

Projektseminar

“Sprachsteuerung im Aufzug”

– Abschlußbericht –

Eric Auer, Claudia Crispi, Christian Dressler, Cordula Klein,
Kerstin Klöckner, Norbert Pfleger, Diana Raileanu
{eric,crispi,dressler,cklein,kekl,pfleger,raileanu}@coli.uni-sb.de

31. Dezember 2000

Inhaltsverzeichnis

1. Einleitung	2
1.1. Grundidee	2
1.2. Kapitelüberblick	2
2. Entwicklung des ersten Szenarios und Dialogdesign	3
3. Wizard of Oz Experiment	5
3.1. Was ist ein Wizard of Oz Experiment?	5
3.2. Unser Experiment	6
3.2.1. Vorbereitung	6
3.2.2. Durchführung	7
3.2.3. Auswertung	8
3.2.4. Konklusion	11
4. Architektur, Modellierung & Implementierung des Dialogsystems	12
4.1. Architektur	12
4.1.1. Anforderungen an das Dialogmodul	13
4.1.2. Kommunikation der Module	14
4.2. Modellierung	14
4.2.1. Dialog	14
4.2.2. Schnittstellen	16
4.2.3. Zeitverwaltung	19
4.3. Implementierung	20
4.3.1. Modularisierung	20
4.3.2. Verwaltung des Arbeitsspeichers	21
4.3.3. Plattformunabhängigkeit	21
4.3.4. Parameter des Systems	22
4.3.5. Versionskontrolle & MakeFile	22
5. Spracherkennung im Fahrstuhl mit HTK	24
5.1. Spracherkennung	26
5.1.1. Trainingsmaterial	27
5.1.2. Trainingsprozess	33
5.1.3. Lexikon, Grammatik	35

5.1.4. Testprozess	37
6. Steuerung	38
6.1. aufzugsteuerung und deren anschluss	38
6.2. protokoll	39
6.3. implementierung	40
6.4. steuerkommandos	40
6.5. interface	40
6.6. Protokoll	40
6.7. Zustände, Informationen	40
6.8. Kommunikation	40
7. Soundkarten (Eric)	41
7.1. Aufgaben des Teilprojektes	41
7.1.1. Grundsätzliche Probleme	42
7.2. Lösungsansätze	43
7.2.1. Verarbeitung analog und zentral	43
7.2.2. Die Sprachdaten kommen digital an einem Rechner an	43
7.2.3. Verwendung mehrerer Rechner: Echte Skalierbarkeit	45
7.2.4. Mehrfachnutzung von Ressourcen: scheinbare Skalierbarkeit	46
7.2.5. Den Aufwand verteilen: Hardware direkt in den Sprechstellen	46
7.3. Verfügbare Soundkarten	48
7.3.1. Midiman Delta 1010 Mehrspur-Soundkarte	48
7.3.2. Terratec EWS 88 MT	49
7.3.3. Standardsoundkarten	49
7.3.4. Spezielle Wandler-Hardware	50
7.3.5. Ein-Chip Wandler-ICs	51
7.3.6. Multiplexing vieler Sprechstellen auf wenige Analoganschlüsse	52
7.4. Treiber für Mehrkanalzugriff	54
7.4.1. Soundtreiber für Windows	54
7.4.2. Soundtreiber für Linux	55
7.4.3. Treiber für die Übertragung von Audiodaten im Netz	56
A. HTK Ettikettierung	61
B. PhonDat–Anpassung für Lautmodelle	64
C. PhonDat–Anpassung für Wortmodelle	73
D. Skripte für den Trainingsprozeß	83
E. Das Lexikon	90
F. Die Grammatik	105

G. Quelltexte für Erweiterungen des Soundtreibers (Eric)	106
G.1. TCP/IP Server für Midiman Delta 1010	106
G.1.1. rec-sock-alsa.h und rec-sock-alsa.c	106
G.1.2. server-sockets.c	112
G.1.3. alsa-ice.c	116
G.2. client2file.c – Der Client zu rec-sock-alsa	118
G.3. play-alsa.c	121
H. Implementierung des Sprachgesteuerten Fahrstuhls	129

Abbildungsverzeichnis

3.1. Beispieldialog	10
4.1. Überblick über die Kern-Module	13
4.2. Streamkommunikation am Beispiel des Spracherkenners	16
5.1. HMM für den Laut [u:]	26
5.2. Kiel Korpus Datei	29
5.3. HTK Datei – Lautmodelle	30
5.4. HTK Datei – Wortmodelle	33
5.5. Training: Initialisierung	34
5.6. Training: Re-estimation	35
5.7. Lexikonausschnitt	36
5.8. Bild 4	37
5.9. Bild 6	37

1. Einleitung

1.1. Grundidee

Das Projektseminar Sprachsteuerung im Aufzug wurde in interdisziplinärer Zusammenarbeit von Informatik, Phonetik und Computerlinguistik durchgeführt. Das Ziel dieses Projektseminars war die Entwicklung eines Prototyps für die dialogbasierte Sprachsteuerung von Aufzügen. Der Prototyp wurde in den Aufzug des Gebäudes 17.2 integriert, da dieser gerade neu eingebaut wurde und wir damit uneingeschränkten Zugriff auf seine Steuerung hatten, um eventuell nötige Änderungen noch veranlassen zu können. Die Leitung des Seminars lag in Händen von Professor Dr. Manfred Pinkal (Computerlinguistik), Professor Dr. William Barry (Phonetik), Dr. Joachim Niehren (Informatik), Jaques Koremann (Phonetik), C.J. Rupp, Malte Gabsdil (beide Computerlinguistik) und Bistra Andreeva (Phonetik).

Zentrale Idee dieses Projekts war es, behinderten Menschen die Bedienung von Aufzügen zu erleichtern. Im Laufe der Entwicklung stellte sich aber heraus, dass ein sprachgesteuerter Aufzug auch eine sehr geeignete Orientierungshilfe für gebäudefremde Benutzer ist, und dieser Aspekt trat nach und nach mehr in den Vordergrund.

1.2. Kapitelüberblick

Zunächst mussten wir uns jedoch einen Überblick der zu erledigenden Arbeiten verschaffen und ein Szenario entwerfen (Kapitel 2). Dennoch blieben noch viele Fragen ungeklärt. Deshalb beschlossen wir, zunächst ein Wizard of Oz Experiment durchzuführen. Dies ist im Kapitel 3 näher beschrieben. Außerdem musste ein Dialogmodul implementiert werden, das den Verlauf des Dialogs zwischen Benutzer und Aufzug steuern sollte (siehe Kapitel 4). Damit der Aufzug die Eingaben des Benutzers verstehen kann, wird ein Spracherkenner benötigt, der in den Dialog eingebunden werden muss. Dies wird genau in Kapitel 4 und 5 beschrieben. Doch eins der größten Probleme bereitete uns der Zugriff auf die Steuerung, die in Kapitel 6 erklärt wird. Ein weiterer wichtiger Punkt war auch die Weiterleitung der Spracheingaben im Fahrstuhl an den Spracherkenner, dazu wurden Soundkarten benötigt, die natürlich erst eingebaut werden mussten, was in Kapitel 8 ausführlich beschrieben wird.

2. Entwicklung des ersten Szenarios und Dialogdesign

Die Entwicklung von Systemen, die einen Dialog mit einem Menschen führen können, ist schon lange eine Herausforderung für die Wissenschaft. In den letzten zehn Jahren haben sich wesentliche Verbesserungen in der Sprachtechnologie ergeben, die die Implementation eines natürlichsprachlichen Dialogsystems ermöglichen. Bei dem sprachgesteuerten Fahrstuhl handelt es sich um ein sprecherunabhängiges System mit recht eingeschränktem Lexikon. Die größten Probleme entstehen dabei aus den unterschiedlichen Äußerungen der Benutzer, da diese ihre Spracheingaben oft nicht in einfachen Sätzen machen. Deshalb wird ein vom System geführter Dialog benötigt.

Um solch einen Dialog zu entwickeln, haben wir zunächst ein einfaches Szenario entworfen. Dazu gehört auch das Besorgen der nötigen Informationen über den aktuellen Aufenthaltsort des Aufzuges (steht er auf einer Etage oder fährt gerade), genauso wie Informationen darüber, ob die Türen (es gibt eine Innentür und eine Aussentür) geöffnet oder geschlossen sind. Bei der Planung des Szenarios entwickelten sich eine Menge Fragen. Sollte der Fahrstuhl mehrsprachig sein? Dies wäre natürlich sinnvoll, da die Computerlinguistik und auch die Phonetik von internationalen Gästen und Studenten besucht wird.

Soll man den Fahrstuhl durch Aussenmikrophone ansprechen können? Wenn nicht, müsste der Aufzug mit Tastendruck von aussen gerufen werden.

Wer spricht zuerst, Aufzug oder Benutzer? Falls der Aufzug zuerst spricht, soll er sich dann vorstellen? Dies würde jedoch erfordern, dass es eine Lichtschranke gäbe, damit die Steuerung feststellen kann, ob sich jemand im Aufzug befindet oder nicht.

Soll der Fahrstuhl ständig aufmerksam sein, oder nur, wenn er nicht fährt? Während der Fahrt ist der Geräuschpegel höher, daher ist es dann auch für den Spracherkenner schwerer, eine eventuelle Eingabe zu verstehen.

Was passiert, wenn dem Aufzug ins Wort gefallen wird? Wird der Spracherkenner dann abbrechen?

Was geschieht, wenn der Benutzer ein bereits gegebenes Kommando korrigieren will, bzw. wenn der Aufzug das Kommando falsch verstanden hat?

All diese Fragen stellten sich und warfen wieder neue Fragen auf (siehe Anhang **** Protokolle ****).

Deshalb beschlossen wir, zunächst ein sehr einfaches Szenario zu konstruieren, das wir später noch erweitern könnten. Dieses Szenario wurde nur für einen Benutzer konstruiert

und sieht wie folgt aus: Eine Sprachsteuerung von aussen wird nicht vorgesehen, da dies wegen der ständigen Geräuschkulisse im Gebäude sehr problematisch werden könnte. Der Spracherkenner würde nicht unterscheiden können, ob jemand einen Fahrbefehl abgegeben hatte, oder ob sich einfach nur Leute miteinander unterhielten. Deshalb soll zunächst das Rufen des Aufzugs von aussen nur durch Tastendruck erfolgen. Um die Aufmerksamkeit des Spracherkenners zu erlangen, sollte der Benutzer den Fahrstuhl in der Kabine mit einem Schlüsselwort ansprechen. Wir entschieden uns für die Wörter “Aufzug” und “Fahrstuhl”.

Das Dialogmodell soll wie folgt aussehen: Der Aufzug wird vom Benutzer mit dem Befehl “Fahrstuhl” gerufen, aber erst, nachdem sich die äussere Tür geschlossen hat. Dann fragt der Fahrstuhl: *“In welchen Stock möchten Sie bitte?”*, oder der Benutzer kann sofort sagen: *“Fahrstuhl erster Stock.”*. Wenn der Aufzug die Zieleingabe verstanden hat, bestätigt er dies durch: *“Ich fahre Sie in den . . . Stock.”* und dann mit der Fahrt beginnen. Bei nicht verstandener Eingabe soll eine Rückfrage des Aufzuges erfolgen. Wenn die Antwort dann immer noch nicht verstanden wird, soll die Aufforderung erfolgen, die Knöpfe zu benutzen, da ein zu häufiges Nachfragen den Benutzer nur stören würde. Für den Fall einer falschverstandenen Eingabe ist eine Korrekturmöglichkeit eingebaut. Korrekturen sind sowohl durch Tastendruck als auch durch Spracheingabe möglich. Im gewünschten Stockwerk angekommen, erfolgt die Ansage des Fahrstuhls: *“Sie sind jetzt im . . . Stock.”*. Mögliche Zieleingaben sind: *Keller, Untergeschoss, Erdgeschoss, erster Stock, zweiter Stock, dritter Stock, vierter Stock, fünfter Stock, Professor Pinkal, Professor Barry, Professor Uszkoreit und Phonetik*. Da sich die Phonetik im vierten und im fünften Stock befindet, enthält der Dialog eine Komponente, die bei der Zieleingabe Phonetik folgende Nachfrage generiert: *“Möchten Sie in den vierten oder fünften Stock?”*. Der Benutzer hat während des gesamten Dialoges die Möglichkeit, den Dialogablauf durch Drücken der Tasten abubrechen.

Dieser Dialog diene als Grundlage für das Wizard of Oz Experiment. Die Implementierung des Dialogs wird ausführlich in Kapitel 4 beschrieben.

3. Wizard of Oz Experiment

In diesem Kapitel wird ein Wizard of Oz Experiment im Zusammenhang mit der Entwicklung des sprachgesteuerten Fahrstuhls beschrieben.

Abschnitt 3.1 beschreibt, was ein Wizard of Oz Experiment ist, sowie Gründe für seine Durchführung. Abschnitt 3.2 behandelt eine konkrete Anwendung eines Wizard of Oz Experiments und gliedert sich in seine Vorbereitung (3.2.1), Durchführung (3.2.2) und Auswertung (3.2.3).

3.1. Was ist ein Wizard of Oz Experiment?

Ein Wizard of Oz (kurz: WOZ) Experiment ist eine Studie, bei der den Versuchspersonen gesagt wird, daß sie mit einem natürlichsprachlichen System interagieren, obwohl sie das in Wirklichkeit nicht tun. Stattdessen wird die Interaktion durch einen menschlichen Operator, den Wizard, vermittelt.

Die Entwicklung eines effektiven Systems (in diesem Teilschritt der Entwicklung des Dialogs mit dem sprachgesteuerten Aufzug) erfordert umfangreiche Experimente mit realen Benutzern. Ein Mensch–Mensch Dialog eignet sich dafür nicht, da Menschen dazu tendieren, bei einem Dialog mit einer Maschine anders zu reden, als wenn die mit einer anderen Person kommunizieren.¹ Die WOZ Technik eignet sich deshalb dafür, weil man so Dialogdaten sammeln kann, die für die effektive Entwicklung eines Systems wichtig sind und Hypothesen testen kann.

Im nächsten Abschnitt wird beschrieben, wie diese Experimentalmethode auf unser Fahrstuhlexperiment angewandt wurde.

¹Bei linguistischen Untersuchungen hat man herausgefunden, daß Sprecher sich den vermeintlichen Eigenschaften des Gesprächspartners anpassen. So unterscheidet sich Sprache, die an Kinder gerichtet ist (auch bekannt als Kindersprache) von Sprache gerichtet an Ausländer (z.B. kurze Sätze oder Erhöhen der Lautstärke) oder an Erwachsene (Korrekturen oder unvollständige/ungrammatische Sätze). Es gibt gute Gründe anzunehmen, daß solche Anpassungen auch beim Dialog mit einer Maschine gemacht werden.

3.2. Unser Experiment

Um zu testen, ob das von uns entwickelte Dialogmodell (siehe Kapitel 4. *tralala* (Norbert) und Anhang **** Dialogmodell ****) von den Benutzern eines sprachgesteuerten Fahrstuhl akzeptiert wird, haben wir ein Wizard of Oz Experiment am “Tag der offenen Tür” an der Universität des Saarlandes durchgeführt, da an diesem Tag mit einer hohen Besucherzahl gerechnet werden kann und somit genügend (potentielle) Versuchspersonen zur Verfügung stehen.

3.2.1. Vorbereitung

Bei der Vorbereitung des WOZ musste zunächst überlegt werden, was wir überhaupt mit diesem Experiment erreichen wollten. Zum einen wollten wir verschiedene Unklarheiten beseitigen: es musste die Frage geklärt werden, ob Benutzer besser auf eine aufgenommene Frauenstimme oder auf eine synthetisierte Computerstimme ansprechen würden. Welche Fahrkommandos würden am ehesten verwendet und akzeptiert? Der Dialog sollte schließlich so sein, dass es dem Benutzer angenehm war, mit dem Aufzug zu kommunizieren. Er sollte sich nicht überfordert oder verunsichert fühlen. Also musste dieses Experiment ganz genau geplant werden.

Entwicklung der Aufgabenstellungen

Wir beschlossen daher, den Besuchern eine Aufgabenstellung zu geben. Zum einen, damit wir das ganze kontrolliert durchführen konnten, zum anderen lag es ja auch in unserer Absicht, ganz bestimmte Ergebnisse zu erhalten. Wir benötigten die aufgenommen Sprachdaten für die Phonetikgruppe, um Lautmodelle zu trainieren, und wir wollten wissen, welche Eingaben der Benutzer machen würde, wenn er die Wahl hatte. Also kamen mündlich gestellte Aufgaben nicht in Frage, denn damit hätte man den Benutzer vorab beeinflusst und er sollte ja unvoreingenommen an die Sache ran gehen. Deshalb formulierten wir die Aufgabenstellung in schriftlicher Form. Doch auch dies barg seine Tücken in sich. Die Aufgabe musste klar definiert sein, sollte aber gleichzeitig so formuliert sein, dass man dem Benutzer nicht den genauen Wortlaut seiner Spracheingabe vorgab.

So wurde ein graphisches Modell für die Aufgabenstellung entwickelt, das einen Plan der fünf Etagen des Gebäudes enthielt. In jeder Etage war die Etagen-Nummer und der Name einer Person oder eines Raumes angegeben (siehe Anhang Nr. ..). Daneben stand eine Aufgabenstellung, die eine fiktive Situation darstellte: “Stellen Sie sich vor, Sie haben einen Brief für Professor XY. Sie fahren zu seinem Büro.”. Dies sollte die Versuchsperson veranlassen, sich den angegebenen Raum selbst auf dem eingezeichneten Plan herauszusuchen und dann mit dem Aufzug in die betreffende Etage zu fahren. Dort musste die Person aussteigen und zu dem Raum hingehen. An der Tür des betreffenden Raumes war dann wieder ein Schild angebracht, das eine weitere Aufgabenstellung enthielt. Somit musste jede Versuchsperson mehrmals den Aufzug benutzen, blockierte ihn aber nicht auf lange Zeit, für den Fall, dass es einen grossen Besucherandrang geben sollte. Ausserdem führen die

Personen ganz kontrolliert alle Etagen an und wir konnten somit von jeder Etage die gleiche Anzahl an Sprachaufnahmen machen. Wir entwickelten zwei verschiedene Aufgabenstellungen, die dann immer abwechselnd ausgegeben wurden. Die erste Aufgabenstellung deckte Etage vier, zwei und den Keller ab, während die zweite Aufgabenstellung die Etagen eins, drei und fünf abdeckte. So erhofften wir uns einen kontrollierten Versuchsablauf. Damit die Besucher wussten, was sie überhaupt tun sollten, entwickelten wir ein Blatt mit Hinweisen über die Funktionsweise des Aufzugs und die Verwendung des Schlüsselworts. Dieses Instruktionsblatt wurde den Versuchspersonen vor Beginn des Experiments vorgelesen. Somit sollte erreicht werden, dass jede Versuchsperson die gleiche Instruktion erhält.

Entwicklung des Fragebogens

Außer den Sprachaufnahmen benötigten wir noch einige Auskünfte, um die Reaktionen der Versuchspersonen bewerten zu können. Dazu entwickelten wir einen Fragebogen. Mit diesem Fragebogen wollten wir herausfinden, welches Kommando den größten Anklang findet, wie die Antworten des Aufzuges beurteilt werden, wie die Stimme des Aufzuges beurteilt wird, da wir ja zwei Varianten hatten, eine synthetisierte und eine aufgenommene Stimme. Außerdem baten wir um persönliche Eindrücke und Verbesserungsvorschläge.

Planung des Versuchs

Damit am Tag der offenen Tür alles reibungslos ablief, musste der Versuchsablauf genau geplant werden. Wir hatten verschiedene Äußerungstypen des Aufzugs vorbereitet, d.h. es gab einen Dialogablauf, der mit der synthetisierten Stimme ablief und einen, der mit der aufgenommenen Frauenstimme durchgeführt wurde. Von beiden Varianten gab es jeweils eine Version, die spezifisch und eine die unspezifisch war. Unter spezifisch wird verstanden, dass der Aufzug seine Frage genauer formuliert, wie z.B. *“In welche Etage möchten Sie bitte?”* während die unspezifische Rückfrage nur aus *“Wohin möchten Sie bitte?”* bestand. Wir wollten damit herausfinden, was von den Testpersonen besser angenommen wurde. Ferner entschieden wir uns dafür jeden Benutzer wenigstens bei einer Fahrt, (jeder Benutzer musste drei mal fahren), nicht zu verstehen, d.h. der Aufzug würde um eine erneute Angabe des Zieles bitten.

3.2.2. Durchführung

Am Tag der offenen Tür sollte der Versuch ab zehn Uhr laufen. Damit die Besucher der Universität auch von unserem Experiment in Kenntniss gesetzt wurden, verteilten wir Flugblätter auf dem Campus. Da wir nicht wussten, wieviele Personen Interesse an einer Testfahrt haben würden, hatten wir eine Terminliste vorbereitet, in die sich die Interessenten eintragen konnten. Entsprechend der eingetragenen Reihenfolge durften sie dann den Aufzug testen. So konnten wir auch bei einem großen Andrang den Überblick bewahren.

Bevor die Versuchsperson den Fahrstuhl bestieg, las ihr ein Helfer die Hinweise vor. Anschließend erhielt sich die Aufgabenstellung und einen Brief, der gemäß der Aufga-

benstellung abzugeben war. Dies sollte die auszuführende Aufgabe realistischer erscheinen lassen. Natürlich lief nicht alles so reibungslos ab, wie wir uns das gedacht hatten. Es gab verschiedene Leute, denen wir den Ablauf doch nochmals erklären mussten. Aber den Leuten schien es ziemlich Spaß zu machen. Sie waren jedesmal sehr überrascht, dass der Aufzug sie so gut verstand. Wieso das so war wurde ihnen erst klar, als wir ihnen hinterher sagten, dass es ein Wizard war, der sie so gut verstanden hatte. Entgegen unserer Befürchtung, dass die Testpersonen darüber verärgert sein würden, nahmen sie es recht gelassen hin. Bevor wir ihnen erklärten, dass es ein WOZ Experiment war, durften die Testpersonen den Umschlag, den wir ihnen gegeben hatten öffnen. Darin befand sich eine kleine Belohnung in Form von Gummibärchen und der Fragebogen, den sie dann ausfüllen sollten.

3.2.3. Auswertung

Anhand der beim Wizard of OZ Experiment gewonnenen Daten — den von den Versuchspersonen nach dem Experiment ausgefüllten *Fragebögen* einerseits und den *Sprachdaten* (die Aufzeichnung der Versuchspersonen beim Dialog mit dem Fahrstuhl) andererseits — erhofften wir uns eine Verbesserung des Prototyps des sprachgesteuerten Aufzugs.

Auswertung der Fragebögen

Die Auswertung der Fragebögen dient dazu, Anregungen und Kritik der Versuchspersonen bezüglich des Systems “sprachgesteuerter Aufzug” konstruktiv zu nutzen, um das System zu erweitern und zu verbessern. Im Anhang **** Fragebogen **** findet sich der beim Wizard of Oz Experiment verwendete Fragebogen, von dem je eine Kopie den Versuchspersonen nach Beendigung der Fahrten ausgehändigt wurde. Die Auswertung ergab folgendes:

Am Versuchstag waren von 51 Personen Fragebögen ausgefüllt worden. Davon hatte die Mehrheit auf Frage eins, welches das bevorzugte Kommando sei, mit “Fahrstuhl” und “Aufzug” geantwortet. Aber es gab auch Leute, die “Computer” oder “Lift” bevorzugten. Einige fanden, man sollte alle vier Befehle zulassen, oder dem Aufzug einen Namen geben. Auf die Frage, wie sie die Antworten bzw. Fragen des Aufzugs fanden, antworteten fast alle mit “normal”. Einer fand die Antworten unhöflich. Zur Ausführlichkeit meinten nur zwei, dass die Antworten zu knapp seien. Die Stimme des Aufzug (Frage drei) fanden drei Leute unangenehm, einer langweilig und alle anderen angenehm. Dass die Stimme unverständlich sei, fand nur einer. Die Leute, die Erfahrung mit sprachgesteuerten Systemen hatten (danach wurde mit Frage Nummer vier gefragt), beurteilten unser System meist mit “gut”, drei mit “sehr gut” und einer mit “befriedigend”; ca. die Hälfte hatte gar keine Erfahrung mit ähnlichen Systemen. Bei der vorletzten Frage, was nach Meinung der Testpersonen fehle, antworteten alle, der Fahrstuhl solle sich verabschieden, wenn man aussteigt und “Bitte” sagen, wenn der Benutzer “Danke” sagt bei der Ankunft. Einige wünschten sich eine Beschreibung, wie man den Raum, in dem eine bestimmte Person sitzt, erreichen kann, oder eine Auskunft über den derzeitigen Aufenthaltsort einer Person, wenn sie nicht im eigenen Büro sei. Ausserdem wurde Mehrsprachigkeit und ein Mikrofon in Kinderhö-

he verlangt. Einige meinten auch, der Fahrstuhl solle die Benutzer ansprechen und ihnen erklären, wie er funktioniere. Zur letzten Frage schrieben alle, dass sie den Aufzug wieder benutzen würden.

Auswertung der Sprachdaten

Die gesammelten Sprachdaten wurden so aufbereitet, dass jeder Fahrt im Fahrstuhl eine .wav-Datei entspricht, die fortlaufend durchnummeriert ist. Insgesamt wurden 1000 .wav-Dateien gesammelt. Mit Hilfe dieser Sprachdaten soll ein angemessenes Sprachmodell (siehe Kapitel 5) erstellt werden. Dazu wurden die Sprachdaten mit einem von uns eigens dafür entwickelten Schema (siehe Anhang **** Transliterationsschema ****) transliteriert.

Das Transliterations-Schema besteht hauptsächlich aus tags, die ein wenig an HTML-tags erinnern, da sie in “<>” eingeschlossen sind. Jede Äußerung des Fahrstuhls (diese entsprechen den Übergängen im Dialogmodell) sind mit einem tag gekennzeichnet, der die Äußerung unverwechselbar identifiziert und der jeweils mit dem Buchstaben “L” beginnt. Die Zustände, in denen sich der Fahrstuhl befinden kann, sind durch einen tag gekennzeichnet, der mit einem “Z” beginnt. Außerdem gibt es noch andere tags die kennzeichnen, ob der Fahrstuhl (<f>) oder der Benutzer () spricht, oder ob es sich um Hintergrundgeräusche (<noise>) handelt. Die Liste aller verwendeter tags inklusive ihrer Bedeutung findet sich in Anhang *** Transliterations-Schema ***.

Eine Transliterationsdatei besteht aus einem **header** (Kopf der Datei) und den einzelnen **turns**. Als turns werden die jeweilige Sprecherbeiträge bezeichnet, was in unserem Fall die Äußerungen des Fahrstuhls und der Versuchspersonen sind. Der header beinhaltet den *Namen* des Transliterierers/der Transliteriererin, das *Datum* der Transliteration, die *Experimentnummer*, die identisch mit dem Namen der .wav-Datei ist, den *Modus* (Werte: Mensch oder Maschine), ein Merkmal Namens “*ques*” — kurz für question — (Werte: spezifisch oder unspezifisch), die *Versuchspersonennummer* und eventuelle *Anmerkungen* des Transliterierers/der Transliteriererin. Der Rest der Datei besteht, wie oben schon erwähnt, aus den einzelnen turns. Jeder turn beginnt mit dem Datum, an dem der Dialog aufgezeichnet wurde und der genauen Uhrzeit. Danach werden die Zustandsänderungen kodiert sowie die Sprecherbeiträge transliteriert. Folgende Abbildung verdeutlicht die Transliteration eines aufgenommenen Dialogs:

Man kann dem header entnehmen, dass die Transliteration von Malte Gabsdil (mg) am 10.07.2000 erstellt wurde. Da es sich um einen fiktiven Dialog handelt, trägt das Experiment keine Nummer, stattdessen ersetzen drei “x” die das Experiment kennzeichnenden Zahlen. Der Modus der Äußerungen des Fahrstuhls wurde mit der menschlichen Stimme ausgegeben (mode: mensch) und die Art der Fragen ist spezifisch. Die Fahrt wurde von der Versuchsperson Nummer x gemacht. Anmerkungen gibt es keine. Danach folgt die Transliteration des Dialogs zwischen Benutzer und Fahrstuhl, der folgendermaßen ablief: Gemäß der Vorgabe beginnt der Benutzer den Dialog mit dem Fahrstuhl, in dem er das Kommando “*Fahrstuhl*” sagt. Dabei geht der Dialog von Zustand Z1 mit dem Übergang L1 in den Zustand Z2 über. Etc.

<h>
bearbeitet von: mg
datum: 10.7.00
experiment: experimentxxx
mode: Mensch
ques: spezifisch
vp: x
Anmerkungen
</h>

2000-06-03__hh-mm-ss: <Z1,L1,Z2> Fahrstuhl.

2000-06-03__hh-mm-ss: <Z2,L2,Z3> <f> In <u> welchen Stock </u> <u> dritte
Etage </u> möchten Sie bitte? <f>

2000-06-03__hh-mm-ss: <Z3,L3_n,Z7> <f> Ich habe Sie nicht <u> verstanden </u>
</f> <Z7,L3_f,Z9> <u> Dritter Stock </u> habe ich
gesagt.

2000-06-03__hh-mm-ss: <Z9,L4_4,Z11> <f> Ich fahre Sie in den vierten Stock. </f>

2000-06-03__hh-mm-ss: <Z11,L7,Z12> <kr> Halt. <i> Aehm </i> in den
dritten. </kr>

2000-06-03__hh-mm-ss: <Z12,L8_4,Z13> <f> Sie sind jetzt im vierten Stock. </f>

2000-06-03__hh-mm-ss: <noise> <Z13,L9,Z1>

Abbildung 3.1.: Beispieldialog

Ist das nicht zu spezifisch mit Zustand und Übergang. Wird doch erst in Kapitel 4 von Norbert eingeführt. Oder wir müssen noch eine Abbildung mit dem DaVinci Dialogmodell machen, damit das klarer wird. Perl-Skript beschreiben
Wortauszählungen → Grammatik. Hier beschreiben oder nur Verweis auf Diana und Kor-
dulas Kapitel, wo die das beschreiben.

3.2.4. Konklusion

Variablen des WOZ nicht konsequent variiert.

Auf Grund der Sprachdaten kein sinnvolles Benutzermodell erstellbar.

Was noch?

Was sollte das nächste Mal besser gemacht werden?

Anhang:

- Fragebogen
- Aufgabenstellung
- Instruktionen
- (Automat???)
- Transliterations-Schema

4. Architektur, Modellierung & Implementierung des Dialogsystems

Im Folgenden werden die wesentlichen Konzepte und Modelle des Sprachgesteuerten Fahrstuhls dargestellt. Aus den Anforderungen an ein solches System lassen sich recht einfach die wesentlichen Komponenten des Systems ableiten. Allgemein betrachtet muß ein Dialogsystem in der Lage sein die Informationen zu sammeln, die benötigt werden, um eine konkrete Aufgabe zu lösen (zum Beispiel Ausgabe von Informationen oder dergleichen). Ein solches System muß also gesprochene Anweisungen verstehen und entsprechend reagieren. Die Reaktionen des Systems können auf der einen Seite aus sprachlichen Äußerungen bestehen, das System führt also einen Dialog mit dem Benutzer und auf der anderen Seite aus Datenbankabfragen oder Befehlen an externe Geräte bestehen.

Die Anforderungen an das Dialogmodul eines sprachgesteuerten Aufzuges sehen ähnlich aus, so sollte das System den Benutzer bei der Eingabe des Fahrtziels unterstützen (Dialog), weiterhin sollte das System die Zieleingabe des Benutzers an die Fahrstuhllogik weiterreichen. Die Frage ist jedoch, welche Äußerungen eines Benutzer muß der Dialog verarbeiten können und wie sollten die Reaktionen des Systems aussehen. Aus diesem Grund haben wir das im vorherigen Kapitel beschriebene Wizard-of-Oz Experiment durchgeführt. Aus den hieraus gewonnen Erfahrungen ist ein recht konkretes Modell des Dialog entstanden, welches die Grundlage für die Implementierung dieser sprachverstehenden Systems bildete. Auch spielten die Erfahrungen aus der Entwicklung des Programms, welches dem Wizard-of-Oz Experiment zugrunde liegt, eine große Rolle bei der Entwicklung dieses Systems. So hat sich zum Beispiel an dem Treiber, der die Fahrstuhllogik anspricht nichts mehr geändert (zum Thema Ansteuerung der Fahrstuhllogik siehe Kapitel 5).

4.1. Architektur

Auch die grobe Architektur wurde aus Wizard-of-Oz System übernommen. Wobei jedoch an Stelle des Menschen nun ein Spracherkenner getreten ist und die Entscheidung, wie es im Dialog weitergeht treffen jetzt Selektionsfunktionen (siehe Abschnitt 3.2.1). Die zentrale Architektur des Systems besteht aus drei Kernmodulen (siehe Abbildung 4.1). Das zentrale

Modul stellt der Dialog dar, hier laufen sämtliche Ein- und Ausgabe-Kanäle zusammen. Der Spracherkenner ermöglicht die Kommunikation des Benutzer mit dem Dialog-System und die Fahrstuhlsteuerung stellt das dritte Modul dar. Eigentlich sollte man von vier Kernmodulen ausgehen, da der Spracherkenner nur die unidirektionale Kommunikation zwischen Benutzer und Dialog ermöglicht. Das vierte Modul wäre somit die Sprachausgabe, aber in dem vorliegenden System ist die Sprachausgabe lediglich durch das Abspielen einzelner WAVE-Files realisiert und es erscheint etwas hochgegriffen, dies als eigenständiges Modul zu bezeichnen. Bei Systemen, die über eine größere Variabilität der Sprachausgabe verfügen sollen, ist es jedoch sinnvoll von einem eigenständigen Modul für die Sprachausgabe zu sprechen.

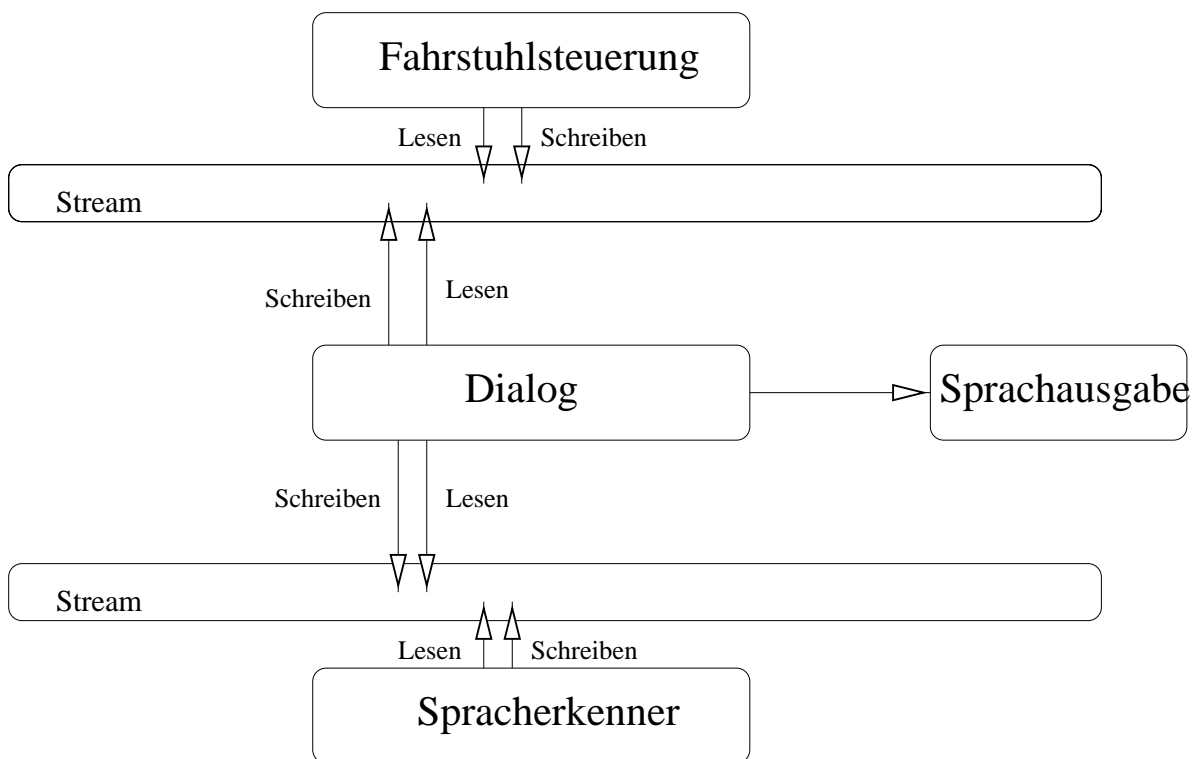


Abbildung 4.1.: Überblick über die Kern-Module

4.1.1. Anforderungen an das Dialogmodul

Wie bereits erwähnt ist das Dialogmodul das zentrale Modul in diesem Design. Denn hier laufen nicht nur alle Ein- beziehungsweise Ausgabe-Kanäle zusammen, sondern sämtliche Reaktionen des System, ob sprachlicher oder mechanischer Hinsicht, werden von diesem Modul veranlaßt. Das Dialogmodul muß also, im Hinblick auf den Dialogverlauf, ein Regelwerk enthalten, daß auf der einen Seite den Benutzer bei der Bedienung des Systems unterstützt (Frage – Antwort) und auf der anderen Seite aus den Anfragen des Benutzers die entsprechenden Fahrtbefehle für die Fahrstuhlsteuerung ableiten kann.

Im Hinblick auf die Steuerung des Dialogverlaufs muß das Dialogmodul eine größt mögliche

Flexibilität aufweisen. So muß das System in der Lage sein auf verschiedenartige natürlich-sprachliche Anfragen des Benutzer zu reagieren und im Falle mehrdeutiger Befehle, den Benutzer zu einem eindeutigen Befehl führen. In dem vorliegenden System ist der grobe Dialogverlauf in der folgenden Form realisiert (Siehe auch Abschnitt ??). Der Benutzer muß zunächst durch die Schlüsselwörter “Fahrstuhl” oder “Aufzug” den Dialog anstoßen und hat danach die Möglichkeit entweder einen direkten Fahrbefehl zu tätigen (zum Beispiel “Fahrstuhl, dritter Stock”) oder er kann die Nachfrage des Systems (“Wohin möchten Sie bitte”) abwarten. Die Struktur der Fahrbefehle ist so flexibel wie möglich gehalten, so kann der Benutzer direkt Etagen (“Ich möchte in den dritten Stock”), Personennamen (“zu Professor Pinkal, bitte”) oder Gruppen beziehungsweise Einrichtungen (“bitte zur Systemgruppe”) benennen. Im Falle einer ambigen Anfrage “Ich möchte zur Phonetik” (der Fachbereich Phonetik befindet sich auf zwei Etagen), fragt das System nach, in welche der beiden Etagen der Benutzer möchte (“ Möchten Sie in den vierten oder fünften Stock”).

4.1.2. Kommunikation der Module

Ein zentraler Punkt bei der Entwicklung eines solchen System ist die Kommunikation der einzelnen Module. So sollte das Dialogmodul ständig die Möglichkeit haben Anfragen an das Spracherkennermodul zu schicken, um zu erfahren, ob der Benutzer etwas gesagt hat. Um keine Äußerungen des Benutzer zu verlieren, sollte der Spracherkenner ständig aktiv sein, also parallel, als eigenständiger Prozeß laufen. Weiterhin sollte der Dialog die Möglichkeit haben ständig Informationen über den Zustand des Fahrstuhls (in welchem Stock er sich befindet, usw.) abzufragen. Hieraus ergibt sich bereits eine starke Anforderung an das Systemdesign. Die Module müssen prinzipiell unabhängig von einander laufen. Diese Nebenläufigkeit erschwert natürlich die Kommunikation der einzelnen Module. Eine Möglichkeit miteinander zu kommunizieren, sind *Streams*. *Streams* sind im Grunde genommen Datenströme, die einen definierten Anfang, aber kein Ende besitzen und bieten den Vorteil, daß der schreibende Prozeß immer am Ende des *Streams* etwas hinzufügen kann und daß der lesende Prozeß trotzdem noch ältere Information vom *Stream* lesen kann (näheres zu Implementierung der *Stream*-Kommunikation siehe Abschnitt 3.??). Ein Problem bei der Kommunikation via *Streams* ist die Ungewißheit, zu welchem Zeitpunkt eine Information auf einen *Stream* geschrieben wurde, diese Problematik wird im Abschnitt Zeitverwaltung näher betrachtet.

4.2. Modellierung

Im folgenden werden die wesentlichen Konzepte, die in die Implementierung dieses Systems eingeflossen sind, dargestellt. Im Mittelpunkt stehen hierbei der Dialog, die Anbindung des Spracherkenners und die Zeitverwaltung.

