitspanne hinzu und setzen danach ihre Uhr. Die korrekte Zeitspanne konnte durch Elimination des Synchronisationsfehlers (Punkt 4) experimentell bestimmt werden.
4. Zum Messen des Synchronisationsfehlers wird von einem der Knoten ein beliebiges Datenpaket als *Broadcast* verschickt, welches alle anderen Knoten gleichzeitig empfangen. Der Eingangszeitpunkt wird in der rx-Interruptroutine erfasst und mit dem Terminalkommando `rxt` auf dem Masterknoten sowie auf einem der anderen synchronisierten Knoten ausgelesen. Die Abweichung sollte nicht mehr als eine Millisekunde betragen.

Natürlich driften die Uhren in den ESB-Knoten nach erfolgter Synchronisation mit der Zeit auseinander. In einer Messreihe (Anhang B.2) wurden die anfangs synchronen Uhren von sechs Knoten über einen Zeitraum von 20 Minuten beobachtet. Die maximale Abweichung betrug am Ende nur 10 ms – ein durchaus zufriedenstellender Wert.

6.3 Mit ESB-RLog die Protokolldateien speichern

Nach Beendigung eines Überlastexperiments werden die gesammelten Protokolldaten auf den PC übertragen. Den Datentransfer und das Speichern der Logdatei (binär oder auch im Textformat) erledigt das Hilfsprogramm `ESB-RLog`. Auf Wunsch werden dabei die Millisekunden des Knotens in echte Millisekunden umgerechnet. Ein Protokollausschnitt im Textformat sieht zum Beispiel folgendermaßen aus:

```
0387.272 OUT MAC: 06 -> 07 (+)[01]  DATAGRAM: 06 -> 09  TTL 10  ID 191
0387.522 IN  MAC: 07 -> 06          DATAGRAM: 02 -> 06  TTL  7  ID 256
0387.983 --- MAC: 04 -> 07          DATAGRAM: 04 -> 02  TTL 10  ID 487
```

Der erste Eintrag ist der Zeitstempel, welcher hier bereits in echte Millisekunden umgerechnet worden ist. Sonst wären die Millisekunden von den Sekunden durch einen Doppelpunkt getrennt und vierstellig. `"---"` steht für ein im *Promiscuous Mode* aufgeschnapptes Datagramm. Einer Erklärung bedürfen noch die beiden Klammern bei ausgehenden Datagrammen: `"(+)"` bzw. `"(-)"` entsprechen dem Protokollpunkt *zugestellt* und die Zahl in den eckigen Klammern ist die Anzahl der erfolgten *Neuübertragungen*.

Kapitel 7

Der Topologiedienst und ESB-Dijkstra

Das Verhalten eines ESB-Netzes bei Überlast soll für unterschiedliche Topologien untersucht werden. Anstatt für eine gegebene Anordnung von ESB-Knoten alle Routing-Tabellen mühsam von Hand anzulegen, kann auch ein Routing-Algorithmus mit dieser Aufgabe betraut werden. Darum wurde zunächst ein Topologiedienst entwickelt, der durch Sammeln aller Nachbarschaftsbeziehungen den dem ESB-Netz zugrundeliegenden Graphen aufstellt. Aus diesem Graphen berechnet dann das PC-Hilfsprogramm `ESB-Dijkstra` mittels eines globalen Link-State-Routing-Algorithmus die optimalen Routing-Tabellen für alle Knoten.

7.1 Funktionsweise des Topologiedienstes

7.1.1 Der Algorithmus

Die prinzipielle Idee besteht darin, mit einem am PC angeschlossenen *Masterknoten* alle Knoten im ESB-Netz zum Erkennen ihrer Nachbarn aufzufordern und die dabei erstellten Nachbartabellen im EEPROM-Speicher dieses Masterknotens zu sammeln. Die Aufforderung, eine Nachbarsuche durchzuführen, wird mit Hilfe eines sogenannten *Tokens* realisiert. Bei Erhalt des Tokens identifiziert ein Knoten seine Nachbarn und reicht dann das Token an sie weiter. Das Token muss dabei von einem Nachbarn zurückgekehrt sein, bevor es an den nächsten weitergegeben werden kann, sodass im gesamten ESB-Netz nur ein einziges Token kursiert. Das Weiterreichen des Tokens an die Nachbarn erfolgt *rekursiv* und bei der Tokenrückgabe wird ein Bündel von Nachbartabellen mitgeschickt. Es enthält neben der Nachbartabelle des zurückgebenden Knotens auch alle Tabellen, die dieser Knoten nach dem selben Prinzip von seinen Nachbarn erhalten hat. Falls aber einem Knoten das Token wiederholt gereicht wird, so gibt er es ohne die Nachbartabel-

len sofort zurück, um das doppelte Sammeln von Tabellen zu verhindern. Der hiermit beschriebene Algorithmus realisiert die komplette Traversierung des Netzes mittels einer verteilt ablaufenden *Tiefensuche* [61]. Er kann etwas formaler folgendermaßen in Pseudocode formuliert werden:

```
if (Token erstmalig erhalten)
{
    Suche alle Nachbarn und erstelle eine Nachbartabelle;
    for all (n Nachbar && n != TokenSender)
    {
        reiche das Token an n;
        warte auf die Rückkehr des Tokens;
        if (Nachbartabellen beigefügt)
            speichere dieses Tabellenbündel im EEPROM;
    }
    return (Token + eigene Nachbartabelle + gespeicherte Tabellen)
           to TokenSender;
}
else
    return Token to TokenSender;
```

Initial reicht der Masterknoten das Token nach Durchführung der Nachbarsuche an einen seiner Nachbarn weiter und wartet darauf, dass es mit einem Bündel von Tabellen zurückkommt. Dann nimmt er sich analog die anderen Nachbarn vor und am Ende sind in den angesammelten Bündeln die Nachbartabellen sämtlicher transitiv erreichbarer Knoten enthalten. Sie ergeben zusammen den Graphen des aufgebauten ESB-Netzes. Abbildung 7.1 demonstriert die Arbeitsweise des Topologiedienstes an einem Beispiel. Zu Beginn des neunteiligen Sequenzausschnitts hat Masterknoten 1 bereits seine beiden Nachbarn (Knoten 2 und 3) erkannt und das Token darauf an Knoten 3 gereicht.

