Integration von ISDN-Teilnehmern in Internet-Multimedia-Konferenzen

Frank Oertel
Mat.Nr. 94301
TU-Berlin

Berlin den
24. Februar 1997

Diplomarbeit am
Fachgebiet für Offene Kommunikationssysteme (OKS)
des Fachbereichs Informatik (FB13)
der Technischen Universität Berlin

Betreut durch
Prof. Adam Wolisz,
TU-Berlin, GMD Fokus, Berlin
Prof. Henning Schulzrinne,
GMD Fokus, Berlin / Columbia University, New York

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Zusammenfassung

Der zunehmende Ausbau des Internets sowie die Entwicklung von Transportprotokollen zur Übertragung von multimedialen Echtzeitdatenströmen fördert zugleich den Ausbau des Multimedia Backbones (MBONE), also dem Teil des Internets, in dem multimediale Konferenzen durch den Einsatz von IP-Multicast geführt werden können. So werden beispielsweise seit einigen Jahren ein Teil der Space-Shuttle-Missionen der NASA im MBONE mit Ton und Bild übertragen. Dennoch haben nicht alle Internet-Benutzer die Möglichkeit, an MBONE-Konferenzen teilzunehmen, sei es, weil sie sich an einem Ort befinden, an dem es keinen Internet-Zugang gibt, oder aber Verbindungen ins Internet benutzen, über die keine MBONE-Konferenzen gerouted werden. Letzteres betrifft vor allem die, auch in Deutschland stark zunehmende Zahl derjenigen Benutzer, die über Verbindungen niedriger Bandbreite, z.B. per Modem, mit dem Internet verbunden sind. Um diesen Benutzergruppen dennoch Zugang zu MBONE-Konferenzen zu ermöglichen, wird in dieser Arbeit eine Kopplung von Telefonnetz und Internet entwickelt und realisiert, deren Anwendungsmöglichkeiten weit über eine bloße Konferenzteilnahme per Telefon hinausgehen.

Neben einem Call-Server, der die Kopplung von Telefonnetz und Internet realisiert, wird eine Skriptsprache entwickelt und implementiert, mit deren Hilfe sich allgemeine Telefonieanwendungen vom einfachen Anrufbeantworter bis hin zu Call-Center-Anwendungen realisieren lassen. Neben der eigentlichen Kopplung beider Netze, die letztlich nur in der Anpassung der unterschiedlichen Datenformate besteht, werden Aspekte wie Anrufsignalisierung, Zugang zu Informationen über aktuelle Konferenzen sowie deren Auswahl am Telefon behandelt.

Zum Inhalt dieser Ausarbeitung

Nach einer genaueren Betrachtung der Problembereiche im nächsten Kapitel, werden diese in Kapitel 3 weiter analysiert. Kapitel 4 stellt schließlich den Call-Server und Kapitel 5 die Skriptsprache vor. Nachdem in Kapitel 6 auf die eigentliche Implementierung eingegangen wurde, werden in Kapitel 7 beispielhafte Anwendungen gezeigt, die mit Hilfe der Skriptsprache realisiert werden können. Eine abschließende Bewertung der Ergebnisse findet im letzten Kapitel 8 statt.

Kapitel 2

Aufgabenstellung

2.1 Einleitung

Ziel dieser Arbeit ist es, Telefonteilnehmern den Zugang zu Multimediakonferenzen im Internet zu ermöglichen. Auch umgekehrt soll es möglich sein, Anrufe aus dem Internet zu einem gewöhnlichen Telefonanschluß weiterzuleiten. Im Rahmen dieser Arbeit sind mit Multimediakonferenzen innerhalb des MBONE übertragene Audio- und Videokonferenzen gemeint. Um beides zu ermöglichen, ist mehr als die Verbindung von Telefon- und Netzwerkkabel erforderlich, wie folgende Fragen zeigen:

- Wie gelangt ein Anrufer aus dem Telefonnetz Kenntnis darüber, welche Konferenzen gerade „laufen“?
- Wie trifft er mit den ihm zur Verfügung stehenden Signalisierungsmitteln (Telefontastatur) seine Auswahl?
- Wie kann er vom Telefon aus Internet-Teilnehmer zu einer Konferenz einladen und umgekehrt?
- Wie werden die unterschiedlichen Formate der Mediendatenströme gehandhabt?

Abbildung 2.1 skizziert die angestrebte Kopplung von Internet und Telefonnetz.

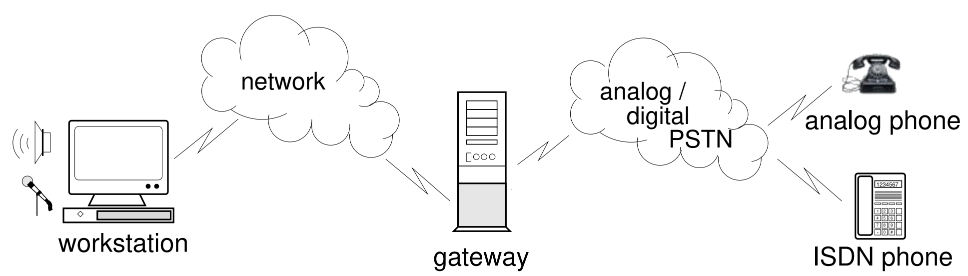

Abb. 2.1: Überblick

Dabei muß der Telefonteilnehmer nicht zwangsläufig über einen ISDN-Anschluß verfügen, in der Grafik werden daher analoges und digitales (ISDN) Telefonnetz gezeigt (PSTN=Public Switched Telephone Network). Lediglich auf der Netzwerkseite wird ein ISDN Basis- (Basic Rate Interface BRI) oder Primärmultiplexanschluß (Primary Rate Interface PRI) vorausgesetzt, da die digitale Natur des ISDN-

Anschlusses die Anbindung an ein Rechnernetz vereinfacht.