4.2.1. Dialog

Die Anforderungen an den Dialog wurden ja bereits in Abschnitt (??) beschrieben, wie lassen sich nun diese Anforderungen im Hinblick auf Effizienz und Wartung beziehungsweise Erweiterung modellieren? Wir haben uns dafür entschieden den Dialog in Form eines endlichen Automaten zu realisieren. Allerdings wurde das Konzept des endlichen Automaten um einen globalen Speicher, der in jedem Zustand des Automaten zugänglich ist, und um die Möglichkeit Aktionen, beim Übergang von einem Zustand in einen anderen auszuführen, erweitert. Diese Aktionen sind Funktionen, die zum Beispiel eine Sprachausgabe oder das Absetzen eines Fahrtbefehls ermöglichen.

Der Dialog besteht aus einer endlichen Menge von Zuständen, diese Zustandsmenge ist eine Liste von Datenstrukturen, die ihrerseits wiederum Datenstrukturen enthalten. So besteht ein Zustand aus einem Feld (*record*), das auf der einen Seite die möglichen Übergänge in einen Folgezustand (auch in Form eines Feldes) und auf der anderen Seite eine Selektionsfunktion enthält. Diese Selektionsfunktion wählt aus der Menge der möglichen Übergänge denjenigen aus, der am ehesten der aktuellen Situation entspricht. In einem Zustand des Dialogs, in dem vom Benutzer eine Äußerung erwartet wird, stellt die Selektionsfunktion zunächst eine Anfrage an das Spracherkennermodul und wählt dann abhängig vom Ergebnis dieser Anfrage den passenden Übergang in einen Folgezustand aus.

Die Spezifikation der Übergänge erfolgt in Form von Funktionen, wobei jede Übergangsfunktion als Argument ein Atom, das dem Folgezustand entspricht, erwartet. Der Rückgabewert dieser Funktionen besteht wiederum aus einem Feld, das neben dem Folgezustand und einem Text, der den Übergang beschreibt, eine anonyme Prozedur enthält, in deren Rumpf der Aufruf der einzelnen Aktionen kodiert ist. Durch diesen rekursiven Aufbau der zum Automaten gehörigen Datenstrukturen und der daraus resultierenden kompakten Darstellung wird sowohl die Entwicklung beziehungsweise Wartung als auch die Verarbeitung des Automaten vereinfacht.

Wie bereits am Anfang dieses Abschnittes erwähnt, verfügt dieser Automat über einen zusätzlichen Speicher. Dieser Speicher dient in erster Linie dazu die Komplexität des Automaten zu verringern, da eventuelle Argumente nicht in Form zusätzlicher Zustände kodiert werden müssen. Das Konzept des zusätzlichen Speichers wird unter anderem für die Verwaltung der zeitlichen Abfolge benötigt, so muß die Anfrage der Selektionsfunktion an den Spracherkenner einen Zeitpunkt enthalten, ab wann eine relevante Äußerung des Benutzer zu erwarten ist. Auch wird ein eventuell erkanntes Ziel in diesen Speicher geschrieben, um dieses Ziel für nachfolgende Zustände zu erhalten.

Um diesen Automaten verarbeiten zu können wird ein Interpretierer benötigt. Hierbei handelt es sich im wesentlichen um eine rekursive Funktion, die ausgehend von einem Zustand mittels der Selektionsfunktion den Folgezustand berechnet und die zu dem berechneten Übergang gehörige Aktion ausführt. Der Endzustand, der eigentlich nur aus

einem leeren Übergang zum Startzustand besteht, muß gesondert gekennzeichnet sein, da der Interpretierer diesen Zustand nutzt, um systemweit den Speicher zu bereinigen. Diese Prozesse der Speicherplatzverwaltung werden im Abschnitt ?? näher erläutert.

4.2.2. Schnittstellen

Streamkommunikation

Wie bereits erwähnt kommuniziert das Dialogmodul mit Spracherkenner und der Fahrstuhlsteuerung über *Streams*. Abstrakt gesehen produzieren oder konsumieren *Streams* sequentiell Datenstrukturen, die durch Ein- oder Ausgabe-Operatoren gelesen beziehungsweise geschrieben werden.

Sowohl beim Spracherkenner als auch bei der Steuerung handelt es sich um externe Prozesse, die in einer eigenen Shell laufen. In dem vorliegenden System werden für jedes dieser Module zwei *Streams* verwaltet. Der erste *Stream* ist die direkte Verbindung mit dem Modul über *std-in* beziehungsweise *std-out*, alle Daten liegen auf diesem *Stream* in ihrer ursprünglich Form vor (*Strings*). Diese rohen Daten werden in Datenstrukturen konvertiert und auf einen zweiten *Stream* geschrieben.

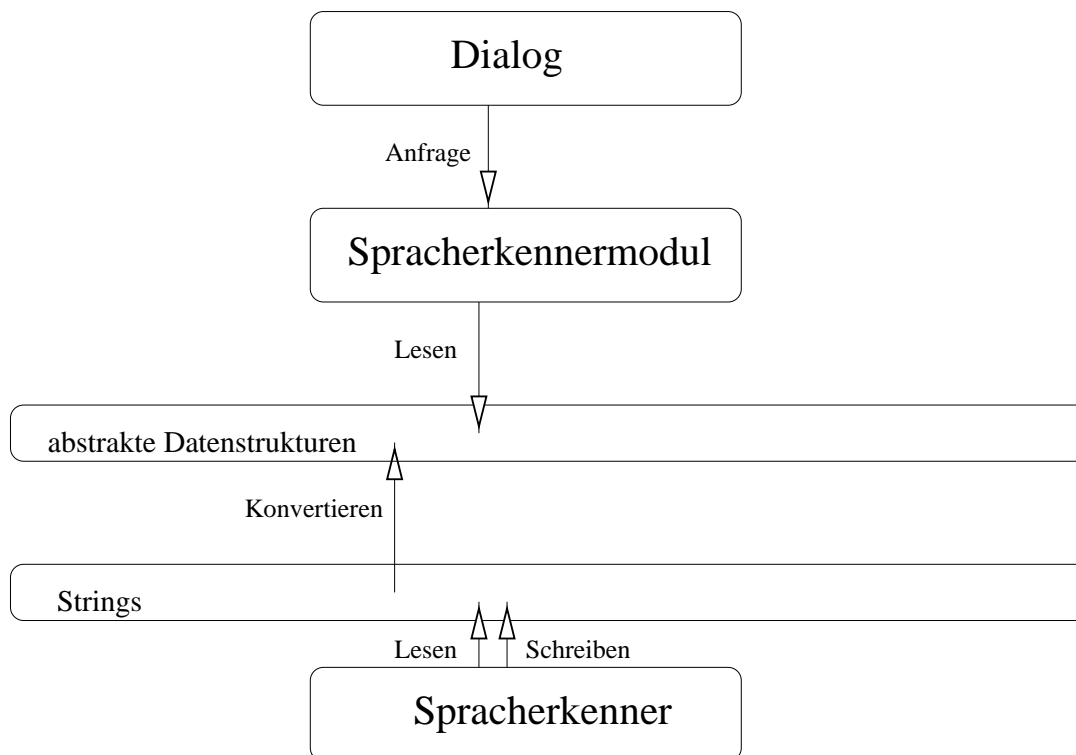


Abbildung 4.2.: Streamkommunikation am Beispiel des Spracherkenners

Im Falle des Spracherkenners (siehe auch Abbildung 4.2) besteht der erste *Stream* aus Paketen in *String*-Form, die Äußerungen darstellen. Diese Äußerungen werden analysiert und auf dem *Stream*, der die komplexe Datenstrukturen enthält, liegen die Wörter dieser Äußerungen in Form einer abstrakten Datenstruktur paketweise an. Diese Wörter verfügen neben den Informationen, um welches Wort es sich handelt und dessen Wahrscheinlichkeit, auch über ein Zeitintervall in dem das Wort aufgetreten ist. Der Vorteil dieses zweiten *Streams* liegt darin, daß Anfragen an den Spracherkenner wesentlich einfacher zu verarbeiten sind. So besteht auch die Möglichkeit mehrmals die gleichen Informationen des *Streams* zu lesen. Auch läßt sich auf diese Art und Weise recht komfortabel ein *Timeout*-Mechanismus realisieren. So kann nachträglich eine Datenstruktur *Timeout* an eine beliebige Stelle des *Streams* eingefügt werden, die es ermöglicht, die Suche auf dem *Stream* nach einer gewissen Zeit abzubrechen.

Anbindung des Spracherkenners

In der Vorliegenden Version des Systems kommt als Spracherkenner ein Programm der Firma Lernout&Hauspie zum Einsatz. Die Ergebnisse des Erkennungsprozesses werden paketweise auf einen *Stream* geschrieben. Es gibt zwei unterschiedliche Arten von Paketen, die jeweils durch die Schlüsselwörter *begin* und *end* umramt sind:

- *Control Statechange* der Form: `control(topic:statechange old:... new:... time:...)` wobei ein Zustandswechsel des Spracherkenners den Beginn einer Äußerung signalisiert, was zur Folge hat, daß der Dialog auf jeden Fall noch auf das Ergebnis dieser Äußerung wartet
- *rec(version:lh tot:... start:... end:...)*: wobei *tot* die Wahrscheinlichkeit und *start* bzw. *end* die Start- und End-Zeit der ganzen Äußerung repräsentieren. Ein solches Paket entspricht dem Ende eines Erkennungsprozesses und enthält zusätzlich die eventuell erkannten Wörter dieser Äußerung, zum Beispiel:

```
begin
rec(version:lh tot:66 start:678594.143 end:678595.653
Fahrstuhl()
in()
den()
dritten()
Stock()
end()
```

Eine wichtige Entscheidung beim Design eines solchen Systems ist die Art und Weise wie die Spracherkennung ablaufen soll. Soll die Spracherkennung immer aktiv sein (*Continuous-speech-mode*), d.h. immer "zuhören", oder soll sie nur dann aktiv sein, wenn ein externer

Prozeß dem Erkennen ein Signal geschickt hat (*Open-Microphone*).

In dem vorliegenden System haben wir uns für die erste Variante, der konstanten Erkennung entschieden. Allerdings mit der Einschränkung, daß dem Erkennen ein *break* Signal geschickt werden kann, das den Erkennen sofort in einen *IDLE*-Modus (Ruhezustand) versetzt. Dieses *break*-Kommando wird dann benötigt, wenn das System selbst eine Sprachausgabe tätigt, um zu verhindern, daß der Erkennen diese Äußerungen verarbeitet und auf den *Stream* schreibt ("Selbstaufnahme").

Auf Seite des Spracherkennermoduls werden die relevanten Pakete, die auf diesem *Stream* liegen, herausgefiltert, konvertiert und auf einen weiteren *Stream* geschrieben. Dieser zweite *Stream* dient dann als Grundlage für Funktionen, die innerhalb von Zeitintervallen nach bestimmten Wörtern suchen. So wartet die Funktion *WaitKeyword* ab einem gewissen Zeitpunkt solange, bis der Spracherkenner eins der Schlüsselwörter erkannt hat, die als Parameter mitgegeben werden. Dazu wird das erste Paket auf dem zweiten *Stream* untersucht, ob dessen Startzeit kleiner als die, durch den Funktionsaufruf übergebene Startzeit der Suchanfrage ist. Ist dies der Fall, wird das nächste Paket auf dem *Stream* untersucht, andernfalls wird getestet, ob eins der Schlüsselwörter in diesem Paket enthalten ist. Die Funktion *WaitKeyword* wartet solange, bis eins der Schlüsselwörter auf dem *Stream* anliegt. Die zweite Funktion, die als Schnittstelle zum Spracherkenner dient, ist die Funktion *Nextutterance*. Diese Funktion wartet nur eine bestimmte Zeit, ob ein Element einer Menge von Wörtern auf dem zweiten *Stream* anliegt. Taucht in dieser Zeitspanne das Wort nicht auf, so gibt die Funktion *nil* zurück. So wird der Dialog abgebrochen, wenn zum Beispiel auf die Frage des Systems "Wohin möchten Sie bitte?" nach einer gewissen Zeitspanne keine Reaktion gekommen ist.

Anbindung der Fahrstuhlsteuerung

Die Schnittstelle zur Fahrstuhlsteuerung besteht im wesentlichen aus einem Treiber, der die Kommunikation mit der Fahrstuhlsteuerung über die Serielle Schnittstelle verwaltet (zur Beschreibung dieses Treibers siehe ??). Dieser Treiber ermöglicht auf der einen Seite Fahrbefehle (in der Form *Ziel(0)*) abzusetzen und auf der anderen Seite kann der augenblickliche Zustand des Fahrstuhl abgefragt werden (in der Form *info*).

Auf der Dialog-Seite sind die Anforderungen an die Fahrstuhlsteuerung recht einfach, Fahrbefehle müssen abgesetzt werden und Anfragen über den Zustand des Fahrstuhl, zum Beispiel in welchem Stock er sich befindet, müssen bearbeitet werden. Dies läßt sich mit der Funktionalität des Treibers einfach umsetzen. Ein Problem bereiten jedoch die Latenzzeiten des Kommunikationsprotokolls. So kann es durchaus zwischen 500 – 750 msec dauern, bis eine Anfrage über den Zustand des Fahrstuhls beantwortet wird. Da die Anfragen sequentiell abgearbeitet werden, hat dies zur Folge, daß in dieser Zeit keine Fahrbefehle abgesetzt werden können. Um jetzt eine Warteschlange für die Befehle und Anfragen zu vermeiden, wurde das folgende Schema für dieses Modul implementiert. Das Fahrstuhlsteuerungsmodul fragt zunächst einmal immer den Zustand des Fahrstuhls ab und beantwortet Anfragen des Dialogs immer mit der zuletzt eingegangenen Information. Wenn der Dialog kurz vor dem Absetzen eines Fahrbefehls steht, wird dies dem Modul Fahrstuhlsteuerung

mitgeteilt und die Steuerung unterbricht die ständige Abfrage des Fahrstuhlzustandes. Nachdem der Fahrbefehl abgesetzt wurde, beginnt die Steuerung wieder mit kontinuierlichen Zustandsabfrage. Dadurch wird sichergestellt, daß kein Fahrbefehl auf eine noch laufende Anfrage warten muß.

4.2.3. Zeitverwaltung

Die Zeitverwaltung (*Timing*) ist in einem System wie diesem, aufgrund der Echtzeitbedingung und durch die Notwendigkeit der Synchronisation verschiedener Prozesse, eine heikle Angelegenheit. Die Synchronisation wird durch die Nebenläufigkeit des Spracherkenners und der Fahrstuhlsteuerung zusätzlich erschwert. Wie bereits im vorherigen Abschnitt beschrieben, kommuniziert das Dialog-Modul mit dem Spracherkenner über zwei Funktionen (*waitKeyword* und *Nextutterance*), die ständig Zugriff auf alle Pakete haben, die der Spracherkenner geliefert hat. Bei dieser Art der Informationsverwaltung stellt sich jedoch die Frage, welches Paket überhaupt zeitlich gesehen das richtige Paket für die aktuelle Anfrage ist.

Wie beschrieben verfügt jedes Paket über eine Start- und End-Zeit. Um das richtige Paket auswählen zu können, muß das Dialog-Modul über die Information verfügen, ab welchem Zeitpunkt vom Benutzer eine Eingabe erwartet wird beziehungsweise erwartet werden könnte. Aus diesem Grund gibt es neben dem Speicher für ein eventuelles Ziel auch einen Speicher für den Zeitpunkt, ab dem relevante Eingaben beginnen könnten. Der Inhalt der Speicherzelle wird dann bei einem Aufruf der Funktionen *Nextutterance* oder *waitKeyword* als Argument übergeben.

Wichtig ist jetzt noch, wann dieser Zeitspeicher gesetzt wird. Wenn das System zum Beispiel selbst eine Sprachausgabe getätigt hatte, wäre es am sinnvollsten direkt nach dem Ende der Äußerung diesen Speicher zu setzen, um eine eventuelle Äußerung des Benutzer nicht zu verpassen.

Dies wirft jedoch das Problem der "Selbstaufnahme" auf. Wenn das System selbst eine Sprachausgabe tätigt, ist es sehr wahrscheinlich, daß der Spracherkenner diese wieder aufnimmt und als Eingabe dem System zur Verfügung stellt. Allerdings benötigt der Spracherkenner nach dem Ende der Äußerung des Systems noch ein gewisse Zeit diese zu Verarbeiten. Während dessen ist der Erkenner allerdings inaktiv, d.h. weitere Sprachsignale können nicht verarbeitet werden und sind verloren. Würde der Benutzer direkt auf die Äußerung des Systems etwas sagen, zum Beispiel gegen eine Zielbestätigung protestieren, würde das System von diesem Protest nichts mitbekommen.

Um dieses Phänomen der Selbstaufnahme zu vermeiden, müssen der Spracherkenner und der Dialog zeitlich synchronisiert sein, d.h. es darf immer nur ein Prozeß den Audiokanal nutzen. Wie bereits im vorherigen Abschnitt beschrieben, ist dies in diesem System durch ein *Break*-Kommando realisiert, der Spracherkenner hat den Vorrang und prinzipiell

immer Zugriff auf den Audiokanal, wenn jedoch von Seiten des Dialogs eine Sprachausgabe geplant ist, schickt dieser dem Spracherkenner ein *Break*-Kommando. Solange diese Kommando nicht widerrufen wurde, ist der Spracherkenner inaktiv. Nach der Sprachausgabe, widerruft der Dialog das *Break*-Kommando, setzt den Zeitspeicher und der Spracherkenner wird wieder aktiv.

Ein weiterer Punkt in dem das Timing eine Rolle spielt, ist die Synchronisation des Dialoges mit der Fahrstuhlsteuerung (zur Implementierung des Treibers zur Fahrstuhlsteuerung siehe Kapitel ??). Solange kein Fahrbefehl abgesetzt werden soll, fragt der Dialog in kurzen Abständen den Zustand des Fahrstuhls ab. Da das verwendete Protokoll relativ lange Latenzzeiten aufweist, würden Probleme entstehen, wenn der Dialog ein Fahrtziel absetzen möchte, aber noch auf ein Ergebnis des Kommandos vorher gewartet werden muß. Zur Vermeidung von längeren Wartezeiten, wird deshalb bereits kurz bevor der Dialog das Fahrtkommando absetzt der Prozeß der Zustandsabfrage angehalten, und erst danach wieder aktiviert.

4.3. Implementierung

Das vorliegende System wurde in der Programmiersprache *Oz* entwickelt. Die Vorteile dieser Programmiersprache liegen im wesentlichen in der Plattformunabhängigkeit, der Netzwerkfähigkeit und der sehr einfachen Handhabung von nebenläufigen Prozessen. Weitere wichtige Eigenschaften dieser Sprache sind auch die direkte Unterstützung rekursiver Datenstrukturen, die effiziente Verarbeitung rekursiver Funktionen und natürlich die automatische Speicherbereinigung (*Garbage collection*). Die eben angeführten Eigenschaften bieten auch andere Hochsprachen, allerdings meist nur eine Teilmenge dieser Eigenschaften. Weiterhin kam auch die Sprache *C* zum Einsatz, da die API (Funktionsbibliothek) des Spracherkenners in *C* vorlag.

4.3.1. Modularisierung

Um die Entwicklung und Wartung zu erleichtern wurde das komplette System weitestgehend modularisiert. Im Anhang ?? befinden sich die einzelnen Dateien, an dieser Stelle werden lediglich die wesentlichen Module namentlich aufgelistet.

- *start.oz*: Initialisierung des Systems
- *lift.oz*: Interpretierer für den Dialog
- *dialog.oz*: Spezifikation des Automaten
- *recognizer.oz*: Verwaltung des *high-level* Daten-Streams
- *recognizer-interface.oz*: Verwaltung des Spracherkenner-Streams (auf *String*-Ebene)

- *lift-control.oz*: Verwaltung des *high-level* Daten-Streams
- *lift-control-interface*: Verwaltung des Fahrstuhlsteuerungs-Streams (auf *String*-Ebene)
- *parameter.oz*: Verwaltung aller für das System relevanter Parameter
- und weiteren “Hilfsprogrammen”: *stack.oz*, *ticket.oz*, *memo.oz*, *time.oz*, ...

4.3.2. Verwaltung des Arbeitsspeichers

Eigentlich fallen bei einem solchen System keine größere Strukturen an, die ein ausgefeiltes Speichermanagement notwendig machen, zumal die Hochsprache *Oz* bereits über einen eingebaute Speicherbereinigung (*Garbagecollector*) verfügt. Diese ermöglicht, während der Laufzeit des Systems, Speicherbereiche, die nicht referenziert werden, wieder freizugeben. Der Einsatz von *Streams* erfordert jedoch gewisse Speicherverwaltungsroutinen, da *Streams* zwar über einen eindeutigen Anfang, aber nie ein Ende verfügen. Dies hat für die Speicher-verwaltung zur Folge, daß diese Datenstruktur immer größer wird und Speicherbereiche vom Anfang des *Streams* nicht frei gegeben werden können, solange noch Referenzen auf diese Teile existieren. Um dieses Problem zu umgehen, muß zu einem geeigneten Zeitpunkt der Lesekopf auf dem *Stream* nach vorne bewegt werden. Alle Informationen, die sich dann vor dem Lesekopf befinden, werden dann vom *Garbagecollector* frei gegeben.

4.3.3. Plattformunabhängigkeit

Die Programmiersprache *Oz* wurde, wie bereits erwähnt, auch wegen ihrer Plattformunabhängigkeit ausgewählt. Diese Entscheidung hat sich als eine sehr glückliche erwiesen. Ursprüngliche wurde dieses Dialogsystem unter *Linux* entwickelt. Damals wurde auch noch der in Kapitel ?? beschriebene Spracherkenner HTK der Firma Entropic verwendet. Während der Entwicklung der Schnittstellen zum Spracherkenner traten jedoch einige Probleme auf, so war es nur mit großem Aufwand möglich zum Startzeitpunkt der Äußerung (Echtzeitverhalten) eine Meldung des Spracherkenners zu erhalten. Auch bereitete die lange Verarbeitungszeit des Erkenners größere Probleme, die allerdings auf noch nicht richtig ausgereizte Einstellungen in der Konfigurationsdatei des Erkenners zurück zu führen sind. Aus diesem Grunde entschlossen wir uns zweigleisig zu fahren und bis die Probleme mit HTK beseitigt sind, einen anderen Erkennen einzusetzen. Augenblicklich kommt in unserem System ein Spracherkenner der Firma Lernout & Hauspie zum Einsatz, der allerdings nur unter *Windows* lauffähig ist.

Bei der Portierung des System nach *Windows* traten zwei Problemfelder hervor. So stellte sich heraus, daß Pipes, das heißt bidirektionale *Streams*, unter *Windows* nicht die gleiche Funktionalität haben wie unter *Linux/Unix*. Aus diesem Grund mußte die Kommunikation zwischen dem Spracherkenner und dem Spracherkennerinterface über eine *Socket*-Verbindung realisiert werden. Da der Treiber für die Fahrstuhlsteuerung bisher nur unter *Linux* zur Verfügung steht, läuft die Kommunikation in diesem Bereich auch über eine Netzverbindung. Ein weiteres Problemfeld ist der Einsatz von Peripherieprogrammen, wie

zum Beispiel ein Tool zur Soundausgabe. Die meisten Programme unter *Windows* kennen keine Kommandozeilen-Argumente, sondern können nur über den Einsatz von Maus und Tastatur gesteuert werden, was die Integration in ein solches System erschwert beziehungsweise fast unmöglich macht.

4.3.4. Parameter des Systems

Um das System so weit wie möglich flexibel zu halten, können in der Datei *parameter.oz* verschiedene Parameter gesetzt werden. Einige dieser Parameter beziehen sich auf die Selektion betriebssystemabhängiger Funktionen andere spezifizieren Verzeichnispfade. Zusätzlich gibt es noch einen Parameter *TimeOut* der direkt Einfluß auf das Dialogverhalten hat, da an dieser Stelle definiert wird, wie lange auf eine Äußerung des Benutzers gewartet wird. Die folgende Aufstellung enthält die wichtigsten Systemparameter, die vor jedem Systemstart überprüft werden sollten:

- *Control* Optionen: *fake / real / none*
Zu Testzwecken man auf die Fahrstuhlsteuerung verzichten bzw. diese vortäuschen
- *ControlInterface* Optionen: *internet / local*
Setzt die Verbindungsart mit der Fahrstuhlsteuerung
- *Recognizer* Optionen: *HTK / LH / fake*
Selektiert den Spracherkenner bzw. zu Testzwecken kann man auf den Spracherkenner verzichten bzw. diesen vortäuschen
- *ControlLocation* Optionen: *coli / ps*
Dieser Parameter ist wichtig für bei der Internetverbindung mit der Steuerung
- *Audio* Optionen: *windows / linux* Selektiert das Tool und das zugrundliegende Betriebssystem für die Audiowiedergabe
- *TimeOut* Optionen: Zeitangabe in Millisekunden
Spezifiziert, wie lange auf eine Äußerung des Benutzer gewartet werden soll

Die restlichen Parameter in dieser Datei müssen normalerweise nicht verändert werden, da sie sich im wesentlichen auf Pfade und interne Vorgänge beziehen. Am Ende dieser Datei kann für Testzwecke und zur Fehlersuche die Ausgabe des Systems modifiziert werden. So kann gezielt die Ausgabe einzelner Module verfolgt werden. Im normalen Betrieb ist diese Option deaktiviert.

4.3.5. Versionskontrolle & MakeFile

Sämtliche Komponenten des Systems wurden in einem *CVS*-Verzeichnis verwaltet. *CVS* ist ein System zur Versionskontrolle und ermöglicht alte Versionen von Dateien wiederherzustellen. Weiterhin verwaltet dieses Tool zu jeder Datei wer zu welchem Zeitpunkt

Änderungen an dieser Datei gemacht hat. Zusätzlich gibt es ein *Makefile*, das auf der einen Seite bei Bedarf die einzelnen Module kompiliert und das komplette System startet.

5. Spracherkennung im Fahrstuhl mit HTK

Obwohl in der automatischen Spracherkennung mittlerweile hohe Erkennungsraten erzielt werden, stellt ein in einen Fahrstuhl eingebauter Spracherkenner besondere Anforderungen und erwartet vom Entwickler bestimmte Entscheidungen, um die optimalste Erkennungsrate zu erzielen.

- Voraussetzung: Sprecherunabhängigkeit

Da ein Fahrstuhl eine potentiell infinite Anzahl an Benutzern hat, die das Gebäude besuchen, muß das verwendete System, anders als z.B. ein Diktiersystem für den PC oder ein Spracherkenner im Auto, sprecherunabhängig sein. Das heißt, daß sprecherspezifische Unterschiede wie Sprechgeschwindigkeit, Sprechtonhöhe (Grundfrequenz) und dialektale Herkunft keine Unterschiede in der Erkennung bewirken dürfen.

- Folge: eingeschränktes Lexikon

Eine optimale Erkennung trotz dieser Variabilität innerhalb der Gruppe der Benutzer kann normalerweise nur bei einem eingeschränkten Lexikon gewährleistet werden. Diese Einschränkung stellt in unserem Fall kein Problem dar, da hauptsächlich Stockwerksangaben, Lehrstuhlbezeichnungen und Professorenennamen erkannt werden müssen.

- Ziel: continuous speech recognition

Da die Variabilität der Benutzeräußerungen in einem Fahrstuhl sehr beschränkt ist, würde ein *keyword spotting* ausreichen. Hierbei wird aus einem kompletten Satz, wie z.B. "Ich möchte in den dritten Stock" nur das Schlüsselwort "dritten" erkannt. Wir haben uns jedoch für continuous speech recognition entschieden.

- Weg: Lautmodelle trainieren

Parallel zum Training der Lautmodelle war eine erste Überlegung auch Wortmodelle zu trainieren. Hierbei werden einzelne Laute nur aus solchen Wörtern geschnitten und trainiert, die hinterher auch erkannt werden sollen. Dies hat den Vorteil, daß Phone, die aus diesen Wörtern genommen werden, weniger variabel sind, als einzelne Phonmodelle aus allen möglichen Kontexten. Da die Datenbank, mit der wir gearbeitet haben, das Kiel Korpus, jedoch nicht genügend Material bezüglich unseres doch sehr speziellen Wortschatzes hergab, und auch weitere, in dem *Wizard-of-Oz* Experiment erhobene Daten nicht die er-

forderliche Menge an Realisierungen eines jeden Wortes brachte, die nötig gewesen wäre, um Wortmodelle zu trainieren, fiel die Wahl auf Lautmodelle.

- Frage: kommerzieller Spracherkenner oder ein eigens trainierter?

Die Entscheidung für einen kommerziellen Spracherkenner (z.B. ASR 1600 von Lernout & Hauspie) hätte einige Vorteile geboten. Dieses System ist beispielsweise im ersten Jahr der Benutzung kostenlos und von anderen Anwendern bereits getestet. Desweiteren sind die Modelle trainiert und ein Lexikon ist bereits vorhanden. Da dieser Erkennen in vielen verschiedenen Umgebungen einsetzbar sein muß, ist er außerdem so trainiert worden, daß er sehr robust gegenüber verschiedenen akustischen Hintergründen, d.h. Geräusch im Sprachsignal, ist.

Der größte Nachteil dieser Lösung ist, daß die Lautmodelle nicht speziell auf den Einsatz im Fahrstuhl trainiert wurden. Die Robustheit des Spracherkenners kann zwar ein großer Vorteil sein, allerdings ist die individuelle Anpassung der bereits vortrainierten Modelle auf unsere spezielle Umgebung im Fahrstuhl nur sehr eingeschränkt möglich. Als weitere Nachteile sind zu nennen, daß die Software zu diesem Erkennen kein fertiges Paket ist, sondern noch weiter angepaßt werden muß, dieser Erkennen nur unter Windows läuft und, sollte er kommerziell genutzt werden, sehr hohe Kosten entstünden.

Demgegenüber stand die Entscheidung für einen selbsttrainierten Spracherkenner mit dem Hidden Markov Tool Kit (HTK) der Firma Entropic. Die Nachteile dieser Variante ergeben sich zum Großteil aus den Vorteilen des kommerziellen Erkenners. So ist ein für einen speziellen akustischen Hintergrund trainierter Erkennen nicht so robust gegenüber anderen Hintergrundgeräuschen, es sei denn, er wird mit einer extrem großen Datenmenge trainiert, von der nicht angenommen wurde, daß sie durch ein einfaches Experiment, bei dem im Fahrstuhl Aufnahmen der Benutzer gemacht werden, erzielt werden kann. Dieses ist außerdem ein Zeitaufwand, der bei einem kommerziellen Spracherkenner nicht aufgebracht werden muß. Desweiteren waren die Kosten für eine kommerzielle Nutzung von HTK unbekannt.

Dennoch bietet dieses System sehr große Vorteile. Abgesehen davon, daß es unter Unix läuft (wie der übrige Teil der für unseren Fahrstuhl entwickelten und eingesetzten Software auch) und wesentlich billiger ist, als der Erkennen von Lernout & Hauspie, hat sich HTK in der Forschung am Institut für Phonetik als flexibel und schnell erwiesen. Der direkte Zugriff auf akustische Parameter und die Möglichkeit eigene Phonmodelle zu trainieren, um den Erkennen individuell an unseren Fahrstuhl anzupassen, überwiegen deutlich die Vorteile des ASR 1600. Die eventuell fehlende Robustheit kann durch eine Vorverarbeitung des Sprachmaterials verbessert und die Datenmenge durch das Kiel Korpus aufgestockt werden. Die verschiedenen Anforderungen des Fahrstuhls und unser Wunsch nach einem robusten, individuellen und ausbaufähigen System legen eine Hidden Markov Modellierung unseres Spracherkenners nahe.

Sprache ist ein Signaltyp, der sich durch hohe Variabilität und durch eine Entwicklung über die Zeit auszeichnet. Um ein solches Signal adäquat widerzuspiegeln, bietet sich eine stochastisch-probabilistische Modellierung besonders an (siehe Abbildung 5.1).

McInnes und Mervyn (1988)¹ definieren Hidden Markov Modelle wie folgt: "The essential

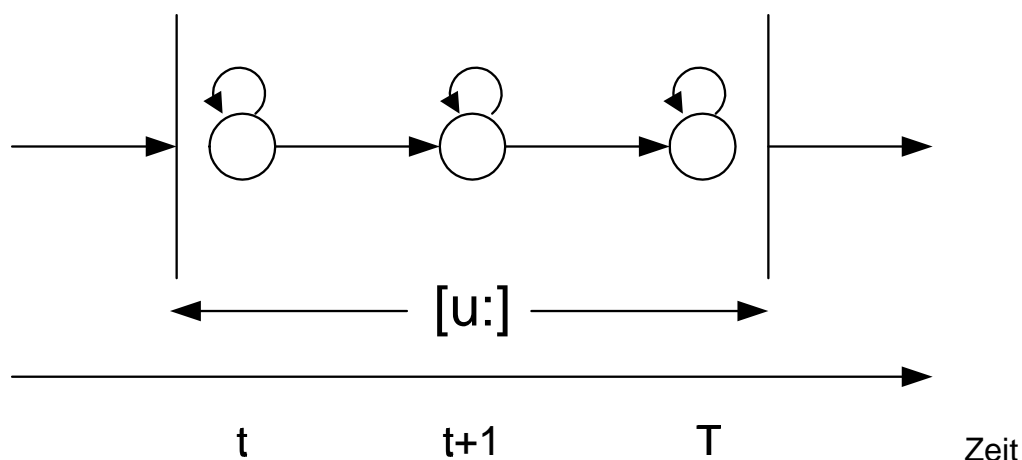


Abbildung 5.1.: HMM für den Laut [u:]

idea of the HMM approach is that each word in the vocabulary is represented by a set of states, including an initial and a final state, with probabilities of transitions from state to state, and for each state a probability distribution for the emission of a vector of acoustic parameters”. In unserem Fall werden nicht Wörter, sondern Laute modelliert, d.h. die drei Kreise in Abbildung 5.1 spiegeln die Zustände wider, in denen sich unser Modell zu einem bestimmten Zeitpunkt t jeweils befindet. Zum Zeitpunkt $t+1$ geht das Modell entweder in den nächsten Zustand über (transition probability) oder es bleibt im momentanen Zustand (self-loop-probability). In unserem Fall wird ein *left-to-right modell* angenommen. D.h. Das Modell kann nur von einem Zustand in den nächsten übergehen, nicht jedoch in einen vorhergehenden, und es können keine Zustände übersprungen werden. In jedem Zustand wird nun ein Output (*observation*) kreiert, solange, bis das Modell seinen Endzustand T erreicht hat. Spezifisch für einen Markov Prozess ist, daß Zustand $t+1$ probabilistisch bestimmt wird und nur von seinem direkt vorhergehenden Zustand t abhängig ist.

Im Folgenden wird der Prozess der Spracherkennung beschrieben, von der Bereitstellung des Trainingsmaterials bis hin zum eigentlichen Erkennungsprozess.

5.1. Spracherkennung

Zunächst wird kurz die Datenbank vorgestellt, die die Basis unseres Trainingsmaterials bildet und deren Anpassung an HTK, sowohl die Anpassung der Signaldateien als auch die der Labeldateien. Weiter werden die zwei Schritte des Trainings (Initialisierung und Re-estimation) beschrieben, deren Ergebnis die Hidden Markov Modelle sind. Im vorletz-

¹F. McInnes und A. Mervyn. Automatic speech recognition using word reference patterns. In Mervyn A. Jack und John Laver. *Edits 4: Aspects of speech technology*. Edinburgh: Edinburgh University Press. 1988

ten Abschnitt wird kurz auf die Struktur von Lexikon und Grammatik eingegangen, die anschließend zusammen mit den HMMs für den Testprozess benutzt werden.

5.1.1. Trainingsmaterial

Das ursprüngliche Trainingsmaterial, das Kiel Corpus, ist eine wachsende gesprochene Sprachdatensammlung der deutschen Lese- und Spontansprache, welche seit 1990 am Institut für Phonetik und digitale Sprachverarbeitung (ipds) aufgezeichnet und segmentell etikettiert wird. Derzeit umfasst das Kiel Corpus über vier Stunden etikettierter Lesesprache sowie knapp vier Stunden etikettierter Spontansprache. Der gelesene Teil umfasst von 26 weiblichen und 27 männlichen Sprechern gesprochene deutsche Prosa. Die drei spontan-sprachlichen Teile enthalten spontane Dialoge und Terminabsprachen von 26, 16 bzw. 10 Sprechern. Das Korpus lässt sich in drei Arten von Dateien untergliedern: Lexika (*.lsv), Signaldateien (spontane Sprache → *.r16, *.l16; gelesene Sprache → *.16) und Labeldateien (*.slh). Letztere sind mit den Signaldateien zeitlich synchronisiert und enthalten deren Verschriftung, in Form von orthographischer und kanonischer Transkription und segmentellen Labels.

Lautmodelle

Da HTK nicht mit dem Sprachsignal direkt, sondern, mit aus ihm abgeleiteten Parametern arbeitet, und auch die im Kiel Korpus verwendeten Label zu komplex für unsere Zwecke sind, wurden im Vorfeld sowohl die Signaldateien als auch die Labeldateien des Kiel Korpus für HTK angepasst.

Das Skript *HCoppy* (siehe Anhang D) leitet aus den Signaldateien akustische Parameter ab (in unserem Fall zwölf MFCC (*Mel Frequency Cepstral Coefficients*), Energie und Delta-Parameter, d.h. die Differenz zwischen den gleichen Parametern mehrerer Vektoren als gewichtete Mittelwerte), die als Merkmalsbündel (*parameter vectors*) in .mfc-Dateien geschrieben werden. Diese Dateien werden anschließend als Trainingsmaterial für den Spracherkenner verwendet.

Die Umwandlung der Labeldateien wurde vorgenommen, da die im Kiel Korpus verwendete Annotation detaillierter und komplexer ist als für unsere Zwecke notwendig. Beispielsweise werden nicht nur zusätzliche Informationen zu Betonung und Melodie (*Prosodie*) gegeben, sondern auch auf lautlicher Ebene Unterscheidungen getroffen, die in unserem Fall nicht relevant sind. Ein Beispiel dafür findet sich in Abbildung 5.2. Hier wird bei nasalem Vorkontext das /d/² in “dreizehn” als [n] realisiert, welches die gleiche Artikulationsstelle hat. Das wird im Kiel Korpus³ mit “d-n” kenntlich gemacht. Für uns ist das Label “n”, das die endgültige Realisierung angibt jedoch ausreichend.

²Ein Laut in // stellt das Phonem dar, während [] die phonetische Realisierung eines Lautes symbolisiert.

³Für die Bedeutung der restlichen Labels, die im Kiel Korpus benutzt wurden, siehe K. J. Kohler. *Arbeitsberichte (AIPUK) Nr. 29*. Kiel: Institut für Phonetik und digitale Sprachverarbeitung Universität Kiel. 1995

Für die Umwandlung der Labeldateien wurden zwei Perl-Skripte angewendet, die sich im Anhang B befinden. *lab_one_mit_grenzen.pl* wandelt eine einzige Datei im Kiel Korpus-Format in eine Datei im HTK-Format um, während *lab_all_mit_grenzen.pl* das gleiche für eine Liste von Dateien macht. Das Ergebnis des zweiten Skripts ist eine *.mlf* Datei (Master Label File), die alle HTK-Label-Dateien enthält, die im weiteren Trainingprozeß benutzt werden. Die Syntax sieht folgendermaßen aus:

```
lab_one_mit_grenzen.pl  Kiel-Korpus-Label-Datei      HTK-Label-Datei
lab_all_mit_grenzen.pl  Kiel-Korpus-Label-Dateien-List  MLF-Datei
```

Die Abbildung 5.2 zeigt eine Beispieldatei des Kiel Korpus. Sie ist wie folgt strukturiert:

1. Dateiname
2. ortographische Repräsentation, endet mit 'oend'
3. kanonische Repräsentation, endet mit 'kend'
4. Transkription, die durch die Plazierung der Labels des Segmentierers erstellt wurde
5. Zeitangaben und Labels/Ettikettierung

HTK braucht nur den fünften Teil, allerdings in einem anderen Format. Die entsprechende HTK Version von der Datei im 5.2 zeigt die Abbildung 5.3. Wir werden zuerst das Kiel Korpus-Format und das HTK-Format detailliert beschreiben, und dann die Umwandlung des Ersten in das Zweite. Dafür muss zuerst den Begriff *Samples* erklärt werden. Bei der Umsetzung eines kontinuierlichen analogen Signals in ein, aus diskreten Werten bestehendes, digitales Signal wird die Spannung, die vom Mikrofon, Kassettenrecorder oder DAT-Recorder kommt, in kleinen zeitlichen Abständen gemessen. Diese Messergebnisse, die abgespeichert werden, nennt man *Samples* oder *Abtastrate*. Beim Kiel Korpus geschieht das 16000 mal pro Sekunde (sampling rate = Abtastrate = 16000 Hz). Ein Sample repräsentiert also 1/16000 Sekunde des Zeitsignals, d.h. 1/16 Millisekunden = 625 100-Nanosekunden-Einheiten⁴.

Das Kiel Korpus-Format werden wir anhand vom Beispiel (5.1.a) erklären. Jede Zeile enthält drei Spalten. In der ersten steht die Anzahl der *Samples*, die vom Aufnahmeanfang bis zu diesem Zeitpunkt im Sprachsignal gezählt wurden (in unserem Beispiel waren es **86**). Das Label selbst (**##g**) befindet sich in der zweiten Spalte. Schließlich registriert die dritte Spalte die Zeit in Sekunden (**0.0053125**) zwischen Aufnahmeanfang und dem jeweiligen Teil des Sprachsignals.