7.1.2 Technische Details

Der Versand des Tokens erfolgt in einem Paket der Sicherungsschicht. Wenn die angehängten Tabellen mehr als 64 Byte belegen, werden sie genau wie die Subpackages bei der Verteilung von Routing-Tabellen (Abschnitt 5.5) auf mehrere Pakete aufgeteilt. Somit skaliert der implementierte Topologiedienst gut und kann auch in größeren ESB-Netzen eingesetzt werden. Die Zwischenspeicherung der Tabellenbündel findet im EEPROM statt. Die aktuelle Position wird durch eine Analyse des EEPROM-Inhalts gewonnen, indem nach einer längeren Kette von auf 0xFF gesetzten Bytes gesucht wird. Dies ist der Beginn von freiem Speicherplatz. Man spart so eine globale Variable ein.

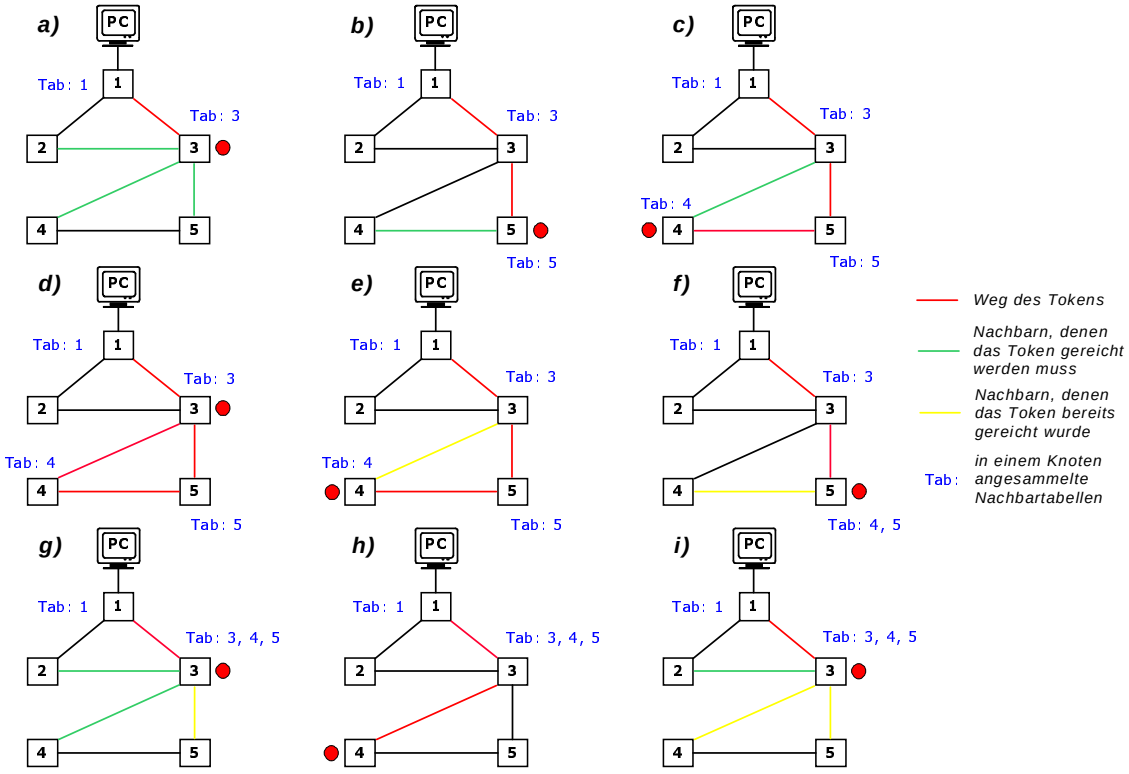


Abbildung 7.1: Der Topologiedienst am Beispiel

Zum Erkunden seiner Nachbarn versendet ein Knoten in Intervallen von 750 ms sieben Mal ein Topologie-Ping-Paket als Broadcast, welches von allen anderen Knoten in Reichweite mit einem Topologie-Pong-Paket beantwortet wird. Mittels eines primitiven Zufallszahlgenerators wird sichergestellt, dass nicht alle Nachbarn das Pong gleichzeitig verschicken, sondern zeitlich versetzt im Laufe von 500 ms nach dem Empfang des Pings antworten. Zur Beurteilung der Verbindungsqualität wird in der Nachbartabelle gezählt, wie oft welcher Knoten auf die insgesamt sieben Pings geantwortet hat. Aus naheliegenden Gründen ist für Topologie-Ping- und Topologie-Pong-Pakete die zuverlässige Datenübertragung deaktiviert. Die Wahl der Zeitintervalle und der Struktur der Tabelleneinträge (5 Bit für die Knoten-ID und 3 Bit für den Pong-Zähler) erfolgte im Hinblick auf Einsätze in einem kleinen ESB-Netz.

Die Topologieerkennung wird am Masterknoten mittels des Terminalkommandos `top` initiiert und kann auf den einzelnen Knoten anhand der LEDs verfolgt werden: Bei der Durchführung der Nachbarsuche wird das *grüne*, beim Weiterreichen des Tokens das *gelbe* und nach Rückgabe des Tokens das *rote* Lämpchen eingeschaltet. Der Masterknoten beendet die Topologieerkennung mit einem akustischen Signal.

7.2 ESB-Dijkstra und effiziente Pfadalgorithmen

Die im EEPROM-Speicher des Masterknotens gesammelten Nachbartabellen werden von `ESB-Dijkstra` auf den Rechner transferiert. Anschließend erfolgt für jeden Knoten die Berechnung der optimalen Routing-Tabelle mittels *Dijkstras Link-State-Routing-Algorithmus* [9]. Da die Nachbartabellen mit dem Pong-Zähler auch eine Aussage über die Qualität der einzelnen Links machen, lag es nahe, diese Information in einem Kostenmaß umzusetzen. Wenn mindestens die Hälfte der Pings beantwortet wurden, teilt `ESB-Dijkstra` dem entsprechenden Link Einheitskosten zu. Sonst werden bei Links zu Nachbarn, von denen in nicht weniger als einem Drittel der Fälle Pongs zurückkamen, doppelte Kosten angesetzt und in den restlichen Fällen erfolgt die Zuweisung der vierfachen Kosten. Eine Zusatzoption erlaubt es, die schlechten Links zu ignorieren und nur Einheitskosten zu verwenden. In diesem Fall kommt zur Berechnung der Routing-Tabellen die *Breitensuche* [61] zum Einsatz. Als Ausgabe schreibt `ESB-Dijkstra` eine Routing-Konfigurationsdatei, die als Zusatzinformation auch die kompletten Pfade und deren Kosten enthält. Sie dient als Eingabe für `ESB-RTable`.