(5.1)	Samples	Label	Dauer in ms
a)	86	##g	0.0053125

⁴Das ist die Einheit, die HTK intern verwendet.


```

g317a005.s1h
SVA005: <#Klicken> gut , um <:<#Klopfen> dreizehn Uhr:> <#Klicken> .
oend
:k g 'u: t , Q U m+ :k d r 'aI t s e: n :k Q 'u: 6 . :k
kend
:k c: &%1^ g -h 'u: t -h , &0 Q- U m+ &0. &1)
:k- d-n r 'aI t s e:-@ n &1. &2^ :k- Q- -q 'u:6 . &2; &PGn :k
hend
      86 #:k          0.0053125
      86 #c:          0.0053125
      86 #&%1^        0.0053125
      86 ##g          0.0053125
      313 $-h         0.0195000
      818 $'u:        0.0510625
      1562 $t         0.0975625
      2132 $-h        0.1331875
      2409 #,         0.1505000
      2409 #&0         0.1505000
      2409 ##Q-        0.1505000
      2409 $U         0.1505000
      3125 $m+        0.1952500
      3877 #&0.        0.2422500
      3877 #&1)        0.2422500
      3877 ##:k-       0.2422500
      3877 $d-n       0.2422500
      4492 $r         0.2806875
      5673 $'aI       0.3545000
      6813 $t         0.4257500
      7689 $s         0.4805000
      8633 $e:-@      0.5395000
      8917 $n         0.5572500
      9702 #&1.        0.6063125
      9702 #&2^        0.6063125
      9702 ##:k-       0.6063125
      9702 $Q-        0.6063125
      9702 $-q        0.6063125
      9702 $'u:6      0.6063125
      12448 #.         0.7779375
      12448 #&2;        0.7779375
      12448 #&PGn      0.7779375
      12448 #:k       0.7779375

```

Abbildung 5.2.: Kiel Korpus Datei

```

"/g317a005.*.lab"
58750    200625    Klicken
58750    58750    ##
58750    200625    g
200625    516250    gh
516250    981250    u
981250    1337500   t
1337500           1510625   th
1510625           1510625   ##
1510625           1958125   U
1958125           2428125   m
2428125           2428125   ##
2428125           2812500   n
2812500           3550625   r
3550625           4263125   aI
4263125           4810625   t
4810625           5400625   s
5400625           5578125   @
5578125           6068750   n
6068750           6068750   ##
6068750           7785000   u6
.

```

Abbildung 5.3.: HTK Datei – Lautmodelle

	313	\$-h	0.0195000
	Anfang	Ende	Label
b)	58750	200625	g

Das Ende des Sprachsignalteils, der mit dem Label gekennzeichnet wurde, kann man berechnen, wenn man die Dauer des entsprechenden Sprachsignalteils kennt. Für prosodische Etikettierung und ähnliches ist die Dauer gleich null. Die Dauer ergibt sich aus dem absoluten Wert der Differenz dieses Sample-Wertes und dem nächst folgenden, der nicht gleich dem aktuellen Wert ist. In Abbildung 5.2 ist die Dauer von “##g” $227 = |86 - 313|$. Wir werden diese zwei Werte

Anfang_Samples_Kiel_Korpus und *Ende_Samples_Kiel_Korpus* nennen.

Beispiel (5.1.b) zeigt das HTK-Format. Die ersten zwei Spalten spezifizieren die Zeitangaben für den Anfang und das Ende des Labels in 100-Nanosekunden-Einheiten. Das Label selbst steht in der dritten Spalte.

Die Umwandlung des Kiel Korpus-Formats in das HTK-Format geschieht in zwei

Schritten. Zuerst werden die neuen Zeitangaben aus den alten abgeleitet, und dann werden die Labels angepasst. Für die HTK-Zeitangaben eines Labels werden die Anfang-Samples- und Ende-Samples-Werte benutzt, die dem Label im Kiel Korpus entsprechen. Da bei der Ableitung der spektralen Parameter ein Fenster von 16 ms benutzt wurde, in dem das Signal in der Mitte des Fensters am stärksten zu den Parameterwerten beigetragen hat (Hamming-Window), soll der Outputvektor von spektralen Parametern für die Mitte des Fensters ausgegeben werden. Es wird eine Korrektur von 8 Samples ($1/2$ von 16) vorgenommen, damit die Vektoren mit den Labeldateien zeitlich übereinstimmen. Die Formeln sind dann:

$$Label_Anfang_HTK = (Anfang_Samples_Kiel_Korpus + 8) * 625$$

$$Label_Ende_HTK = (Ende_Samples_Kiel_Korpus + 8) * 625$$

Die neu berechneten HTK-Zeitwerte für unser Beispiel sieht man in (5.1.b).

HTK verlangt nicht nur andere Zeitwerte, sondern auch andere Labels als das Kiel Korpus. Im Folgenden werden alle Änderungen inklusive Beispiele aufgelistet, die mittels Perl-Skripten durchgeführt wurden.

- Die Labels, die nur Geräusche repräsentieren (ohne Überlappung mit Sprachmaterial), werden durch die Geräusche selbst ersetzt.
 $:k \rightarrow Klicken \quad s \rightarrow Schmatzen$
- Geräusche, die sich mit Sprache oder mit vorherigen Geräuschen überlappen, werden gelöscht (*Klopfen* in Abbildung 5.2).
- Geräusche, die am Ende einer Aufnahme vorkommen, werden gelöscht (das letzte *Klicken* in Abbildung 5.2) .
- ## markiert Wortgrenzen, also werden Labels, die ## enthalten, in zwei Labels zerlegt. Die Dauer des Signalteils mit dem Label ## ist gleich null.

$$\begin{array}{ccccccc} 58750 & 200625 & ##g & \longrightarrow & 58750 & 58750 & ## \\ & & & & 58750 & 200625 & g \end{array}$$

- Wenn ein Label für einen Plosiv (p, t, k, b, d, g) von einem -h Label gefolgt wird (Lösungsphase des Plosivs), wird -h mit *Plosivh* ersetzt.

$$\begin{array}{ccccccc} 58750 & 200625 & g & \longrightarrow & 58750 & 200625 & g \\ 200625 & 516250 & -h & & 200625 & 516250 & gh \end{array}$$

- weitere Änderungen, die genau in dieser Reihenfolge durchgeführt werden müssen:

(# \$)&...	→ nichts
a	→ A
(a e i o u y 2):	→ a e i o u y 2
E:	→ EE
#c:	→ nichts
#(, . ; ? -MA _)	→ nichts
(, . ; ? -MA _)	→ nichts
\$(-MA = / + - / - / + = / + = / - ; _ - ~ =)	→ nichts
+\$	→ nichts
\$#	→ \$
\$before-after	→ \$after
#before-after	→ #after
(# \$)(// /)"label	→ label
(:k l: q: r:s: w:g: q v: z:n:)	→ nichts

Wortmodelle

Nach der Anpassung des Kiel Korpus, haben wir das für den Trainingsprozeß nötige Korpus bekommen. Dieses Korpus war allerdings dazu geeignet, HMMs für Lautmodelle zu trainieren. Da eine Überlegung war, HMMs für Wortmodelle zu trainieren, haben wir dafür das Kiel Korpus ein zweites mal angepasst. Die notwendige Perl-Skripte befinden sich im Anhang C. Die neue Anpassung erfolgt in zwei Schritten. Zuerst haben wir mittels des Variantenlexikons *kielcd4.lsv* ein Aussprachelexikon erstellt. *kielcd4.lsv* ist ein Variantenlexikon des bisher gesammelten spontansprachlichen Materials des Kiel Korpus.

- (5.2) a) "ähnlich Q'E:nlIC Q-:q'E:nlIC 2 1 G124A002 8
 b) "ähnlich EEnlIC

Jeder der Lexikoneinträge hat folgende Struktur: (siehe Beispiel (5.2.a)):

1. orthographische Repräsentation des lexikalischen Eintrags
2. kanonische Repräsentation des lexikalischen Eintrags
3. Transkription der Aussprachevariante
4. Auftretenshäufigkeit des lexikalischen Eintrags im Korpus
5. Auftretenshäufigkeit der Aussprachevariante
6. Platzierung des Eintrags im Dialog (Dialog- und Textreferenz)
7. Platzierung des Eintrags im Text

Ein Aussprachelexikon wurde erstellt, indem die orthographische Form (erste Spalte) und die Aussprachevariante (dritte Spalte) behalten wurden. Letztere wurde natürlich an das HTK Format angepasst. Wichtig ist, daß jede Variante nur einmal im ganzen Lexikon vorkommen darf. Wenn zwei Wörter dieselbe Variante haben, wird das Wort ins Aussprachelexikon aufgenommen, das alphabetisch zuerst aufgetreten ist. (5.2.b) zeigt den Eintrag von (5.2.a) nach der Anwendung unseres Skriptes. Die entsprechende Syntax ist:

```
erzeugt_aussprache_lexikon.pl kielcd4.lsv aussprache_lexikon.txt
```

Der zweite Schritt unseres Ansatzes war die Herstellung einer .mlf Datei für das Training von Wortmodellen. *create_master_file_lexicon.pl* nimmt als Input die .mlf Datei, die für Lautmodelle erzeugt wurde. Alle Labels, die zum selben Wort (Labels zwischen zwei Wortgrenzen) gehören, werden aneinander gereiht. So entsteht eine Aussprachevariante, die mit der orthographischen Form vom Aussprachelexikon ersetzt wird. Wortanfang ist der Anfang vom ersten und Wortende das Ende vom letzten Label. Geräusche werden als einzelne Wörter betrachtet. Ein Beispiel für das Output des Skriptes sieht man in Abbildung 5.4. Das Input dafür war die Datei in Abbildung 5.3. Die Syntax für diese Herstellung ist:

```
"*/g317a005.*.lab"
58750 200625 Klicken
58750 1510625 gut
1510625 2428125 um
2428125 6068750 dreizehn
6068750 7785000 Uhr
.
```

Abbildung 5.4.: HTK Datei – Wortmodelle

```
create_master_file_lexicon.pl phondat.mlf lex.mlf
```

Entgegen unserer ursprünglichen Überlegungen, haben wir nur Lautmodelle trainiert.

5.1.2. Trainingsprozess

Das Ziel dieser Phase ist die eigentliche Erstellung der Lautmodelle. Das Training besteht aus zwei Schritten — Initialisierung und Re-estimation, die mittels zweier Skripte (HInit, HRest) durchgeführt werden (siehe Anhang D). Bei der Initialisierung werden aus jeder Signaldatei die Sequenzen an Sprachmaterial genommen, die einem bestimmten Label entsprechen. Das Label ist immer das gleiche (phonemische Label), während die entsprechenden Signaleile abhängig vom Kontext, in dem ein Laut gesprochen wurde, variieren. Beispielsweise wird [u] in Wörtern wie “Hut” und “Tuch” unterschiedlich ausgesprochen,

aber mit dem gleichen Label erfasst (siehe Anhang A). Alle Signalsequenzen, die einem Label entsprechen, werden für die Erstellung eines HMMs benutzt. Um eine Lautsequenz gut zu beschreiben, sollten verschiedene HMMs verschiedene Anzahlen von Zuständen haben. Wir haben uns für fünf Zustände bei Diphthongen und drei Zustände bei allen anderen Lauten entschieden. Für jeden Zustand müssen beide Übergangswahrscheinlichkeiten (Zustand verlassen, im Zustand bleiben) zusammen 1 ergeben. Wir haben uns für eine Einteilung von je 0.5 entschieden (*uniform segmentation*). In einem HMM mit n Zuständen und einer Dauer von d in ms, beschreibt jeder Zustand genau d/n ms. D.h., bei einem [u] mit einer Dauer von 60 ms, entspricht jeder der drei Zustände (Transition vom vorherigen Laut zu [u], das [u] selbst, Transition von [u] zum nachfolgenden Laut) genau 20 ms (siehe Abbildung 5.5).

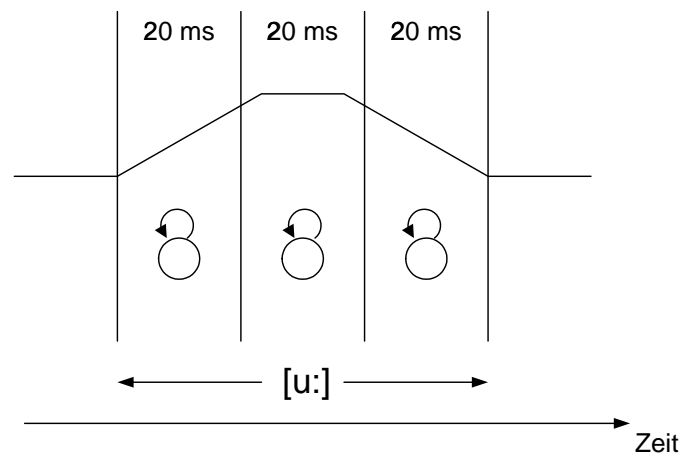


Abbildung 5.5.: Training: Initialisierung

Diese äquidistante Einteilung einer Lautsequenz wird der Variabilität eines Sprachlautes nicht gerecht. Die Transition von einem [d] zu einem [i:] verläuft beispielweise schneller als die von einem [d] zu einem [a:]. Im ersten Fall ist die Zunge bei beiden Lauten an einer ähnlichen Position, während im zweiten Fall die beiden Laute an sehr unterschiedlichen Positionen artikuliert werden. Auch ist die Realisierung eines Lautes von Sprecher, Artikulationsgenauigkeit oder Kontext abhängig. Um sich also der artikulatorischen Realität anzunähern, und, um gleichzeitig die Erkennungsrate zu verbessern, wird als zweiter Schritt die Re-estimation durchgeführt. Hierbei wird die Wahrscheinlichkeit eines Zustandes ermittelt und die Zustände dann gewichtet (siehe Abbildung 5.6).

Re-estimation versucht mittels eines *vorwärts-rückwärts Algorithmus* (forward/backward oder Baum-Welsh algorithm) herauszufinden, wie hoch die Wahrscheinlichkeit für einen bestimmten Zustand ist. Hierbei wird ein Zeitfenster durch das zu evaluierende HMM "geschoben" und für jeden zeitlichen Abschnitt anhand der ermittelten Parameterwerte bestimmt, in welchem der drei bzw. fünf Zustände sich das HMM am ehesten befindet. Eine solche Schätzung kann mehrfach wiederholt werden bis die Zustände

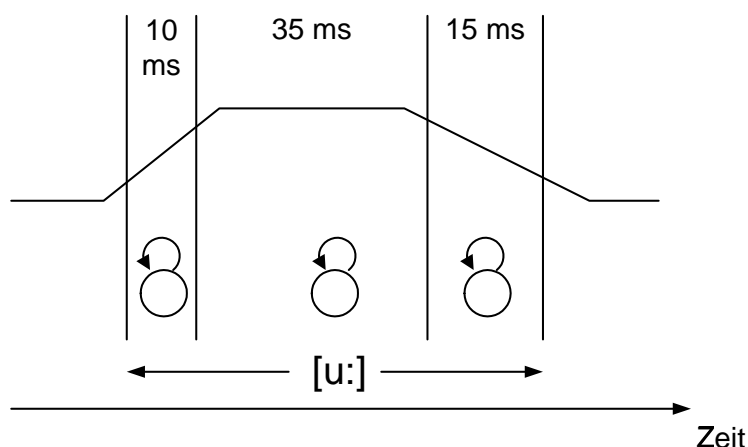


Abbildung 5.6.: Training: Re-estimation

als sehr wahrscheinlich und das HMM als trainiert gelten. Dafür wird zunächst die maximale Anzahl der Wiederholungen festgelegt (in unserem Fall ist sie 20). Wenn sie erreicht wird, hört die Re-estimation auf. Der Prozeß kann auch früher beendet werden, wenn der Unterschied zwischen dem Input und Output einer Wiederholung unter einen bestimmten Schwellenwert fällt (Konvergenzbedingung).

Zu erwähnen ist noch, daß HInit sowohl die spontansprachlichen als auch die gelesenen Sprachdaten des Kiel Korpus verwendet, da soviel Material wie möglich benutzt werden sollte. HRest dagegen verwendet nur den spontansprachlichen Teil, da hier viele Reduktionsprozesse auftreten, genau wie später in den Benutzeräußerungen im Fahrstuhl.

Der Trainingsprozeß liefert also die trainierten HMMs, die zusammen mit einem Lexikon und einer Grammatik die Grundlage des Erkennungsprozesses bilden.

5.1.3. Lexikon, Grammatik

Ausser der Lautmodelle benötigt der Testprozess ein Lexikon und eine Grammatik. Das Lexikon wurde manuell erstellt. !PHONE-SET bezeichnet die Menge aller trainierten Lautmodelle, inklusive verschiedene Geräusche. Für den Rest der Einträge gilt, daß die erste Spalte das zu erkennende Wort und die dritte Spalte seine entsprechende Lautsequenz enthält, während die zweite Spalte das Output spezifiziert, welches im Fall der Erkennung an das Dialogmodul weitergegeben wird. Jedes Wort wird mit mehreren Aussprachevarianten aufgelistet (siehe Abbildung 5.7). Wörter, die im Dialog eine Funktion erfüllen (Schlüsselwörter), bilden den grössten Teil der Lexikoneinträge. Wörter wie “bitte” werden aufgrund ihrer Häufigkeit als Zielwörter erkannt, werden jedoch zur Zeit noch nicht im Dialog berücksichtigt (die zweite Spalte bleibt leer). !ANY und !SIL sind das Garbage-Modell des Spracherkenners, d.h. wenn keine Wörter erkannt werden können, versucht der Spracherkennner irgendetwas zu erkennen. Im Anhang E findet sich das vollständige Lexikon. Die möglichen Syntaxvarianten der Benutzeräußerungen werden in der Grammatik im Anhang

```

!PHONE-SET          hmm_p hmm_t hmm_k hmm_b hmm_d hmm_g hmm_Q hmm_f ...
...  hmm_U hmm_U6 hmm_aI hmm_aU hmm_OY hmm_@ hmm_6 hmm_Pause  hmm_Atmung
hmm_Schmatzen  hmm_Raeuspern hmm_Schlucken hmm_Lachen hmm_Klicken
hmm_Klopfen   hmm_Rascheln  hmm_Geraeusch

!SIL              []      hmm_Pause
!ANY              []      hmm_p
...
fahrstuhl        [F]      hmm_f hmm_a6 hmm_S hmm_t hmm_u hmm_l
fahrstuhl        [F]      hmm_f hmm_a6 hmm_C hmm_t hmm_u hmm_l
...
#"Erst(er)"
1                [1]      hmm_Q hmm_E6 hmm_s hmm_t hmm_6
...
#"Bitte"
bitte            []      hmm_b hmm_I hmm_t hmm_@

```

Abbildung 5.7.: Lexikonausschnitt

F definiert.

5.1.4. Testprozess

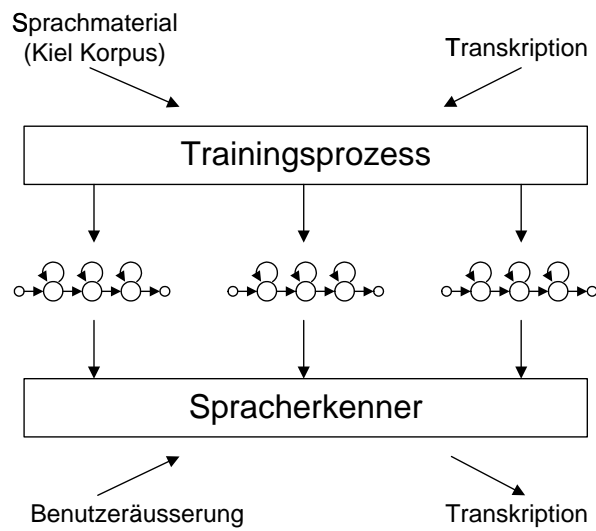


Abbildung 5.8.: Bild 4

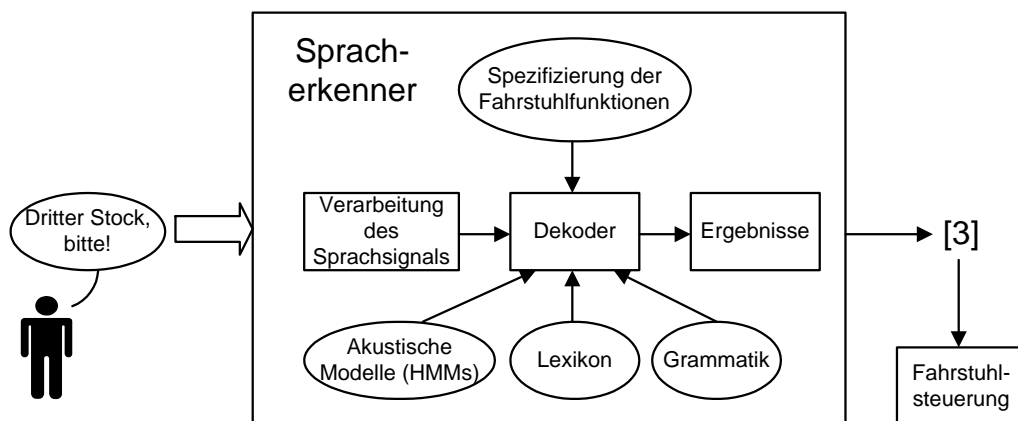


Abbildung 5.9.: Bild 6

6. Steuerung

im folgenden abschnitt soll ein grober ueberblick ueber die aufzugsteuerung sowie dem dazugehoerigen treiber, welcher die kommunikation zwischen dialogsystem und steuerung realisiert, dargestellt werden. als treiber ist ein programm zu verstehen, welches in direkter kommunikation zu einer entsprechenden hardware-einheit steht und damit anderen programmen eine abstrakte kommunikationsschnittstelle zur verfuegung stellt.

die hier beschriebene loesung versteht sich im augenblicklichen entwicklungsstand als spezialloesung, die, im gegensatz zu dem dialogsystem, nicht modular aufgebaut ist und sich nicht zwangslaefig direkt auf aufzugsteuerungen anderer hersteller uebertragen laesst.

im rahmen dieses projektes beschraenkt sich die aufgabe des treibers auf die moeglichkeit den jeweiligen status des aufzugs an das dialogsystem weiterzuleiten, sowie tastendrucke auf dem bedienfeld des fahrkorbes zu simulieren. weitere funktionen koennen bei bedarf implementiert werden.

6.1. aufzugsteuerung und deren anschluss

bei der verwendeten aufzugsteuerung handelt es sich um das von der firma newlifthergestellte modell "KST". dass dieses modell bereits eine schnittstelle zum anschluss an einen pc, bzw. ein modem vorgesehen hat, war zum einen sehr hilfreich, da somit keine eigene schnittstelle mehr entwickelt werden musste. zum anderen mussten aber damit auch einige einschraenkungen in kauf genommen werden. genannt seinen hier limitierungen der maximalen kabellaenge zwischen rechner und steuerung, sowie probleme, die das design des protokolls mitbrachte (siehe abschnitt "protokoll").

die verbindung zwischen steuerung und rechner wird ueber eine serielle schnittstelle (RS232) hergestellt. da diese schnittstelle in verschiedenen modi betrieben werden kann und schon von seiten der KST ein modus verwendet wird, der nur drei leitungen der schnittstelle benutzt, wurde der gleiche modus auch auf rechnerseite verwendet. die signale dieser leitungen sind:

- *TX*: transceive signal
- *RX*: receive signal
- *GROUND*: signal-masse

damit ergibt sich intuitiv fuer eine verbindung zwischen steuerung und rechner folgende pin-belegung der D-Sub-stekcer wie in abbildung 5.1 dargestellt.

(abbildung 5.1)

um nun das oben angesprochene problem der auf 15m limitierten kabellaenge zu bewaeltigen und die benoetigten 70m zu ueberwinden, wurde eine vorhandene netzwerkinfrastruktur genutzt. dabei handelt es sich um hochwertige, abgeschirmte twisted-pair-kabel, wobei allerdings der vorteil der verdrehten draechte der leitungen nicht genutzt wird, da die signale der seriellen schnittstelle absolut und nicht differenziell uebertragen werden. jedoch reicht messungen zufolge allein die abschirmung aus, um eine stoerungsfreie datenuebertragung zu gewaehrleisten.

abbildung 5.2 zeigt nun die erweiterte darstellung der pin-belegung der D-Sub-stecker sowie der fuer die twisted-pair-leitungen verwendeten RJ-45-stecker.

(abbildung 5.2)

6.2. protokoll

dieser abschnitt soll einen knappen ueberblick ueber die beiden verwendeten protokollen, die beide unter dem name *NLF401* (NewLiftFormat 401) zusammengefasst sind, vermitteln. da die firma newlift selbst dokumentation zu diesen prokollen zur verfuegung stellt, wird hier nur auf wichtige details eingegangen.

bei den beiden protokollen handelt es sich ein transportprotokoll und um ein sogenanntes application-protokoll. als problem stellte sich heraus, dass diese beiden protokolle an einigen stellen miteinander verflochten sind, was eine modulare implementierung leider unmoeglich gemacht hat.

bei dem transportprotokoll handelt es sich um ein einfaches bouncing-protokoll, dass jeweils ein packet mit der groesse von einem byte versenden kann. bei einem bouncing-protokoll erwartet der sender nach jedem gesendeten packet eine antwort des empfaengers, bevor das naechste packet gesendet wird. diese eigenart wird u.a. dazu benutzt, um den ausfall eines endpunktes (ueber timeout) oder aber eine desynchronisation zu entdecken.

eine fehler-erkennung und auch "korrektur" wird durch das zweimalige senden und vergleichen jedes datenpaketes erreicht, wobei hierfuer eine einbusse der uebertragungsgeschwindigkeit um faktor zwei in kauf genommen wird. dies kann bei haeufigen und zeitkritischen anfragen an die stuerung zu problemen fuehren.

um die funktionsweise des protokolls zu verdeutlichen, wird das versenden des bytes *A* anhand der abbildung 5.4 demonstriert. abbildung 5.3 zeigt den aufbau eines datenpaketes. zu erkennen ist, dass die obere haelfte (bits 4 bis 7) dazu dienen, um statusinformationen

auf protokoll-ebene zu uebertragen. die unteren vier bits dienen als container fuer die zu uebertragenden nutzdaten.

(abbildung 5.3)

(abbildung 5.4)

6.3. implementierung

6.4. steuerkommandos

6.5. interface

das interface des treibers ist durch eine interaktive kommandozeile realisiert. in der bisherigen implementation des treibers werden die beiden folgenden kommandos unterstuetzt:

- *info()*
gibt die aktuelle position, fahrtrichtung und den status der tueren zurueck.
- *ziel(n)*
setzt ein fahrkommando in den n-ten stock.

Beispiel:

```
[root /] # nlf401
info()
info(kabinentuer:auf drehtuer:zu pos:1 ziel:255 richtung:keine)
ziel(5)
ok()
info()
info(kabinentuer:zu drehtuer:zu pos:5 ziel:5 richtung:keine)
```

6.6. Protokoll

6.7. Zustände, Informationen

6.8. Kommunikation

7. Soundkarten (Eric)

7.1. Aufgaben des Teilprojektes

Da ursprünglich für den sprechenden Fahrstuhl eine Sprechstelle pro Stockwerk vorgesehen war und nicht nur eine in der Kabine, befasse ich mich mit der Auswahl von Mehrkanal-„Soundkarten“ und ihrer Einbindung in das Gesamtsystem der Fahrstuhlsteuerung. Den Begriff „Soundkarte“ verwende ich dabei als Oberbegriff für eine Vorrichtung, um Sprachsignale oder sonstige Töne mit einem PC zu verarbeiten. Es muss sich dabei nicht um ein einzelnes Gerät handeln, es ist beispielsweise auch denkbar, jede Sprechstelle mit einem eigenen Wandler auszustatten, der die Sprachsignale vor Ort verarbeitet und an ein Computernetzwerk (zum Beispiel Ethernet) angeschlossen ist.

In den folgenden Abschnitten meine ich mit Soundkarte nicht mehr den Oberbegriff, sondern eine Baugruppe, die in einen PC eingebaut wird, um ihn in die Lage zu versetzen, Töne zu verarbeiten. Meistens hat eine Soundkarte direkt Anschlüsse für Lautsprecher / Mikrofon etc., manchmal befinden sich diese aber auch in einer speziellen Box, die an die Soundkarte angeschlossen wird. So werden die Anschlüsse besser erreichbar und man kann den Einfluss von Störsignalen reduzieren, die im Innern des PC vorhanden sind.

Um ein „Tonverarbeitungsgerät“ in Verbindung mit einem PC einsetzen zu können, braucht man immer auch einen passenden Treiber, der eine geräteunabhängige Schnittstelle zu den auf dem PC laufenden Programmen zur Verfügung stellt. Das geschieht zum Beispiel über eine simulierte Datei, aus der die empfangenen Töne wie aus einer Audiodatei gelesen oder zu sendende Töne geschrieben werden können, oder über einen festen Satz von Funktionen wie „spiele diesen Datensatz als Ton ab“, die dann von den anderen Programmen aufgerufen werden.

Für unsere Anwendung benötigen wir insgesamt 7 Sprechstellen (1 pro Stockwerk und 1 in der Kabine) für die volle Ausbaustufe. Da sich bald zeigte, dass eine Sprechstelle in jedem Stockwerk das ganze Projekt sehr komplex machen würde, wurden nur wenige praktische Experimente in dieser Richtung durchgeführt. Ich beschreibe hier also Möglichkeiten, die erst in einer weiteren Ausbaustufe auch wirklich zum Einsatz kommen werden.

Die Sprachsignale müssen letztlich von Software zur Spracherkennung analysiert werden, also ist es wichtig, auf die Schnittstelle zur verwendeten Software zu achten. Zuerst wurde das Paket HTK (Hidden Markov Toolkit) von Entropic unter Linux (x86) eingesetzt, später dann der Spracherkenner ASR 1600 M und das Toolkit dazu von Lernout & Hauspie unter Windows NT. HTK gibt es auch für Windows, der L & H Erkennen läuft

auch mit Windows 95/98. Je nach verwendetem Spracherkenner und Betriebssystem muss jeweils ein passender Treiber vorhanden sein.

7.1.1. Grundsätzliche Probleme

Um die Komplexität des Projektes überschaubar zu halten, ist der Fahrstuhl bisher (Dezember 2000) noch auf 1 Sprechstelle in der Kabine beschränkt. Dadurch kommt im Programm weniger Nebenläufigkeit vor, ausserdem ist es technisch schwierig, alle Stockwerke so zu verkabeln, dass eine gute Tonqualität erreicht wird – die Akustik im Flur ist ohnehin für Spracherkennung nicht besonders gut.

Ein weiteres Problem ist die Aufmerksamkeitssteuerung: Wenn auch ausserhalb des Fahrstuhls die Steuerung nur durch Sprache möglich sein soll (ohne Rufknopf), muss ein Spracherkenner ständig bereit sein, trotz wechselnder Umgebungsgeräusche sofort beim Erkennen eines Schlüsselwortes („Fahrstuhl“, „Aufzug“, ...) anzusprechen und im Optimalfall gleich den Rest der Äusserung zu analysieren („Fahrstuhl, ich möchte in den 3. Stock“). Im Zweifelsfall müsste man für jede Sprechstelle ständig einen Spracherkenner laufen lassen. Dabei besteht die Gefahr, so viel Speicher oder Rechenleistung zu verbrauchen, dass die Verarbeitung auf mehrere Computer verteilt werden muss.

Das Problem lässt sich entschärfen, indem man wie bei einer Sprechanlage das jeweilige Mikrofon nur einschaltet, solange der Rufknopf gedrückt ist. Damit laufen die Spracherkenner nur nach Bedarf und müssen nicht dauernd Nebengeräusche analysieren, während ohnehin kein Dialog stattfindet.

Eine elegantere Lösung arbeitet mit einem Schlüsselwort, zum Beispiel „Fahrstuhl“. Nur dieses Wort startet einen Dialog. Schlüsselwort und Rufknopf lassen sich kombinieren: Wird das Wort nicht genannt, kommt einfach der Aufzug (Rufknopf im alten Sinn), sonst wird ein Dialog begonnen (Rufknopf wie bei einer Sprechanlage). Ein Spracherkenner muss bei Verwendung eines Schlüsselwortes nicht versuchen, alle in der Nähe des Mikrofons gesprochenen Worte zu erkennen und entscheiden, ob die Worte an den Fahrstuhl gerichtet waren. Das Schlüsselwort kann auch leichter erkannt werden, da nur eine Ja/Nein-Entscheidung gebraucht wird und nicht mehrere vielleicht ähnliche Worte aus einem grossen Wortschatz in Betracht gezogen werden müssen. Man kann weiterhin vom Benutzer fordern, nachdem er das Schlüsselwort gesagt hat, erst auf eine Bestätigung zu warten:

Fahrstuhl

Wohin möchten Sie bitte?

Ich möchte in den 3. Stock

Wenn mehr Ressourcen (Speicher, Rechenleistung) verbraucht werden als ein einzelner Rechner anbietet, aus anderem Grund nicht genug Spracherkenner auf einem einzelnen Rechner laufen, oder einfach die Soundkarten oder sonstigen Geräte auf mehr als einen Rechner verteilt sind, müssen Sprachdaten in einem Rechnernetzwerk übertragen werden. Dabei kann es sich um unverarbeitete Mikrofonsignale oder schon um Ergebnisse der Spracherkennung handeln. Entsprechende Treiber müssen die Spracherkenner dann mit dem Netzwerk verbinden, da die Spracherkenner selten bereits mit einer Option „Mikrofon via Netzwerk abfragen“ oder „Ergebnisse ans Netzwerk schicken“ ausgestattet sind.

Ich befasse mich in Abschnitt 7.4.3 mit dem Problem, HTK unter Linux via Netzwerk Mikrofondaten empfangen zu lassen. Da sowohl bei HTK als auch beim ASR 1600 eine Schnittstelle für eigene Treiber vorgesehen ist, ist es zumindest prinzipiell auch unter Windows denkbar, Mikrofondaten im Netzwerk zu verschicken. Unter Linux ist allerdings der Sound-Treiber sehr gut dokumentiert. Auch sein Programmtext wird mitgeliefert, es ist also sogar denkbar, den Treiber direkt in zwei Teile aufzuspalten, die per Netzwerk miteinander kommunizieren – die Spracherkenner müssen dann gar nicht angepasst werden, obwohl die Soundkarte sich nicht im selben Rechner befindet. Diesen Ansatz habe ich nicht weiter verfolgt, da er umfangreiche Veränderungen am Soundtreiber erfordert.

7.2. Lösungsansätze

7.2.1. Verarbeitung analog und zentral

Dieser Lösungsansatz geht davon aus, dass die Mikrofone mit normaler Tontechnik bedient werden und die Umwandlung in digitale Computerdaten erst direkt in dem Rechner erfolgt, in dem die Spracherkennungssoftware läuft (oder direkt an diesen angeschlossener Hardware).

Für 6 Stockwerke plus die Kabine sind 7 Eingänge und 7 Ausgänge nötig, damit in keinem Fall die Ressourcen ausgehen können. Eine typische Wahl wäre damit eine Soundkarte mit 8 Eingängen und 8 Ausgängen, wie sie in semiprofessionellen Tonstudios und für Harddisk-Recording (ein PC ersetzt ein mehrspuriges Tonband im Studio) verwendet wird.

Soweit genügend Steckplätze zur Verfügung stehen, lassen sich zumindest bei modernen PCI-Soundkarten auch einfach mehrere einfache Karten gleichzeitig in einen PC einbauen und verwenden.

Im Abschnitt 7.3 „Verfügbare Soundkarten“ werden verschiedene Produkte verglichen und besprochen, zunächst führe ich aber noch einige grundsätzlich andere Ansätze auf.

7.2.2. Die Sprachdaten kommen digital an einem Rechner an

Für weitere Skalierbarkeit auf mehr Sprechstellen ist es denkbar, mehrere Rechner mit Soundkarten zu versehen und die aufgenommene Sprache per Netzwerk weiterzureichen. Umgekehrt kann die Verwendung eines Computernetzes auch ermöglichen, den Ressourcenverbrauch für die Spracherkennung auf mehrere Rechner zu verteilen.

Eine eher dezentrale Lösung sieht vor, direkt bei den Mikrofonen das Tonsignal zu digitalisieren und über Datenkabel zum Auswerterechner zu transportieren. Das verkompliziert die einzelnen Sprechstellen, vereinfacht aber je nach Konstruktion der Sprechstellen die Hardware am Auswerterechner und ist weniger störanfällig als ein analoges Signal. Gerade in der Nähe des Fahrstuhles und von Computern treten verstärkt elektromagnetische Störungen auf.

Die Schwierigkeit bei den „digitalen Sprechstellen“ liegt in der Umwandlung zwischen analoger Sprache und digitalen Daten im Netz.

Wegen der geringeren Anforderungen an die Tonqualität sollte man überlegen, zumindest die Sprachausgabe weiterhin analog im Rechner (oder den Rechnern) zu erzeugen, was die Sprechstellen vereinfacht und das Netz entlastet.

Eine interessante Möglichkeit für die Digitalisierung der Mikrofonsignale ist die Verwendung von sogenannten **GSM-Encodern** (in Form kleiner Spezialcomputer, solche kann man fertig kaufen), die die aufgenommene Sprache mit einer auf Sprache optimierten Methode komprimieren und dann über seriellen Schnittstellen übertragen. Dabei fallen weniger als 14000 bit/s an, was eine robuste Signalübertragung ohne besondere Hardware erlaubt.

Die GSM-Kompression wird auch für Handies verwendet, die Sprachqualität ist nach der Kompression also nur noch so gut wie wenn der Fahrstuhl mit den Sprechstellen per Handy sprechen würde.

Wenn die Spracherkennung auf einem einzelnen PC laufen soll, benötigt dieser dann ausser mehrerer serieller Schnittstellen keine besondere Hardware. Entsprechende Schnittstellenkarten sind verfügbar und ausgereift, früher wurden sie gerne bei Netzwerkanbietern verwendet, um mehrere Modems an einen Rechner anzuschliessen. Treiber sind für Linux, Windows, ... vorhanden.

Die grössere Schwierigkeit liegt bei der Spracherkennungssoftware, die oft GSM (noch dazu von einer seriellen Schnittstelle) nicht als Eingabeformat vorsieht. Gerade HTK ist hier aber erweiterbar: Man kann mit (ggf. selber zu schreibenden) Zusatzmodulen Audiodaten aus einem Datenstrom einlesen, den man wiederum durch Entkomprimierung der GSM-Daten (Rückumwandlung GSM in normale Audiodaten) zum Beispiel durch das Unix-Tool **sox** gewinnt. Sox ist ein Programm, welches verschiedene Audio-Datenformate ineinander konvertiert, zum Beispiel GSM, WAV und Rohdaten zum Austausch mit Treibern über /dev/dsp oder /dev/audio.

Für höhere Sprachqualität sollte weniger stark oder gar nicht komprimiert werden. Dazu ist allerdings spezielle Hardware nötig, zum Beispiel der Einsatz **digitaler Signalprozessoren (DSP)** die das Signal digitalisieren, komprimieren und dann in einem geeigneten Protokoll digital weiterleiten.

Unkomprimierte digitale Daten bekommt man bereits als paralleles oder serielles Signal aus relativ preiswerten **ADC (Analog Digital Konverter)** Bausteinen, das hohe Datenaufkommen und das Fehlen eines „intelligenten“ Protokolls fordern aber gut abgeschirmte Kabel und eine spezielle Verarbeitung im Zielrechner.

Bei Verwendung eines echtzeitfähigen Betriebssystems kann dazu eine einfache **PIO (Parallel Input Output) Steckkarte** genügen, wobei der PC die seriellen Signale aller Sprechstellen unter relativ hohem Rechenaufwand wieder zusammensetzt. Eine sinnvollere Methode ist die Verwendung mehrerer serieller Schnittstellen im PC, die Geschwindigkeit und Datenformat bewältigen (115200 kbit/s reichen nur für etwa 8 kHz **Samplingrate**, also Amplitudenmessungen pro Sekunde, bei einer Genauigkeit von 12 Bit, was gerade noch etwa Telefonqualität ist).

Noch höhere Datenraten lassen sich mit speziellen seriellen Schnittstellenbausteinen oder einer **DSP Steckkarte** am empfangenden PC erreichen. Den DSP zu programmieren ist aber wegen der Forderung nach paralleler Verarbeitung in Echtzeit eine schwierige Aufgabe, die Spezialwissen erfordert.

Für die digitale Übertragung in hoher Qualität ist es also ab einem bestimmten Punkt sinnvoller, die Sprechstellen selber mit Computern oder DSPs auszustatten. Diese könnten dann z.B. über normales **Ethernet** kommunizieren, gegebenenfalls sogar über ein einzelnes **BNC-Kabel** (nur bis 10 Mbit/s). Damit muss man nicht mehr getrennte Kabel für jede Sprechstelle verlegen, sofern sowohl die Aufnahme als auch die Wiedergabe digitalisiert werden. Gerade bei BNC muss allerdings die Netzwerkkapazität beachtet werden:

Das Datenaufkommen im Netzwerk ist bei heute verbreitetem **Fast Ethernet** (100 Mbit/s) kein Problem: Bei 44100 Amplitudenmessungen mit 16 Bit Genauigkeit fallen 0.7 Mbit/s pro Kanal an. Für 7 Kanäle ergibt das 5 Mbit/s, wenn man also auch noch die Dinge die der Lift sagt per Netz übertragen will, droht bei 10 Mbit/s Ethernet ein „Datenstau“.