Der Dijkstra-Algorithmus ist für die Berechnung der kürzesten Wege von *einem* Startknoten zu allen anderen Knoten ausgelegt, was in der Literatur als das *Single Source Shortest Paths (SSSP)* Problem bezeichnet wird. Mit der Berechnung sämtlicher Routing-Tabellen liegt hier aber eigentlich das *All Pairs Shortest Paths (APSP)* Problem vor. Eine direkte Lösung dieses Problems erlaubt der klassische Floyd-Warshall-Algorithmus [54] in $O(n^3)$[1], und es existieren hierfür auch neuere, etwas effizientere Algorithmen, wobei der derzeit schnellste [6] in $O(n^3/\ log\ n)$ läuft[1]. Indirekt wird das APSP-Problem natürlich auch durch n-maliges Anwenden des Dijkstra-Algorithmus gelöst. Wird dieser mittels eines effizienten *Heaps* (auch *Prioritätswarteschlange* genannt) implementiert, so ist er auf einem dünnen Graphen[2] sogar schneller als die direkten APSP-Algorithmen. In `ESB-Dijkstra` kam eine frei verfügbare C++ Implementierung [3] des Fibonacci-Heaps [17] zum Einsatz. Mit diesem Heap erfordert ein Durchlauf von Dijkstras Algorithmus $O(m + n\ log\ n)$ viele Schritte[1], wobei m die Zahl der Kanten und n die Anzahl der Knoten im Graphen ist. Ein asymptotisch noch effizienterer Heap für das SSSP-Problem ist in [62] vorgestellt. In der Praxis bringen komplexe Heaps aber nur bei sehr großen Graphen Geschwindigkeitsvorteile. Im Falle eines Graphen mit Einheitskosten weicht `ESB-Dijkstra` zum Berechnen der Routing-Tabellen auf die mittels einer gewöhnlichen Warteschlange realisierte *Breitensuche* aus, die in $O(m+n)$ läuft[1].

[1] Zu jedem Ziel werden die Kosten und das Gateway ermittelt. Die Berechnung der Pfade erfolgt separat.
[2] Bei fixierter Knotenzahl n ist hier die Anzahl der Kanten deutlich kleiner als der Maximalwert $n(n-1)$.

Kapitel 8

Resümee

Diese Bachelorarbeit hat alle Voraussetzungen geschaffen, um auf den ESB-Sensorknoten Überlastexperimente durchführen zu können.

Die bereitgestellte *Vermittlungsschicht* ermöglicht mit statischem Routing eine Multihop-Kommunikation zwischen den ESB-Knoten, wobei spezielle Dienstmerkmale die Zustellungschancen von besonders wichtigen Datagrammen erhöhen können. Zum Festhalten des gesamten Datagrammverkehrs steht ein konfigurierbarer *Protokollierungsdienst* zur Verfügung, der dank der Wahl eines platzsparenden Formats in jedem Knoten über mehr als 8000 Datagramme Buch führen kann. Die Auswertung und der Vergleich von Protokollen werden durch eine dem Experiment vorangehende *Synchronisation der Knotenuhren* erleichtert. Ein auf Tiefensuche basierender *Topologiedienst* kann für eine gegebene Anordnung von ESB-Knoten sämtliche Nachbarschaftsbeziehungen und die Qualität der Links ermitteln.

Mit den PC-Hilfsprogrammen `ESB-RTable`, `ESB-RLog` und `ESB-Dijkstra` sind drei Werkzeuge gegeben, um die ESB-Knoten unter Minimierung des Arbeitsaufwandes für die Überlastexperimente vorzubereiten (Routing-Tabellen auf Wunsch automatisiert erstellen und über Funk verteilen) sowie um die gesammelten Protokolldaten anschließend am Rechner analysieren zu können.

Bei allen Implementierungen stand der *algorithmische Aspekt* im Vordergrund. Auf den ESB-Knoten wird mittels einer leicht optimierten binären Suche das schnelle Durchsuchen der im Flash-Speicher abgelegten Routing-Tabelle ermöglicht. Der Topologiedienst und die Verteilungsmaschinerie für Routing-Tabellen sind so implementiert, dass sie auch für Einsätze in größeren ESB-Netzen geeignet sind. In `ESB-RTable` ist eine professionelle Technik zum *Parsen von Konfigurationsdateien* verwendet worden. Schließlich erfolgte in `ESB-Dijkstra` eine effiziente Umsetzung von *Dijkstras Routing-Algorithmus* mittels eines Fibonacci-Heaps.

Anhang A

Terminalkommandos

Die folgenden Tabellen geben eine Übersicht der wichtigsten Terminalkommandos. Die fett gedruckten Befehle waren nicht Teil der ursprünglichen ScatterWeb Software sondern wurden im Rahmen dieser Bachelorarbeit hinzugefügt. Manche Befehle haben einen oder mehrere Parameter, die durch ein Leerzeichen getrennt dem jeweiligen Kommando hinten angefügt werden. Falls der Parameter eine Bitmaske (8 Bit) oder eine Adresse (16 Bit) ist, so erfolgt die Eingabe als zwei- bzw. vierstellige Hexadezimalzahl. Fast alle Terminalkommandos können auch auf entfernten Sensorknoten ausgeführt werden, wenn man dem eigentlichen Kommando „@X “ davorgestellt, wobei X die *dezimale* ID des entfernten Knotens ist. Die wenigen Befehle, bei denen nur eine lokale Ausführung möglich ist, sind mit dem Sternsymbol ⋆ versehen.