Diese Betrachtungen sollten aber als „worst case“ angesehen werden, da sie von Mono CD-Qualität ausgehen. Zumindest für die Äusserungen des Fahrstuhls sollte sogar bei 8 Bit Genauigkeit und einer Samplingrate (Amplitudenangaben pro Sekunde) von 22050 Hz noch eine angenehme Tonqualität erreicht werden. Für die Spracherkennung ist jedoch 16 Bit Genauigkeit wünschenswert, da das **Signal/Rausch-Verhältnis (SNR)** verbessert wird und noch Reserven vorhanden sind um schlecht ausgesteuerte Signale zu verarbeiten. Die Lautstärke kann dann noch per Software korrigiert werden. Ein Spracherkennner kann bei entsprechendem Design sogar mit 8000 Hz Samplingrate und 8 Bit Genauigkeit arbeiten, da dies aber nur noch mässige Telefonqualität ist, sind 22050 Hz und 16 Bit (Qualität etwa wie Musik-Kassette oder UKW-Radio) für zuverlässige Erkennung sinnvoller.

7.2.3. Verwendung mehrerer Rechner: Echte Skalierbarkeit

Wie im letzten Abschnitt beschrieben, ist es möglich und für grössere Anlagen sinnvoll, die Sprachverarbeitung auf mehrere Rechner zu verteilen.

Dabei kann man entweder die Sprachsignale direkt an mehreren Stellen empfangen – jeder Rechner hat eine eigene Soundkarte oder sonstige Empfangsmöglichkeit für genau so viele Kanäle wie er analysiert – oder die Sprachsignale digital von zentraler Stelle aus verteilen, wobei dann aber ggf. die Kapazität der Netzwerkverbindung entsprechend vorhanden sein muss.

Bei der digitalen Verteilung kann man die Sprachdaten vor dem Verteilen komprimieren. Dazu bieten sich z.B. der **GSM-Algorithmus** aus dem Handy-Bereich oder **MP3 (mpeg audio layer 3)** an, welches versucht, für Menschen unhörbare Klanganteile wegzulassen und das deshalb gerne für die Kompression von Musik ohne hörbaren Qualitätsverlust verwendet wird.

Dabei ist aber mit deutlichem Verbrauch an Rechenleistung zu rechnen, da GSM eigentlich besonders für spezielle Signalprozessoren mit extra auf GSM abgestimmten Recheneinheiten geeignet ist und bei MP3 die hörbare Qualität und nicht der Verbrauch an Rechenleistung vorrangiges Entwicklungsziel war.

Sobald der Ressourcenverbrauch in der Grössenordnung dessen liegt, was die Spracherkennung an Rechnerkapazität benötigt, wird die Kompression jedoch nutzlos. Wie oben gesagt, ist das Datenaufkommen ohnehin zumindest bei Verwendung von Fast Ethernet selbst für dutzende von Tonkanälen auch ohne Kompression noch akzeptabel.

Für besonders grosse Anlagen ist es trotzdem sinnvoller, jedem Rechner eine eigene Soundkarte einzubauen bzw. bei digitalen Sprechstellen jeweils wenige Sprechstellen direkt mit einem Rechner zu verbinden statt über ein gemeinsames Netz. Über das gemeinsame Netz laufen dann nur noch Daten der Art von „An Sprechstelle 42 wurde das Wort FÜNF mit einer Wahrscheinlichkeit von 80 Prozent gehört“ – das bedeutet minimale Anforderungen an das gemeinsame Netz selbst bei riesigen Systemen.

7.2.4. Mehrfachnutzung von Ressourcen: scheinbare Skalierbarkeit

Ein alternativer Ansatz besteht darin, weniger Kanäle als Sprechstellen komplett auszubauen. Zum Beispiel würde man überall Mikrofone installieren, die Soundkarte am Rechner hätte aber zu wenig Anschlüsse. Da es sehr unwahrscheinlich ist, dass an vielen Sprechstellen absolut gleichzeitig gesprochen wird, lässt sich eine einfachere Soundkarte ohne im Normalfall feststellbare Beeinträchtigungen verwenden.

Bei diesem Ansatz benötigt man als Zusatzhardware einen elektronischen Umschalter, der die vorhandenen Soundkartenanschlüsse computergesteuert den jeweils aktiven Sprechstellen zuteilt. So ein Gerät lässt sich einfach und preiswert konstruieren und beispielsweise über die Druckerschnittstelle ansteuern.

Ich schätze für unsere Anwendung genügen 2-4 Eingänge und 2-4 Ausgänge: Eine typische PC-Soundkarte, zum Beispiel ein **SoundBlaster** hat 2 Eingänge und 2 bis 4 Ausgänge die völlig unabhängig verwendet werden können. Weitere Anschlüsse sind oft über einen Mixer verbunden und können nicht eigenständig eingesetzt werden.

Bei Verwendung einer einzelnen Standardsoundkarte müsste der Fahrstuhl also erst dann Benutzer warten lassen, wenn er bereits an 4 Stellen absolut gleichzeitig spricht. Angesichts der Kürze typischer Fahrstuhläusserungen wie „Ich fahre Sie jetzt ins Untergeschoss“ ist dieser Fall extrem unwahrscheinlich.

Das grössere Problem beim Einsatz einer sehr einfachen Soundkarte ist die Spracherkennung: Das Mikrofon in der Kabine sollte zumindest bei stillstehendem Aufzug ständig aktiv sein, wodurch für alle Sprechstellen auf den Stockwerken zeitweise nur noch ein Kanal übrig bleibt.

7.2.5. Den Aufwand verteilen: Hardware direkt in den Sprechstellen

Ein einfacher, aber uneleganter Ausweg ist die Verwendung von **Rufknöpfen** – Der Fahrstuhl öffnet nur für die Mikrofone einen Kanal, an denen ein Benutzer den Sprechknopf gedrückt hält. Es ist dann sogar möglich, falls kein Kanal mehr frei ist, den Benutzer um Geduld zu bitten (per Sprachausgabe oder indem eine „Sprechen“ Lampe erst aufleuchtet, wenn nach wenigen Sekunden wieder ein Kanal freigeworden ist). Alternativ kann der Fahrstuhl wie bei einem klassischen Aufzug einfach zu dem Stockwerk kommen, in dem der Knopf gedrückt wurde, sofern keine Kapazitäten für einen gesprochenen Dialog mehr frei sind. Der Benutzer würde dann erst *im* Fahrstuhl das gewünschte Ziel nennen.

Der Einsatz von „intelligenten Sprechstellen“ kann die Spracherkennung hier wesentlich vereinfachen: Die Firma Sensory (<http://www.voiceactivation.com>) bietet für ca. 100 DM mit dem **VoiceDirect Modul** ein vorprogrammiertes DSP-Modul an, welches an ein analoges Sprachsignal oder Mikrofon angeschlossen direkt bis zu 15 verschiedene Phrasen von jeweils bis zu 3 Sekunden Länge erkennt. Das Modul verwendet dazu ein **neuronales Netz** , also ein ähnlich wie die **Hidden Markov Chains** von HTK fehlertolerantes und einfach an Beispielen lehrbares System.

Die Anwendung so eines Moduls kann z.B. darin bestehen, dem Modul vor Ort Schlüsselwörter wie „Fahrstuhl“ vorzusprechen (Vorteil: die Raumakustik wird dadurch gleich mit berücksichtigt), die es dann später erkennt und ein einfaches binäres Signal schickt. Der Wortschatz von 15 Phrasen reicht bei kleineren Gebäuden sogar zur Erkennung aller relevanten Wörter.

Bei grösseren Systemen eignet sich eine Spracherkennungshardware besonders zur Erkennung von **„Aufwachbegriffen“** : Die Hardware erkennt, dass ein Benutzer den Fahrstuhl sprechen möchte, das entsprechende digitale Signal geht an den Rechner mit der komplexen Spracherkennungssoftware, der daraufhin einen Tonkanal zu dem jeweiligen Mikrofon öffnet und ggf. einen Prozess der Spracherkennungssoftware startet. Dadurch laufen nicht unnötig ständig viele Prozesse komplexer Software nur um auf ein dialogeinleitendes Stichwort zu warten.

Wenn keine Spracherkennungshardware in jeder Sprechstelle zur Verfügung steht, kann man sich wie folgt behelfen: die zentrale Spracherkennung wird reihum mit den Sprechstellen verbunden an denen überhaupt ein Geräusch erkannt wurde. Für die Geräuscherkennung reicht eine einfache Hardware, oder man lässt das zentrale System jeweils für Sekundenbruchteile an jeder Sprechstelle den Geräuschpegel messen (was natürlich ausgesetzt werden muss, wenn gerade keine Eingangskanäle frei sind).

Eine noch einfachere Methode wären Zeitfenster, die reihum auf alle Sprechstellen verteilt werden: Alle z.B. 5 Sekunden „hört“ der Fahrstuhl jeder Sprechstelle kurz zu (kann dem Benutzer signalisiert werden), und wenn in dieser Zeit ein Dialog begonnen wird (Aufwachbegriff), wird die Aufmerksamkeitsphase für die betreffende Sprechstelle verlängert. Da das zu z.B. nur 20 Prozent Chance für den Benutzer führt, ohne Signalisierung den Aufwachbegriff im richtigen Moment zu sagen (während 5 Sprechstellen sich 1 freien Kanal teilen), ist diese Methode nicht zu empfehlen.

Bei gleichem Verkabelungsaufwand wie für die Signalisierung nötig ist sind dann Rufknöpfe zur Aktivierung der Mikrofone die bessere Methode. Es ist auch denkbar, vorhandene Knöpfe und Signalleuchten zu verwenden: Wird der Rufknopf gedrückt, aber der Aufwachbegriff fällt nicht, kommt der Aufzug einfach in das betreffende Stockwerk. Oder man verzichtet auf den Stockwerken ganz auf Aufwachbegriffe: Wird der Knopf gedrückt, fragt der Aufzug per Sprachausgabe nach dem Wunsch des Benutzers.

Das hat gleich mehrere Vorteile: Man kann mit der Aufforderung warten, bis die Spracherkennungssoftware gestartet und das Mikrofon eingeschaltet ist, die Aufforderung durch den Aufzug erspart eine vorherige Einweisung der Benutzer (wer nicht mit einem sprechen rechnet, drückt dennoch den Knopf) und Umstehende werden wahrscheinlich weniger Nebengeräusche verursachen – ein Mensch kann leichter mitten im Gespräch erkennen,

dass der Aufzug einen Dialog beginnen will, als umgekehrt der Fahrstuhl erkennen kann, wenn ein Gespräch unterbrochen wird um den Aufzug anzusprechen.

7.3. Verfügbare Soundkarten

7.3.1. Midiman Delta 1010 Mehrspur-Soundkarte

Die erste Karte die bei uns ankam war eine **Midiman Delta 1010** (<http://www.midiman.de>), eine **MT-Soundkarte** (MT: Mehrspur / Multitracking) für semiprofessionellen Studioeinsatz. Diese Karte bietet 8 analoge und 2 digitale Eingänge, ebensoviele Ausgänge und eine **MIDI-Schnittstelle** zum Anschluss von z.B. Synthesizern oder Effektgeräten. Dabei stehen beachtliche 96 kHz Samplingrate und 24 Bit Genauigkeit zur Verfügung, was einem Signal/Rausch-Verhältnis (SNR) von 109 dB entspricht. Bei 96 kHz Samplingrate können Frequenzen bis zu 48 kHz verarbeitet werden (Nyquist-Theorem): Menschen können allerdings keine Frequenzen über 20 kHz wahrnehmen. Die Karte bietet also absolute Studioqualität, zumal der Analogteil in einem abgeschirmten 19 Zoll Gehäuse getrennt aufgebaut ist. Mit einem Preis von 1600 DM ist sie allerdings auch die teuerste mir bekannte Alternative.

Ein wichtiger Vorteil der Midiman ist die Verfügbarkeit von Treibern für verschiedene Windowsversionen (mitgeliefert) und Linux (kostenlos: **Advanced Linux Sound Architecture (ALSA)** <http://www.alsa-project.org> kommerziell: **Open Sound System (OSS)** <http://www.opensound.com>).

Für Windows werden die Treibermodelle DirectSound, ASIO und MME unterstützt. Eine Spracherkennungssoftware ohne besondere Unterstützung der genannten Treiber kann immernoch jeweils ein Stereopaar als normale Sounddevice öffnen, damit sind für die unterstützten Windowsversionen (95, 98, 98 SE, ME, 2000, NT) unter allen Umständen 4 gleichzeitig laufende Spracherkenner und sehr häufig sogar volle 8 auf 1 Rechner möglich. Dabei droht natürlich leicht eine Überlastung oder Instabilität des Rechners: Das zur Zeit verwendete L & H System läuft trotz eines einzigen Erkenners kaum einen Tag am Stück durch. L & H empfehlen je nach Einsatzgebiet und Betriebssystem 32 bis 96 MByte RAM, für einzelne Erkenner einen Pentium, für mehrere Erkenner eine besondere Version ihres Programmes und ca. 300 MHz. Midiman empfiehlt bei 96 kHz allein zum „Datenschaukeln“ mindestens einen Pentium II 350 MHz oder einen entsprechenden AMD Prozessor und 64 MByte RAM. Für 48 kHz werden immernoch 300 MHz MMX benötigt.

Sollen mehrere Erkenner gleichzeitig laufen, ergeben sich verschiedene Probleme: Bei HTK besteht die Gefahr, dass recht viel Rechenleistung verbraucht wird. Die Windowsversionen der verwendeten Erkenner unterstützen alle kein ASIO oder DirectSound und können daher nur Stereopaare öffnen, also können nur die Hälfte der Kanäle unabhängig voneinander benutzt werden. Der Erkenner von L & H ist unter Windows NT aus unklarem Grund nicht in der Lage, mehr als einen Kanal zu benutzen – will man einen weiteren Kanal öffnen, bricht der Zugriff mit einem internen Fehler ab. Unter Windows 98 ist das Problem schlicht, dass wir nur die Lizenz zur Benutzung von 2 Kanälen haben und zusätzliche

Zugriffe abgelehnt werden.

Lernout und Hauspie bieten noch einen anderen Erkennen an (**ASR 1500 TSO** , der in Verbindung mit einem speziellen Telefonanschluss die Sprache von acht Telefonaten gleichzeitig erkennen soll. Dieser Erkennen ist für computergesteuerte Auskunft- und Bestellsysteme gedacht und auf die schlechte Tonqualität von Telefonaten abgestimmt. Es ist allerdings fraglich, ob er ausreichend gut mit einer Soundkarte (z.B. über ASIO) verbunden werden kann. Lernout und Hauspie (die ohnehin in Schwierigkeiten stecken) haben eine entsprechende Anfrage noch nicht beantwortet, wir haben diesen Spezialerkennen nicht angeschafft.

7.3.2. Terratec EWS 88 MT

Eine preiswertere Alternative zur Midiman ist die **EWS 88 MT von Terratec** (<http://www.terratec.de>). Der Analogteil ist dort in einen 5.25 Zoll Laufwerkseinschub statt in ein 19 Zoll Gehäuse eingebaut, ausserdem fehlt die Midi-Schnittstelle zur Ansteuerung von Synthesizern und sonstigen elektronischen Musikinstrumenten und Effektgeräten in der Standardausführung. Die restlichen technischen Daten sind praktisch identisch, aber unter Linux wird die Karte nur vom kommerziellen Open Sound System Treiber (siehe 7.3.1 oben) unterstützt: Da dieser aber nur 60 Dollar kostet, bleibt die Kombination EWS 88 MT (1000 DM) plus Treiber trotzdem preiswerter als die Lösung von Midiman.

Praktische Tests haben wir mit dieser Karte nicht durchgeführt, da recht bald klar wurde, dass wir in der ersten Phase des Projektes nur eine Sprechstelle in der Kabine benutzen würden und deshalb die Experimente zunächst mit einer einfachen PC Soundkarte für unter 100 DM fortgesetzt wurden.

7.3.3. Standardsoundkarten

Bereits ab weit unter 100 DM bekommt man heute verschiedene einfache (z.B. **Sound-Blaster 16** kompatible) Soundkarten, wie sie sich in vielen PCs befinden. Ursprünglich für Geräuscheffekte oder Musik aus dem Computer vorgesehen, lassen sie sich auch für unsere Zwecke einsetzen. Normalerweise bieten solche einfachen Soundkarten 16 Bit Genauigkeit und Samplingraten bis 44100 Hz. 16 Bit sind bei sinnvoll eingestellter Lautstärke mehr als ausreichend zur Spracherkennung, 24 Bit Genauigkeit würde sogar rechnerisches Nachregeln der Lautstärke bei ausreichender Qualität erlauben. Da aber praktisch alle Soundkarten einen digital steuerbaren Mischer haben, es ist nicht unbedingt erforderlich, am Signal herumzurechnen, um die Lautstärke zu korrigieren. Oft reicht es aus, den Mischer von Hand auf eine mittlere Lautstärke zu programmieren und auf Nachregelung ganz zu verzichten.

Beim Einsatz einfacher Soundkarten kann gerade bei **PCI Soundkarten** oft problemlos mehr als eine Karte in einen Rechner eingebaut und so die Kanalzahl erhöht werden, sofern genügend Steckplätze vorhanden und frei sind. Oft sind 4 bis 6 PCI Steckplätze vorhanden. Grafikkarte, Netwerkkarte und Festplattencontroller können jeweils einen davon belegen, falls die entsprechende Funktionalität benötigt wird und nicht schon direkt auf der

Hauptplatine vorhanden ist. Ausser der Anzahl freier Steckplätze im Computer begrenzt meistens die Zahl freier **Interruptleitungen** und **DMA Kanäle** die Zahl einbaubarer Karten:

Eine typische Soundblaster 16 Soundkarte (noch für die alten **ISA Steckplätze** mit 16 Bit Breite) benötigt eine solche Interruptleitung, um den Computer über Statusänderungen zu informieren (ein Interrupt löst ein Unterprogramm im PC aus, so dass dieser nicht ständig den Status abfragen muss). Ausserdem werden 2 DMA-Kanäle verbraucht, die das Kopieren der Daten zwischen Computer und Soundkarte automatisieren. Ein normaler PC stellt 16 Interruptleitungen zur Verfügung und 8 DMA Kanäle, wovon jeweils einer nicht nutzbar ist. Festplatten mit DMA-Datenübertragung brauchen ggf. auch noch DMA Kanäle.

Bei einem normalen Heim-PC werden für Zeitgeber, Tastatur, Maus, Drucker, Uhr, Netzwerkkarte, Gleitkomma-Prozessoreinheit und Festplatten bereits insgesamt 9 Interruptleitungen verbraucht, Modem, Midi-Anschluss und ggf. besondere Einstellungen der Grafikkarte würden bis zu weiteren 3 Leitungen verbrauchen.

Bei neueren Steckkarten können sich mehrere Geräte eine Interruptleitung teilen („IRQ-sharing“), das funktioniert jedoch nur mit geeigneten Treibern. Gerade bei PCI sind die Chancen gut, aber es ist schwierig, einen DMA Kanal zwischen mehreren Geräten aufzuteilen.

Ein Interrupt ist ein einzelnes **Ereignis**, während bei DMA der **Prozess** des automatischen Kopierens eines ganzen Speicherbereichs im Mittelpunkt steht. Während für eine Karte also Daten bewegt werden, ist der ganze Kanal belegt. Bei einem Interrupt reicht es dagegen, den Verursacher herauszufinden (sofern der Interrupt von mehreren Geräten gemeinsam verwendet wird) und dem zugehörigen Gerätetreiber zu melden. Während dieser das Ereignis weiter verfolgt, ist die Interruptleitung schon wieder frei.

Im Normalfall macht es also aus verschiedenen Gründen kaum Sinn, mehr als zwei oder drei Soundkarten in einen einzelnen Rechner einzubauen. Da bei Spiele- und DVD-Freunden vierkanaliger Surround Sound zunehmend in Mode kommt, hat selbst eine einfache Soundkarte inzwischen oft 4 Ausgänge, aber nur 2 Eingänge. Drei einfache Soundkarten können also nicht die geforderten sieben Sprechstellen gleichzeitig bedienen – dieses Problem lässt sich aber meistens verschmerzen, da selten alle Sprechstellen gleichzeitig von Benutzern „besprochen“ werden. Mehr dazu gleich im Anschluss.

7.3.4. Spezielle Wandler-Hardware

Für die Verwendung mehrerer Sprechstellen kann auch spezielle Hardware eingesetzt werden, die oft wesentlich preiswerter als entsprechende Mehrkanal-Studiosoundkarten ist. Dabei sind besonders integrierte Wandler zwischen analogen und digitalen Signalen interessant, die kaum zusätzliche Beschaltung brauchen und oft schon eine serielle oder parallele Schnittstelle zum Anschluss an einen Computer eingebaut haben.

Für Studioeinsatz gibt es auch direkt über Kabel oder Glasfaser anschliessbare Wandler. Entsprechende Schnittstellen sind teilweise auf Soundkarten auch der unteren Preisklasse schon vorhanden, insgesamt würde aber eine solche Lösung dennoch ähnlich teuer ausfallen

wie eine Mehrspur-Studiosoundkarte, da mehrere Wandler (und Schnittstellen) benötigt werden.

Ebenfalls ungeeignet sind die meisten Oszilloskop- oder Mess-Karten für PC. Erstere bieten ihre enorme zeitliche Auflösung / Samplingrate oft nicht im Dauerbetrieb an, sondern können nur einzelne kurze Signale mit hoher Samplingrate in einem internen Speicher aufzeichnen und dann deutlich langsamer an den PC übertragen. Die Samplingrate bei durchgehender Aufzeichnung ist dann deutlich niedriger, ausserdem stehen nur wenige Kanäle zur Verfügung. Bei Messkarten sind zwar oft mehrere Kanäle verfügbar, aber diese teilen sich dann intern einen einzelnen hochgenauen aber langsamen Wandler. Versucht man, mehrere Kanäle gleichzeitig einzulesen, sinkt also die Samplingrate pro Kanal auf einen für Audioanwendungen zu niedrigen Wert.

Es gibt allerdings Messkarten, die sich mit 12 Bit Genauigkeit begnügen und dafür eine bessere sogenannte Summenabtastrate von etwa 100 kHz erreichen. Indem man diese Rate auf mehrere Kanäle aufteilt, lassen sich immerhin 4 bis 6 Eingänge mit ausreichender Qualität realisieren, zu einem angemessenen Preis von 250 DM. Diese Karten verwenden einen Wandler mit eingebauter Umschaltmöglichkeit zwischen mehreren (z.B. 8 oder 16) Eingängen, was eine Mehrfachnutzung erlaubt: Man schliesst zum Beispiel 8 Sprechstellen an einen Wandler an und beschränkt sich auf maximal 4 gleichzeitige Dialoge. Siehe oben unter 7.2.4 „scheinbare Skalierbarkeit“ und unten unter 7.3.6 „Multiplexing“.

7.3.5. Ein-Chip Wandler-ICs

Will man mehr Kanäle und trotzdem eine ausreichende Samplingrate, kann man mehrere integrierte Wandlerbausteine gleichzeitig verwenden. Integrierte 8 Bit Wandler sind zwar sehr preiswert (z.B. TLC 549, etwa 4 DM), für die Mikrofoneingänge sind aber weniger als 12 Bit Genauigkeit nicht sinnvoll. Entsprechende Bausteine kosten dann schon etwa 30 bis 50 DM, können aber mit einer Samplingrate von 50 kHz und mehr eventuell zwei oder mehr Kanäle bedienen. Entsprechende Umschalter zwischen mehreren Eingängen sind teilweise schon integriert.

Ein besonders schneller Baustein (LTC 1419, 70 DM) kann ganze 800000 Messungen pro Sekunde machen. Diese Rate auf viele Kanäle zu verteilen ohne das Messergebnis zu sehr zu verfälschen ist aber schwierig. Mit einer perfekten Verteilung der Messleistung könnten ganze 36 Kanäle über 22000 mal pro Sekunde gemessen werden! Für grosse Anlagen sollte diese Möglichkeit dennoch in Betracht gezogen werden: Zwei dieser Bausteine mit einer komplizierten Schaltung an 50 Sprechstellen anzubinden kann immernoch einfacher sein, als 50 einzelne Wandler gleichzeitig anzusteuern.

Sogar Studiosoundkarten stossen hier an ihre Grenzen: Beispielsweise sollte man nicht mehr als vier Midiman 1010 gleichzeitig in einen PC einbauen und ist damit auf 40 Kanäle begrenzt – allerdings dann mit 24 Bit Genauigkeit und 96 kHz Samplingrate pro Kanal, also perfekter Tonqualität. Gerade bei sehr vielen Sprechstellen ist es aber kaum noch machbar, auf allen gleichzeitig Dialoge zuzulassen. Hier sollte man unbedingt auf „scheinbare Skalierung“ (siehe 7.2.4 oben) zurückgreifen, und zum Beispiel für 20 Sprechstellen an einem einzigen Fahrstuhl maximal Ressourcen für 8 gleichzeitige Dialoge anschaffen.

Genauere Werte bekommt man durch Tests im praktischen Betrieb oder durch Simulation. Einerseits sollen die Benutzer nicht zu lange warten, bis der Fahrstuhl den Dialog akzeptiert, andererseits sollen nicht 95 Prozent der Hard- und Software die meiste Zeit mit dem Warten auf einen Dialog beschäftigt sein!

7.3.6. Multiplexing vieler Sprechstellen auf wenige Analoganschlüsse

Beschränkt man den Zugriff per Software auf wenige Kanäle gleichzeitig, sind zumindest schnellere Messkarten für den Einsatz in einem System denkbar, welches seine „Aufmerksamkeit“ immer nur einer oder wenigen Sprechstellen zuteilt. Eine mittlere Messkarte für 250 DM erreicht bei ausreichenden 12 Bit Genauigkeit eine Summenabtastrate von 100 kHz, es sind mindestens 8 Eingänge vorhanden. Will man jeden Kanal mit einer Samplingrate von 22 kHz betreiben, lassen sich also maximal 4 der Eingänge gleichzeitig benutzen.

Eine Schaltung, die viele Kanäle und wenige Wandler verbindet (**Multiplexer / Demultiplexer**) lässt sich auch recht einfach auch „von Hand“ passend für die jeweilige Anwendung konstruieren und beispielsweise an eine Soundkarte anschliessen. Die Steuerung kann dabei ohne Probleme über den Druckerport oder eine digitale Schnittstellenkarte geschehen. Ein Multiplexer ist dabei eine Schaltung, die einen von mehreren Eingängen mit einem Wandler verbindet, ein Demultiplexer verteilt das Ausgangssignal eines Wandlers auf mehrere Ausgänge. Die Kombination mehrerer (De-) Multiplexer nenne ich der Einfachheit halber im Folgenden einen Multiplexer.

Dazu ein Beispiel mit einfacher Hardware: Es sind 7 Sprechstellen vorhanden, aber nur eine Soundkarte mit 4 Ausgängen und 2 Eingängen. Es ist nicht praktikabel, einen Anschluss an zwei Geräte mit jeweils halber Samplingrate zu verteilen: Die Kanäle würden stark ineinander übersprechen, da Tiefpassfilter auf der Karte den vom Kanalwechsel bedingten Sprung im Signal verwischen. Der Fahrstuhl muss also während einem Dialog einen Teil seiner Ressourcen jeweils einer Sprechstelle fest zuordnen. Auch während auf den Beginn eines Dialoges gewartet wird, ist es besser, jeweils mindestens für Zehntelsekunden an jeder Sprechstelle den Geräuschpegel zu messen oder sogar auf ein Stichwort zu warten, als zu versuchen, alle 8000tel Sekunde ein Sample von jeder Sprechstelle zu bekommen. Es wäre schwierig, die Samples mit ausreichender Qualität zu erfassen, so dass wirklich für jede Sprechstelle parallel ständig auf ein Stichwort gewartet werden könnte.

Es ergeben sich hier zwei Szenarios: Sofern die Sprechstellen **per Hardware** selber das Stichwort zum Dialogbeginn erkennen können oder man den Dialogbeginn vereinfacht **über einen Rufknopf** einleitet, sind alle Kanäle für Dialoge verfügbar. Andernfalls muss immer ein Kanal freigehalten werden, der die Sprechstellen reihum abhört und auf neu beginnende Dialoge wartet. Da mehr Ausgänge als Eingänge vorhanden sind, kann der Benutzer bereits zum Sprechen aufgefordert oder um noch etwas Geduld gebeten werden, bevor die Ressourcen für einen neuen Dialog wirklich frei sind.

Insbesondere kann und sollte das Mikrofon einer Sprechstelle, an der gerade der Fahrstuhl selber spricht abgeschaltet werden. Dadurch besteht keine Gefahr, dass die Spra-

cherkennung versucht, Äusserungen des Fahrstuhls zu interpretieren. Ausserdem kann der freigewordene Kanal währenddessen anderweitig verwendet werden.

Um alle Möglichkeiten offenzuhalten, sollten alle Kanäle frei den Sprechstellen zuzuordnen sein. Für die Auswahl von einer aus 7 Sprechstellen sind 3 digitale Steuerleitungen nötig, bei 2 Eingängen und 4 Ausgängen also 18 Leitungen. Man kann sich leicht klarmachen, dass 3 je Steuerleitungen je 8 Bitkombinationen / Binärwerte annehmen können. Der achte Wert kann dabei fest für mit „kein Signal“ verbunden werden, um zum Beispiel Mikrofone per Hardware abzuschalten. Manche Spracherkenner bieten keine Softwareschnittstelle an, über die sie zum „Weghören“ aufgefordert werden können, hier bietet sich also eine einfache Alternative zum Eingriff in die Software des Soundtreibers.

Ohnehin lohnt es sich kaum, durch komplizierte Codierungen Leitungen einsparen zu wollen: Mit jeder Sprechstelle ist zu jedem Zeitpunkt maximal ein Eingang oder ein Ausgang verbunden. Umgekehrt ist jeder Eingang oder Ausgang zu jedem Zeitpunkt mit genau einer Sprechstelle verbunden oder signalfrei geschaltet. Lässt man die Möglichkeit „kein Signal“ weg, gibt es für den ersten Anschluss sieben mögliche Sprechstellen, für den zweiten sechs, und so weiter: Insgesamt gibt es dann 5040 sinnvolle Verschaltungen, zu deren Codierung immernoch 13 Leitungen nötig sind. Die Ansteuerung dieser 13 Leitungen ist aber deutlich komplizierter, als jedem von 6 Kanälen (oder jeder von 7 Sprechstellen) einfach jeweils drei Steuerleitungen zuzuordnen, die dann jeweils eine Sprechstelle (oder einen Kanal) auswählen.

Mit Bausteinen wie dem **CMOS IC 4051** (analoger bidirektionaler 1 aus 8 Umschalter, unter 2 DM) lässt sich so ein digital steuerbares „Patchfeld“ zur Verteilung und Verschaltung analoger Signale einfach realisieren. Eine ausreichende Zahl von Steuerleitungen bekommt man über preiswerte **digitale PIO Schnittstellenkarten** (zum Beispiel 48 Leitungen, in verschiedenen Gruppen als Eingänge, Ausgänge oder bidirektional konfigurierbar: 80 DM) oder einfach durch Verwendung von zwei Druckerports. Druckerports sind auf weniger als 100 Kilobyte pro Sekunde, einfache Schnittstellenkarten sind ähnlich langsam, die oben genannte Karte wird über einen **ISA-Bus** mit 8 Bit Breite und 8 MHz Takt an den PC angeschlossen, eine Übertragungsrate von maximal einigen Megabyte pro Sekunde ist zu erwarten.

Ein **Druckerport** hat je 8 Datenleitungen (bei modernen Ausführungen bidirektional, sonst in Richtung Drucker), 5 Statuseingänge (Drucker an PC, einer davon kann auch zusätzlich einen Interrupt auslösen) und 4 Steuerausgänge. Ohne besondere Anpassungen an den Treiber oder direkten Hardwarezugriff (braucht unter Linux **root** oder **kmem** Rechte) sollte man sich aber darauf beschränken, die Datenleitungen zu verwenden. Werden wirklich viele Anschlüsse benötigt, kann man an spezielle **integrierte PIO-Bausteine wie den 8255** (PIO: parallel input/output) verwenden, der auch auf digitalen Schnittstellenkarten eingesetzt wird. Diese Bausteine werden z.B. über einen 8 Bit breiten Datenbus und zwei bis drei Adressleitungen angeschlossen und bieten dann z.B. jeweils 24 Anschlüsse. Gerade bei Anschluss über den Druckerport muss hier aber beachtet werden, dass oft nur eine Übertragungsrate von unter 100 Kilobyte pro Sekunde möglich ist.

Den PC um eine Druckerschnittstelle zu erweitern kostet etwa 30 bis 40 DM, mehr als drei Schnittstellen sind normal nicht sinnvoll. Für die vorgeschlagene Multiplexerschaltung

ist es auch denkbar, einen einzelnen Druckerport zu verwenden: Drei Datenleitungen wählen eine Sprechstelle, drei weitere einen Kanal, der Rest kann zum Beispiel als Taktsignal dienen. Noch einfacher ist es, die vorhandene Steuerleitung „Strobe“ zu verwenden, und mit zwei Datenleitungen eines von vier **Registern** zu wählen, das dann jeweils die 6 Bit der restlichen Datenleitungen speichert: Damit lassen sich über eine Druckerschnittstelle mit geringem Aufwand 24 Steuerleitungen ohne betriebssystemabhängige Spezialtreiber ansteuern. Ein solches Register für bis zu 8 Bit ist zum Beispiel der Standardbaustein (IC in Low Power Schottky TTL Bauweise) **74LS273 / 74LS373** für unter 2 DM. Viele andere Bauarten sind denkbar. Ein digitales „Patchfeld“ für 6 Kanäle und 7 Sprechstellen und nur einen Druckerport wird insgesamt für etwa 30 bis 50 DM zu realisieren sein, vorverstärkte Pegel von den Mikrofonen vorausgesetzt.

7.4. Treiber für Mehrkanalzugriff

Hat man sich für eine Hardwarelösung entschieden, muss noch das Dialogsystem auf diese Hardware abgestimmt werden. Wenn die Spracherkenner zu viele Ressourcen für einen Rechner brauchen, die Soundhardware sich aber in einem Rechner konzentriert, müssen die Audiodaten an mehrere Rechner per Netzwerk verteilt werden. Verteilt sich die Soundhardware ohnehin auf mehrere Rechner, empfiehlt es sich, die Spracherkenner auf allen Rechnern laufen zu lassen, damit nur noch der erkannte Text über das Netzwerk verschickt werden muss, beziehungsweise Status- und Steuer- Informationen mit dem Rechner ausgetauscht werden müssen, der mit der Fahrstuhlsteuerung verbunden ist.

Wenn Soundhardware UND Spracherkenner jeweils auf mehrere Rechner verteilt sind, können normale Soundtreiber verwendet werden (wenige Kanäle pro Rechner) und die Kommunikation zwischen den Rechnern beschränkt sich auf die Dialogkomponente: Alle Erkenner schicken ihre Ergebnisse an einen Rechner für die Dialogauswertung, der wiederum mit der Fahrstuhlsteuerung kommuniziert. Oder aber jeder Rechner wertet die Dialoge „seiner“ Kanäle aus und schickt dann nur noch Fahrtziele über den entsprechenden Rechner oder sogar direkt an die Fahrstuhlsteuerung. Erfolgt die Steuerung des Fahrstuhls nämlich über die Simulation von Knopfdrücken, können über eine einfache Verschaltung mehrere Rechner gleichzeitig direkt mit der Fahrstuhlsteuerung / dem „Knopfsimulator“ verbunden werden.

7.4.1. Soundtreiber für Windows

Ich beschränke mich im Folgenden auf die Probleme bei Bündelung der Soundhardware auf einem Rechner. Unter Windows stehen verschiedene Treibermodelle zur Verfügung: Klassische **digital Wave Audio (PCM) Geräte** , **ASIO** , **DirectSound** und **MME** . ASIO und MME sind dabei besonders für Mehrspur-Soundkarten aus dem Studiobereich vorgesehen. Gerade die Mehrspurkarten bringen oft Treiber für alle neueren Windowsversionen, also sowohl 95/98/ME als auch NT/2000 mit, wobei ASIO und DirectSound nur auf den Versionen für Spieler und Heimanwender (95/98/ME) verfügbar sind, die wieder-

um beim gleichzeitigen Betrieb mehrerer Spracherkennerprozesse leicht überfordert werden können. Der Betrieb des Systems ist dann nicht mehr so zuverlässig wie man das von einer Fahrstuhlsteuerung erwartet. Die von uns verwendeten Spracherkenner unterstützen nur den einfachen PCM Treiber, was zur Folge hat, dass man linken und rechten Kanal eines Stereopaars nicht auf zwei Erkennen verteilen kann. Der ASR 1600 Erkennen liegt uns sowieso nur mit einer Lizenz zur Analyse von maximal zwei Kanälen gleichzeitig vor, wegen einem Fehler im Treiber oder im Erkennen geht sogar das nur mit Windows 98, nicht mit Windows NT, wenn man die Midiman Soundkarte verwendet.

Das klassische Gerätemodell wird von allen Spracherkennern und Soundkarten unterstützt, oft lassen sich die Kanäle aber nur als Stereopaare ansprechen. Wenn der verwendete Spracherkenner mit halben Stereopaaren nicht richtig umgehen kann, kann man also nur die Hälfte der Kanäle verwenden! Mögliche Probleme: Der Erkennen kann nicht auf „nur links“ oder „nur rechts“ festgelegt werden, oder der Treiber erlaubt für jedes Stereopaar nur einen parallelen Zugriff, also nur einen Erkennen. Optimal wäre hier ein Erkennen, der ein Stereopaar öffnen und zugleich beide Kanäle getrennt auf einmal analysieren kann.

Die spezielleren Treibermodelle erlauben den Zugriff auf jeden Kanal einzeln. Soweit der verwendete Erkennen und die verwendete Hardware es also ermöglichen, führt diese Methode zuverlässiger zum Erfolg. Für speziell angefertigte Hardware stehen leider kaum Treiber zur Verfügung, zumindest Messkarten (siehe 7.3.4 oben) oder über den Druckeranschluss gesteuerte Multiplexer (siehe 7.3.6 oben) lassen sich aber mit erträglichem Aufwand ansteuern. Das Problem ist die zu erwartende Inkompatibilität zum Spracherkenner: Während ein Multiplexer ohnehin von der Dialogkomponente angesteuert wird, ist die Verbindung von Spracherkenner und Messkarte durch die nicht aufeinander abgestimmten **Programmerschnittstellen (API, Application Programmer Interface)** ein ernstes Problem.

7.4.2. Soundtreiber für Linux

Für Linux stehen hauptsächlich die zwei Treibermodelle **ALSA** und **OSS** (siehe 7.3.1 oben) zur Verfügung: ALSA bietet einen einfachen Gerätezugriff über die **/dev/dsp Gerätedatei**, komplexere Funktionen werden überwiegend über eine API realisiert, die die beim Compilieren von Programmen Einbindung der **ALSA Bibliotheken** erfordert. OSS verwendet **ioctl Aufrufe**, um komplexere Funktionen über Zugriffe auf die Gerätedatei abzubilden. Die ioctl-Aufrufe brauchen keine besondere Bibliothek, wohl aber besondere Headerdateien, da die verwendeten Funktionen über (nicht ausserhalb von OSS standardisierte) Nummern ausgewählt werden.

Soll nur eine Standardsoundkarte verwendet werden, genügt prinzipiell der direkte Zugriff auf die Gerätedatei mit Standard-Dateifunktionen, weitere Befehle sind nur zum Beispiel zur Einstellung der Samplingrate nötig. Da wir jedoch mehrere Kanäle getrennt öffnen und getrennt der Spracherkennung zuführen wollen, ergeben sich weitere Schwierigkeiten.

Der ALSA-Treiber für Mehrspur-Soundkarten wie Midiman Delta 1010 erlaubt nur das gleichzeitige Öffnen aller Kanäle, man muss also über ein spezielles Programm die Soundkarte zugreifen und die Kanäle dann einzeln an die verschiedenen Spracherkenner weiterleiten. Ein solches Programm stelle ich im nächsten Abschnitt vor: Es leitet die Kanäle

über TCP/IP Netzwerkverbindungen weiter, ein kleines Hilfsprogramm erlaubt dann den Empfang. Die empfangenen Daten können in eine Datei oder eine „Pipeline“ geschrieben werden. Sofern also der Spracherkenner Pipelines unterstützt, ist eine Weiterverarbeitung in Echtzeit möglich. Für den Spracherkenner HTK von Entropic lässt sich eine entsprechende Erweiterung über die Programmierschnittstelle (API) einbauen, mit zusätzlicher Verbesserung der Erweiterung könnte sogar das Hilfsprogramm entfallen.