Kommando	Parameter	Beschreibung
pps⋆	x (y)	schickt ein Ping-Paket der Sicherungsschicht an Knoten x optional: y Datenbytes mitsenden
dpg⋆	x (y) (z)	schickt ein routbares Ping-Datagramm an Knoten x optional: y Datenbytes mitsenden und Service-Flags-Bitmaske auf z setzen Service-Flags-Bitmaske: • `0x01` – *hohe Priorität* • 7 Stufen für die Signalstärke: `0x00` – *default* `0x02` – *very bad* `0x04` – *bad* `0x06` – *medium1* `0x08` – *medium2* `0x0A` – *good* `0x0C` – *very good* `0x0E` – *excellent* • 4 Stufen für Übertragungswiederholungen: `0x00` – *default* `0x10` – *0,5·default* `0x20` – *2·default* `0x30` – *3·default*
dps⋆	x (y) (z)	schickt ein routbares Datagramm an Knoten x optional: y Datenbytes mitsenden und Service-Flag-Bitmaske auf z setzen
rrt⋆		zeigt die Routing-Tabelle an
rrl		zeigt die Einstellungen für Routing und die Protokollierung an
srl	x	setzt die Bitmaske für die Routing- und Protokollierungseinstellungen auf x `0x01` – *keine Link-Layer-ACKs für Datagramme* `0x02` – *eingehende* `0x04` – *fremde (Promiscuous Mode)* `0x08` – *abgehende* Datagramme protokollieren
top⋆		startet die Maschinerie zur Erkundung der Topologie des gesamten Netzes
syn⋆		synchronisiert die Uhren aller anderen Knoten in Reichweite mit der eigenen Uhr und fixiert diesen Zeitpunkt als den Beginn des Überlastexperiments
rxt⋆		zeigt die Zeit des letzten Paketempfangs an (nützlich, um z. B. die durchgeführte Synchronisation der Uhren zu beurteilen)
ext		zeigt den Zeitpunkt an, an dem das Überlastexperiment begann (vgl. `syn`)
gtr		fängt an, für das Überlastexperiment periodisch Datagramme zu verschicken
str		beendet den periodischen Versand von Datagrammen
rtp		gibt die gegenwärtig eingestellte Stärke des Funksignals an
stp	x	setzt die Stärke des Funksignals auf x *(0 – 100)*
rfr		zeigt die für den Funkempfänger eingestellte Schwellspannung an
sfr	x	setzt die Schwellspannung des Funkempfängers auf x *(0 – 4095)*
rrp		gibt die am Funkempfänger aktuell induzierte Spannung aus

Tabelle A.1: Terminalkommandos Funkschnittstelle

Kommando	Parameter	Beschreibung
epr	x	liest x Bytes aus dem EEPROM-Speicher und misst die dazu benötigte Zeit
epw	x	schreibt x Bytes in den EEPROM-Speicher und misst die dazu benötigte Zeit
rid		liest die ID des Sensorknotens aus
sid	x	setzt die ID des Sensorknotens auf x *(dezimal eingeben!)*
rlo		zeigt die gegenwärtig aktive Stufe für Systemmeldungen an
slo	x	setzt die gegenwärtig aktive Stufe für Systemmeldungen auf x `0x01` – *no* `0x02` – *low* `0x03` – *medium* `0x04` – *high* `0x05` – *verbose*, für *Meldungen über Netzwerkereignisse* außerdem `0x80` dazuaddieren
rlb		zeigt die beim Booten zu aktivierende Stufe für Systemmeldungen an
slb	x	setzt die beim Booten zu aktivierende Stufe für Systemmeldungen auf x
tim		gibt das Datum und die aktuelle Uhrzeit aus
tiz	x	setzt die interne Uhr des Sensorknotens, Format: x = *„jj-mm-tt hh:mm:ss“*
rfl*	x y	liest den Flash- bzw. Arbeitsspeicher von Adresse x bis Adresse y aus
rer*	x y	liest den EEPROM-Speicher von Adresse x bis Adresse y aus
web	x y	schreibt das Byte y in den EEPROM-Speicher an Adresse x
dea		löscht den gesamten EEPROM-Speicher (mit Ausnahme der dort gespeicherten Konfiguration)
mem		zeigt an, wieviel Arbeitsspeicher noch frei ist
RST		führt einen Reset des Knotens aus
lst*		zeigt alle verfügbaren Terminalkommandos an

Tabelle A.2: Terminalkommandos System

Kommando	Parameter	Beschreibung
snd	x	spielt für x = 1 – 5 jeweils einen Testton
rtt		zeigt die gegenwärtig gemessene Temperatur an
rmm		zeigt die am Mikrofon aktuell induzierte Spannung an *(0 – 4095)*
rvb		gibt die aktuelle Spannung der Batterien an, der Wert sollte im Bereich 2000 – 2560 liegen, dann Spannung $\geq$ 3,2V [13]
rve		zeigt analog zu *rvb* die aktuelle Spannung am externen Stromanschluss an
sir*	x	sendet das Wort x (16 Bit) über Infrarot
rsm		gibt die gegenwärtig aktiven Sensoren an
ssm	x	setzt die Bitmaske für die aktiven Sensoren auf x `0x01` – *Mikrofon* `0x02` – *Temperatur* `0x04` – *Infrarot* `0x08` – *Bewegung* `0x10` – *Vibration* `0x40` – *Batteriespannung* `0x80` – *Knopf*
raf		zeigt an, wie über Sensorereignisse informiert wird
saf	x	setzt die Bitmaske für die Meldung von Sensorereignissen auf x `0x01` – *am Terminal ausgeben* `0x02` – *Broadcast an andere Knoten über Funk*
swg		grüne LED ein- oder ausschalten
swy		gelbe LED ein- oder ausschalten
swr		rote LED ein- oder ausschalten
swb		Lautsprecher: Heulton ein- oder ausschalten
rlg		zeigt den Status der grünen LED an
rly		zeigt den Status der gelben LED an
rlr		zeigt den Status der roten LED an