Der OSS-Treiber unterstützt Mehrspur-Soundkarten nur in der kommerziellen Version (60 Dollar), bietet dann aber sehr umfangreiche Unterstützung des Konzepts von Gerätedateien: für jedes Stereopaar wird eine Gerätedatei eingerichtet, ausserdem kann beim Öffnen von `/dev/dsp` optional angegeben werden, wieviele und ggf. welche Kanäle man zugreifen will. Wenn man also aufzeichnet, welche Kanal wie zugeteilt wurde, kann man einfach mehrere Spracherkenner jeweils `/dev/dsp mono` öffnen lassen, um mehrere Kanäle gleichzeitig zu analysieren.

Soll spezielle Hardware zum Einsatz kommen, muss ggf. ein entsprechender Treiber erst noch geschrieben werden. Während unter Windows 95/98/ME ein Programm direkt auf die Hardware zugreifen kann, sind bei Linux und Windows NT besondere Rechte nötig. Bei Linux muss das Programm mit den erhöhten Rechten von **root oder kmem** laufen. Bei allen genannten Betriebssystemen kann alternativ ein normaler Treiber geschrieben und ins Betriebssystem eingebunden werden, was wegen des verfügbaren **Quellcodes** anderer Treiber unter Linux deutlich einfacher ist:

Zum Beispiel kann man den Treiber einer Messkarte und den einer Soundkarte als Vorlage nehmen, um einen Treiber zu schreiben, der die Messkarte über `/dev/dsp` auf OSS-kompatible Weise zugänglich macht. Verwendet man stattdessen den direkten Zugriff durch ein Programm, sollte nur ein Hilfsprogramm die Root-Rechte und direkten Hardwarezugriff bekommen, während die Weiterverarbeitung der Daten ohne besondere Rechte erfolgt. Das hat den Vorteil, bei einem Fehler im Hauptprogramm keine fehlerhafte Ansteuerung der Hardware oder Veränderung von wichtigen Daten zu riskieren. Also: Sicherheit vor Abstürzen, Systemveränderungen, unberechtigtem Zugriff durch Benutzer die Unsicherheiten in der Fahrstuhl- Software ausnutzen.

7.4.3. Treiber für die Übertragung von Audiodaten im Netz

Für die Midiman Delta 1010 Soundkarte (jeweils 8 analoge Ein- und Ausgänge) habe ich eine spezielle Software für Linux entwickelt, die bisher im Zusammenhang mit HTK getestet wurde. Da der ALSA Treiber für diese Soundkarte nur erlaubt, alle Kanäle einer Richtung gleichzeitig zu öffnen, kann HTK nicht direkt auf den Treiber zugreifen. HTK kann immer nur einen Kanal pro Prozess untersuchen, muss also mehrfach gestartet werden, wenn mehrere Sprechstellen aktiv sind. Mein **Server** hält alle Kanäle ständig offen, liest alle Daten von ihnen und verteilt die Daten der einzelnen Kanäle an HTK Prozesse oder andere **Clients** nach Bedarf. Hat ein Kanal zu einem Zeitpunkt keinen Client, werden die aufgezeichneten Daten gleich wieder verworfen. Mein Programm bietet also die Sprachdaten zentral an (Server) und gibt sie auf Anfrage an „Kunden“ (Clients) weiter.

Diese Aufteilung in Client und Server mit Verbindung übers Netzwerk hat den Vorteil,

dass die Spracherkennung auf mehrere Rechner verteilt werden kann beziehungsweise ein Operator an übers Netzwerk mithören oder zur weiteren Untersuchung mitschneiden kann, was der Fahrstuhl hört. Sofern Client und Server auf dem selben Rechner laufen, funktioniert die Datenübertragung sehr schnell und ressourcenschonend (über das sogenannte local **loopback** unter Linux), über eine externe Verbindung sind zum Beispiel bei einer Samplingrate von 22050 Hz und 16 Bit Genauigkeit pro Kanal 44100 Byte pro Sekunde zu übertragen, was in einem normalen Ethernet-Netzwerk bei den von uns angestrebten 7 Sprechstellen kein Problem sein sollte.

Für die Übertragung wird eine Verbindung auf Basis von **TCP/IP** aufgebaut, einem Standard-Protokoll zur Datenübertragung im Netzwerk. Nachdem der Server gestartet ist, kann jederzeit ein Client eine Verbindung aufbauen. Der Server antwortet dann mit dem Text „Hello!“ und erwartet dann vom Client eine Anfrage. Es gibt bisher nur 2 Arten von Anfrage: Sendet der Client den Text „CLOSE“, beendet sich der Server. Sendet der Client den Text „GETx“, wobei an Stelle des x eine Zahl stehen muss, antwortet der Server mit einem Strom von signed short int Werten, die das Eingangssignal von Kanal x enthalten. Will der Client keine weiteren Daten empfangen, beendet er einfach die Verbindung. Tritt beim Server ein Fehler auf, beendet dieser seinerseits die Verbindung. Ausserdem ist der Server auf zwei Clients pro Kanal begrenzt. Beim anderen Anfragen als „CLOSE“ gibt der Server Fehlermeldungen aus:

„FAIL: Pardon?“ im Falle einer nicht interpretierbaren Anfrage, „FAIL: Channel?“ wenn ein nicht vorhandener Kanal gelesen werden soll, und „FAIL: Busy“ wenn ein Kanal bereits zwei Clients hat.

Das Programm ist bisher recht elementar aufgebaut, kann aber einfach erweitert werden. In **rec-sock-alsa** (G.1.1) werden von **main()** aus zuerst die Einstellungen für die Soundkarte gesetzt und alle Kanäle geöffnet (**open_sound**). Dann wird **open_socket** gerufen, welches einen **Thread** (nebenläufigen Verarbeitungsprozess, verwendet die **pthread** Funktions-Bibliothek von Linux) startet, der fortan auf eingehende Verbindungen auf dem beim Serverstart angegebenen Port (TCP/IP-Kanalnummer) wartet. Dieser Thread ist in **server-sockets** (G.1.2) implementiert und wird später noch genauer erklärt. Das Programm wird dann mit der Funktion **read_sound** fortgesetzt, die solange Daten von der Soundkarte liest, bis ein Fehler auftritt oder der Server eine „CLOSE“ Aufforderung empfängt. Dann wird der Server beendet, vorher werden noch **close_socket** und **close_sound** aufgerufen, um restliche offene TCP/IP Verbindungen und die Verbindung zur Soundkarte zu beenden.

open_sound und **close_sound** sind in **alsa-ice** (G.1.3) implementiert und verwenden Funktionen der ALSA-Bibliothek zum Zugriff auf die Soundkarte. Die Funktionen dieser Bibliothek sind auf der ALSA-Homepage (<http://www.alsa-project.org>) detailliert beschrieben, die Beschreibung kann auch zusammen mit Treiber und Bibliothek dort heruntergeladen werden.

read_sound liest Daten in einen Puffer und setzt dann ein Signal, dass neue Daten eingetroffen sind. Nach kurzer Wartezeit werden die Daten für ungültig erklärt und neue Daten gelesen. Alte Daten werden also einfach überschrieben. Mehrere Puffer reihum als Ringpuffer zu verwenden um mehr Zeit zum Abholen der Daten anzubieten ist robuster,

wurde hier aber noch nicht implementiert. Egal wie die Daten gepuffert werden, zu alte Daten sollten besser verworfen werden, damit die Spracherkennung möglichst zeitnah erfolgen kann.

`send_buffer` wandelt die Daten vor dem Senden um und filtert sie, so dass nur ein Kanal an den Client verschickt wird. Für komplexere Aufgaben dieser Art bietet ALSA ein Plugin-Konzept an, bei dem man vorgefertigte Funktionen zwischen den eintreffenden Rohdaten und die letztlich in `send_buffer` eintreffende Version „einklinken“ kann. Damit würde `send_buffer` einfacher, aber die Initialisierung der Plugins und die Notwendigkeit von Puffern für jeden Kanal machen den Server insgesamt eher komplexer. Für einen Server mit mehreren Funktionen empfiehlt sich die Methode trotzdem, da die Verarbeitung dann modularer und übersichtlicher stattfindet.

Die Abwicklung von Verbindungen ist in `server-sockets` (G.1.2) implementiert und besteht aus drei Funktionen: `open_socket` öffnet einen TCP/IP „Socket“ und startet einen weiteren Thread `accept_conn`, der auf eingehende Verbindungen wartet. Dann kehrt `open_socket` zurück, `accept_conn` läuft weiter bis der Server aufgefordert wird, sich zu beenden. Sobald eine Verbindung eingeht, wird ein neuer Thread `serve_partner` gestartet, der diese Verbindung dann bedient und bis solange läuft, wie die jeweilige Verbindung besteht. `serve_partner` ruft nach Bedarf `send_buffer` aus `rec-sock-alsa` auf, welches wartet, bis gültige Daten im Puffer sind und dann daraus die Daten für einen einzelnen Kanal ausliest und über die TCP/IP Verbindung in Form eines Feldes von signed short int Werten (16 Bit) abschickt.

Eine *wichtige Erweiterung* des Servers ist ein Befehl zur Stummschaltung eines Kanals: Oft liegen Lautsprecher und Mikrofon nahe beieinander, falls der Spracherkenner also nicht während einer Sprachausgabe des Aufzugs an den Benutzer auf Pause gestellt werden kann, wird der Spracherkenner sonst versuchen, die Sprache des Aufzuges zu analysieren. Kurz: Der Aufzug würde sich selber zuhören, was unnötig ist und leicht zu Fehlverhalten führt. Eine entsprechende Funktion muss in `serve_partner` aktiviert werden, indem dort ein Befehl erkannt wird (Syntax zum Beispiel `MUTEx` und `HEARx` wobei `x` eine Kanalnummer angibt). Dort wird dann eine Tabelle aktualisiert, die für jeden Kanal angibt, ob er gerade stummgeschaltet ist. Ebenfalls in `serve_partner` muss dann diese Tabelle abgefragt werden, um vor jedem Aufruf von `send_buffer` zu entscheiden, ob wirklich `send_buffer` aufgerufen werden soll oder eine Funktion `send_silence`, die einen Puffer gleicher Grösse aber ohne Sprachdaten sendet. Stattdessen werden Daten gesendet, die einem Signal der Lautstärke Null entsprechen.

Ein Beispielclient ist `client2file` (G.2) : Das Programm sendet sein drittes Argument als String an den im ersten und zweiten Argument angegebenen Rechner und Port des Servers. Die empfangenen Daten werden dann über die Standardausgabe ausgegeben und können per Umleitung dieser in eine Datei oder eine „Pipeline“ weitergeleitet werden. Das Client-Programm ist sehr einfach. Der Spracherkenner kann entweder so eingestellt oder erweitert werden, dass er Mikrofondaten aus einer „Pipeline“ akzeptiert, oder man kann den Spracherkenner direkt so erweitern, dass er direkt mit dem Server kommuniziert.

Wird die Ausgabe von `client2file` in eine sogenannte „named pipes“ geschrieben (eine „Pipeline“ die einen eigenen Dateinamen besitzt), kann der Spracherkenner die Daten sogar

fast wie aus einer normalen Datei lesen – die Option, Daten aus einer Datei zu analysieren, ist in vielen Spracherkennern vorhanden. `client2file` kann so erweitert werden, dass die Daten in ein anderes Dateiformat umgewandelt werden, ausserdem kann man über die „Pipelines“ alle Daten durch das Konvertierungsprogramm `sox` leiten und so ein anderes Datenformat erreichen. Durch das einfache Protokoll ist es nicht schwierig, einen Client aus Standardkomponenten in verschiedenen Programmiersprachen zu implementieren. Sogar von der Unix Kommandozeile kann man auf den Server zugreifen, indem man die Daten zum Beispiel mit `netcat` übers Netzwerk schickt (telnet ginge zur Not auch).

Hier nochmal ein Ablaufbeispiel. Zu Beginn:

```
rec-sock-alsa 1234 & (Server startet)
```

```
client2file localhost 1234 GET5 | erkenner & (ein Erkenner)
```

```
client2file localhost 1234 GET3 | erkenner & (weiterer Erkenner)
```

Auf dem Rechner des Operators:

```
client2file soundserver 1234 GET5 >/dev/dsp & (Mithören was der erste Erkenner hört)
```

Am Ende des Experiments:

```
client2file soundserver 1234 CLOSE (Server wird beendet)
```

Zum Abspielen von Sprache an einer Sprechstelle steht bisher nur ein einfaches Programm namens `play-alsa` (G.3) bereit, mit dem man eine an der Kommandozeile angegebene Audiodatei auf einem vorgegebenen Kanal abspielen kann. Da auch zum Abspielen alle Kanäle gleichzeitig zugegriffen werden müssen, ist es mit dem Programm nicht möglich, den Fahrstuhl an mehreren Sprechstellen zugleich sprechen zu lassen. Dazu und um die oben beschriebene MUTE Funktion zu vereinfachen, *sollte der Server um eine Abspielfunktion erweitert werden*. Eine denkbare Syntax ist `PLAYx dateiname`, um eine Datei auf einem Kanal abzuspielen und zu Beginn und Ende der Wiedergabe automatisch MUTE und HEAR aufzurufen.

Den Server auch beim Abspielen per Netzwerk mit Audiodaten zu versorgen ist weniger sinnvoll, da der Fahrstuhl nur wenige vorgegebene Sätze verwendet und die Bereitstellung neuer Audiodaten mit einem Sprachsyntheseprogramm nur wenig Ressourcen verbraucht. Ausserdem kann man die Audiodateien mit Dateiübertragungsprogrammen wie `ftp` oder `scp` übers Netzwerk aktualisieren, wenn sie nicht sogar auf einem übers Netzwerk verbundenen Laufwerk liegen. Soll ein PUT für in Echtzeit erzeugte Audiodaten dennoch vorgesehen werden, braucht der Server auch zum Abspielen einen Puffer, um einen gleichmässigen Datenstrom zur Soundkarte sicherzustellen.

Weitere Verbesserungsvorschläge:

Das Protokoll kann kompatibel zu `http` Version 1.0 und `icecast` gestaltet werden. Die Syntax lautet dann zum Empfangen `GET /5**` (* steht für einen Zeilenwechsel) und zum Senden (wie bei PUT) `SOURCE password /5**`, wobei im Beispiel die Sprechstelle 5 angesprochen wird. Entsprechende Anpassungen in `serve_partner` sind ohne grossen Aufwand zu machen. Ausserdem kann man in Server und Client Programme wie `sox` oder `lame3` (LameENC) und `mpeg123` aufrufen, um die Audiodaten in Formaten wie GSM oder MP3 komprimiert zu übertragen und so Netzwerkbandbreite zu sparen. Das PLAY Kommando

kann im Voraus auf bestimmte beim Serverstart genannte Dateien beschränkt werden, die dann gleich im Arbeitsspeicher eingelagert werden. Dadurch greift der Server nur beim Start auf die Festplatte zu, was manchmal wünschenswert ist (zum Beispiel wenn die Festplatte langsam ist oder sogar nur übers Netzwerk erreichbar und gar nicht im Rechner mit der Soundkarte eingebaut ist).

A. HTK Ettikettierung

I) Vokale

<i>Label</i>	<i>Beispielwörter/Erklärung</i>
2	Möbel, nötig
26	
6	Messer, bitter
9	Köche, öffnen
96	Mörder, dörfllich
@	Bitte, Puste
A	Pass, Dackel
A6	
E	Rest, fett
E6	Erster, bersten
EE	Däne, ähnlich
EE6	Ähre, Fähre
I	Mitte, bis
I6	Birke, Wirt
O	Woche, doch
O6	Orbit, Dorf
OY	Beule, heute
U	Mutter, lustig
U6	Urlaub, urteilen
Y	Mütter, lüften
Y6	Fürth, Bürde
a	Bahn, baden
a6	Bar, Narr
aI	Meinung, beißen
aU	Haut, raufen
e	See, leben
e6	Meer, Leer
i	Biene, gießen
i6	Bier, wir
o	Hose, Not

o6	Ohr, Tor
u	Fuge, gut
u6	Fuhre, Uhr
y	Tüte, müde
y6	Tür, beführworten

II) Konsonanten

<i>Label</i>	<i>Beispielwörter/Erklärung</i>
C	ich, Gelächter
N	Spange, eng
Q	<Glottalverschluß, z.B. zwischen der ersten und zweiten Silbe in Beamter>
S	Schuh, schlank
Z	Garage, Rouge
b	Bitte, Bus
bh	<Plosivlösung des /b/>
d	Dank, Durst
dh	<Plosivlösung des /d/>
f	Fuß, finden
g	Gas, gut
gh	<Plosivlösung des /g/>
h	Habe, husten
j	Jod, Jacke
k	Kopf, küssen
kh	<Plosivlösung des /k/>
l	Liebe, lösen
m	Mut, machen
n	Nase, nehmen
p	Park, persönlich
ph	<Plosivlösung des /p/>
r	Rose, raten
s	Tasse, besser
t	Tür, tanzen
th	<Plosivlösung des /t/>
v	Wirt, was
x	Wucht, kochen
z	Sonne, sieben

III) Sonstiges

Label

Atmung

Geräusch

Klicken

Klopfen

Lachen

Pause

Räuspern

Rascheln

Schlucken

Schmatzen

B. PhonDat–Anpassung für Lautmodelle

lab_one_mit_grenzen.pl

```
#!/usr/bin/perl -w

## it takes into consideration sentences with
## combination _ +/.../+ _ and
## segments which contain noises and are being overlapped
## by other noises
## prints the labels in the new format

%begin_end = ();
$previous_label = "";

##### SUBROUTINES #####
#####

sub belongs_to_array
{
    my ($elem, $arr) = @_;
    my @arr = @$arr;
    my $res = 0;
    my $i = 0;
    while (($res == 0) && ($arr[$i]))
    {
        if ($arr[$i] eq $elem)
        {
            $res = 1;
        }
        ++$i;
    }
    return $res;
}
```

```

}
sub process_overlapping_segment
{
    my ($string) = @_ ;
    @arr = split(/\s+/, $string);
    my $noise = $arr[0];
    my $temp;
    $noise =~ s/(\<|\:|\||\>)//g;
    $arr[$#arr] =~ s/:\>//;
    my $words_nr = 0;
    my $i = 1;
    while ($arr[$i])
    {
        if (($arr[$i] ne ",") && ($arr[$i] ne ".") &&
            ($arr[$i] ne "<#Mikrowind>"))
        {
            if ($arr[$i] =~ /(\<Schmatzen\>)/)
            {
                $second_noise = $1;
                $second_noise =~ s/(\<|\>|\#)//g;
                ++$words_nr;
                $temp = $noise . "->$words_nr";
                push(@sounds, $temp);
                push(@sounds, $second_noise);
                $words_nr = 0;
            }
            elsif ($arr[$i] =~ /(\<Lachen\>|\<\#Klopfen\>)/)
            {
                $second_noise = $1;
                $second_noise =~ s/(\<|\>|\#)//g;
                ++$words_nr;
                $temp = $noise . "->$words_nr";
                push(@sounds, $temp);
                push(@sounds, $second_noise);
                $words_nr -= 2;
            }
            else
            {
                ++$words_nr;
            }
        }
        ++$i;
    }
}

```

```

    if ($string =~ /\_\\s+\\+\\/(.*?)\\/+\\s+\\_/)
    {
        $words_nr -= 2;
    }
    if ($words_nr != 0)
    {
        $noise .= "->$words_nr";
        push (@sounds, $noise);
    }
    return 1;
}

sub modify_label
{
    my ($in, $begin) = @_;
    my $prev;

    if ($previous_label =~ /[e|a|S|A]/)
    {
        $prev = "h";
    }
    else
    {
        $prev = $previous_label . "h";
    }

    if ($in =~ /(\\$|\\#)&/) ## (#|$)&... -> nichts
    {
        $in = "";
    }
    else
    {
        if ($in =~ /\\#\\#/)
        {
            print OUT "$begin \\t $begin \\t \\#\\#\\n";
            $in =~ s/\\#\\#(.*)/\\#$1/;
        }
        if (($in =~ /a/) && (!($in =~ /a(:|E|I|U)/)) &&
            (!($in =~ /(Lachen|R\\"auspern|Schmatzen|Ger\\"ausch|Rascheln)/)))
        {
            $in =~ s/a/A/;
        }
        $in =~ s/\\$\\-h/\\$$prev/; #0
    }
}

```

```

    $in =~ s/(a|e|i|o|u|y|2):/$1/g; #1
    $in =~ s/E:/EE/g; #2
    $in =~ s/\#c://g; #3
    $in =~ s/\#?(,|\.|;|\?|-MA|\\_)//; #4
    $in =~ s/\$(-MA|=\\/+|-|\\/-|\\/+|=\\/+|=\\/+|=\\/-|;|_|-~|=)//; #5
    $in =~ s/%//g; #6
    $in =~ s/\\+$//; #7
    $in =~ s/\\$\\#\\/\\$/; #8
    $in =~ s/(\\#|\\$)(.*?)-(.*)/$1$3/; #9
    $in =~ s/(\\#|\\$)(\\'?'\\'|\\")?(.*)/$3/; #10
    $in =~ s/(\\:k|l:l|q:r|s:w:g|q|v:z:n:)//g; #11
}
return $in;
}

sub print_new_line
{
    ## converts the old line format in the new format
    ## begin_label end_label label

    my ($arr) = @_ ;
    my @arr = @$arr;
    my $begin; my $end;

    $begin = ($arr[1] + 8) * 625;
    $arr[2] = modify_label($arr[2], $begin);
    $previous_label = $arr[2];
    if (($arr[2] ne "") && ($begin_end{$arr[1]}))
    {
        $end = ($begin_end{$arr[1]} + 8) * 625;
        print OUT "$begin \\t $end \\t $arr[2]\\n";
    }
    return 1;
}

##### MAIN PROGRAM #####
#####

## go through the all labels and find for every label_begin the
## label_end; store all pairs in a hash

$ff = $ARGV[0];
open(HASH, $ff);
open(OUT, ">$ARGV[1]");

```

```

@list = <HASH>;

$region = 1;
foreach $line (@list)
{
    if ($line =~ /^hend/)
    {
        $region = 3;
    }
    elsif ($region == 3)
    {
        $begin = $line;
        $begin =~ s/\s+(\d+)(.*)/$1/;
        $region = 4;
    }
    elsif ($region == 4)
    {
        chop($line);
        $temp = $line;
        $temp =~ s/\s+(\d+)(.*)/$1/;
        if ($begin != $temp)
        {
            $begin_end{$begin} = $temp;
        }
        $begin = $temp;
    }
}
close(HASH);

## start the real processing of the file

open(IN, $ff);
open(OUT, ">$ARGV[1]");

$name = $ARGV[0];
$name =~ s/\.slh//;
print OUT "\"*/$name.*.lab\"\\n";

@signals = (<Lachen>, "<Husten>", "<R\\\"auspern>", "<Schmatzen>",
            "<Schlucken>", "<Ger\\\"ausch>", "<#Klicken>", "<#Klingeln>",
            "<#Klopfen>", "<#Mikrobe>", "<#Mikrowind>", "<#Quietschen>",
            "<#Rascheln>", "<#>");

```

```

$region = 1;
$sentence = "";
@sounds = ();
$being_the_first_noise = 0;

while(<IN>)
{
    chop();
    if ($_ eq "oend")
    {
        $region = 2;

        ## process the text from region 1 ##
        while ($sentence ne "")
        {
            if ($sentence =~ /</)
            {
                $sentence =~ s/^(.*?)<(.*)/<$2/;
                if ($sentence =~ /<:/) ## noise and words overlapp
                {
                    $temp = $sentence;
                    $temp =~ s/<:(.*?):>(.*)/<:~>/;
                    $sentence =~ s/<:(.*?):>(.*)/$2/;
                    process_overlapping_segment($temp);
                }
            }
            else ## isolated noise
            {
                $temp = $sentence;
                $temp =~ s/<(.*?)>(.*)/<$1>/;
                $res = belongs_to_array($temp, \@signals);
                if ($res)
                {
                    if ($temp ne "<#>")
                    {
                        $temp =~ s/(\<|\#|\>)//g;
                    }
                    push (@sounds, $temp);
                }
                $sentence =~ s/^(.*?)>(.*)/$2/;
            }
        }
    }
    else
    {

```

```

        $sentence = "";
    }
}
}
elseif ($_ eq "hend")
{
    $region = 4;
    $index = 0;
}
elseif ($region == 1)
{
    $sentence .= $_;
}
elseif ($region == 4)
{
    if (/\\#(\\:k|l\\:|q\\:|r\\:|s\\:|w\\:|g\\:)/) ## labels for different noises
    {
        $being_the_first_noise = 1;
        @temp = split(/\\s+/);

        if ($sounds[$index])
        {
            $subst = $sounds[$index]; ## the noise the label stands for

            if ($subst =~ /->/) ## overlapping noise
            {
                $nr = $subst;
                $nr =~ s/(.*?)->(.*)/$2/;
                if ($nr == 1) ## last word in overlapping segment
                {
                    ++$index;
                }
                else ## still in the overlapping segment
                {
                    --$nr;
                    $sounds[$index] =~ s/(.*?)->(.*)/$1->$nr/;
                }
                if ($temp[2] =~ /\\#\\#/) ## if word border keep it ##
                {
                    $temp[2] = "##";
                    print_new_line(\\@temp);
                }
            }
        }
    }
}

```

```

else ## isolated noise
{
    # if word border, keep it

    $temp[2] =~ s/\#\#(.*?)\-\#\#/;
    $temp[2] =~ s/\#\#(.*?)\-\//;

    if ($subst ne "<#>")
    {
        $temp[2] =~ s/(\:k|l\:|q\:|r\:|s\:|w\:|g\:)/$subst/;
    }
    else ## $subst = <#>
    {
        $temp[2] =~ s/(\:k|l\:|q\:|r\:|s\:|w\:|g\:)//;
    }
    ++$index;
    print_new_line(\@temp);
}
}
}
elseif ((/\$s:|\$l:|\$k:/) ## isolated noise contained in an overlapping \
segment
{
    if ($being_the_first_noise == 1)
    {
        @temp = split(/\s+/);
        if (($sounds[$index]) && (!($sounds[$index] =~ /->/)))
        {
            $subst = $sounds[$index];
            $temp[2] =~ s/(\$s:|\$l:|\$k:)/$subst/;
            ++$index;
            print_new_line(\@temp);
            $being_the_first_noise = 0;
        }
    }
}
else
{
    @temp = split(/\s+/);
    print_new_line(\@temp);
}
}
}

```

```
print OUT ".\n";
close(IN);
close(OUT);
```

lab_all_mit_grenzen.pl

```
#!/usr/bin/perl -w

open(IN1, $ARGV[0]);
open(OUT1, ">$ARGV[1]");
print OUT1 "##MLF!\n";
close(OUT1);

while(<IN1>)
{
    chop();
    print "process $_\n";
    system "lab_one_mit_grenzen.pl $_ /tmp/label$$";
    print "finished processing $_\n";
    system "cat $ARGV[1] /tmp/label$$ > /tmp/temp$$";
    system "cp /tmp/temp$$ $ARGV[1]";
}
system "\\rm /tmp/temp$$ /tmp/label$$";
close(IN1);
```

C. PhonDat–Anpassung für Wortmodelle

erzeugt _aussprache _lexikon.pl

```
#!/usr/bin/perl -w

##### SUBROUTINES #####
#####

sub convert_kiel_labels_to_our_labels
{
    my ($string) = @_;

    ## here come the modifications ##

    ## labels to be deleted: %-: % ' " # =-MA * +
    $string =~ s/^(%-:|%|\'|\"|\#|=-MA|\*|\+)/g;

    ## delete at the beginning: =-:k =-;
    $string =~ s/^(=-:k|=-;)/g;

    ## delete: =-~
    $string =~ s/=-~/g;

    ## delete at the beginning: :k-: or :k followed by anything but -
    ## or :k:k followed by anything but -
    $string =~ s/^(k-:)/g;
    $string =~ s/^(k:k)([^-])/$2/g;
    $string =~ s/^(k)([^-])/$2/g;

    ## delete at the beginning: Q-:
    $string =~ s/^Q-:/g;
```

```

## delete at the end: l: q: r: s: w: g: v: z: n:
$string =~ s/(l:|q:|r:|s:|w:|g:|v:|z:|n:)$/g;

## adapt vowels
$string =~ s/a([^(|I|U|)])/A$1/g;
$string =~ s/E:/EE/g;
$string =~ s/(a|e|i|o|u|y|2):/$1/g;

## delete z:-: and l:-:
$string =~ s/(z|l):-://g;

## z:= -> =
$string =~ s/z:=/=/g;

# delete: ==
$string =~ s/==//g;

## lab-: --> nichts
$string =~ s/(i6|y6|I6|Y6|e6|26|E6|EE6|EE|96|a6|A6|O6|o6|u6|U6|aI|aU|OY|\w|\@)\
-://g;

## :-lab --> lab
$string =~ s/:(i6|y6|I6|Y6|e6|26|E6|EE6|EE|96|a6|A6|O6|o6|u6|U6|aI|aU|OY|\w|\@)\
/$1/g;

## lab1-lab2 --> lab2
$string =~ s/(i6|y6|I6|Y6|e6|26|E6|EE6|EE|96|a6|A6|O6|o6|u6|U6|aI|aU|OY|\w|\@)-\
(i6|y6|I6|Y6|e6|26|E6|EE6|EE|96|a6|A6|O6|o6|u6|U6|aI|aU|OY|\w|\@)/$2/g;

## delete q
$string =~ s/q//g;

## ph -> pph, th -> tth, kh -> kkh,
## bh -> bbh, dh -> ddh, gh -> ggh
$string =~ s/(p|t|k|b|d|g)h/$1$1h/g;

## delete: l: z: g: p: h:
$string =~ s/(l|z|g|p|h)://g;

## delete once: ; =
$string =~ s/(;|=|\~)//g;

```

```

##### end modifications #####

return $string;
}

##### MAIN PROGRAM #####
#####

open(IN, $ARGV[0]); ## input file - first parameter
open(OUT, ">$ARGV[1]"); ## output file - second parameter

## contains all pronunciation variants that appeared in the third column
%global_hash = ();

while(<IN>)
{
    chop(); ## remove end of line
    @temp_array = split(/\s+/);
    ## $temp[0] contains the word (column 1)
    ## $temp[2] contains its pronunciation variant (column 3)

    ## convert the variant in our labels
    $variant = convert_kiel_labels_to_our_labels($temp_array[2]);

    ## check if the variant was already met
    if (!$global_hash{$variant}) ## first occurrence of this variant
    {
        ## delete % and <;T> from words in the first column
        $temp_array[0] =~ s/(\<;T\>|\%)/g;

        if ($variant ne "")
        {
            ## checks if after adapting the labels, the variant is still not empty
            ## this case appeared just once in kielcd4.lsv, for:
            ## Sie      zi:+    z-:i:-:++      340      1      G426A008      2

            $global_hash{$variant} = 1; ## introduce the variant in the hash
            print OUT "$temp_array[0]\t$variant\n";
        }
    }
}

close(IN);
close(OUT);

```

create_master_file_lexicon.pl

```
#!/usr/bin/perl -w

## hash that contains all labels for noises; they must be alone in one line
## the noises are different for different phondat.mlf variants
## that's why, this hash must be new written for every new phondat.mlf

%noise_hash = ("p:", 1, "h:", 1, "Schmatzen", 1, "R\"auspern", 1,
               "Schlucken", 1, "Lachen", 1, "Mikrobe", 1, "Klicken", 1,
               "Klopfen", 1, "Rascheln", 1, "Ger\"ausch", 1,
               "Quietschen", 1);

open(OUT, ">$ARGV[1]");

## hash that contains all pairs: pronunciation variant - word
## example: ybbh6A1 - "uberall"
%variant_word_hash = ();

## all pronunciation variants from phondat.mlf which where not
## found in aussprache_lexikon.txt
%not_found_in_aussprache_lexikon = ();

## hash with all pronunciation variant from aussprache_lexikon.txt
## which were also found in phondat.mlf
%found_in_phondat = ();

##### SUBROUTINES #####
#####

sub all_noises
{
    ## checks if all the labels from the input array are noises

    my ($temp) = @_ ;
    @labels = @$temp;

    my $i = 0;
    my $noises = 1;
    while(($labels[$i]) && $noises)
    {
        if (!$noise_hash{$labels[$i]})
    
```

```

        {
            $noises = 0;
        }
        ++$i;
    }
    return $noises;
}

sub process_new_word
{
    ## takes a word (pronunciation variant), contained in three arrays
    ## (begins, ends, labels) and looks for the the corresponding word
    ## in aussprache_lexikon.txt; noises on the first or last position
    ## in the word are written alone; noises in the middle of the word
    ## are ignored; if all labels are noises, each of them gets
    ## a separate line

    my ($first_parameter, $second_parameter, $third_parameter) = @_ ;

    my @labels = @$first_parameter;
    my @begins = @$second_parameter;
    my @ends   = @$third_parameter;

    if (all_noises(\@labels)) ## all the labels in this word are noises
    {
        my $k = 0;
        while($labels[$k])
        {
            print OUT "$begins[$k]\t$ends[$k]\t$labels[$k]\n";
            ++$k;
        }
    }
    else ## the word contains also normal labels
    {
        my $final_word = "";

        $number_of_labels = $#labels;

        my $begin_word;
        if ($begins[0])
        {
            $begin_word = $begins[0];
        }
    }
}

```

```

else
{
    $begin_word = -1;
}
my $i = 0;
my $word = "";

while($labels[$i]) ## check all labels
{
    if ($i == 0) ## first position in word
    {
        if ($noise_hash{$labels[$i]}) ## noise
        {
            print OUT "$begins[$i]\t$ends[$i]\t$labels[$i]\n";
            $j = $i + 1;
            if ($labels[$j])
            {
                $begin_word = $begins[$j];
            }
            else
            {
                $begin_word = -1;
            }
        }
        else ## normal label
        {
            $word .= $labels[$i]; ## add it to the other normal labels
        }
    }
    elseif ($i == $number_of_labels) ## last position in word
    {
        if ($noise_hash{$labels[$i]}) ## noise
        {
            $j = $i - 1;
            if ($variant_word_hash{$word})
            {
                ## the variant was found in aussprache_lexikon.txt,
                ## so the variant belongs to both files
                $found_in_phondat{$word} = 1;
                ## take the corresponding word
                $word = $variant_word_hash{$word};
            }
            else

```

```

        {
            ## the variant was not found in aussprache_lexikon.txt
            $not_found_in_aussprache_lexikon{$word} = 1;
            $word .= "\t***"; ## mark the variant with ***
        }
        print OUT "$begin_word\t$ends[$j]\t$word\n";
        print OUT "$begins[$i]\t$ends[$i]\t$labels[$i]\n";
        $begin_word = -1;
    }
    else ## normal label
    {
        $word .= $labels[$i];
    }
}
else ## position in the middle of the word
{
    if (!$noise_hash{$labels[$i]}) ## normal label
    {
        ## normal labels are concatenated, noises are ignored
        $word .= $labels[$i];
    }
}
++$i;
}
--$i;
$end_word = $ends[$i];
if ($begin_word != -1)
{
    if ($variant_word_hash{$word})
    {
        ## the variant was found in aussprache_lexikon.txt
        ## so the variant belongs to both files
        $found_in_phondat{$word} = 1;
        ## take the corresponding word
        $word = $variant_word_hash{$word};
    }
    else
    {
        ## the variant was not found in aussprache_lexikon.txt
        $not_found_in_aussprache_lexikon{$word} = 1;
        $word .= "\t***"; ## mark the variant with ***
    }
}
print OUT "$begin_word\t$end_word\t$word\n";

```

```

    }
}
return 1;
}

##### MAIN PROGRAM #####
#####

## load the information from aussprache_lexikon.txt in %variant_word_hash
open(IN1, "aussprache_lexikon.txt");
while(<IN1>)
{
    chop();
    @temp = split(/\s+/);
    ## temp[0] --> word
    ## temp[1] --> pronunciation variant

    $variant_word_hash{$temp[1]} = $temp[0];
}
close(IN1);

## open for writing files that will contain list of not found words
open(OUT1, ">not_found_in_aussprache_lexikon.list");
open(OUT2, ">not_found_in_phondat_mlf.list");

## open for reading phondat.mlf version
open(IN, $ARGV[0]);

@labels = (); ## will contain labels
@begins = (); ## will contain labels beginnings
@end = (); ## will contain labels ends

while(<IN>)
{
    chop();
    if (/!MLF!/) ## first line
    {
        print OUT "$_\n";
    }
    elsif (/lab/) ## starts a new file
    {
        print OUT "$_\n";
    }
}

```

```

    }
    elseif (/^\./) ## ends a file AND the last word from this file
    {
        process_new_word(\@labels, \@begins, \@ends);
        \@labels = ();
        \@begins = ();
        \@ends = ();
        print OUT "$_\n";
    }
    else ## normal labels and word borders
    {
        if (/#\#/ ) ## word border; ends an old word and starts a new one
        {
            if ($labels[0])
            {
                process_new_word(\@labels, \@begins, \@ends);
                \@labels = ();
                \@begins = ();
                \@ends = ();
            }
        }
        else ## normal label
        {
            @temp = split(/\s+/);
            push(@labels, $temp[2]);
            push(@begins, $temp[0]);
            push(@ends, $temp[1]);
        }
    }
}
close(IN);
close(OUT);

## print all variants from phondat.mlf which were not found
## in aussprache_lexikon.txt
foreach $variant (keys %not_found_in_aussprache_lexikon)
{
    print OUT1 "$variant\n";
}

## print all variants from aussprache_lexikon.txt which were
## not found in phondat.mlf
foreach $variant (keys %variant_word_hash)

```

```
{
    if (!$found_in_phondat{$variant})
    {
        print OUT2 "$variant\n";
    }
}
close(OUT1);
close(OUT2);
```

D. Skripte für den Trainingsprozeß

HCopy.sh

```
#!/bin/bash

export WORKDIR=/courses/koreman.Fahrstuhl.00/PhonDat

# Derive MFCC_0_D_A_Z files for Read speech files:

mkdir MFCC_0_D_A_Z

export DATADIR=/proj/PhonDat/Data/Read
for file in $DATADIR/*.16; do
    name='mybasename.pl $file .16';
    echo $name;
    # Computation of parameters with log energy normalisation
    HCopy -C $WORKDIR/HCopy_MFCC_0_D_A_Z.cfg \
        $file MFCC_0_D_A_Z/Read/$name.CEP.htk
    chmod 444 MFCC_0_D_A_Z/Read/$name.CEP.htk
done

export DATADIR=/proj/PhonDat/Data/Spontan
for file in $DATADIR/*/*.{l,r}16; do
    name='mybasename.pl $file .[lr]16';
    echo $name;
    # Computation of parameters with log energy normalisation
    HCopy -C $WORKDIR/HCopy_MFCC_0_D_A_Z.cfg \
        $file MFCC_0_D_A_Z/Spontan/$name.CEP.htk
    chmod 444 MFCC_0_D_A_Z/Spontan/$name.CEP.htk
done
```

HCopy_MFCC_0_D_A_Z.cfg

```
# X-Waves -> MelSpec configuration file
# 28/6/96 bojan
# 22/7/96 adapted by JK
# 7/4/00 adapted by AE
# 22/6/00 adapted by JK

SOURCEFORMAT = NOHEAD    # headerless source format
SOURCERATE = 625         # 16k sampling rate
SOURCELABEL = HTK        # label format
BYTEORDER = NONVAX       # byte order is important to be cpu independent

TARGETKIND = MFCC_0_D_A_Z      #
TARGETFORMAT = HTK
TARGETRATE = 50000           # 5ms frame rate
WINDOWSIZE = 256000.0       # 25.6-ms window size
ZMEANSOURCE = T             # subtract DC component from signal
ENORMALISE = F             # live audio!
USEHAMMING = T             # use Hamming window
PREEMCOEFF = 0.97           # use pre-emphasis 0.97
CEPLIFTER = 22

# A. NUMBER OF MEL-SCALED CHANNELS
#
# To have a discrimination of 100 Hz under 1 kHz (to be able to tell all
# vowels apart) we need the following number of filters:
#
#  $n_{filt} = a / b$    where  $a$  = number of mel-scales at 8 kHz
#                         $b$  = size of mel-scale at 100 Hz
#
#  $a = 2595 \log_{10} (1 + 8000/700) = 2840.023$ 
#  $b = 2595 \log_{10} (1 + 100/700) = 150.4891$ 
#  $a / b = 2840.023 / 150.4891 = 18.871952$ 
#
# B. NUMBER OF BARK-SCALED CHANNELS
#
#  $n_{filt} = 13 \arctan (0.76 \times 8000/1000 + 3.5 \arctan (8000/7500)**2)$ 
#
#  $= 18.990343$  (critical bands)
#
# Formulas from O'Shaughnessy (1950). Speech Communication.
# Human and Machine, p. 150. Reading (MA), Addison-Wesley.
```



```
#
# CONCLUSION FROM A and B:
#
# NUMCHANS has to be greater than 18 to model critical bands. To be on
# the safe side, I'll use 24.

NUMCHANS = 24          # number of filterbank channels
LPCORDER = 12          # order of LPC analysis
NUMCEPS = 12           # number of cepstral coefficients

TRACE = 02             # trace level setting
V1COMPAT = F           # Version 1.5 compatibility mode
SAVECOMPRESSED = F     # don't compress the output data
SAVEWITHCRC = F        # don't mess data with CRC checksum
```

makefile.PhonDat_MX16tied

```
# makefile.PhonDat_MX16

# written by AE,BA,JK to train PhonDat database

#Original Data
DATADIR=/proj/PhonDat/Data

#base directory of the Experiment
BASEDIR=/courses/koreman.Fahrstuhl.00/
MFCCDIR=${BASEDIR}/PhonDat/MFCC_0_D_A_Z

#Subdir for different variants of the experiment
PROJDIR=${BASEDIR}/HMM16

HMMBASE=hmm_mx16

# Rare labels (n<50): will be modelled with 1 mixture/state
#           1x  Quietschen (not modelled, too few obs. seq. even
#                               for s-state, 1-mixture HMM)
#           2x  /96/
#           3x  /Z/
#           6x  Geraeusch
#           6x  Raeuspern
#           11x Mikrobe (not modelled, since this will not occur
#                               in the lift)
```

```

#                18x Schlucken
#                19x /26/
#                19x Lachen
#                39x Rascheln
# Infrequent labels (50<=n<100): will be modelled with 8 mixtures/state
#                56x U6
#                64x Klopfen
#                78x EE6
#                95x A6
# Pause wird mit nur einem Mixture modelliert (wie "rare"), da invariant.

LABELS_norm=      p ph t th k kh b bh d dh g gh Q f s S C x v z m n N \
                  l r j h i y I Y e 2 E EE 9 a A O o u U @ 6 \
                  Schmatzen Klicken Atmung
LABELS_normdiphth= aI aU OY i6 y6 I6 Y6 e6 E6 a6 O6 o6 u6
LABELS_infreq=     Klopfen
LABELS_infreqdiphth= U6 EE6 A6
LABELS_rare=       Z Geraeusch Raeuspfern Schlucken \
                  Lachen Rascheln Pause
LABELS_rarediphth= 96 26
LABELS= $(LABELS_norm) $(LABELS_normdiphth) $(LABELS_infreq) \
        $(LABELS_infreqdiphth) $(LABELS_rare) $(LABELS_rarediphth)

HMMINIT_norm=      $(foreach lab,$(LABELS_norm),Init/hmm_$(lab))
HMMINIT_normdiphth= $(foreach lab,$(LABELS_normdiphth),Init/hmm_$(lab))
HMMINIT_infreq=     $(foreach lab,$(LABELS_infreq),Init/hmm_$(lab))
HMMINIT_infreqdiphth= $(foreach lab,$(LABELS_infreqdiphth),Init/hmm_$(lab))
HMMINIT_rare=       $(foreach lab,$(LABELS_rare),Init/hmm_$(lab))
HMMINIT_rarediphth= $(foreach lab,$(LABELS_rarediphth),Init/hmm_$(lab))
HMMINIT=$(HMMINIT_norm) $(HMMINIT_normdiphth) $(HMMINIT_infreq) \
        $(HMMINIT_infreqdiphth) $(HMMINIT_rare) $(HMMINIT_rarediphth)
HMMREST=$(foreach lab,$(LABELS),Rest/hmm_$(lab))
HMMTIED=$(foreach lab,$(LABELS),Tied/hmm_$(lab))
HMMRESTTIED=$(foreach lab,$(LABELS),RestTied/hmm_$(lab))

HMMEVAL=HResults.out
#      HResults_words.out

all:    $(HMMBASE).init $(HMMBASE).rest $(HMMINIT) $(HMMREST) $(HMMTIED) \
        $(HMMRESTTIED) test.mlf HVite.list $(HMMBASE).SLF HVite.mlf $(HMMEVAL) \
        deutsch_tied.mmf

init:   $(HMMINIT)

```

```

rest:    $(HMMREST)
tied:    $(HMMTIED)
resttied:    $(HMMRESTTIED)

$(HMMINIT_norm):    HMMMODEL=$(HMMBASE)_norm
$(HMMINIT_norm):    $(HMMBASE)_norm.cfg
$(HMMINIT_normdiphth):    HMMMODEL=$(HMMBASE)_normdiphth
$(HMMINIT_normdiphth):    $(HMMBASE)_normdiphth.cfg
$(HMMINIT_infreq):    HMMMODEL=$(HMMBASE)_infreq
$(HMMINIT_infreq):    $(HMMBASE)_infreq.cfg
$(HMMINIT_infreqdiphth):    HMMMODEL=$(HMMBASE)_infreqdiphth
$(HMMINIT_infreqdiphth):    $(HMMBASE)_infreqdiphth.cfg
$(HMMINIT_rare):    HMMMODEL=$(HMMBASE)_rare
$(HMMINIT_rare):    $(HMMBASE)_rare.cfg
$(HMMINIT_rarediphth):    HMMMODEL=$(HMMBASE)_rarediphth
$(HMMINIT_rarediphth):    $(HMMBASE)_rarediphth.cfg

$(HMMBASE).init:
    find ${MFCCDIR} -name \*.CEP.htk > $(HMMBASE).init

Init Rest Tied RestTied:
    mkdir $@

$(HMMINIT):    $(HMMBASE).init Init
    cd Init && \
    echo "Initializing label /$(subst hmm_,,$(notdir $@))/" && \
    HInit -l $(subst hmm_,,$(notdir $@)) -o $(notdir $@) \
        -I ../../PhonDat/Labels/phondat_read.mlf \
        -I ../../PhonDat/Labels/phondat_spontan.mlf \
        -T 1 -S ../$(HMMBASE).init \
        ../$(HMMMODEL).cfg

$(HMMBASE).rest:    $(HMMBASE).init
    grep Spontan $(HMMBASE).init > $(HMMBASE).rest

Rest/%: Init/% $(HMMBASE).rest Rest
    cd Rest && \
    echo "Reestimating label /$(subst hmm_,,$(notdir $@))/" && \
    HRest -w 2 -m 1 -l $(subst hmm_,,$(notdir $@)) -v 0.0001 \
        -I ../../PhonDat/Labels/phondat_spontan.mlf -T 1 -S \
        ../$(HMMBASE).rest    ../Init/hmm_$(subst hmm_,,$(notdir $@))

Tied/%: $(HMMREST) $(HMMBASE)_ties.hhed $(HMMBASE).list HHed.cfg Tied

```

```

    echo "Tying reestimated HMMs" && \
    HHed -C HHed.cfg -T 1 -d Rest/ -M Tied/ $(HMMBASE)_ties.hhed hmm.list

RestTied/%:      Tied/% $(HMMBASE).rest RestTied
    cd RestTied && \
    echo "Reestimating label /$(subst hmm_,,$(notdir $@))/ after tying" && \
    HRest -w 2 -m 1 -l $(subst hmm_,,$(notdir $@)) -v 0.0001 \
        -I ../../PhonDat/Labels/phondat_spontan.mlf -T 1 \
        -S ../$(HMMBASE).rest -H ../Tied/newMacros \
        ../Tied/hmm_$(subst hmm_,,$(notdir $@))

deutsch_tied.mmf:      $(HMMRESTTIED)
    HHed -w $@ -d RestTied hhedplain.cfg hmm.list

#test.mlf HVite.list:
#    echo "#!MLF!#" >test.mlf && \
#    rm -f HVite.list && \
#    cutmlf.pl ${MFCCDIR}/Spontan ../../PhonDat/Labels/phondat_spontan.mlf \
#    HVite.list test.mlf

#$(HMMBASE).SLF: $(HMMBASE).net
#    HParse $(HMMBASE).net $@

#$(HMMBASE).list:
#    cd Rest && \
#    ls -1 * > ../$@
#
#HVite.mlf:      $(HMMBASE).SLF HVite.list $(HMMBASE).list
#    HVite -w $(HMMBASE).SLF -S HVite.list -d Rest -i HVite.mlf -T 1 \
#    $(HMMBASE).dict $(HMMBASE).list

#HResults.out:  HVite.mlf test.mlf HResults.list
#    HResults -p -I test.mlf HResults.list \
#    HVite.mlf > HResults.out

#HResults_words.out:  HVite.mlf test.mlf HResults_words.list
#    HResults -p -I test.mlf HResults_words.list \
#    HVite.mlf > HResults_words.out

#eval:  $(HMMEVAL)

clean:
    rm -f $(HMMBASE).init $(HMMBASE).rest $(HMMINIT) $(HMMREST) test.mlf \

```

```
HVite.list $(HMMBASE).SLF HVite.mlf $(HMMEVAL)
```

E. Das Lexikon

```
!PHONE-SET          hmm_p hmm_t hmm_k hmm_b hmm_d hmm_g hmm_Q hmm_f hmm_s hmm_S \
hmm_C hmm_x hmm_v hmm_z hmm_Z hmm_m hmm_n hmm_N hmm_l hmm_r hmm_j hmm_h hmm_i hmm_i6 \
hmm_y hmm_y6 hmm_I hmm_I6 hmm_Y hmm_Y6 hmm_e hmm_e6 hmm_2 hmm_26 hmm_E hmm_E6 hmm_EE \
hmm_EE6 hmm_9 hmm_96 hmm_a hmm_a6 hmm_A hmm_A6 hmm_O hmm_O6 hmm_o hmm_o6 hmm_u hmm_u6 \
hmm_U hmm_U6 hmm_aI hmm_aU hmm_OY hmm_@ hmm_6 hmm_Pause hmm_Atmung hmm_Schmatzen \
hmm_Raeuspern hmm_Schlucken hmm_Lachen hmm_Klicken hmm_Klopfen hmm_Rascheln hmm_Geraeusch
```

```
!SIL                []      hmm_Pause
```

```
!ANY                []      hmm_p
!ANY                []      hmm_t
!ANY                []      hmm_k
!ANY                []      hmm_b
!ANY                []      hmm_d
!ANY                []      hmm_g
!ANY                []      hmm_Q
!ANY                []      hmm_f
!ANY                []      hmm_s
!ANY                []      hmm_S
!ANY                []      hmm_C
!ANY                []      hmm_x
!ANY                []      hmm_v
!ANY                []      hmm_z
!ANY                []      hmm_Z
!ANY                []      hmm_m
!ANY                []      hmm_n
!ANY                []      hmm_N
!ANY                []      hmm_l
!ANY                []      hmm_r
!ANY                []      hmm_j
!ANY                []      hmm_h
!ANY                []      hmm_i
!ANY                []      hmm_i6
!ANY                []      hmm_y
!ANY                []      hmm_y6
```

!ANY	[]	hmm_I
!ANY	[]	hmm_I6
!ANY	[]	hmm_Y
!ANY	[]	hmm_Y6
!ANY	[]	hmm_e
!ANY	[]	hmm_e6
!ANY	[]	hmm_2
!ANY	[]	hmm_26
!ANY	[]	hmm_E
!ANY	[]	hmm_E6
!ANY	[]	hmm_EE
!ANY	[]	hmm_EE6
!ANY	[]	hmm_9
!ANY	[]	hmm_96
!ANY	[]	hmm_a
!ANY	[]	hmm_a6
!ANY	[]	hmm_A
!ANY	[]	hmm_A6
!ANY	[]	hmm_0
!ANY	[]	hmm_06
!ANY	[]	hmm_o
!ANY	[]	hmm_o6
!ANY	[]	hmm_u
!ANY	[]	hmm_u6
!ANY	[]	hmm_U
!ANY	[]	hmm_U6
!ANY	[]	hmm_aI
!ANY	[]	hmm_aU
!ANY	[]	hmm_OY
!ANY	[]	hmm_@
!ANY	[]	hmm_6
#!ANY	[]	hmm_Atmung
#!ANY	[]	hmm_Schmatzen
#!ANY	[]	hmm_Raeuspern
#!ANY	[]	hmm_Schlucken
#!ANY	[]	hmm_Lachen
#!ANY	[]	hmm_Klicken
#!ANY	[]	hmm_Klopfen
#!ANY	[]	hmm_Rascheln
#!ANY	[]	hmm_Geraeusch
#"Fahrstuhl"		
fahrstuhl	[F]	hmm_f hmm_a6 hmm_S hmm_t hmm_u hmm_l
fahrstuhl	[F]	hmm_f hmm_a6 hmm_S hmm_t hmm_U hmm_l
fahrstuhl	[F]	hmm_f hmm_a6 hmm_S hmm_t hmm_u

fahrstuhl	[F]	hmm_f	hmm_a6	hmm_S	hmm_t	hmm_U		
fahrstuhl	[F]	hmm_f	hmm_a6	hmm_S	hmm_t	hmm_u	hmm_Q	
fahrstuhl	[F]	hmm_f	hmm_a6	hmm_S	hmm_t	hmm_U	hmm_Q	
fahrstuhl	[F]	hmm_f	hmm_a6	hmm_s	hmm_t	hmm_u	hmm_l	
fahrstuhl	[F]	hmm_f	hmm_a6	hmm_s	hmm_t	hmm_U	hmm_l	
fahrstuhl	[F]	hmm_f	hmm_a6	hmm_s	hmm_t	hmm_u		
fahrstuhl	[F]	hmm_f	hmm_a6	hmm_s	hmm_t	hmm_U		
fahrstuhl	[F]	hmm_f	hmm_a6	hmm_s	hmm_t	hmm_u	hmm_Q	
fahrstuhl	[F]	hmm_f	hmm_a6	hmm_s	hmm_t	hmm_U	hmm_Q	
fahrstuhl	[F]	hmm_f	hmm_a6	hmm_C	hmm_t	hmm_u	hmm_l	
fahrstuhl	[F]	hmm_f	hmm_a6	hmm_C	hmm_t	hmm_U	hmm_l	
fahrstuhl	[F]	hmm_f	hmm_a6	hmm_C	hmm_t	hmm_u		
fahrstuhl	[F]	hmm_f	hmm_a6	hmm_C	hmm_t	hmm_U		
fahrstuhl	[F]	hmm_f	hmm_a6	hmm_C	hmm_t	hmm_u	hmm_Q	
fahrstuhl	[F]	hmm_f	hmm_a6	hmm_C	hmm_t	hmm_U	hmm_Q	
fahrstuhl	[F]	hmm_h	hmm_a6	hmm_S	hmm_t	hmm_u	hmm_l	
fahrstuhl	[F]	hmm_h	hmm_a6	hmm_S	hmm_t	hmm_U	hmm_l	
fahrstuhl	[F]	hmm_h	hmm_a6	hmm_S	hmm_t	hmm_u		
fahrstuhl	[F]	hmm_h	hmm_a6	hmm_S	hmm_t	hmm_U		
fahrstuhl	[F]	hmm_h	hmm_a6	hmm_S	hmm_t	hmm_u	hmm_Q	
fahrstuhl	[F]	hmm_h	hmm_a6	hmm_S	hmm_t	hmm_U	hmm_Q	
fahrstuhl	[F]	hmm_h	hmm_a6	hmm_s	hmm_t	hmm_u	hmm_l	
fahrstuhl	[F]	hmm_h	hmm_a6	hmm_s	hmm_t	hmm_U	hmm_l	
fahrstuhl	[F]	hmm_h	hmm_a6	hmm_s	hmm_t	hmm_u		
fahrstuhl	[F]	hmm_h	hmm_a6	hmm_s	hmm_t	hmm_U		
fahrstuhl	[F]	hmm_h	hmm_a6	hmm_s	hmm_t	hmm_u	hmm_Q	
fahrstuhl	[F]	hmm_h	hmm_a6	hmm_s	hmm_t	hmm_U	hmm_Q	
fahrstuhl	[F]	hmm_h	hmm_a6	hmm_C	hmm_t	hmm_u	hmm_l	
fahrstuhl	[F]	hmm_h	hmm_a6	hmm_C	hmm_t	hmm_U	hmm_l	
fahrstuhl	[F]	hmm_h	hmm_a6	hmm_C	hmm_t	hmm_u		
fahrstuhl	[F]	hmm_h	hmm_a6	hmm_C	hmm_t	hmm_U		
fahrstuhl	[F]	hmm_h	hmm_a6	hmm_C	hmm_t	hmm_u	hmm_Q	
fahrstuhl	[F]	hmm_h	hmm_a6	hmm_C	hmm_t	hmm_U	hmm_Q	
#"Aufzug"								
aufzug	[F]	hmm_Q	hmm_aU	hmm_f	hmm_t	hmm_s	hmm_u	hmm_k
aufzug	[F]	hmm_aU	hmm_f	hmm_t	hmm_s	hmm_u	hmm_k	
aufzug	[F]	hmm_Q	hmm_aU	hmm_f	hmm_t	hmm_s	hmm_U	hmm_k
aufzug	[F]	hmm_aU	hmm_f	hmm_t	hmm_s	hmm_U	hmm_k	
aufzug	[F]	hmm_Q	hmm_aU	hmm_f	hmm_s	hmm_u	hmm_k	
aufzug	[F]	hmm_aU	hmm_f	hmm_s	hmm_u	hmm_k		
aufzug	[F]	hmm_Q	hmm_aU	hmm_f	hmm_s	hmm_U	hmm_k	
aufzug	[F]	hmm_aU	hmm_f	hmm_s	hmm_U	hmm_k		
aufzug	[F]	hmm_Q	hmm_aU	hmm_f	hmm_t	hmm_s	hmm_u	hmm_x
aufzug	[F]	hmm_aU	hmm_f	hmm_t	hmm_s	hmm_u	hmm_x	

aufzug	[F]	hmm_Q hmm_aU hmm_f hmm_t hmm_s hmm_U hmm_x
aufzug	[F]	hmm_aU hmm_f hmm_t hmm_s hmm_U hmm_x
aufzug	[F]	hmm_Q hmm_aU hmm_f hmm_s hmm_u hmm_x
aufzug	[F]	hmm_aU hmm_f hmm_s hmm_u hmm_x
aufzug	[F]	hmm_Q hmm_aU hmm_f hmm_s hmm_U hmm_x
aufzug	[F]	hmm_aU hmm_f hmm_s hmm_U hmm_x

#"Keller"

keller	[0]	hmm_k hmm_E hmm_l hmm_6
keller	[0]	hmm_k hmm_E hmm_l hmm_@
keller	[0]	hmm_k hmm_@ hmm_l hmm_6
keller	[0]	hmm_k hmm_@ hmm_l hmm_@

#"Untergeschoss"

UG	[0]	hmm_Q hmm_U hmm_n hmm_t hmm_6 hmm_g hmm_@ hmm_S hmm_O hmm_s
UG	[0]	hmm_U hmm_n hmm_t hmm_6 hmm_g hmm_@ hmm_S hmm_O hmm_s
UG	[0]	hmm_Q hmm_U hmm_n hmm_t hmm_@ hmm_g hmm_@ hmm_S hmm_O hmm_s
UG	[0]	hmm_U hmm_n hmm_t hmm_@ hmm_g hmm_@ hmm_S hmm_O hmm_s
UG	[0]	hmm_Q hmm_U hmm_n hmm_t hmm_6 hmm_g hmm_@ hmm_S hmm_@ hmm_s
UG	[0]	hmm_U hmm_n hmm_t hmm_6 hmm_g hmm_@ hmm_S hmm_@ hmm_s
UG	[0]	hmm_Q hmm_U hmm_n hmm_t hmm_@ hmm_g hmm_@ hmm_S hmm_@ hmm_s
UG	[0]	hmm_U hmm_n hmm_t hmm_@ hmm_g hmm_@ hmm_S hmm_@ hmm_s

#"CIP-Raum"

CIP	[0]	hmm_t hmm_s hmm_I hmm_p hmm_r hmm_aU hmm_m
CIP	[0]	hmm_s hmm_I hmm_p hmm_r hmm_aU hmm_m

#"Systemgruppe"

SYS	[0]	hmm_z hmm_Y hmm_s hmm_t hmm_e hmm_m hmm_g hmm_r hmm_U hmm_p hmm_@
SYS	[0]	hmm_z hmm_s hmm_t hmm_e hmm_m hmm_g hmm_r hmm_U hmm_p hmm_@
SYS	[0]	hmm_s hmm_Y hmm_s hmm_t hmm_e hmm_m hmm_g hmm_r hmm_U hmm_p hmm_@
SYS	[0]	hmm_s hmm_t hmm_e hmm_m hmm_g hmm_r hmm_U hmm_p hmm_@

#"Eins"

1	[1]	hmm_Q hmm_aI hmm_n hmm_s
1	[1]	hmm_aI hmm_n hmm_s
1	[1]	hmm_Q hmm_aI hmm_n hmm_t hmm_s
1	[1]	hmm_aI hmm_n hmm_t hmm_s

#"Erst(er)"

1	[1]	hmm_Q hmm_E6 hmm_s hmm_t
1	[1]	hmm_Q hmm_E6 hmm_s hmm_t hmm_@
1	[1]	hmm_Q hmm_E6 hmm_s hmm_t hmm_6
1	[1]	hmm_Q hmm_E6 hmm_s hmm_t hmm_@ hmm_n

```

1          [1]      hmm_Q hmm_E6 hmm_s hmm_t hmm_n
1          [1]      hmm_Q hmm_E6 hmm_s hmm_t hmm_@ hmm_s
#"Erst(er)"
1          [1]      hmm_E6 hmm_s hmm_t
1          [1]      hmm_E6 hmm_s hmm_t hmm_@
1          [1]      hmm_E6 hmm_s hmm_t hmm_6
1          [1]      hmm_E6 hmm_s hmm_t hmm_@ hmm_n
1          [1]      hmm_E6 hmm_s hmm_t hmm_n
1          [1]      hmm_E6 hmm_s hmm_t hmm_@ hmm_s
#"Erss(er)"
1          [1]      hmm_Q hmm_E6 hmm_s
1          [1]      hmm_Q hmm_E6 hmm_s hmm_@
1          [1]      hmm_Q hmm_E6 hmm_s hmm_6
1          [1]      hmm_Q hmm_E6 hmm_s hmm_@ hmm_n
1          [1]      hmm_Q hmm_E6 hmm_s hmm_n
1          [1]      hmm_Q hmm_E6 hmm_s hmm_@ hmm_s
#"Erss(er)"
1          [1]      hmm_E6 hmm_s
1          [1]      hmm_E6 hmm_s hmm_@
1          [1]      hmm_E6 hmm_s hmm_6
1          [1]      hmm_E6 hmm_s hmm_@ hmm_n
1          [1]      hmm_E6 hmm_s hmm_n
1          [1]      hmm_E6 hmm_s hmm_@ hmm_s
#"Est(er)"
1          [1]      hmm_Q hmm_E hmm_s hmm_t
1          [1]      hmm_Q hmm_E hmm_s hmm_t hmm_@
1          [1]      hmm_Q hmm_E hmm_s hmm_t hmm_6
1          [1]      hmm_Q hmm_E hmm_s hmm_t hmm_@ hmm_n
1          [1]      hmm_Q hmm_E hmm_s hmm_t hmm_n
1          [1]      hmm_Q hmm_E hmm_s hmm_t hmm_@ hmm_s
#"Est(er)"
1          [1]      hmm_E hmm_s hmm_t
1          [1]      hmm_E hmm_s hmm_t hmm_@
1          [1]      hmm_E hmm_s hmm_t hmm_6
1          [1]      hmm_E hmm_s hmm_t hmm_@ hmm_n
1          [1]      hmm_E hmm_s hmm_t hmm_n
1          [1]      hmm_E hmm_s hmm_t hmm_@ hmm_s
#"Ess(er)"
1          [1]      hmm_Q hmm_E hmm_s
1          [1]      hmm_Q hmm_E hmm_s hmm_@
1          [1]      hmm_Q hmm_E hmm_s hmm_6
1          [1]      hmm_Q hmm_E hmm_s hmm_@ hmm_n
1          [1]      hmm_Q hmm_E hmm_s hmm_n
1          [1]      hmm_Q hmm_E hmm_s hmm_@ hmm_s
#"Ess(er)"

```

```

1          [1]      hmm_E hmm_s
1          [1]      hmm_E hmm_s hmm_@
1          [1]      hmm_E hmm_s hmm_6
1          [1]      hmm_E hmm_s hmm_@ hmm_n
1          [1]      hmm_E hmm_s hmm_n
1          [1]      hmm_E hmm_s hmm_@ hmm_s
#"Derst(er)"
1          [1]      hmm_Q hmm_96 hmm_s hmm_t
1          [1]      hmm_Q hmm_96 hmm_s hmm_t hmm_@
1          [1]      hmm_Q hmm_96 hmm_s hmm_t hmm_6
1          [1]      hmm_Q hmm_96 hmm_s hmm_t hmm_@ hmm_n
1          [1]      hmm_Q hmm_96 hmm_s hmm_t hmm_n
1          [1]      hmm_Q hmm_96 hmm_s hmm_t hmm_@ hmm_s
#"Derst(er)"
1          [1]      hmm_96 hmm_s hmm_t
1          [1]      hmm_96 hmm_s hmm_t hmm_@
1          [1]      hmm_96 hmm_s hmm_t hmm_6
1          [1]      hmm_96 hmm_s hmm_t hmm_@ hmm_n
1          [1]      hmm_96 hmm_s hmm_t hmm_n
1          [1]      hmm_96 hmm_s hmm_t hmm_@ hmm_s
#"Derss(er)"
1          [1]      hmm_Q hmm_96 hmm_s
1          [1]      hmm_Q hmm_96 hmm_s hmm_@
1          [1]      hmm_Q hmm_96 hmm_s hmm_6
1          [1]      hmm_Q hmm_96 hmm_s hmm_@ hmm_n
1          [1]      hmm_Q hmm_96 hmm_s hmm_n
1          [1]      hmm_Q hmm_96 hmm_s hmm_@ hmm_s
#"Derss(er)"
1          [1]      hmm_96 hmm_s
1          [1]      hmm_96 hmm_s hmm_@
1          [1]      hmm_96 hmm_s hmm_6
1          [1]      hmm_96 hmm_s hmm_@ hmm_n
1          [1]      hmm_96 hmm_s hmm_n
1          [1]      hmm_96 hmm_s hmm_@ hmm_s
#"Desst(er)"#
1          [1]      hmm_Q hmm_9 hmm_s hmm_t
1          [1]      hmm_Q hmm_9 hmm_s hmm_t hmm_@
1          [1]      hmm_Q hmm_9 hmm_s hmm_t hmm_6
1          [1]      hmm_Q hmm_9 hmm_s hmm_t hmm_@ hmm_n
1          [1]      hmm_Q hmm_9 hmm_s hmm_t hmm_n
1          [1]      hmm_Q hmm_9 hmm_s hmm_t hmm_@ hmm_s
#"Desst(er)"#
1          [1]      hmm_9 hmm_s hmm_t
1          [1]      hmm_9 hmm_s hmm_t hmm_@
1          [1]      hmm_9 hmm_s hmm_t hmm_6

```

```

1          [1]      hmm_9 hmm_s hmm_t hmm_@ hmm_n
1          [1]      hmm_9 hmm_s hmm_t hmm_n
1          [1]      hmm_9 hmm_s hmm_t hmm_@ hmm_s
#"Oess(er)"#
1          [1]      hmm_Q hmm_9 hmm_s
1          [1]      hmm_Q hmm_9 hmm_s hmm_@
1          [1]      hmm_Q hmm_9 hmm_s hmm_6
1          [1]      hmm_Q hmm_9 hmm_s hmm_@ hmm_n
1          [1]      hmm_Q hmm_9 hmm_s hmm_n
1          [1]      hmm_Q hmm_9 hmm_s hmm_@ hmm_s
#"Oess(er)"#
1          [1]      hmm_9 hmm_s
1          [1]      hmm_9 hmm_s hmm_@
1          [1]      hmm_9 hmm_s hmm_6
1          [1]      hmm_9 hmm_s hmm_@ hmm_n
1          [1]      hmm_9 hmm_s hmm_n
1          [1]      hmm_9 hmm_s hmm_@ hmm_s

#"Erdgeschoss"
EG         [1]      hmm_Q hmm_e6 hmm_t hmm_g hmm_@ hmm_S hmm_O hmm_s
EG         [1]      hmm_e6 hmm_t hmm_g hmm_@ hmm_S hmm_O hmm_s
EG         [1]      hmm_Q hmm_E6 hmm_t hmm_g hmm_@ hmm_S hmm_O hmm_s
EG         [1]      hmm_E6 hmm_t hmm_g hmm_@ hmm_S hmm_O hmm_s
EG         [1]      hmm_Q hmm_E hmm_t hmm_g hmm_@ hmm_S hmm_O hmm_s
EG         [1]      hmm_E hmm_t hmm_g hmm_@ hmm_S hmm_O hmm_s
EG         [1]      hmm_Q hmm_e6 hmm_t hmm_g hmm_@ hmm_S hmm_@ hmm_s
EG         [1]      hmm_e6 hmm_t hmm_g hmm_@ hmm_S hmm_@ hmm_s
EG         [1]      hmm_Q hmm_E6 hmm_t hmm_g hmm_@ hmm_S hmm_@ hmm_s
EG         [1]      hmm_E6 hmm_t hmm_g hmm_@ hmm_S hmm_@ hmm_s
EG         [1]      hmm_Q hmm_E hmm_t hmm_g hmm_@ hmm_S hmm_@ hmm_s
EG         [1]      hmm_E hmm_t hmm_g hmm_@ hmm_S hmm_@ hmm_s

#"Pinkal"
pinkal     [1]      hmm_p hmm_I hmm_N hmm_k hmm_a hmm_l
pinkal     [1]      hmm_p hmm_I hmm_N hmm_k hmm_@  hmm_l
pinkal     [1]      hmm_p hmm_I hmm_N hmm_k hmm_A hmm_l
pinkal     [1]      hmm_p hmm_I hmm_N hmm_k hmm_l
pinkal     [1]      hmm_p hmm_i hmm_N hmm_k hmm_a hmm_l
pinkal     [1]      hmm_p hmm_i hmm_N hmm_k hmm_@ hmm_l
pinkal     [1]      hmm_p hmm_i hmm_N hmm_k hmm_A hmm_l
pinkal     [1]      hmm_p hmm_i hmm_N hmm_k hmm_l

2          [2]      hmm_t hmm_s hmm_v hmm_aI
2          [2]      hmm_s hmm_v hmm_aI
2          [2]      hmm_t hmm_s hmm_v hmm_aI hmm_t

```

2 [2] hmm_s hmm_v hmm_aI hmm_t
 2 [2] hmm_t hmm_s hmm_v hmm_aI hmm_t hmm_@
 2 [2] hmm_s hmm_v hmm_aI hmm_t hmm_@
 2 [2] hmm_t hmm_s hmm_v hmm_aI hmm_t hmm_6
 2 [2] hmm_s hmm_v hmm_aI hmm_t hmm_6
 2 [2] hmm_t hmm_s hmm_v hmm_aI hmm_t hmm_@ hmm_n
 2 [2] hmm_s hmm_v hmm_aI hmm_t hmm_@ hmm_n
 2 [2] hmm_t hmm_s hmm_v hmm_aI hmm_t hmm_@ hmm_s
 2 [2] hmm_s hmm_v hmm_aI hmm_t hmm_@ hmm_s
 2 [2] hmm_t hmm_s hmm_v hmm_aI hmm_Q hmm_@ hmm_n
 2 [2] hmm_s hmm_v hmm_aI hmm_Q hmm_@ hmm_n
 2 [2] hmm_t hmm_s hmm_v hmm_aI hmm_Q hmm_n
 2 [2] hmm_s hmm_v hmm_aI hmm_Q hmm_n

##Konferenzraum"

KR [2] hmm_k hmm_0 hmm_n hmm_f hmm_e hmm_r hmm_E hmm_n hmm_t hmm_s \
 hmm_r hmm_aU hmm_m
 KR [2] hmm_k hmm_0 hmm_n hmm_f hmm_@ hmm_r hmm_E hmm_n hmm_t hmm_s \
 hmm_r hmm_aU hmm_m
 KR [2] hmm_k hmm_0 hmm_n hmm_f hmm_r hmm_E hmm_n hmm_t hmm_s hmm_r \
 hmm_aU hmm_m
 KR [2] hmm_k hmm_0 hmm_n hmm_f hmm_6 hmm_r hmm_E hmm_n hmm_t hmm_s \
 hmm_r hmm_aU hmm_m

KR [2] hmm_k hmm_0 hmm_n hmm_f hmm_e hmm_r hmm_E hmm_n hmm_t hmm_s \
 hmm_r hmm_aU
 KR [2] hmm_k hmm_0 hmm_n hmm_f hmm_@ hmm_r hmm_E hmm_n hmm_t hmm_s \
 hmm_r hmm_aU
 KR [2] hmm_k hmm_0 hmm_n hmm_f hmm_r hmm_E hmm_n hmm_t hmm_s \
 hmm_r hmm_aU
 KR [2] hmm_k hmm_0 hmm_n hmm_f hmm_6 hmm_r hmm_E hmm_n hmm_t hmm_s \
 hmm_r hmm_aU

KR [2] hmm_k hmm_0 hmm_m hmm_f hmm_e hmm_r hmm_E hmm_n hmm_t hmm_s \
 hmm_r hmm_aU hmm_m
 KR [2] hmm_k hmm_0 hmm_m hmm_f hmm_@ hmm_r hmm_E hmm_n hmm_t hmm_s \
 hmm_r hmm_aU hmm_m
 KR [2] hmm_k hmm_0 hmm_m hmm_f hmm_r hmm_E hmm_n hmm_t hmm_s hmm_r \
 hmm_aU hmm_m
 KR [2] hmm_k hmm_0 hmm_m hmm_f hmm_6 hmm_r hmm_E hmm_n hmm_t hmm_s \
 hmm_r hmm_aU hmm_m

KR [2] hmm_k hmm_0 hmm_m hmm_f hmm_e hmm_r hmm_E hmm_n hmm_t hmm_s \
 hmm_r hmm_aU
 KR [2] hmm_k hmm_0 hmm_m hmm_f hmm_@ hmm_r hmm_E hmm_n hmm_t hmm_s \

```

hmm_r hmm_aU
KR [2]      hmm_k hmm_O hmm_m hmm_f hmm_r hmm_E hmm_n hmm_t hmm_s hmm_r \
hmm_aU
KR [2]      hmm_k hmm_O hmm_m hmm_f hmm_6 hmm_r hmm_E hmm_n hmm_t hmm_s \
hmm_r hmm_aU

3 [3]      hmm_d hmm_r hmm_aI
3 [3]      hmm_d hmm_x hmm_aI
3 [3]      hmm_t hmm_r hmm_aI
3 [3]      hmm_t hmm_x hmm_aI

3 [3]      hmm_d hmm_r hmm_I hmm_t
3 [3]      hmm_d hmm_r hmm_I hmm_t hmm_@
3 [3]      hmm_d hmm_r hmm_I hmm_t hmm_6
3 [3]      hmm_d hmm_r hmm_I hmm_t hmm_@ hmm_n
3 [3]      hmm_d hmm_r hmm_I hmm_t hmm_n
3 [3]      hmm_d hmm_r hmm_I hmm_t hmm_@ hmm_s
3 [3]      hmm_d hmm_r hmm_I hmm_Q hmm_n
3 [3]      hmm_d hmm_r hmm_I hmm_Q hmm_@ hmm_n

3 [3]      hmm_d hmm_x hmm_I hmm_t
3 [3]      hmm_d hmm_x hmm_I hmm_t hmm_@
3 [3]      hmm_d hmm_x hmm_I hmm_t hmm_6
3 [3]      hmm_d hmm_x hmm_I hmm_t hmm_@ hmm_n
3 [3]      hmm_d hmm_x hmm_I hmm_t hmm_n
3 [3]      hmm_d hmm_x hmm_I hmm_t hmm_@ hmm_s
3 [3]      hmm_d hmm_x hmm_I hmm_Q hmm_n
3 [3]      hmm_d hmm_x hmm_I hmm_Q hmm_@ hmm_n

3 [3]      hmm_t hmm_r hmm_I hmm_t
3 [3]      hmm_t hmm_r hmm_I hmm_t hmm_@
3 [3]      hmm_t hmm_r hmm_I hmm_t hmm_6
3 [3]      hmm_t hmm_r hmm_I hmm_t hmm_@ hmm_n
3 [3]      hmm_t hmm_r hmm_I hmm_t hmm_n
3 [3]      hmm_t hmm_r hmm_I hmm_t hmm_@ hmm_s
3 [3]      hmm_t hmm_r hmm_I hmm_Q hmm_n
3 [3]      hmm_t hmm_r hmm_I hmm_Q hmm_@ hmm_n

3 [3]      hmm_t hmm_x hmm_I hmm_t
3 [3]      hmm_t hmm_x hmm_I hmm_t hmm_@
3 [3]      hmm_t hmm_x hmm_I hmm_t hmm_6
3 [3]      hmm_t hmm_x hmm_I hmm_t hmm_@ hmm_n
3 [3]      hmm_t hmm_x hmm_I hmm_t hmm_n
3 [3]      hmm_t hmm_x hmm_I hmm_t hmm_@ hmm_s
3 [3]      hmm_t hmm_x hmm_I hmm_Q hmm_n

```

```

3          [3]      hmm_t hmm_x hmm_I hmm_Q hmm_@ hmm_n

#"Uszkoreit"
uszkoreit  [3]      hmm_Q hmm_U hmm_s hmm_k hmm_o hmm_r hmm_aI hmm_t
uszkoreit  [3]      hmm_U hmm_s hmm_k hmm_o hmm_r hmm_aI hmm_t
uszkoreit  [3]      hmm_Q hmm_U hmm_S hmm_k hmm_o hmm_r hmm_aI hmm_t
uszkoreit  [3]      hmm_U hmm_S hmm_k hmm_o hmm_r hmm_aI hmm_t
uszkoreit  [3]      hmm_Q hmm_U hmm_s hmm_k hmm_o hmm_x hmm_aI hmm_t
uszkoreit  [3]      hmm_U hmm_s hmm_k hmm_o hmm_x hmm_aI hmm_t
uszkoreit  [3]      hmm_Q hmm_U hmm_S hmm_k hmm_o hmm_x hmm_aI hmm_t
uszkoreit  [3]      hmm_U hmm_S hmm_k hmm_o hmm_x hmm_aI hmm_t

4          [4]      hmm_f hmm_i6
4          [4]      hmm_f hmm_i6 hmm_t
4          [4]      hmm_f hmm_i6 hmm_t hmm_@
4          [4]      hmm_f hmm_i6 hmm_t hmm_6
4          [4]      hmm_f hmm_i6 hmm_t hmm_@ hmm_n
4          [4]      hmm_f hmm_i6 hmm_t hmm_n
4          [4]      hmm_f hmm_i6 hmm_t hmm_@ hmm_s
4          [4]      hmm_f hmm_i6 hmm_Q hmm_n
4          [4]      hmm_f hmm_i6 hmm_Q hmm_@ hmm_n

4          [4]      hmm_f hmm_I6
4          [4]      hmm_f hmm_I6 hmm_t
4          [4]      hmm_f hmm_I6 hmm_t hmm_@
4          [4]      hmm_f hmm_I6 hmm_t hmm_6
4          [4]      hmm_f hmm_I6 hmm_t hmm_@ hmm_n
4          [4]      hmm_f hmm_I6 hmm_t hmm_n
4          [4]      hmm_f hmm_I6 hmm_t hmm_@ hmm_s
4          [4]      hmm_f hmm_I6 hmm_Q hmm_n
4          [4]      hmm_f hmm_I6 hmm_Q hmm_@ hmm_n

4          [4]      hmm_f hmm_Y hmm_r hmm_t
4          [4]      hmm_f hmm_Y hmm_r hmm_t hmm_@
4          [4]      hmm_f hmm_Y hmm_r hmm_t hmm_6
4          [4]      hmm_f hmm_Y hmm_t hmm_@ hmm_n
4          [4]      hmm_f hmm_Y hmm_t hmm_n
4          [4]      hmm_f hmm_Y hmm_t hmm_@ hmm_s
4          [4]      hmm_f hmm_Y hmm_Q hmm_n
4          [4]      hmm_f hmm_Y hmm_Q hmm_@ hmm_n

4          [4]      hmm_f hmm_Y6 hmm_t
4          [4]      hmm_f hmm_Y6 hmm_t hmm_@
4          [4]      hmm_f hmm_Y6 hmm_t hmm_6
4          [4]      hmm_f hmm_Y6 hmm_t hmm_@ hmm_n

```

```

4          [4]      hmm_f hmm_Y6 hmm_t hmm_n
4          [4]      hmm_f hmm_Y6 hmm_t hmm_@ hmm_s
4          [4]      hmm_f hmm_Y6 hmm_Q hmm_n
4          [4]      hmm_f hmm_Y6 hmm_Q hmm_@ hmm_n

```

```

#"Barry"

```

```

barry      [4]      hmm_b hmm_E hmm_r hmm_i
barry      [4]      hmm_b hmm_E hmm_r hmm_I
barry      [4]      hmm_b hmm_E hmm_v hmm_i
barry      [4]      hmm_b hmm_E hmm_v hmm_I
barry      [4]      hmm_b hmm_E hmm_U hmm_i
barry      [4]      hmm_b hmm_E hmm_U hmm_I
barry      [4]      hmm_b hmm_A hmm_r hmm_i
barry      [4]      hmm_b hmm_A hmm_r hmm_I