Tabelle A.3: Terminalkommandos Sensoren

Anhang B

Messreihen

B.1 Die EEPROM-Datenrate

Für die Messung der EEPROM-Datenrate wurden die beiden Terminalkommandos `epw` und `epr` eingerichtet. Sie stoppen mit Hilfe der softwaregesteuerten Uhr die Zeit, um n Bytes in den EEPROM-Speicher zu schreiben oder daraus zu lesen, und geben das verstrichene Zeitintervall anschließend am Terminal aus. Im Gegensatz zum Flash-Speicher müssen beim Beschreiben des EEPROMs die Interrupts nicht deaktiviert werden, sodass die Uhr normal weiterläuft. Während die Leserate unabhängig von der Blockgröße bei 16 KB/s liegt, ist die Schreibrate nicht linear. Sie fällt von 12,8 KB/s bei 128 Byte Blockgröße auf 10,2 KB/s bei einer Blockgröße von 512 Byte ab. Die Tatsache, dass das Schreiben von 136 Byte erheblich länger als das Schreiben von 128 Byte dauert, hängt mit der Seitengröße des EEPROM-Speichers zusammen. Er kann nämlich bis zu 128 Byte an Daten zwischenspeichern, bevor sie im Hintergrund geschrieben werden [12].

Blockgröße in Byte	*Dauer in ms (1024 ms $\widehat{=}$ 1 s)*	
	Schreiben	*Lesen*
32	3	2
64	5	4
128	10	8
136	14	9
256	23	16
512	50	32

Tabelle B.1: Die EEPROM-Datenraten

B.2 Auseinanderdriften der Knotenuhren

Obwohl es möglich ist, die Knotenuhren bis auf eine Millisekunde genau zu synchronisieren, driften sie mit der Zeit leicht auseinander. In der folgenden Messreihe wurden die Knoten 2, 3, 4, 6, 7 und 8 nach dem im Abschnitt 6.2 vorgestellten Verfahren zunächst synchronisiert. Dann erfolgte über einen Zeitraum von 20 Minuten die Messung des Synchronisationsfehlers mit Hilfe eines weiteren Knotens. In diesem Hilfsknoten sind hierzu das Registrieren von Knopfdrücken sowie das Broadcasten von Sensorereignissen aktiviert gewesen (Terminalkommandos `ssm` und `saf`). So wurde am Hilfsknoten alle fünf Minuten der linke Knopf gedrückt, was den Versand eines Broadcastpakets auslöste[1]. Der von den anderen Knoten erfasste Empfangszeitpunkt konnte darauf mit dem Terminalkommando `rxt` ausgelesen werden und wurde mit dem Empfangszeitpunkt im Knoten 2 verglichen. Interessanterweise liefen alle Uhren im Vergleich zur Uhr dieses Referenzknotens nach.

Knoten	*Abweichung zu Knoten 2 in ms nach*			
	5 min	*10 min*	*15 min*	*20 min*
3	-2	-5	-7	-9
4	-1	-2	-4	-4
6	-1	-1	-2	-2
7	-3	-5	-8	-10
8	-2	-4	-6	-7

Tabelle B.2: Auseinanderdriften der Knotenuhren

[1] Da das Drücken und Loslassen des Knopfes zwei separate Sensorereignisse sind, werden eigentlich zwei Pakete verschickt. Dies kann dadurch unterbunden werden, dass man beim Loslassen des linken Knopfes gleichzeitig auch den rechten Reset-Knopf betätigt.

Anhang C

Übersicht der Quelltextdateien

Die Quelltexte sind durchgehend kommentiert (auf Englisch), insbesondere enthalten die Headerdateien Beschreibungen der verwendeten Funktionen und Strukturen.

C.1 ESB-RTable

```
obj
  ├── Objektdateien zu den Quelltexten
Release
  ├── ESB-RTable.exe    Programmdatei für Windows
  ├── ESB-RTable        Programmdatei für Linux
```

crc16.h	*CRC-16 Prüfsumme berechnen*
ESB-RTable.bison.cpp / .h	*von Bison erzeugter Parser*
ESB-RTable.cpp / .h	*Hauptprogramm*
ESB-RTable.flex.cpp	*von Flex erzeugter lexikalischer Scanner*
ESB-RTable.l	*Beschreibungsdatei für Flex (reguläre Ausdrücke)*
ESB-RTable.y	*Beschreibungsdatei für Bison (kontextfreie Grammatik)*
FlexBison.bat / .txt	*Skript und Anmerkung, um Flex und Bison unter Windows zu nutzen*
makefile	*Makefile für Linux*
rlserial.cpp / .h	*Klasse zum Zugriff auf die serielle Schnittstelle unter Windows und Linux [45]*
unistd.h	*Header für Flex unter Windows*