```

```

5          [5]      hmm_f hmm_Y hmm_n hmm_f
5          [5]      hmm_f hmm_Y hmm_m hmm_f
5          [5]      hmm_f hmm_Y hmm_n hmm_f hmm_t
5          [5]      hmm_f hmm_Y hmm_m hmm_f hmm_t
5          [5]      hmm_f hmm_Y hmm_n hmm_f hmm_t hmm_@
5          [5]      hmm_f hmm_Y hmm_m hmm_f hmm_t hmm_@
5          [5]      hmm_f hmm_Y hmm_n hmm_f hmm_t hmm_6
5          [5]      hmm_f hmm_Y hmm_m hmm_f hmm_t hmm_6
5          [5]      hmm_f hmm_Y hmm_n hmm_f hmm_t hmm_@ hmm_n
5          [5]      hmm_f hmm_Y hmm_m hmm_f hmm_t hmm_@ hmm_n
5          [5]      hmm_f hmm_Y hmm_n hmm_f hmm_t hmm_n
5          [5]      hmm_f hmm_Y hmm_m hmm_f hmm_t hmm_n
5          [5]      hmm_f hmm_Y hmm_n hmm_f hmm_t hmm_@ hmm_s
5          [5]      hmm_f hmm_Y hmm_m hmm_f hmm_t hmm_@ hmm_s
5          [5]      hmm_f hmm_Y hmm_n hmm_f hmm_t hmm_@ hmm_n
5          [5]      hmm_f hmm_Y hmm_m hmm_f hmm_t hmm_@ hmm_n
5          [5]      hmm_f hmm_Y hmm_n hmm_f hmm_t hmm_@ hmm_n
5          [5]      hmm_f hmm_Y hmm_m hmm_f hmm_t hmm_@ hmm_n

```

```

5          [5]      hmm_f hmm_y hmm_n hmm_f
5          [5]      hmm_f hmm_y hmm_m hmm_f
5          [5]      hmm_f hmm_y hmm_n hmm_f hmm_t
5          [5]      hmm_f hmm_y hmm_m hmm_f hmm_t
5          [5]      hmm_f hmm_y hmm_n hmm_f hmm_t hmm_@
5          [5]      hmm_f hmm_y hmm_m hmm_f hmm_t hmm_@
5          [5]      hmm_f hmm_y hmm_n hmm_f hmm_t hmm_6
5          [5]      hmm_f hmm_y hmm_m hmm_f hmm_t hmm_6
5          [5]      hmm_f hmm_y hmm_n hmm_f hmm_t hmm_@ hmm_n
5          [5]      hmm_f hmm_y hmm_m hmm_f hmm_t hmm_@ hmm_n

```

```

5          [5]      hmm_f hmm_y hmm_n hmm_f hmm_t hmm_n
5          [5]      hmm_f hmm_y hmm_m hmm_f hmm_t hmm_n
5          [5]      hmm_f hmm_y hmm_n hmm_f hmm_t hmm_@ hmm_s
5          [5]      hmm_f hmm_y hmm_m hmm_f hmm_t hmm_@ hmm_s
5          [5]      hmm_f hmm_y hmm_n hmm_f hmm_t hmm_@ hmm_n
5          [5]      hmm_f hmm_y hmm_m hmm_f hmm_t hmm_@ hmm_n
5          [5]      hmm_f hmm_y hmm_n hmm_f hmm_t hmm_@ hmm_n
5          [5]      hmm_f hmm_y hmm_m hmm_f hmm_t hmm_@ hmm_n

#"Stockwerk"
stock      []      hmm_S hmm_t hmm_O hmm_k hmm_v hmm_E6 hmm_k
stock      []      hmm_S hmm_t hmm_O hmm_k hmm_v hmm_6  hmm_k
stock      []      hmm_S hmm_t hmm_O hmm_k hmm_v hmm_E6 hmm_Q
stock      []      hmm_S hmm_t hmm_O hmm_k hmm_v hmm_6  hmm_Q

stock      []      hmm_C hmm_t hmm_O hmm_k hmm_v hmm_E6 hmm_k
stock      []      hmm_C hmm_t hmm_O hmm_k hmm_v hmm_6  hmm_k
stock      []      hmm_C hmm_t hmm_O hmm_k hmm_v hmm_E6 hmm_Q
stock      []      hmm_C hmm_t hmm_O hmm_k hmm_v hmm_6  hmm_Q

#"Stock"
stock      []      hmm_S hmm_t hmm_O hmm_k
stock      []      hmm_C hmm_t hmm_O hmm_k
stock      []      hmm_S hmm_t hmm_O hmm_Q
stock      []      hmm_C hmm_t hmm_O hmm_Q

#"Etage"
etage      []      hmm_Q hmm_e hmm_t hmm_a hmm_Z hmm_@
etage      []      hmm_e hmm_t hmm_a hmm_Z hmm_@
etage      []      hmm_Q hmm_@ hmm_t hmm_a hmm_Z hmm_@
etage      []      hmm_@ hmm_t hmm_a hmm_Z hmm_@
etage      []      hmm_Q hmm_E hmm_t hmm_a hmm_Z hmm_@
etage      []      hmm_E hmm_t hmm_a hmm_Z hmm_@

etage      []      hmm_Q hmm_e hmm_t hmm_a hmm_S hmm_@
etage      []      hmm_e hmm_t hmm_a hmm_S hmm_@
etage      []      hmm_Q hmm_@ hmm_t hmm_a hmm_S hmm_@
etage      []      hmm_@ hmm_t hmm_a hmm_S hmm_@
etage      []      hmm_Q hmm_E hmm_t hmm_a hmm_S hmm_@
etage      []      hmm_E hmm_t hmm_a hmm_S hmm_@

#"Professor"
professor  []      hmm_p hmm_r hmm_o hmm_f hmm_E hmm_s hmm_06
professor  []      hmm_p hmm_r hmm_@ hmm_f hmm_E hmm_s hmm_06
professor  []      hmm_p hmm_r hmm_O hmm_f hmm_E hmm_s hmm_06

```

professor	[]	hmm_p hmm_r hmm_o hmm_f hmm_E hmm_s hmm_6
professor	[]	hmm_p hmm_r hmm_@ hmm_f hmm_E hmm_s hmm_6
professor	[]	hmm_p hmm_r hmm_0 hmm_f hmm_E hmm_s hmm_6
#"Herrn"		
herrn	[]	hmm_h hmm_E6 hmm_n
herrn	[]	hmm_h hmm_E hmm_n
#"Bitte"		
bitte	[]	hmm_b hmm_I hmm_t hmm_@
bitte	[]	hmm_b hmm_I hmm_d hmm_@
#"moecht(e)"		
moechte	[]	hmm_m hmm_9 hmm_C hmm_t
moechte	[]	hmm_m hmm_9 hmm_C hmm_t hmm_@
#"gern(e)"		
gerne	[]	hmm_g hmm_E6 hmm_n hmm_@
gerne	[]	hmm_g hmm_E6 hmm_n
gerne	[]	hmm_k hmm_E6 hmm_n hmm_@
gerne	[]	hmm_k hmm_E6 hmm_n
#"ich"		
ich	[]	hmm_Q hmm_I hmm_C
ich	[]	hmm_Q hmm_I hmm_S
ich	[]	hmm_I hmm_C
ich	[]	hmm_I hmm_S
#"wir"		
wir	[]	hmm_v hmm_i6
wir	[]	hmm_v hmm_I6
#"in"		
in	[]	hmm_Q hmm_I hmm_n
in	[]	hmm_Q hmm_I hmm_n hmm_@ hmm_n
in	[]	hmm_Q hmm_I hmm_n hmm_n
in	[]	hmm_I hmm_n
inen	[]	hmm_I hmm_n hmm_@ hmm_n
inn	[]	hmm_I hmm_n hmm_n
ins	[]	hmm_I hmm_n hmm_s
#"den"		
den	[]	hmm_d hmm_e hmm_n
den	[]	hmm_d hmm_@ hmm_n
den	[]	hmm_d hmm_@

#"die"

die	[]	hmm_d hmm_i
die	[]	hmm_d hmm_I

zu	[]	hmm_t hmm_s hmm_u
zu	[]	hmm_t hmm_s hmm_U
zu	[]	hmm_t hmm_s hmm_u hmm_Q
zu	[]	hmm_t hmm_s hmm_U hmm_Q
zu	[]	hmm_t hmm_s hmm_U hmm_m
zu	[]	hmm_t hmm_s hmm_u6
zu	[]	hmm_t hmm_s hmm_U6

zu	[]	hmm_s hmm_u
zu	[]	hmm_s hmm_U
zu	[]	hmm_s hmm_u hmm_Q
zu	[]	hmm_s hmm_U hmm_Q
zu	[]	hmm_s hmm_U hmm_m
zu	[]	hmm_s hmm_u6
zu	[]	hmm_s hmm_U6

!ANY	[]	hmm_n hmm_aI hmm_n
!ANY	[]	hmm_S hmm_t hmm_O hmm_p
!ANY	[]	hmm_f hmm_A hmm_l hmm_S
!ANY	[]	hmm_f hmm_h hmm_A hmm_l hmm_S
!ANY	[]	hmm_h hmm_A hmm_l hmm_t

#"Crocker"

crocker	[]	hmm_k hmm_r hmm_O hmm_k hmm_6
crocker	[]	hmm_k hmm_x hmm_O hmm_k hmm_6
crocker	[]	hmm_k hmm_d hmm_O hmm_k hmm_6
crocker	[]	hmm_k hmm_r hmm_O hmm_k hmm_@
crocker	[]	hmm_k hmm_x hmm_O hmm_k hmm_@
crocker	[]	hmm_k hmm_d hmm_O hmm_k hmm_@
crocker	[]	hmm_k hmm_r hmm_A hmm_k hmm_6
crocker	[]	hmm_k hmm_x hmm_A hmm_k hmm_6
crocker	[]	hmm_k hmm_d hmm_A hmm_k hmm_6
crocker	[]	hmm_k hmm_r hmm_A hmm_k hmm_@
crocker	[]	hmm_k hmm_x hmm_A hmm_k hmm_@
crocker	[]	hmm_k hmm_d hmm_A hmm_k hmm_@

#"Wahlster"

wahlster	[]	hmm_v hmm_a hmm_l hmm_s hmm_t hmm_6
----------	----	-------------------------------------

wahlster	[]	hmm_v	hmm_A	hmm_l	hmm_s	hmm_t	hmm_6
wahlster	[]	hmm_v	hmm_a	hmm_l	hmm_s	hmm_t	hmm_@
wahlster	[]	hmm_v	hmm_A	hmm_l	hmm_s	hmm_t	hmm_@
wahlster	[]	hmm_v	hmm_a	hmm_l	hmm_t	hmm_s	hmm_t
wahlster	[]	hmm_v	hmm_A	hmm_l	hmm_t	hmm_s	hmm_t
wahlster	[]	hmm_v	hmm_a	hmm_l	hmm_t	hmm_s	hmm_t
wahlster	[]	hmm_v	hmm_A	hmm_l	hmm_t	hmm_s	hmm_t

F. Die Grammatik

```
$ziel1 = 1 | 2 | 3 | 4 | 5;
$ziel2 = keller | UG | CIP | SYS | EG | KR;
$prof = [herrn] [professor] (pinkal|uszkoreit|barry|wahlster|crocker);
$etage = stock|etage;
$in = (in [den|die])|ins;
$moechte = [ich|wir] {!SIL|!ANY} [moechte] {!SIL|!ANY} [gerne] {!SIL|!ANY};
$ziel = ($ziel1 [$etage]) | $ziel2;

(
  {!SIL|!ANY}
  [
    ( (fahrstuhl | aufzug ) [bitte] ) | ( $ziel [bitte] ) |
    ( [fahrstuhl | aufzug] {!SIL|!ANY} [$moechte] ( ([ $in] $ziel)|([zu] $prof) ) [bitte] )
  ]
  {!SIL|!ANY}
)
```

G. Quelltexte für Erweiterungen des Soundtreibers (Eric)

Dieser Teil des Anhangs enthält die Quelltexte zu meinen Hilfsprogrammen zur Verwendung der Midiman Delta 1010 unter Linux. Die Programme werden im Kapitel über Soundkarten genauer beschrieben, und zwar speziell im Abschnitt über Treiber für die Übertragung von Audiodaten im Netz (7.4.3) .

G.1. TCP/IP Server für Midiman Delta 1010

Dieses Programm stellt die Kanäle der Midiman Soundkarte via TCP/IP zur Verfügung.

Bisher kann der Server nur Audiodaten von der Soundkarte ins Netz schicken, der umgekehrte Weg sollte noch eingebaut werden (siehe mein Bericht). Ausserdem ist es sinnvoll, dem Server per Netz einen Befehl zum „weghören“ geben zu können. Auch das wird in meinem Bericht im Abschnitt über Treiber für Audiodaten im Netzwerk (7.4.3) beschrieben. Dort finden sich auch weitere Beschreibungen der Programme in diesem Teil des Anhangs.

Zum Compilieren:

```
gcc -Wall -lasound -lpthread -g server-sockets.c rec-sock-alsa.c alsa-ice.c -o
rec-sock-alsa
```

G.1.1. rec-sock-alsa.h und rec-sock-alsa.c

Diese beiden Dateien sind der zentrale Teil des Servers.

Die Headerdatei:

```
#ifndef R_S_ALSA
#define R_S_ALSA

#define nCHN 12
// Channels, muss bei Ice1712 capture 12 sein!

#define snd_HZ 22050 /* z.B. 22050 */
/* Ice1712pro/Linux kann nur S32_LE, aber 8-48kHz Samplingrate... */

#define fmtSOCK signed short int
// Datenformat fuer die Uebertragung per Socket
```

```
#define snd_inBITS 16
/* *** Achtung, zur Zeit werden nur 16 Bit signed verwendet!!! *** */

/* ***** */

#include <sys/asoundlib.h>
#include <linux/asound.h>
#include <stdio.h>
#include <string.h> // strncpy
#include <sys/types.h>
#include <errno.h> // ...
#include <fcntl.h> // open
#include <sys/stat.h> // stat
#include <stdlib.h> // malloc
#include <unistd.h> // usleep
#include <sys/uio.h> // iovec

#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <pthread.h>

#define BACKLOG 24      /* how many pending connections queue will hold */

/* ***** */

// #define DEBUG
#define USLEEP
#define RBUF 4096 /* ReadBufferSize in Samples */

#define snd_SHL (31-snd_inBITS) /* 24 MSB of 32 used bei ice1712 */
// Ich habe ein MSB weggelassen (absichtlich)
// 386: __BYTE_ORDER == __LITTLE_ENDIAN

// ice1712pro kann nur genau 10 voices playback und 12 voices capture
// ausserdem geht im stream mode nur write/interleaved richtig!?

// Werte < 0 bzw. != 0 signalisieren Fehler.

pthread_mutex_t mutex;
#define LOCK (void)pthread_mutex_lock(&mutex)
#define UNLOCK (void)pthread_mutex_unlock(&mutex)
// evtl. Errorhandling

int run_server;
```

```

int in_buffer;

int open_sound(snd_pcm_t ** handle /* ptr auf ptr */);
int close_sound(snd_pcm_t * handle);

int open_socket(int port); /* liefert handle als return value */
int close_socket(int handle);

int send_buffer(int sockfd, int channel);
// verbindet beide Welten!

#endif

```

Die C Datei:

```

/*
 * PCM-Capture nach TCP / 16bit signed auf Ice1712 Pro mit ALSA/Linux/x86
 * Aufruf [Programm] [Portnummer]
 *
 * (Compilieren fuer Mono oder Stereo und auf eine feste Samplingrate zw.
 * 8000 und 48000Hz, hat bisher keine eigene Erkennung fuer WAV-Header!)
 * Benoetigte Libraries: asound pthread
 *
 * Frei kopierbarer haesslicher "proof of concept" Code ohne jegliche
 * Garantien, nicht mal die, die oben versprochene Funktion zu erfuellen.
 *
 * Beteiligt: Eric Auer, Attilio Erriquez, Dan Hollis, Christian Dressler
 *
 * Achtung: Der ALSA Treiber fuer diese Karte erlaubt nur, ALLE Kanaele
 * gleichzeitig zu oeffnen, daher funktioniert dafuer auch keine OSS-Emu.
 * Fuer Ausgabe sind das exakt 10, fuer Eingabe exakt 12 Monokanaele.
 *
 * 30.05.2000 - 01.06.2000
 */

#include "rec-sock-alsa.h"

/* ***** */

void help(void)
{
    printf("Usage: Give me a port and I will write,\n"
           "signed %dbit "\n"
           "mono"\n"
           " PCM %dHz data to it.\n",
           snd_inBITS,snd_HZ);
}

```



```

    _exit(1); return;
};

/* ***** */

signed int mybuf[RBUF*nCHN]; // nCHN wg nCHN voices
int want_data; // for locking

/* ***** */

int send_buffer(int sockfd, int channel)
// verbindet beide Welten!
// Aktion: fetch channel from buffer (locking), send it, recv ack (ign. ack)
{
    int cnt,cnt2,left;
    fmtSOCK mywrbuf[RBUF+1];
    char combuf[200];

    LOCK;
    want_data++;
    UNLOCK;

    while (in_buffer<=0) { /* usleep(1); */ };

    LOCK;
    if ((left=in_buffer)<=0) return 1; // Pech gehabt?
    left /= sizeof(mybuf[0]) * nCHN;
    if (left>=RBUF) return 2; // dont overflow yourself

    for (cnt=0,cnt2=channel; cnt<left; cnt++)
    {
        mywrbuf[cnt] = mybuf[cnt2] >> snd_SHL;
        cnt2+=nCHN;
    };
    if (want_data>0) want_data--;
    UNLOCK;

    fprintf(stderr,"send_buffer: %d Samples\n",left);
    if (left<=0) return 3; // should never happen

    if (send(sockfd,mywrbuf,left * sizeof(mywrbuf[0]),0) == -1)
        { perror("send"); return 4; };

    if (recv(sockfd,combuf,sizeof(combuf),0) == -1)
        { perror("recv"); return 5; };
}

```

```

/* Ack. from client is ignored here, there only has to be some message */

return 0;
};

/* ***** */

int read_sound(snd_pcm_t * handle)
{
    int left, cnt;

    if ((cnt = snd_pcm_capture_go(handle)))
        { fprintf(stderr, "Error with snd_pcm_capture_go: %s\n",
            snd_strerror(cnt)); return 1;
        };
    want_data=0;
    usleep(1000);

    while (run_server>0)
    {
        LOCK;
        in_buffer = 0;
        left = in_buffer = snd_pcm_read(handle, mybuf, sizeof(mybuf));
        UNLOCK;
        left = (left>0) ? left / (nCHN * sizeof(mybuf[0])) : left;
        if (left<=0)
            { fprintf(stderr, "Error with snd_pcm_read: %s\n", snd_strerror(left));
              left = snd_pcm_capture_go(handle);
              fprintf(stderr, "Tried to restart capture: %s\n", snd_strerror(left));
              left = 0;
              fprintf(stderr, "Trying to continue\n");
            };
        if ((left<0) && (left!= -EAGAIN))
            { fprintf(stderr, "Error with snd_pcm_read: %s\n", snd_strerror(left));
              LOCK;
              run_server = -1;
              in_buffer = 0;
              UNLOCK;
              return 1;
            };
        if (left>0)
            { // wait a bit if someone wants to copy the buffer
              // but only if there is something in it.
              usleep(1000);
              while (want_data) usleep(1000);
            }
    }
}

```

```
    };
#ifdef DEBUG
    if (left>0) { fprintf(stderr,"%d samples fetched\n",left); };
#endif
    };

LOCK;
run_server = -1;
in_buffer = 0;
UNLOCK;
return 0;
};

/* ***** */

int main(int argc, char ** argv)
{
    snd_pcm_t * soundhandle=NULL;
    int sockfd;
    long int port;

    run_server = 1;
    in_buffer = 0;

    if ((argc!=2) || (!argv)) help(); if (!argv[1]) help();

    port = strtol(argv[1],NULL,10);
    if ((port < 1024) || (port > 65519)) help();

    fprintf(stderr,"Opening sound for capture\n");
    if (open_sound(&soundhandle)) return 1;

    fprintf(stderr,"Starting to listen to port %d\n",(int)port);
    if ((sockfd = open_socket((int)port)) < 0) return 1;

    if (read_sound(soundhandle))
        { fprintf(stderr, "Error reading from PCM\n"); };
    // kehrt nur bei Fehler zurueck oder bei "CLOSE" Kommando

    fprintf(stderr,"Closing server sockets\n");
    if (close_socket(sockfd)) return 1;

    fprintf(stderr,"Closing sound\n");
    if (close_sound(soundhandle)) return 1;
```

```
return 0;
};
```

G.1.2. server-sockets.c

Hier wird das Protokoll zwischen Client und Server aufgebaut.

```
#include "rec-sock-alsa.h"

/* ***** */

/* thread for dialog with client */
void * serve_partner(void * ptr);

// global var:
int running_servers = 0;
int sockfd_server = 0;
int conn1[nCHN];
int conn2[nCHN];

/* ***** */

void * accept_conn(void * ptr)
{
    pthread_t partner;
    int new_fd;
    struct sockaddr_in their_addr;
    int HasToBeVar = sizeof(their_addr);
    int sockfd = ((int *)ptr)[0];

    fprintf(stderr,"Starting thread waiting for new connections\n");

    bzero(&their_addr,sizeof(their_addr));

    while ((run_server) && (sockfd_server > 0))
    {
        if ((new_fd = accept(sockfd, &their_addr, &HasToBeVar)) == -1)
            { perror("accept"); continue; };
        fprintf(stderr,"Starting additional server thread\n");
        pthread_create (&partner, NULL, serve_partner, (void *)&new_fd);
        fprintf(stderr,"Additional server thread started\n");
    }

    LOCK;
    run_server = 0; // please stop serving
```

```

// free some memory ... do other stuff ...
UNLOCK;

fprintf(stderr, "Waiting for servers to stop\n");
while (running_servers > 0) usleep(1000);
close_socket(sockfd);

return ptr;
};

/* ***** */

int open_socket(int port) /* liefert ein handle zurueck */
{
    struct sockaddr_in my_addr;
    int ch;
    int sockfd;
    pthread_t acceptor;

    if ((port < 1024) || (port > 65519)) return 1;

    fprintf(stderr, "Initialising server sockets\n");

    pthread_mutex_init(&mutex, NULL);
    for (ch=0; ch<nCHN; ch++) conn2[ch] = conn1[ch] = 0;

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        { perror("socket"); return -1; }; /* open socket */

    bzero(&my_addr, sizeof(my_addr));
    my_addr.sin_family = AF_INET;
    my_addr.sin_port = htons(port); /* short int ... */
    my_addr.sin_addr.s_addr = INADDR_ANY; /* use my address */

    if (bind(sockfd, &my_addr, sizeof(my_addr)) == -1)
        { perror("bind"); return -1; }; /* bind name to socket */

    if (listen(sockfd, 24 /* backlog queue size */) == -1)
        { perror("listen"); return -1; }; /* listen for connections */

    sockfd_server = sockfd;
    fprintf(stderr, "Starting server thread\n");
    pthread_create(&acceptor, NULL, accept_conn, (void *)&sockfd_server);
    fprintf(stderr, "Server thread started\n");

```

```

    return sockfd;
};

/* ***** */

int close_socket(int handle)
{
    fprintf(stderr,"closing down server\n");
    close(handle);
    return 0;
};

/* ***** */

void * serve_partner(void * ptr)
{
    int my_fd = ((int *) ptr)[0];
    int len, retval, channel=-1;
    char combuf[200];

// ->
#define WIMP(n) LOCK;  running_servers--; \
                if (channel>=0) \
                { \
conn1[channel] = (conn1[channel] == my_fd) ? 0 : conn1[channel]; \
conn2[channel] = (conn2[channel] == my_fd) ? 0 : conn2[channel]; \
                }; \
                UNLOCK; close(my_fd); \
                fprintf(stderr,"End of server thread, fd=%d\n",my_fd); \
                return ptr;

// <-
// could use n as some kind of error code...

    fprintf(stderr,"Server thread for another client started, fd=%d\n",my_fd);

    LOCK;
    running_servers++;
    UNLOCK;

    if (send(my_fd,"Hello!",strlen("Hello!")+1,0) == -1)
        { perror("send"); WIMP(1); };

    if (recv(my_fd,combuf,sizeof(combuf),0) == -1)
        { perror("recv"); WIMP(1); };
    combuf[sizeof(combuf)-1] = 0;

```

```

if ((len>=0) && (len<sizeof(combuf))) combuf[len] = 0;

fprintf(stderr,"Received request <%s>\n",combuf);

if (strncmp(combuf,"GET",3))
{ /* not GET, is it CLOSE ? */
    if (!strncmp(combuf,"CLOSE",5))
    { /* it is CLOSE */
        LOCK;
        run_server = 0;           // tell the others to stop too
        UNLOCK;
        fprintf(stderr,"CLOSE request detected, about to shut down\n");
        WIMP(0);
    } else
    { /* unknown command */
        fprintf(stderr,"Unknown command: %s\n",combuf);
        if (send(my_fd,"FAIL: Pardon?",strlen("FAIL: Pardon?")+1,0) == -1)
            { perror("send"); WIMP(1); };
        WIMP(1);
    };
} else
{ /* command is GETn */
    channel = atoi(combuf+3);
    if ((channel<0) || (channel >= nCHN))
    { /* invalid channel selection */
        fprintf(stderr,"Invalid channel in GET %d\n",channel);
        if (send(my_fd,"FAIL: Channel?",strlen("FAIL: Channel?")+1,0) == -1)
            { perror("send"); WIMP(1); };
        WIMP(1);
    };

    LOCK;
    if ((conn1[channel]) && (conn2[channel]))
    {
        fprintf(stderr,"Only two clients per channel allowed for now\n"
            "Channel %d busy (%d,%d)\n",
            channel,conn1[channel],conn2[channel]);
        if (send(my_fd,"FAIL: Busy",strlen("FAIL: Busy")+1,0) == -1)
            { perror("send"); UNLOCK; WIMP(1); };
        UNLOCK; WIMP(1);
    };
    if (conn1[channel]) conn2[channel]=my_fd;
    else
        conn1[channel]=my_fd;
    // and maybe alloc buffer and stuff
    UNLOCK;

```

```

fprintf(stderr,"GET for channel %d accepted\n", channel);
while (run_server) // send until client goes away or global shutdown
{ /* loop to satisfy GET */

    if ((retval = send_buffer(my_fd, channel)))
        { fprintf(stderr,"Error %d while transmitting channel %d\n",
                    retval,channel);
          WIMP(1);
        } else { /* fprintf(stderr,"."); */ };

    // convert buffer (locked), send buffer, recv (discard)
};

}; // end GET

WIMP(0);
};

```

G.1.3. alsa-ice.c

Hier wird die Soundkarte via ALSA initialisiert. Die eigentlichen Schreib-Lese-Zugriffe finden sich zur Zeit nur in rec-sock-alsa.c !

```

#include "rec-sock-alsa.h"

/* ***** */

int open_sound(snd_pcm_t ** handle /* ptr auf ptr */)
{
    int retval;
    snd_pcm_channel_params_t params;

    if ((retval = snd_pcm_open(handle, 0 /* card */, 0 /* device */,
                               SND_PCM_OPEN_CAPTURE)))
        { fprintf(stderr,"Error with snd_pcm_open: %s\n",snd_strerror(retval));
          return 1; };
    // auch mgl.: snd_pcm_open_subdevice(h,c,d, s ,m) ...
    // mode: | SND_PCM_OPEN_NONBLOCK ist mgl.

    params.channel=0;
    params.format.format=SND_PCM_SFMT_S32_LE;
    // Ice1712pro/Linux does only this, and without the use of plugins,
    // ALSA offers no conversion.
    // btw. the hardware only uses the 24 MSBs.
    params.format.interleave = 1;

```



```

    // interleaved vs. blocked data format
    // for vector based i/o, interleave sucks.
params.format.rate = snd_HZ;
params.format.voices = nCHN;
    // Ice1712pro/Linux driver supports only opening all voices at once!
    // for capture, voices *must* be 12, for playback, it *must* be 10.
params.channel = SND_PCM_CHANNEL_CAPTURE; // or ... CAPTURE
params.mode = SND_PCM_MODE_STREAM; // or ... BLOCK
    // not set: params.digital.*, params.format.reserved,
    // params.reserved, params.format.special
params.start_mode = SND_PCM_START_DATA; // or ... FULL or ... GO
params.stop_mode = SND_PCM_STOP_ROLLOVER;
    // ... STOP or ... ERASE or ... ROLLOVER
    // STOP klappt, aber ROLLOVER muss nach einem Buffer
    // overrun/underrun nicht neu angeworfen werden.
params.time = 0; // set to get gettimeofday time of beginning of transfer
                // -> sys/time.h ... struct timeval -> tv_sec ...
params.ust_time = 0; // set to get time in UST format

    params.buf.stream.queue_size = (snd_HZ >> 1) * 4 * nCHN;
                // muss ein Vielfaches von 4*nCHN sein, min 1/2k...
                // 1/16 Sekunde mit (snd_HZ >> 4) * 40 funktioniert z.B. ok.
                // Groessere Queue -> mehr Latency aber weniger kritisches
                // Timing bei Multitasking. Also sinnvoll einstellen!
params.buf.stream.fill = SND_PCM_FILL_SILENCE_WHOLE;
                // or ... NONE or ... SILENCE
params.buf.stream.max_fill = 480;
                // not really used if fill!=...SILENCE
/*
* params.buf.block.frag_size=40960;
* params.buf.block.frag_min=1;
* params.buf.block.frag_max=10;
*/
if ((retval = snd_pcm_channel_params(*handle,&params)))
{ fprintf(stderr,"Format not supported by soundcard: ");
  fprintf(stderr,"%s\n",snd_strerror(retval)); return 1; };

if ((retval = snd_pcm_capture_prepare(*handle)))
{ fprintf(stderr,"Error preparing capture: ");
  fprintf(stderr,"%s\n",snd_strerror(retval)); return 1; };

// je nach START setting fehlt nun noch snd_pcm_capture_go(),
// ansonsten passiert der START automatisch.
return 0;
};

```

```

/* ***** */

int close_sound(snd_pcm_t * handle)
{
int retval;
if ((retval = snd_pcm_close(handle)))
    { fprintf(stderr,"Error closing sound: %s\n",snd_strerror(retval)); };
return retval;
};

/* ***** */

```

G.2. client2file.c – Der Client zu rec-sock-alsa

Dieses Programm stellt einen einfachen Beispiel-Client für rec-sock-alsa dar, mit dem man Audiodaten vom Server abholen kann.

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define MAXBUFFER (256*1024)

#define fmtSOCK signed short int

// #define PROGRESS

/* ***** */

int conn2server (char *hostname, int port, int *sockfd)
{
    struct hostent *he;
    struct sockaddr_in their_addr;

    if ((he=gethostbyname(hostname)) == NULL) /* get the host info */
        { perror("gethostbyname"); return 1; };
}

```

```

    if ((*sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) /* open socket */
        { perror("socket"); return 1; };

    bzero(&their_addr, sizeof(their_addr));
    their_addr.sin_family = AF_INET;          /* IPv4 */
    their_addr.sin_port = htons(port);        /* port (netw byte order, short) */
    their_addr.sin_addr = *((struct in_addr *)he->h_addr);

    if (connect(*sockfd, (struct sockaddr *)&their_addr,
                sizeof(struct sockaddr)) == -1) /* connect to target */
        { perror("connect"); return 1; };

    return 0;
};

/* ***** */

int main(int argc, char *argv[])
{
    int sockfd, numbytes, putbytes;
    static fmtSOCK buf[MAXBUFFER];
    char message[100];

    if (argc != 4)
        { fprintf(stderr, "Usage: %s hostname port GET<n>|CLOSE\n"
          "GET writes received Data to standard output!\n",
          argv[0]); return 1; };

    /* try to connect to server */
    if (conn2server (argv[1], atoi (argv[2]), &sockfd))
        { fprintf(stderr, "Could not connect server\n"); return 1; };

    numbytes=recv(sockfd, message, 10, 0);
    message[(numbytes>0) ? numbytes : 0] = '\0';
    if (strcmp (message, "Hello!"))
        { fprintf (stderr, "Protocol not recognized\n"); return 1; };

    fprintf (stderr, "C - Received %s, now sending %s\n", message, argv[3]);

    /* send command to server */
    if (send (sockfd, argv[3], strlen (argv[3])+1, 0)<0)
        { perror("send"); return 1; };

    /* if command is GET<n>, then loop, else return */
    if (strncmp (argv[3], "GET", 3))

```

```

    { /* if (!strcmp(argv[3],"CLOSE",5)) return 0; */
      numbytes=recv(sockfd, message, 99, 0);
      if (numbytes<0) { perror("recv"); return 1; };
      message[(numbytes>0) ? numbytes : 0] = '\0';
      fprintf (stderr,"Response was: %s\n",message);
      return 0;
    };

while (1)
{
    /* receive buffer */
    numbytes=recv(sockfd, buf, sizeof(buf), 0);

    if (numbytes > 0)
        { if (send (sockfd,"OK",strlen ("OK")+1, 0)<0)
            { perror("send"); return 1; };
          } else if (numbytes < 0)
            { perror("recv"); return 1; };

    if (numbytes > 0)
        {
            if (!strcmp((char *)buf,"FAIL",4))
                { (void)strncpy(message,(char *)buf,sizeof(message));
                  if (message[sizeof(message)-1])
                      fprintf(stderr,"Protocol failure: [Msg too long]\n");
                  else fprintf(stderr,"Protocol failure: %s\n",message);
                };

            putbytes = write(1,buf,numbytes);
            if (putbytes != numbytes) { perror("write error"); return 1; };
        };

        usleep(1000);
#ifdef PROGRESS
        if (numbytes>0) fprintf (stderr,"C - received %i bytes\n",numbytes);
#endif

    }; // never leave this unless an error occurred...

    return 1;
};

```

G.3. play-alsa.c

Die Abspielfunktion sollte später auf jeden Fall in ein Client-Server Modell eingebaut werden, dieses Programm dient nur der Demonstration der ALSA-Schnittstelle zum Abspielen von Audi-odaten.

```

/*
 * PCM-Player fuer 16bit signed Dateien auf Ice1712 Pro mit ALSA/Linux/x86
 * Aufruf [Programm] [Dateiname]
 *
 * (Compilieren fuer Mono oder Stereo und auf eine feste Samplingrate zw.
 * 8000 und 48000Hz, hat bisher keine eigene Erkennung fuer WAV-Header!)
 * Benotigte Library: asound
 *
 * Frei kopierbarer haesslicher "proof of concept" Code ohne jegliche
 * Garantien, nicht mal die, die oben versprochene Funktion zu erfuellen.
 *
 * Geschrieben von Eric Auer, Tipps von Dan Hollis und Christian Dressler
 * Achtung: Der ALSA Treiber fuer diese Karte erlaubt nur, ALLE Kanale
 * gleichzeitig zu oeffnen, daher funktioniert dafuer auch keine OSS-Emu.
 * Fuer Ausgabe sind das exakt 10, fuer Eingabe exakt 12 Monokanale.
 *
 * Ausserdem ist 32bit signed little endian das einzig moegliche Format,
 * wobei nur die 24 MSB vom Wandler verarbeitet werden. Das ALSA PCM
 * Plugin System bietet aber bei Bedarf Formatanpassungen per Software.
 *
 * 22.05.2000
 */

#define FORCE_REGULAR 0
// 0 = auch Pipes etc. erlaubt
// #define snd_inSTEREO
/* Wenn define, wird die Datei von Stereo auf Mono runtergemischt */
#define snd_HZ 22050 /* z.B. 22050 */
/* Ice1712pro/Linux kann nur S32_LE, aber 8-48kHz Samplingrate... */

#define iCARD signed int
#define iFILE signed short int
#define nCHN 10

/* ***** */

#include <sys/asoundlib.h>
#include <linux/asound.h>
// #include <linux/asoundid.h> // Kartentypen
// #include <linux/asequencer.h> // Sequencer

```

```

#include <stdio.h>
#include <string.h> // strncpy
#include <sys/types.h>
#include <errno.h> // ...
#include <fcntl.h> // open
#include <sys/stat.h> // stat
#include <stdlib.h> // malloc
#include <unistd.h> // usleep
#include <sys/uio.h> // iovec

/* ***** */

// #define DEBUG
#define USLEEP
#define WBUF 4096 /* WriteBufferSize in Samples */

#define snd_inBITS (8*sizeof(iFILE))
/* *** Achtung, zur Zeit werden nur 16 Bit signed verwendet!!! *** */
#define snd_SHL (31-snd_inBITS) /* 24 MSB of 32 used bei ice1712 */
// Ich habe ein MSB weggelassen, besser analog auf volles Volume bringen...
// 386: __BYTE_ORDER == __LITTLE_ENDIAN

// ice1712pro kann nur genau 10 voices playback und 12 voices capture
// ausserdem geht im stream mode nur write/interleaved richtig!?

/* ***** */

void help(void) { printf("Usage: Give me a filename and I will play it,\n"
    "assuming the data is signed %dbit "
#ifdef snd_inSTEREO
    "stereo"
#else
    "monaural"
#endif
    " PCM %dHz.\n",
    snd_inBITS,snd_HZ);
    _exit(1); return; };

/* auf files gibts z.B. getc() und feof() und int getw()
 * coz EOF is a valid int, ferror() should be used to detect getw() errors.
 * interessant bei sun: man -s 5 intro -> Tipps fuer Manpages aus Sektion 5,
 * z.B. regex, math, pam_unix, socket, signal/siginfo, ... flockfile fread
 */

/* ***** */

```

```

int file_size(char * filename, int force_regular)
{
    struct stat status;
    int fd, fsize;
    fd = open(filename, O_RDONLY /* | O400000 */ ); // 04... ist O_NOFOLLOW
    if (!fd) { perror("could not open sound file"); return 0; };
    if (fstat(fd, &status))
        { perror("could not stat file");
          close(fd);
          return 0;
        };
    close(fd);
    fsize = status.st_size / sizeof(iFILE);
#ifdef snd_inSTEREO
    fsize >>= 1;
#endif
    if (force_regular)
        {
            if (!S_ISREG(status.st_mode))
                { fprintf(stderr, "not a regular file\n");
                  return 0; };
        } else
        {
            if (!S_ISREG(status.st_mode))
                { fprintf(stderr, "not a regular file, assuming pipe\n");
                  return 12*3600*snd_HZ;
                };
        };
    return (fsize>0) ? fsize : 0;
};

/* ***** */

int open_sound(snd_pcm_t ** handle /* ptr auf ptr */)
{
    int retval;
    snd_pcm_channel_params_t params;

    if ((retval = snd_pcm_open(handle, 0 /* card */, 0 /* device */,
                               SND_PCM_OPEN_PLAYBACK)))
        { fprintf(stderr, "Error opening sound "); perror(snd_strerror(retval));
          return 1; };
    // auch mgl.: snd_pcm_open_subdevice(h,c,d, s ,m) ...
    // mode: | SND_PCM_OPEN_NONBLOCK ist mgl.

```

```

params.channel=0;
params.format.format=SND_PCM_SFMT_S32_LE;
    // je nach iCARD !
    // Ice1712pro/Linux does only this, and without the use of plugins,
    // ALSA offers no conversion.
    // btw. the hardware only uses the 24 MSBs.
params.format.interleave = 1;
    // interleaved vs. blocked data format
    // for vector based i/o, interleave sucks.
params.format.rate = snd_HZ;
params.format.voices = nCHN;
    // Ice1712pro/Linux driver supports only opening all voices at once!
    // for capture, voices *must* be 12, for playback, it *must* be 10.
params.channel = SND_PCM_CHANNEL_PLAYBACK; // or ... CAPTURE
params.mode = SND_PCM_MODE_STREAM; // or ... BLOCK
    // not set: params.digital.*, params.format.reserved,
    // params.reserved, params.format.special
params.start_mode = SND_PCM_START_DATA; // or ... FULL or ... GO
params.stop_mode = SND_PCM_STOP_STOP;
    // ... STOP or ... ERASE or ... ROLLOVER
    // STOP klappt, aber ROLLOVER muss nach einem Buffer
    // overrun/underrun nicht neu angeworfen werden.
params.time = 0; // set to get gettimeofday time of beginning of transfer
    // -> sys/time.h ... struct timeval -> tv_sec ...
params.ust_time = 0; // set to get time in UST format

    params.buf.stream.queue_size = (snd_HZ >> 1) * sizeof(iCARD) * nCHN;
        // muss ein Vielfaches von ... sein, min 1/2k...
        // 1/16 Sekunde mit (snd_HZ >> 4) * ... funktioniert z.B. ok.
        // Groessere Queue -> mehr Latency aber weniger kritisches
        // Timing bei Multitasking. Also sinnvoll einstellen!
    params.buf.stream.fill = SND_PCM_FILL_SILENCE_WHOLE;
        // or ... NONE or ... SILENCE
    params.buf.stream.max_fill = sizeof(iCARD) * nCHN * 10;
        // not really used if fill!=...SILENCE
/*
* params.buf.block.frag_size=4096 * nCHN;
* params.buf.block.frag_min=1;
* params.buf.block.frag_max=10;
*/
if ((retval = snd_pcm_channel_params(*handle,&params)))
{ fprintf(stderr,"Format not supported by soundcard: ");
  fprintf(stderr,"%s\n",snd_strerror(retval)); return 1; };

```



```

if ((retval = snd_pcm_playback_prepare(*handle)))
{ fprintf(stderr, "Error preparing playback: ");
  fprintf(stderr, "%s\n", snd_strerror(retval)); return 1; };

// je nach START setting fehlt nun noch snd_pcm_playback_go(),
// ansonsten passiert der START automatisch.
return 0;
};

/* ***** */

int write_sound(snd_pcm_t * handle, FILE * pcmdata, int samples)
{
int left, written;
int cnt, cnt2, retval;
int towrite, toread;
// struct iovec bufv[nCHN];
iCARD mybuf[1+(WBUF*nCHN)];
iFILE filebuf[1+(WBUF*2)]; // *2 wg snd_inSTEREO
iCARD * myrest;
int restsize;
int underrun=0;

if (samples <= 0) return 1;
left = samples;
written = 0;
#ifdef DEBUG
fprintf(stderr, "About to play %d samples\n", samples);
#endif

while (left>0)
{
  toread = (left>WBUF) ? WBUF : left;
  towrite = fread(&filebuf,
#ifdef snd_inSTEREO
    2 *
#endif
    sizeof(filebuf[0]), toread, pcmdata);
  if (ferror(pcmdata)) { perror("fread PCM file failed"); return 1; };
  if (towrite<toread)
  { if (feof(pcmdata))
    { if (left<WBUF)
      { left=0; towrite=0;
        fprintf(stderr, "EOF reached (PCM file)\n");
      };
    }
  }
}

```

```

        clearerr(pcmdata);
#ifdef DEBUG
        fprintf(stderr,"EOF ");
#endif
        } else
        { perror("Short read but no EOF?"); return 1;
        };
        towrite = (towrite<0) ? 0 : towrite;
        underrun = 1;
    };
    clearerr(pcmdata);

    if (towrite)
    for (cnt=cnt2=0;cnt<towrite;cnt++)
    {
        cnt2 = cnt * nCHN;
#ifdef snd_inSTEREO
        mybuf[cnt2] = filebuf[cnt] << snd_SHL;
#else
        mybuf[cnt2] = (filebuf[cnt+cnt] + filebuf[cnt+cnt+1]) << (snd_SHL-1);
//      mybuf[cnt2] = filebuf[cnt+cnt] << snd_SHL;
                // pick one side instead of mixing
#endif
    for (cnt2++ ; cnt2<((cnt+1)*nCHN) ; cnt2++) mybuf[cnt2]=0;
    };
    // eingelesen...