C.2 ESB-RLog

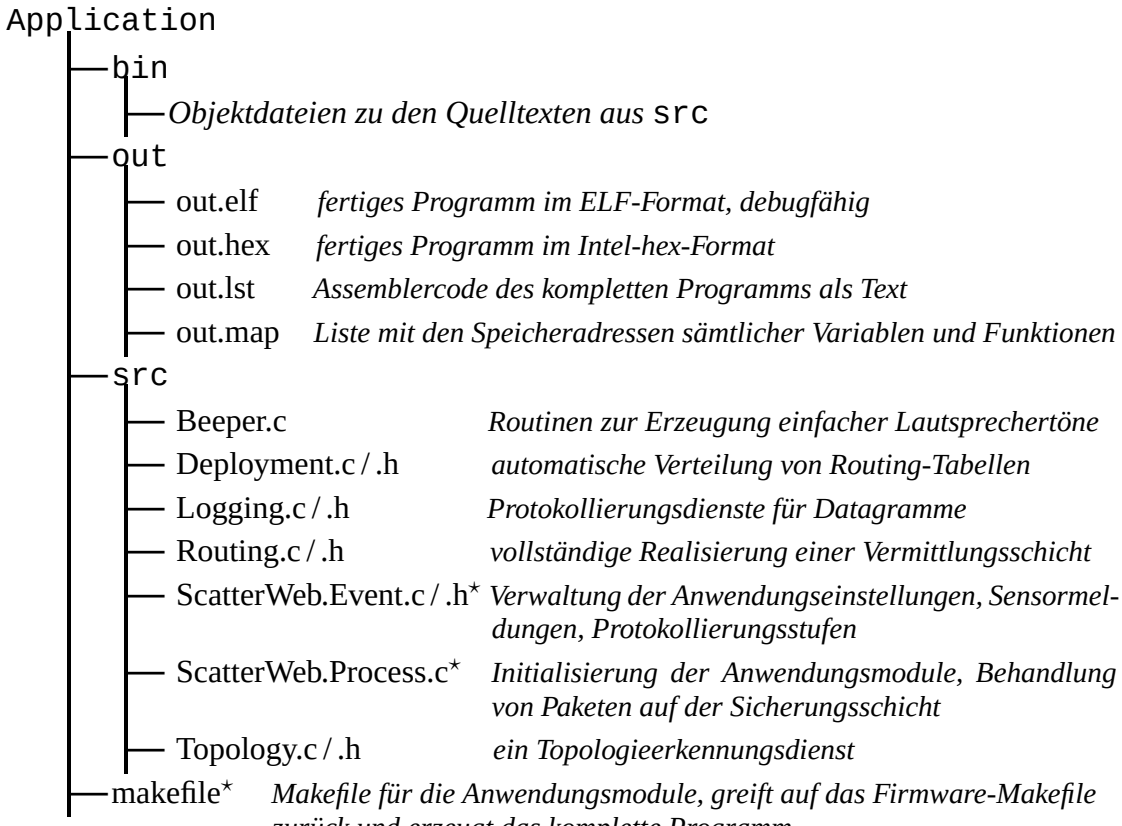
`obj`	
├─ *Objektdateien zu den Quelltexten*	
`Release`	
├─ ESB-RLog.exe	*Programmdatei für Windows*
├─ ESB-RLog	*Programmdatei für Linux*
crc16.h	*CRC-16 Prüfsumme berechnen*
ESB-RLog.cpp / .h	*Hauptprogramm*
makefile	*Makefile für Linux*
rlserial.cpp / .h	*Klasse zum Zugriff auf die serielle Schnittstelle unter Windows und Linux [45]*

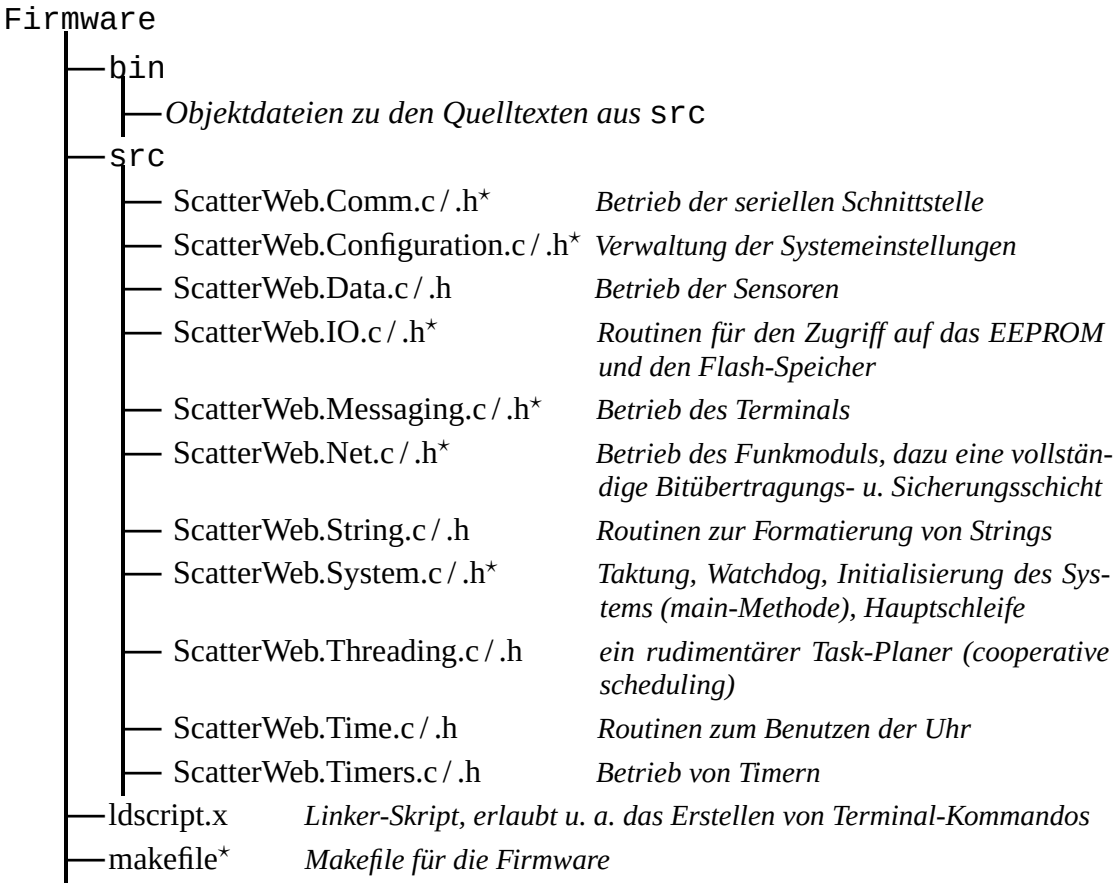
C.3 ESB-Dijkstra

`obj`	
├─ *Objektdateien zu den Quelltexten*	
`Release`	
├─ ESB-Dijkstra.exe	*Programmdatei für Windows*
├─ ESB-Dijkstra	*Programmdatei für Linux*
crc16.h	*CRC-16 Prüfsumme berechnen*
ESB-Dijkstra.cpp / .h	*Hauptprogramm*
FibHeap.cpp / .h	*ein Fibonacci-Heap [3]*
makefile	*Makefile für Linux*
rlserial.cpp / .h	*Klasse zum Zugriff auf die serielle Schnittstelle unter Windows und Linux [45]*

C.4 ScatterWeb

Die im Rahmen dieser Bachelorarbeit neu hinzugekommenen Quelltexte tragen keinen *ScatterWeb*-Präfix im Dateinamen. Außerdem wurden einige Teile des vorhandenen ScatterWeb-Codes revidiert, modifiziert oder um neue Funktionen erweitert. Die betreffenden Dateien sind mit einem Stern ⋆ gekennzeichnet.