    // ...ausgeben
    if (underrun && towrite)
    {
        underrun = 0;
        retval = snd_pcm_playback_go(handle);
#ifdef DEBUG
        fprintf(stderr,"snd_pcm_playback_go after underrun returned %s\n",
                snd_strerror(retval));
#endif
    };
#ifdef DEBUG
    fprintf(stderr,"Underrun state\n");
#endif
    myrest = mybuf /* &mybuf[0] */;
    restsize = towrite;
    while (restsize>0)
    {
        written = snd_pcm_write(handle, myrest, restsize*sizeof(iCARD)*nCHN );
    }

```

```

    // snd_pcm_writev(handle, bufv, nCHN);

    if ((written<0) && (written != -EAGAIN))
        { fprintf(stderr,"Error with snd_pcm_write, trying to recover: %s\n",
            snd_strerror(written));
#ifdef USLEEP
            usleep(1000);
#endif
//          (void)snd_pcm_playback_drain(handle);
//          (void)snd_pcm_playback_flush(handle);
//          retval = snd_pcm_playback_go(handle);
//          fprintf(stderr,"Tried to flush and restart playback / %s\n",
//              snd_strerror(retval));

        if ((retval = snd_pcm_close(handle)))
            { fprintf(stderr,"Error closing sound for re-open: %s\n",
                snd_strerror(retval));
                return 1;
            };
        if ((retval = open_sound(&handle)))
            { fprintf(stderr,"Error re-opening sound: %s\n",
                snd_strerror(retval));
                return 1;
            };
    };
    written = (written>0) ? (written / (sizeof(iCARD) * nCHN)) : 0;
    left -= written;
#ifdef DEBUG
    if (written!=restsize)
        { fprintf(stderr,"Only %d of %d (rest of %d) samples accepted\n",
            written,restsize,towrite);
        } else fprintf(stderr,"%d samples written\n",written);
#endif
    restsize -= written;
    myrest += written * nCHN;
#ifdef USLEEP
    if (left>0) usleep(1000);
#endif
    }; // Ende while... write
#ifdef USLEEP
    if (!towrite) usleep(1000);
#endif
    }; // Ende while... read
return 0;
};

```

```
/* ***** */

int main(int argc, char ** argv)
{
FILE * pcmdata;
int retval, fsize;
snd_pcm_t * soundhandle=NULL;

if ((argc!=2) || (!argv)) help(); if (!argv[1]) help();

if (! (fsize = file_size(argv[1],FORCE_REGULAR)))
    { fprintf(stderr,"%s:",argv[1]); perror("Error checking file"); return 1; };
if (! (pcmdata = fopen(argv[1],"r")))
    { fprintf(stderr,"%s:",argv[1]); perror("Error opening file"); return 1; };

if (open_sound(&soundhandle))
    { fprintf(stderr,"Error opening ALSA"); return 1; };

if (write_sound(soundhandle, pcmdata, fsize-1)) /* fsize hat Einheit Samples */
    { fprintf(stderr, "Error writing to PCM\n"); };
    // but hang on...

if ((retval = snd_pcm_close(soundhandle)))
    { fprintf(stderr,"Error closing sound: ");
      fprintf(stderr,"%s\n",snd_strerror(retval)); };
    // hang on if an error occured, do not return 1.

if (fclose(pcmdata))
    { perror("Could not close file (!)"); return 1; };

return 0;
};
```

H. Implementierung des Sprachgesteuerten Fahrstuhls

lift.oz

```
functor
import

    L(make:MakeLift)          at 'lift.ozf'
    D(make:MakeAutomaton)    at 'dialog.ozf'

export
define

    Automaton = {MakeAutomaton Lift}
    Lift = {MakeLift Automaton}
    {Lift.start}

end
```

lift.oz

```
functor
import
    Ozcar
    LiftControl          at 'lift-control-selection.ozf'

    History              at 'history.ozf'
    Memory               at 'memo.ozf'

    Time                 at 'time.ozf'
    Browser              at 'browser.ozf'
%   Panel                at 'panel.ozf'
    T(start:Thread)      at 'threads.ozf'
    Int(toAtom:ToAtom)   at 'int.ozf'
export
```

```

    Make
define

    % {Panel.make}
    % {Panel.browseMemory}
    Browse = Browser.debug.dialog

    proc{ProcessAutomaton StateName Lift}
        {Ozcar.breakpoint}
        {Browse state(StateName)}
        unit(automaton:Automaton
            protocol:Protocol ...) = Lift

        unit(selection:Selection
            transition:Transition
            states:States ...) = Automaton

        {Protocol.addInfo state(StateName)}
        State = States.StateName
        SelectName = State.selection
        TransName = if SelectName == unit
            then % there is only one possible transition
                case {Arity State.trans}
                of [TransName]
                then TransName
                else raise error(unique_Transition(State)) end unit
            end
            else % the SelectionName is specified
                {Selection.SelectName}
            end
        end
        NextState = State.trans.TransName
        {Browse trans(TransName)}
        Trans = {Transition.TransName NextState}
    in
        try
            {Protocol.addInfo transition(Trans.text)}
            {Trans.action} % execute the transition's action
            {Protocol.collectEvent} % update the protocol
            if StateName == Automaton.'end'
            then {Protocol.saveStory} % save the protocol in the end state
            else skip
            end
            {ProcessAutomaton NextState Lift} % go to the next state
        catch All then
            {Browser.debug.error exception(All)}

```

```

        end
    end

    fun{Make Automaton}
        {Browse 'hallo lift'}

        Memo = {Memory.make Automaton.memo}
        proc{Start}
            {Memo.set 'time' {Time.real}}
        proc{Check}
            Pos = {Memo.get pos}
        in
            if Pos == {Memo.get ziel} % vorsicht hier
                                    % das ist zu grosszügig
            then {Automaton.action.sprechen sentence
                ['Ankunft' 'Stock' {Int.toAtom Pos}]}
                {Memo.set Pos false}
            else skip
            end
            {Delay 3000}
            {Check}
        end
    in
        % {Thread Check} % keine Ansage der Ankunft
        % % in einem Stockwerk !!
        {ProcessAutomaton Automaton.start Lift}
    end

    Lift =
    unit(automaton:Automaton
        memo:Memo
        protocol:{History.make}
        control: {LiftControl.make Lift}
        start:Start)
    in
        Lift
    end
end

```

dialog.oz

```

functor
import

    Ozcar
    System(gcDo)

```

```

fun{Make Lift}

  unit(protocol:Protocol
    memo:Memo ...) = Lift

  %%%% Actions %%%%%%%%%%%%%%

  local
    proc{Inform Info}
      {Protocol.addInfo Info}
    end

    proc{Sprechen Type Path}
      Text#Wav =
        case Type of question then
          {Sound.select.selectQues Lift text Path}
          #{Sound.select.selectQues Lift sound Path}
        else
          {Sound.select.selectSent Lift text Path}
          #{Sound.select.selectSent Lift sound Path}
        end
    end

    in
      {Recognizer.break}
      {Browser.debug.recognizerStringSend breaksend({RealTime})}
      {Inform speak(Text)}
      {Browse speak(Text)}
      {Play Wav}
      {Recognizer.start}
      {Memo.set time {RealTime}}
    end

    proc{Fahre}
      {Action.inform 'Fahrt'}
      Dest = {Memo.get dest}
      Where = if {Member Dest Words.prof}
        then profs
        else Etage
        end
    in
      {Memo.set {DestToInt Dest} true}
      {Lift.control.fahre {DestToInt Dest}}
      {Browse 'Lift:FAHRE!'#Dest}
      {Sprechen sentence [SoundWords.fahre Where Dest]}
    end

  in

```

```

        Action = unit(sprechen:Sprechen
                      fahre:Fahre
                      inform:Inform
                      )
    end

%%% Transitions %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% The type of all transitions is:
%
% Atom -> unit(to:Atom
%              text:Atom
%              action: proc{$} ... end)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

local
  fun{CrockerNichtHier NextState}
    unit(to:NextState
          text:'Crocker im Nebengebäude'
          action:proc{$}
            {Action.sprechen sentence [Crocker]}
          end)
  end
  fun{WahlsterNichtHier NextState}
    unit(to:NextState
          text:'Wahlster DFKI'
          action:proc{$}
            {Action.sprechen sentence [Wahlster]}
          end)
  end
  fun{BibliothekNichtHier NextState}
    unit(to:NextState
          text:'Bibliothek im Nebengebäude'
          action:proc{$}
            {Action.sprechen sentence [Bibliothek]}
          end)
  end

  fun{NeuerDialog NextState}
    unit(to:NextState
          text:'Endzustand '
          action: proc{$}

```

```

        {CollectUntil {RealTime}}
        {System.gcDo}
    end
)
end
fun{NichtVerstanden NextState}
    Text = 'Eingabe nicht verstanden'
in
    unit(to:NextState
        text:Text
        action: proc{$}
            {Action.sprechen sentence ['Nicht Verstanden']}
        end)
end
fun{NachfragePhonetik NextState}
    unit(to:NextState
        text:'Nachfrage Phonetik'
        action: proc{$}
            {Action.sprechen sentence ['Nachfrage Phonetik']}
            {Lift.control.wait}
            {Browse 'Lift:WAIT!'}
        end)
end
fun{Abbruch NextState}
    unit(to:NextState
        text:'Abbruch'
        action: proc{$}
            {Action.sprechen sentence ['Abbruch']}
        end)
end
fun{Fahrstuhl NextState}
    unit(to:NextState
        text:'Token Fahrstuhl erkannt'
        action:proc{$}
            {Action.inform erkannt('Fahrstuhl')}
        end
    )
end
fun{Bereits NextState}
    unit(to:NextState
        text:'Sie sind bereits im Stock '
        action:proc{$}
            Arg = {Memo.get dest}

```

```

        in
            {Action.sprechen sentence
              ['Fehlerhafte Eingabe' Etage Arg]}
        end)
    end
    fun{Fahre NextState}
        unit(to:NextState
            text:'Fahre nach Stock '
            action:Action.fahre)
    end
    fun{Zieleingabe NextState}
        unit(to:NextState
            text:'Eingabe verstanden'
            action:proc{$}
                {Action.inform 'Zieleingabe: '#{Memo.get dest}}
            end)
    end
    fun{FrageNachZiel NextState}
        unit(to:NextState
            text:'Wohin?'
            action: proc{$}
                {Action.sprechen question [Wohin]}
            end)
    end
    fun{TastenBenutzen NextState}
        unit(to:NextState
            text:'Tasten benutzen'
            action: proc{$}
                {Action.sprechen sentence ['Abbruch']}
            end)
    end
    fun{Start NextState}
        unit(to:NextState
            text:'nächster Dialog'
            action: proc{$}
                if {Memo.isDef time}
                then skip % not the first dialog
                else {Memo.set time {RealTime}}
                end
            end)
    end
    fun{Bitte NextState}
        unit(to:NextState
            text:'bitte schoen'

```

```

        action:proc{$}
            {Action.sprechen sentence [bitte]}
        end)
    end

in
    Transition = unit(start:Start
        fahre:Fahre
        wohin:FrageNachZiel
        ziel:Zieleingabe
        keyword:Fahrstuhl
        bereits:Bereits
        bitte:Bitte
        ende:NeuerDialog
        abbruch:Abbruch
        Phonetik:NachfragePhonetik
        nichtVerstanden:NichtVerstanden
        Wahlster:WahlsterNichtHier
        Crocker:CrockerNichtHier
        Bibliothek:BibliothekNichtHier
        tasten:TastenBenutzen
    )

end

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% understand goals in utterances
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

fun{Select List}
    case List
    of nil then unit    % unit is returned if nothing is selected
    [] [X1] then X1    % the remainng element of the List is selected
    [] X1|X2|Rest then
        if {GetProb X1} =< {GetProb X2}
        then {Select X1|Rest}
        else {Select X2|Rest}
        end
    end
end
end

```

```

fun{Filter Atoms Words}    % the Words are records
                            % of the form Word(...) where Word is an
                            % atom that is checked to belong to the
                            % list of Atoms or not.

    {List.filter Words

```

```
% process destinations %
```

[illegible]

```
%% Select transition %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% The type of all selection function is: %
%
%      (noarg) -> atom %
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
local
```

```
fun{Keyword}
  {Ozcar.breakpoint}
  Utterance = {WaitKeyword Words.key {Memo.get time}}
  _#Input = {SplitAt Words.key Utterance.words}
  {Memo.set time Utterance.'end'}
  Dest = {Label {Select {Filter Words.dest Input}}}
in
  if Input == nil
  then keyword
  else
    {Goto Dest}
  end
end

fun{Verarbeite}
  Utterance = {NextUtterance {Memo.get time}}
  {Memo.set time Utterance.'end'}
  WordsSaid = case Utterance
    of timeout(...) then nil
    else Utterance.words
  end
  Dest = {Label {Select {Filter Words.dest WordsSaid}}}
in
  {Goto Dest}
end

fun{Bestätige}
  Floor = {Memo.get Infos.floor}
  Door = {Memo.get Infos.door2}
  IntDest = {DestToInt {Memo.get dest}}
in
  if Floor == IntDest andthen Door == Infos.auf
```

```

        then bereits
        else fahre
        end
    end
end

fun{PhonetikAuswahl}
    Utterance = {NextUtterance {Memo.get time}}
    {Memo.set time Utterance.'end'}
    WordsSaid = case Utterance
                  of timeout(...) then nil
                  else Utterance.words
                  end
    Dest = {Label {Select {Filter Words.phonetik WordsSaid}}}}
in
    if Dest == unit
    then nichtVerstanden
    else {Goto Dest}
    end
end
fun{Danke}
    Utterance = {NextUtterance {Memo.get time}}
    WordsSaid = case Utterance
                  of timeout(...) then nil
                  else Utterance.words
                  end
    Phons = {Map WordsSaid Label}
in
    if Phons == [Lexicon.danke]
    then    % danke said
        {Memo.set time Utterance.'end'}
        bitte    % dialog ended with bitte
    elseif
        {Filter Words.key WordsSaid} \= nil
    then    % fahrstuhl starts the next dialog
        {Memo.set time {Memo.get time}-0.1}
            % hack wegen eindeutiger Timeouts
        keyword
            % the actual utterance is reprocessed
            % in the Keyword state
    else    % timeout or nothing relevant said
        {Memo.set time Utterance.'end'}
        ende % the dialog ends here
    end
end
end
in

```



```

        Selection = unit(keyword:Keyword
                        verarbeite:Verarbeite
                        bestätige:Bestätige
                        Phonetik:PhonetikAuswahl
                        danke:Danke)
end

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

State9 = unit(name:'Begin'
              trans:unit(start:'Keyword')
              selection:unit)

```

```

State0 = unit(name:'Keyword'
              trans:
                unit(keyword:'Frage wohin'
                    ziel:'Bestätige Ziel'
                    Phonetik:'Nachfrage Phonetik'
                    nichtVerstanden:'Verarbeite Eingabe2'
                    Wahlster:'Danke'
                    Crocker:'Danke'
                    Bibliothek:'Danke')
              selection:keyword)

```

```

State1 = unit(name:'Frage wohin'
              trans:unit(wohin:'Verarbeite Eingabe')
              selection:unit)

```

```

State2 = unit(name:'Verarbeite Eingabe'
              trans:
                unit(ziel:'Bestätige Ziel'
                    Phonetik:'Nachfrage Phonetik'
                    nichtVerstanden:'Verarbeite Eingabe2'
                    Wahlster:'Danke'
                    Crocker:'Danke'
                    Bibliothek:'Danke')
              selection:verarbeite)

```

```

State3 = unit(name:'Bestätige Ziel'
              trans: unit(fahre:'Danke'
                          bereits:'Ende')
              selection:bestätige)

```

```

State4 = unit(name:'Nachfrage Phonetik'

```

```

        trans:
            unit(ziel:'Bestätige Ziel'
                nichtVerstanden:'Verarbeite Eingabe2')
        selection:Phonetik)

State5 = unit(name:'Verarbeite Eingabe2'
    trans:
        unit(ziel:'Bestätige Ziel'
            Phonetik:'Nachfrage Phonetik2'
            nichtVerstanden:'Tasten benutzen'
            Wahlster:'Danke'
            Crocker:'Danke'
            Bibliothek:'Danke'
            abbruch:'Ende')
        selection:verarbeite)

State6 = unit(name:'Ende'
    trans:unit(ende:'Begin')
    selection:unit)

State7 = unit(name:'Nachfrage Phonetik2'
    trans:
        unit(ziel:'Bestätige Ziel'
            abbruch:'Begin'
            nichtVerstanden:'Ende')
        selection:Phonetik)

State8 = unit(name:'Tasten benutzen'
    trans:unit(tasten:'Ende')
    selection:unit)

State10 = unit(name:'Danke'
    trans:unit(bitte:'Ende'
        keyword:'Keyword'
        ende:'Ende')
    selection:danke)

StateList = [State0 State1 State2 State3
    State4 State5 State6 State7
    State8 State9 State10]

Pairs= {Map StateList fun{$ S} (S.name)#S end}
States = {List.toRecord unit Pairs}
in
    unit(states: States

```

```

        transition:Transition
        selection:Selection
        start:'Begin'
        'end':'Ende'
        memo:unit(type:MemoType init:MemoInit)
        action:Action)
    end
end

```

recognizer.oz

```

functor
import
    System

    Cell                at 'cell.ozf'
    Browser              at 'browser.ozf'
    Interface            at 'recognizer-interface.ozf'
    Word(getInfo:GetInfo) at 'word.ozf'
    Time                at 'time.ozf'
    Threads(start:Thread) at 'threads.ozf'
    P(print:Print)       at 'print.ozf'

%   D(until:DelayUntil)      at 'delay.ozf'
%   P(answerDelay:AnswerDelay) at 'parameter.ozf'
export
    Make
define

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% To be done:                                     %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Invariants on the stream should be checked %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                     %%
%                                     %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

End = 'end'

% The speech recognizer returns word packages
% per utterance.
%
% We do not assume that words in word packages are
% ordered in time but we assume that word packages
% are ordered.

```

```

% We get the packages on a stream where words
% come individually rather than in packages. The
% stream is a list of so called items. An item is
% either a control item or a word item, both of which
% are computed by the speech recognizer. A package on
% a stream is respresented by a subsequence of items:
%
% 1) a single start control item
% 2) a sequence of word items for all words in the package
% 3) a single end control item
%
% We assume that words items of earlier packages come
% earlier on the stream.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%% Stream %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Browse      = Browser.debug.recognizer
BrowseSend  = Browser.debug.recognizerDataSend
BrowseReceived = Browser.debug.recognizerDataReceived
BrowseDialog = Browser.debug.dialog
BrowseStream = Browser.debug.streamSize

fun{Make}
% The FirstItemPointer is a pointer to the actual
% first item of interest on the stream.
local
  Stream

  {Thread proc{$}{ForAll Stream
    proc{$ Item}
      case Item
      of word(...) then
        {BrowseReceived Item}
        {Print Item.info}
        {Print Item.package#{Time.real}}
        [] control(info:start ...) then
          {BrowseReceived Item}
          {Print start}
        else skip
        end
      end}
    end}
end}

```

```

    % enforce an invariant on the stream
    {Thread proc{$} {SetEndTimes Stream} end}

    StreamPort = {Port.new Stream}
in
    FirstItemPointer = {Cell.make}
    {FirstItemPointer.set Stream}
    StreamSize = {Cell.make}
    {StreamSize.set 0}
    fun{OpenListSize OL N}
        if {IsDet OL}
            then case OL
                of _|T then {OpenListSize T N+1}
                end
            else N
            end
        end
    end
    proc{BrowseStreamSize}
        {BrowseStream {OpenListSize {FirstItemPointer.get} 0}}
    end
    {BrowseStreamSize}
    proc{Send Item}
        {BrowseSend send_to_data_stream#Item}
    in
        {StreamSize.set {StreamSize.get}+1}
        {BrowseStreamSize}
        {Port.send StreamPort Item}
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Start the speech recognizer
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    Recognizer = {Interface.make Send}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Invariants of the stream
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% grounding start control items %%%%%%%%%%%%%%%

% The package end time in a start control
% item may be unspecified. It is determined once

```

```
% the succeeding item is known (which belongs to
% to same package and knows its end time). This
% can be done by the procedure EndTimes
```

```
proc{SetEndTimes Items}
  case Items
  of control(info:start
    package:Package
    ...) | Item | Is then
    Package.End = Item.package.End
    {SetEndTimes Is}
  [] _ | Is then {SetEndTimes Is}
  end
end
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%% For garbage collection, it is necessary
%%%% to advance the FirstItemPointer eventually
%%%% but only access to the previous items on the
%%%% stream is no more needed.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
local
  proc{AdvanceUntil TimePoint}
    {BrowseStream advanceUntil(TimePoint)}
    Stream = {FirstItemPointer.get}
  in
    if {IsDet Stream}
    then
      case Stream
      of Item | Is then
        if Item.package.End < TimePoint
        then
          {FirstItemPointer.set Is}
          {StreamSize.set {StreamSize.get}-1}
          {BrowseStreamSize}
          {AdvanceUntil TimePoint}
        else
          skip
        end
      end
    else
      skip
    end
  end
end
```

```

in
  proc{CollectUntil TimePoint}
    {Thread proc{$}
      {AdvanceUntil TimePoint}
      {BrowseStream before_gc}
      {System.gcDo}
      {BrowseStream gc_done}
    end}
  end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FetchWords fetches all words of the next package
%%%%% on a stream that end after a given time point
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

local
  proc{AdvanceTo StartTime Stream RelevantSection}
    case Stream
    of Item|Items then
      if {IsDet Item.package.End}
      then
        if Item.package.End >= StartTime
        then RelevantSection = Stream
        else {AdvanceTo StartTime Items RelevantSection}
        end
      else
        RelevantSection = Stream
      end
    end
  end
  proc{FetchNextPackage Stream Filter Words}
    {Browse 'FNP'#isdet({IsDet Stream})}
    case Stream
    of Item|Items then
      {Browse 'FNP'#{IsDet Stream}}
      case Item
      of control(info:!End ...) then % next package ended
        Words = nil
      [] control(info:start ...) then % ignore and continue
        {FetchNextPackage Items Filter Words}
      [] word(...) then % pick the word if it passes the
        % filter and continue anyway
        RestWords
      end
    end
  end
in

```



```

local

    fun{WaitFirst Stream KeyWords StartTime}
        {Browse waitFirst#isdet({IsDet Stream})}
        NextStartTime = {Max StartTime {Time.real}}
    in
        {Recognizer.start}
        {CollectUntil StartTime}
        case Stream
        of Item|Items then
            case Item
            of control(...) then
                {WaitFirst Items KeyWords NextStartTime}
            [] word(...) then
                if Item.info.start < StartTime
                then {WaitFirst Items KeyWords NextStartTime}
                else % Item is a good candidate.
                    if {Member {GetInfo Item.info} KeyWords}
                    then % Item is the first keyword
                        % garbage collection
                        {Browse 'waitfirst'(sucessful:Item)}
                        Item
                    else % Item is another word
                        {WaitFirst Items KeyWords NextStartTime}
                    end
                end
            end
        end
    end
end

in
    fun{WaitFor KeyWords StartTime}
        {BrowseDialog keyWords(KeyWords)}
    in
        {WaitFirst {FirstItemPointer.get} KeyWords StartTime}
    end
end

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% Match selects all words uttered in the first
%%%%% package that end after the StartTime and belong
%%%%% to the Words list.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

fun{Match Words StartTime}
    {Browse match(Words StartTime)}

```

```

{Recognizer.start}
{BrowseDialog match(Words StartTime)}

CandidateItems = {PickIn StartTime}
                                % may be bound late
CandidateWords
{Thread proc{$}
    CandidateWords = {Map CandidateItems
                      fun{$ Item} Item.info end}
    end}
{BrowseDialog match_candidates(CandidateWords)}
Out
{Thread proc{$}
    Out = {Filter CandidateWords
           fun{$ W}
             {Member {GetInfo W} Words}
           end }
    end}
{BrowseDialog matching_words(Out isdet:{IsDet Out})}
in
    Out
end
in
    unit(sendToStream:Send
        collectUntil:CollectUntil
        firstItemPointer:FirstItemPointer
        waitFor:WaitFor
        match:Match
        recognizer:Recognizer) % start stop quit
    end
end

```

recognizer-interface.oz

```

functor
import
    Open

R(start:ShellCmd
   makeCmds:MakeCmds)      at 'recognizer-selection.ozf'

C(toInfo:ToInfo)          at 'parse-info.ozf'
Browser                    at 'browser.ozf'
Counter                    at 'counter.ozf'
PS(pretty:Pretty)         at 'prettyStream.ozf'
Lexicon(toWord:ToWord)    at 'lexicon.ozf'

```

```

Threads(start:Thread)      at 'threads.ozf'
S(newLine:Convert)         at 'windows-special.ozf'

export
  Make % starts the speech recognizer, creates an interface to it,
        % and returns a record of procedures sending by which to
        % send commands to the recognizer
define

  BrowseSend = Browser.debug.recognizerStringSend
  BrowseReceived = Browser.debug.recognizerStringReceived
  BrowseAux = Browser.debug.recognizerInterface
  BrowseWarning = Browser.debug.recognizerWarnings

  fun{Make Send} % starts the speech recognizer and makes an
                  % interface to. The input obtained from the
                  % recognizer is forwarded by the Send
                  % procedure which is an parameter of Make.
                  %
                  % The interface receives strings from the recognizer,
                  % puts them onto the InputStream. An concurrent process
                  % converts all strings on the InputStream to data
                  % structures and sends them somewhere by the Send
                  % procedure. (In the lift application, the data structures
                  % are send to the dialog data stream)

  % start the speech recognizer

  RecognizerPipe = {New Open.pipe ShellCmd}
  proc{SendToRecognizer Cmd}
    {BrowseSend string_send_to_recognizer(Cmd)}
    {RecognizerPipe write(vs:Cmd#'\n')}
  end
  Recognizer = {MakeCmds SendToRecognizer}

  % process the input produced by the speech recognizer

  InputStreamPure
  {Thread proc{$}
    {RecognizerPipe read(size:all tail:_
                      list:InputStreamPure)}}
  end}
  InputStream = {Convert InputStreamPure}

```

```

{BrowseReceived {Value.byNeed fun{$} {Pretty InputStream} end}}
{BrowseAux string_received_from_recognizer#InputStream}

{Thread proc{$}
    {Process InputStream}
    end}

% define the processing steps

PackageCounter = {Counter.make}

fun{ConvertPhon Info}
    Phon = {Label Info}
    PairList = {Record.toListInd Info}
in
    {List.toRecord {ToWord Phon} PairList}
end

% Cut off spaces on the left
fun{LTrim Str}
    {List.dropWhile Str Char.isSpace}
end

% Map String to LowerCase
fun{LCase Str}
    {Map Str Char.toLower}
end

% main recursion

proc{Process InputStream}
    RestStream = {ProcessPackage InputStream}
    {BrowseAux getpackage_done}
in
    {Process RestStream}
end

% ProcessPackage picks the first string package from Stream0
% converts it into a data structure and sends it to the Dialog.
% ProcessPackage returns the remaining string on Stream0

fun{ProcessPackage Stream0}
    GarbageLines Stream1
    {Thread proc{$}

```

```

        {ForAll GarbageLines
          proc{$ L}
            {BrowseAux garbageLine#L}
          end}
        end}
    {Split Stream0 "begin" GarbageLines Stream1}
    {ForAll GarbageLines proc{$ Line}
      if Line==" " then skip
      else {BrowseWarning garbageLine#Line}
      end
    end}

    PackageLines Stream2
    {Thread proc{$}
      {ForAll PackageLines
        proc{$ L}
          {BrowseAux packageLine#L}
        end}
      end}
    {Split Stream1 "end" PackageLines Stream2}
    PackageInfos = {Map PackageLines
      fun{$ Line}
        {ToInfo {LCase {LTrim Line}}}}
      end}
    {SendToDialog PackageInfos}
  in
    Stream2
  end

% Splits the Stream into those lines before and
% after the Str

proc{Split Stream Str BeginLines EndStream}
  WholeLine RestStream
  {String.token Stream &\n WholeLine RestStream}
  Line = {LTrim WholeLine}
in
  if Line == Str
  then
    EndStream = RestStream
    BeginLines = nil
  else
    NextBeginLines
  in
    BeginLines = Line|NextBeginLines
    {Split RestStream Str NextBeginLines EndStream}
  end
end

```

```

    end
  end

  % SendToDialog gets a package as a list whose first
  % element determines the of kind of the package (control
  % or utterance) and some auxiliary information. The remaining
  % elements specify the content of the package (i.e. the words
  % of the utterance.
  %
  % SendToDialog converts the input strings into records
  % of one of the following type where the type Word is
  % determined by the speech recognizer:
  %
  %   control(topic:start
  %           time:Float
  %           number:Int)
  %
  %   utterance(words:List(Word)
  %             start:Float
  %             end:Float
  %             number:Int)
  %
  proc{SendToDialog PackageInfos}
    Kind|Infos = PackageInfos
    {BrowseReceived sendToDialog_called(Kind)}
  in
    case Kind
    of control(topic:statechange old:'5' new:'6' time:Time ...) then
      {Send control(topic:utteranceStarted
                    time:Time
                    number:{PackageCounter.next})}
    [] control(topic:statechange old:'6' new:'5' time:Time ...) then
      {Send control(topic:utteranceEnded
                    time:Time
                    number:{PackageCounter.next})}
    [] rec(start:Start 'end':End ...) then
      {Send utterance(words:{Map Infos ConvertPhon}
                      number:{PackageCounter.next}
                      start:Start
                      'end':End)}
    else
      {BrowseWarning 'strange packages'}
      {ForAll PackageInfos BrowseWarning}
    end
  end
end

```

```

    in
        Recognizer
    end
end

```

lift-control.oz

```

functor
import
    I(fahre:Fahre
        getInfo:GetInfo)    at 'lift-control-interface-make.ozf'
    Parameter(
        control:Control
        infos:Infos)        at 'parameter.ozf'
    Browser                  at 'browser.ozf'
    T(start:Thread)         at 'threads.ozf'

export
    Make
define

    Browse = Browser.debug.control

    unit(waitTime:WaitTime
        repeatTime:RepeatTime ...) = Control

    class WaitClass
        attr active    % the number of active wait requests
        prop locking
        meth init
            active <- 0
        end
        meth wait(MilliSeconds)
            active <- @active+1
            {Thread proc{$}
                {Time.delay MilliSeconds}
                active <- @active-1
            end}
        end
        meth free($)
            @active==0
        end
    end

    fun{Make Wizard}
        LiftStream % commands send to the lift are
                   % send to the LiftStream.
    end
end

```

```

LiftPort = {NewPort LiftStream}
WaitOffice = {New WaitClass init}
    % The InfoLoop processes commands
    % on the ListStream if there are any.
    % Otherwise, the state of the lift
    % is asked for.
proc{InfoLoop Stream}
    if {IsDet Stream}
    then
        case Stream
        of fahre(X)|Rest then
            _ = {Fahre X}          % antwortet ok
        in
            {InfoLoop Rest}
        [] wait|Rest then
            {WaitOffice wait(WaitTime)} % milliseconds
            {InfoLoop Rest}
        end
    else
        if {WaitOffice free($)}
        then
            Info = {GetInfo}
        in
            {Browse [lift_info Info]}
            if {Label Info} == info
            then
                {Record.forAllInd Info
                proc{$ Key Value}
                    ConvValue = if {Member Key [Infos.floor Infos.goal]}
                        then {String.toInt {Atom.toString Value}}
                        else Value
                    end
                in
                    {Wizard.memo.set Key ConvValue}
                end}
            else
                skip
            end
        else
            skip
        end
        {Time.delay RepeatTime}
        {InfoLoop Stream}
    end
end
end

```



```

        {Thread proc{$}
            try
                {InfoLoop LiftStream}
            catch All then
                {Browser.debug.error
                 error(lift_control)#exception(All)}
            end
        end}

    in
        unit(fahre:proc{$ X} {Port.send LiftPort fahre(X)} end
            wait: proc{$} {Port.send LiftPort wait} end)
    end
end

```

lift-control-interface.oz

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% creates an interface to either the lift-control
% which is either of the following:
%
% nlf401 or fake-lift-control
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

functor
import
    Browser at 'browser.ozf'
    Open
    Commands(fahre:Fahre
              getInfo:GetInfo
              toInfo:ToInfo) at 'lift-commands.ozf'
    Parameter(control:Control) at 'parameter.ozf'
export
    Make
define
    Browse = Browser.debug.control
    unit(shellCmd:ShellCmd ...) = Control
    class TextPipe
        from Open.pipe Open.text
    end

    fun{Make}
        Pipe = {New TextPipe ShellCmd}
        fun{ReadLine}

```

```

        {Pipe getS($)}
    end
    fun{Send Cmd}
        {Browse ['lift control command:' Cmd]}
        {Pipe putS(Cmd)}
        AnswerStr = {ReadLine}
        Answer = {ToInfo AnswerStr}
        {Browse ['lift control answer:' Answer]}
    in
        Answer
    end
    proc{Close} {Pipe close} end
in
    unit(close:Close
        fahre:fun{$ Stock}
            {Send {Fahre Stock}}
        end
        getInfo:fun{$}
            {Send GetInfo}
        end)
    end
end
end

```

parameter.oz

```

functor
import
    Lex(words:Words) at 'lexicon.ozf'

export
    Experiment
    SoundWords
    Time
    Sound          % loud speaker
    Recognizer     % for speech input
    Control        % of the lift
    ControlInterface
    ControlInfoType
    ControlTicketURL
    Font
    Infos
    Debug
    MemoType
    MemoInit
    Protocols
    TimeOut

```

```

    Home          % of lift control server
    User          % of lift control server
define

%%%%%%%% main parameter %%%%%%%%%%%%%%

CONTROL = real          % fake %real %none
ControlInterface = 'internet' % 'local' % internet
ControlLocation = 'coli' % coli % ps
Recognizer = 'LH'       % 'HTK' % 'LH' % fake
Audio = windows         % windows % linux
TimeOut = 3.000         % seconds until an answer is
                        % expected in the lift

% starting the lift control by User at Home

Home = '/home/CE'       % '/home/MP' % '/home/CE'
User = 'pfleger'       % pfleger % niehren

%%%%%%%% fonts %%%%%%%%%%%%%%

Font = '*-helvetica-bold-r*--*-180-*'

%%%%%%%% Sound %%%%%%%%%%%%%%

Mensch ='Mensch'
Maschine='Maschine'
Spez='Spezifisch'
Unspez='Unspezifisch'

Experiment =
unit(mensch:Mensch
     machine:Maschine
     spez:Spez
     unspez:Unspez)
SoundWords =
unit(etage: 'Stock'    % Etage or Stockwerk
     keller:'Keller'
     wohin: 'Frage wohin?'
     bereits: 'bereits im'
     fahre: 'Fahre')

%%%%%%%% Auxiliary files and directories %%%%%%%%%

local

```

```

SoundLinux    = unit(dir:'../Synthesis/Sound3/'
                    player:'play_wav_gz')

SoundWindows = unit(dir:'c:\\Synthesis\\Sound3\\'
                    player:'wav_wav')

SoundVariants = unit(linux:SoundLinux
                    windows:SoundWindows)
in
    Sound =SoundVariants.Audio
end

Protocols = 'Protocols'

%%%%%%%% lift control used %%%%%%%%%

local
    RealControl =
        unit(shellCmd:init(cmd:'/usr/local/bin/nlf401'
                            args:nil)
              waitTime:1000    % milliseconds
              repeatTime:1000) % milliseconds

    FakeControl =
        unit(shellCmd:init(cmd:'ozengine'
                            args: ['fake.ozf' '--control'])
              waitTime:500     % milliseconds
              repeatTime: 500) % milliseconds

    ControlVariant = unit(real:RealControl
                          fake:FakeControl
                          none:unit)
in
    Control = ControlVariant.CONTROL
end

local
    Coli='http://www.coli.uni-sb.de/~'#User#/lift-control-ticket'
    Ps  ='http://www.ps.uni-sb.de/~'#User#/lift-control-ticket'
    ControlTicketURLs = unit(coli:Coli ps:Ps)
in
    ControlTicketURL = ControlTicketURLs.ControlLocation
end

```

```
%%%%%%%%%% Informations of the lift control %%%%%%%%%%
```

```
Goal=ziel
Door=drehtuer
Door2=kabinentuer
Floor=pos      % Position
Dir=richtung
```

```
Infos = unit(goal:Goal
              door2:Door2
              door:Door
              floor:Floor
              dir:Dir
              auf:auf
              zu:zu)
```

```
%%%%%%%%%% Memory of the lift %%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
ControlInfoType =
info(Goal:oz(int)
     Door2:list([auf zu])      % kabinentuer
     Door:list([auf zu])      % drehtuer
     Floor:oz(int)            % position
     Dir:list([auf ab keine]) % richtung
    )
```

```
DialogMemoType =
unit(dest:list(Words.dest)
     0:oz(bool)
     1:oz(bool)
     2:oz(bool)
     3:oz(bool)
     4:oz(bool)
     5:oz(bool)
     time:oz(float)
     mode:list([Mensch Maschine])
     ques:list([Spez Unspez])
    )
```

```
ControlInfoInit =
unit(Goal:3
     Door2:zu      %kabinentuer
     Door:zu       %drehtuer
     Floor:1       % pos
```

```

        Dir: auf)      %richtung
DialogMemoInit =
unit(mode:Mensch
    ques:Spez
    0:false
    1:false
    2:false
    3:false
    4:false
    5:false)

```

```

MemoType = {Adjoin ControlInfoType DialogMemoType}
MemoInit = {Adjoin ControlInfoInit DialogMemoInit}

```

%%%%%%%% Browser configuration for debugging %%%%%%%%%%

```

DebugNone
= unit(control:false
    recognizer:false
    controlStringSend:false
    controlStringReceived:false
    recognizerStringSend:false
    recognizerStringReceived:false
    recognizerData:false
    recognizerInterface:false
    dialog:true
    fake:false
    parse:false
    error:true
    recognizerWarnings:false
    history:false
    runningProtocol:false
    memory:false
    threads:false)
DebugTest
= unit(control:false
    controlStringSend:false
    controlStringReceived:false
    fake:false
    recognizerStringSend:false
    recognizerStringReceived:false
    recognizerInterface:false
    recognizer:false

```

```
recognizerData:false
dialog:false
parse:false
error:true
recognizerWarnings:false
history:false
runningProtocol:false
memory:false
threads:false)
Variants = unit(test:DebugTest
                none:DebugNone)

Debug = Variants.none

end
```