Firmware
bin
Objektdateien zu den Quelltexten aus src
src
ScatterWeb.Comm.c / .h*
Betrieb der seriellen Schnittstelle
ScatterWeb.Configuration.c / .h*
Verwaltung der Systemeinstellungen
ScatterWeb.Data.c / .h
Betrieb der Sensoren
ScatterWeb.IO.c / .h*
Routinen für den Zugriff auf das EEPROM und den Flash-Speicher
ScatterWeb.Messaging.c / .h*
Betrieb des Terminals
ScatterWeb.Net.c / .h*
Betrieb des Funkmoduls, dazu eine vollständige Bitübertragungs- u. Sicherungsschicht
ScatterWeb.String.c / .h
Routinen zur Formatierung von Strings
ScatterWeb.System.c / .h*
Taktung, Watchdog, Initialisierung des Systems (main-Methode), Hauptschleife
ScatterWeb.Threading.c / .h
ein rudimentärer Task-Planer (cooperative scheduling)
ScatterWeb.Time.c / .h
Routinen zum Benutzen der Uhr
ScatterWeb.Timers.c / .h
Betrieb von Timern
ldscript.x
Linker-Skript, erlaubt u. a. das Erstellen von Terminal-Kommandos
makefile*
Makefile für die Firmware

Literaturverzeichnis

[1] Die Elektrosmog-Homepage: Online-Kursheft "Wirkmodell auf Lebewesen", Die Antennenspannung und der Mensch, Die optimale Antennenlänge. http://www.e-smog.ch/wirkmodell/seiten/die_optimale_antennenlange.htm.

[2] Free Software Foundation (FSF) – Bison. http://www.gnu.org/software/bison.

[3] John Boyer. The Fibonacci Heap. *Dr. Dobb's Journal*, Januar 1997. http://www.ddj.com/articles/1997/9701.

[4] Ulrich Breymann. *Die C++ Standard Template Library*. Addison-Wesley, 2002. *Auch online unter* http://www.informatik.hs-bremen.de/~brey/stlb.html *kostenlos erhältlich!*

[5] British Standards Institute. *The C Standard (ISO/IEC 9899-1999)*. John Wiley and Sons Ltd, 2003.

[6] Timothy M. Chan. All-pairs shortest paths with real weights, Mai 2005. www.cs.uwaterloo.ca/~tmchan/apsp.ps.

[7] The Contiki Operating System. http://www.sics.se/~adam/contiki.

[8] CXCC: Cooperative Crosslayer Congestion Control. http://www.cn.uni-duesseldorf.de/projects/CXCC http://www.tm.uka.de/forschung/SPP1140/inc/description.php?pid=29&ph=2.

[9] E.W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959. http://www-gdz.sub.uni-goettingen.de/cgi-bin/digbib.cgi?PPN362160546_0001.

[10] Adam Dunkels, Laura Marie Feeney, Björn Grönvall, and Thiemo Voigt. An integrated approach to developing sensor network solutions. In *Proceedings of the Second International Workshop on Sensor and Actor Network Protocols and Applications*, Boston, Massachusetts, USA, August 2004.
http://www.sics.se/~adam/sanpa2004.pdf.

[11] Adam Dunkels, Thiemo Voigt, Niclas Bergman, and Mats Jönsson. The Design and Implementation of an IP-based Sensor Network for Intrusion Monitoring. In *Swedish National Computer Networking Workshop*, Karlstad, Sweden, November 2004.
http://www.sics.se/~adam/sncnw2004.pdf.

[12] Microchip Technology, 512K I^2C^{TM} CMOS Serial EEPROM.
http://ww1.microchip.com/downloads/en/DeviceDoc/21754E.pdf.

[13] Documentation for the ESB Nodes: Power supply.
http://postone.ti5.tu-harburg.de/WiKi/wsn:index. Artikelrevision vom 23. Juni 2005.

[14] uIP web server test page.
http://uip-demo.sics.se.

[15] Fachgespräch Sensornetze: Beiträge.
http://www.fachgespraech-sensornetze.de/beitraege.html.

[16] Free Software Foundation (FSF) – Flex.
http://www.gnu.org/software/flex.

[17] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the Association for Computing Machinery (ACM)*, 34(3):596–615, 1987.
http://doi.acm.org/10.1145/28869.28874.

[18] GCC – GNU Compiler Collection.
http://gcc.gnu.org.

[19] GNU Compiler Collection (GCC) – Extensions to the C Language Family.
http://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html.

[20] Gunter Saake und Kai-Uwe Sattler. *Algorithmen & Datenstrukturen – Eine Einführung mit Java*. Dpunkt Verlag, 2002.

[21] Thomas Haenselmann. Vorlesung Sensornetze WS 2004/2005 – Motivation. http://www.informatik.uni-mannheim.de/pi4/lib/lectures/ws0405/sensornetze.

[22] Herbert Schildt. *The Annotated ANSI C Standard*. Osborne McGraw-Hill, 1993.

[23] HI-TECH Software – Embedded C Compilers. http://www.htsoft.com.

[24] IAR Systems – Embedded Development Tools. http://www.iar.com.

[25] IEEE Standards Association. *IEEE 802.11 (ISO/IEC 8802-11)*, 1999 edition. http://standards.ieee.org/getieee802/802.11.html.

[26] ImageCraft – Embedded C Development Tools. http://www.imagecraft.com/software.

[27] Insight – grafisches Frontend zum gdb-Debugger.
Dokumentation: http://www.redhat.com/docs/manuals/edk/EDK-1.0-Manual/getting-started-guide/gsdebug.html
MSP430-Version für Windows: http://prdownloads.sourceforge.net/mspgcc
MSP430-Version für Linux: http://cdk4msp.sourceforge.net.

[28] Wikipedia: Integrationsdichte. http://de.wikipedia.org/wiki/Integrationsdichte. Artikelrevision vom 01. März 2005.

[29] Olimex: MSP430-JTAG. http://www.olimex.com/dev/msp-jtag.html.

[30] Enrico Köppe, Achim Liers, Hartmut Ritter, and Jochen Schiller. Low-Power Image Transmission in Wireless Sensor Networks using ScatterWeb Technology. In *BROADNETS 2004*, San José, California, USA, Oktober 2004. http://www.broadnets.org/2004/workshop-papers/Basenets/Schiller.pdf.

[31] John Levine, Tony Mason, and Doug Brown. *lex & yacc*. O'Reilly, 1992.

[32] Martin Mauve. Mobile Ad-hoc-Netzwerke: Kommunikation ohne Infrastruktur. *Jahrbuch der Heinrich-Heine-Universität Düsseldorf*, 2003.
http://www.uni-duesseldorf.de/HHU/Jahrbuch/2003/Mauve.

[33] Maxwell Walter. Serial Port Programming in Windows and Linux, November 2003.
Der Artikel ist online leider nicht mehr verfügbar, eine PDF-Fassung befindet sich jedoch auf der beiliegenden CD-ROM.

[34] Michael Becker. Eine Einführung in Makefiles.
http://www.ijon.de/comp/tutorials/makefile.html.

[35] Wikipedia: Memory Mapped I/O.
http://de.wikipedia.org/wiki/Memory_Mapped_I/O. Artikelrevision vom 11. März 2005.

[36] Texas Instruments, MSP430x1xx Family User's Guide (Rev. E).
http://focus.ti.com/lit/ug/slau049e/slau049e.pdf.

[37] Texas Instruments, MSP430F149 – 16-Bit Ultra-Low-Power Microcontroller.
http://focus.ti.com/docs/prod/folders/print/msp430f149.html.

[38] mspgcc – GCC toolchain for MSP430.
http://mspgcc.sourceforge.net.

[39] mspgcc: ein fertiges Installationspaket für Linux.
http://www.informatik.uni-mannheim.de/pi4/lib/lectures/ss2004/msp430.

[40] mspgcc – Tools: gdbproxy, pyJTAG.
http://mspgcc.sourceforge.net/tools.html.

[41] Mikrocontroller.net – MSPGCC.
http://www.mikrocontroller.net/articles/MSPGCC. Artikelrevision vom 02. Juli 2005.

[42] Chris Nagy. *Embedded Systems Design Using the TI MSP430 Series*. Newnes, 2003.

[43] P. J. Plauger. *The Standard C Library*. Prentice Hall PTR, 1991.

[44] Process Visualization – ProcessViewBrowser.
http://pvbrowser.org.

[45] Rainer Lehrig. rl-library: rlSerial Class Reference.
http://www.pvbrowser.org/pvbrowser/sf/manual/rllib/html/classrlSerial.html.

[46] IR-Fernbedienung – der RC-5 Code.
http://www.sprut.de/electronic/ir/rc5.htm.

[47] RFC 791 – Internet Protocol.
http://www.faqs.org/rfcs/rfc791.html.

[48] Hartmut Ritter, Jochen Schiller, Thiemo Voigt, and Adam Dunkels. Experimental Evaluation of Lifetime Bounds for Wireless Sensor Networks. In *2nd European Workshop on Wireless Sensor Networks (EWSN)*, Istanbul, Türkei, Februar 2005.
http://www.sics.se/~thiemo/ewsn2005.pdf.

[49] Jochen Schiller, Achim Liers, Hartmut Ritter, Rolf Winter, and Thiemo Voigt. ScatterWeb – Low Power Sensor Nodes and Energy Aware Routing. In *Proceedings of the 38th Hawaii International Conference on System Sciences (HICSS 2005)*, Hawaii, USA, Januar 2005.
http://page.mi.fu-berlin.de/~winter/scatterweb-hicss2005.pdf.

[50] Stefan Schmidt, Holger Krahn, Stefan Fischer, and Dietmar Wätjen. A Security Architecture for Mobile Wireless Sensor Networks. In *First European Workshop (ESAS 2004)*, Heidelberg, Deutschland, August 2004.
http://www.sse.cs.tu-bs.de/publications/ESAS2004_Schmidt_et_al.pdf.

[51] Freie Universität Berlin, Computer Systems Telematics, ScatterWeb.
http://www.inf.fu-berlin.de/inst/ag-tech/scatterweb_net/home.

[52] Freie Universität Berlin, Computer Systems & Telematics, ScatterWeb, Embedded Sensor Board.
http://www.inf.fu-berlin.de/inst/ag-tech/scatterweb_net/esb/index.shtml.

[53] Freie Universität Berlin, Computer Systems Telematics, ScatterWeb, Embedded Sensor Board, Equipment.
http://www.inf.fu-berlin.de/inst/ag-tech/scatterweb_net/esb/equipment.shtml.

[54] Robert Sedgewick. *Algorithmen in C*. Addison-Wesley, 1992.

[55] SGI – *hash_map* template class.
http://www.sgi.com/tech/stl/hash_map.html.

[56] SGI – *slist* template class.
http://www.sgi.com/tech/stl/Slist.html.

[57] SGI – Standard Template Library (STL) Programmer's Guide.
http://www.sgi.com/tech/stl.

[58] SGI – *vector* template class.
http://www.sgi.com/tech/stl/Vector.html.

[59] Steve Underwood. mspgcc Handbuch und FAQ.
http://sourceforge.net/projects/mspgcc *unter Files*.

[60] Herb Sutter. Part 1: Vectors and Deques. *C++ Report*, 11(7), 1999.
http://www.gotw.ca/publications/mill10.htm.

[61] Thomas Ottmann und Peter Widmayer. *Algorithmen und Datenstrukturen*. Spektrum Akademischer Verlag, 2002.

[62] Mikkel Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. In *STOC '03: Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 149–158, 2003.
http://doi.acm.org/10.1145/780542.780566.

[63] RFM, TR1001 868.35 MHz Hybrid Transceiver.
http://www.rfm.com/products/data/tr1001.pdf.

[64] The uIP Embedded TCP/IP Stack.
http://www.sics.se/~adam/uip.

[65] Thiemo Voigt, Hartmut Ritter, and Jochen Schiller. Solar-aware Routing in Wireless Sensor Networks. In *9th International IEEE Symposium on Computers and Communications (ISCC)*, Alexandria, Griechenland, Juli 2004.
http://www.sics.se/~thiemo/PWC2003.pdf.

[66] Microsoft Visual Studio 6.0.
http://msdn.microsoft.com/vstudio/previous/vs6.

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, den 10. August 2005 Yves Jerschow