Der Schwerpunkt dieser Arbeit liegt also nicht in der Erhöhung des Telefonkomforts sondern vielmehr darin, Telefonteilnehmern, die augenblicklich nicht über einen IP-multicast-fähigen Internetanschluß verfügen, den Zugang zu MBONE-Konferenzen zu ermöglichen. In dieser Situation befindet sich insbesondere die permanent steigende Zahl der privaten Internet-Nutzer, die über serielle Leitungen per Modem oder ISDN mittels SLIP oder PPP über einem Provider mit dem Internet verbunden sind. Neben der zu geringen Bandbreite dieser Verbindungsarten besteht das Hauptproblem darin, daß IP-Multicast-Ströme im allgemeinen nicht über SLIP/PPP-Verbindungen gerouted werden.

2.2 Unterstützte Szenarien

Es werden folgende Szenarien unterstützt:

2.2.1 Passive Konferenzteilnahme per Telefon

Ein Anrufer aus dem Telefonnetz kann passiv an einer Multimedia-Konferenz teilnehmen, d.h. der Sprachkanal des Anrufers wird bidirektional mit dem Audiostrom einer entsprechenden Konferenz unter Berücksichtigung der unterschiedlichen Formate verbunden. So können beispielsweise Studenten eine bestimmte Nummer anrufen und einer Vorlesung zuhören.

2.2.2 Aktive Teilnahme per Telefon

Zusätzlich ist es möglich, per Telefon beliebige (also auch mehr als einen) Netzwerkteilnehmer zu einer Konferenz einzuladen. So kann ein Anrufer durch Angabe einer e-mail-Adresse oder einer anderen personenbezogenen Identifikation einen Benutzer an seiner Workstation anrufen. Während des folgenden Gespräches können weitere Teilnehmer eingeladen werden.

Abb. 2.2: Anrufsignalisierung durch den Conference-Controller

Für das Einladen von Netzwerkbenutzern in Multimedia-Konferenzen dient u.a. das Session Initiation Protocol SIP [HSS96]. Durch die Verwendung von e-Mail-Adressen realisiert SIP eine personenbezogene Adressierung der Teilnehmer. Der Einladungswunsch wird in Form eines SIP-Requests an einen User-Agent oder einen Proxy-Server geschickt, welche die Aufgabe haben, den adressierten Benutzer ausfindig und auf die Einladung aufmerksam zu machen. Die Signalisierung des Anrufes an der Workstation erfolgt durch einen Conference-Controller, der in einem Fenster auf den Anruf aufmerksam macht (siehe Abbildung 2.2). Der Angerufene kann den Anruf durch Drücken von `Accept` annehmen oder mit `Reject` ablehnen.

2.2.3 Zugang zu Konferenzinformationen

Sofern einem Anrufer die Parameter einer Multimedia-Konferenz wie IP-Adresse, Portnummer, etc. nicht vorab bekannt sind, muß er sich einen Überblick über gerade laufende Konferenzen und deren Parameter verschaffen können.

Zur Ankündigung von Multimedia-Konferenzen wird im MBONE das Session Description Protocol SDP [HJ96] in Verbindung mit dem Session Announcement Protocol SAP [Han96] verwendet. Die hier propagierten Informationen müssen für potentielle Anrufer gesammelt und aufbereitet werden. Steht nur eine Audioverbindung zur Verfügung, so müssen dem Anrufer diese Informationen per Sprachsynthese vorgelesen werden.

2.2.4 Anrufe vom Internet ins Telefonnetz

Von seiner Workstation aus, kann ein Netzwerkbenutzer einen Telefonteilnehmer anrufen, also zu einer Konferenz einladen. Für beide Teilnehmer bleibt dies transparent, d.h. speziell für den Anrufer gibt es keinen sichtbaren Unterschied, ob er einen anderen Netzwerkbenutzer oder aber einen Teilnehmer im Telefonnetz einlädt/anruft. Da der Netzwerkbenutzer den gewünschten Teilnehmer mit Hilfe seiner e-Mail-Adresse adressiert, muß diese von einem Proxy-Server in eine Rufnummer im Telefonnetz umgewandelt werden. Bei Rufannahme erfolgt wie bereits erwähnt eine Kopplung der Mediendatenströme von ISDN und Internet.

2.2.5 Allgemeine Telefonieanwendungen

Neben der Teilnahme an Konferenzen kann der Anrufer auch auf andere Informationen zugreifen. So kann er beispielsweise seine Mailbox abfragen und Nachrichten hinterlassen oder sich zufaxen lassen.

Kapitel 3

Problemanalyse

3.1 Einleitung

Die im vorherigen Kapitel aufgestellten Anforderungen beinhalten also eine Kopplung von ISDN und Internet. Benötigt wird eine Instanz, die beide Netze miteinander verbindet, Anrufe von beiden Seiten behandelt und die Mediendatenströme entsprechend umwandelt, Abbildung 3.1 skizziert dies.

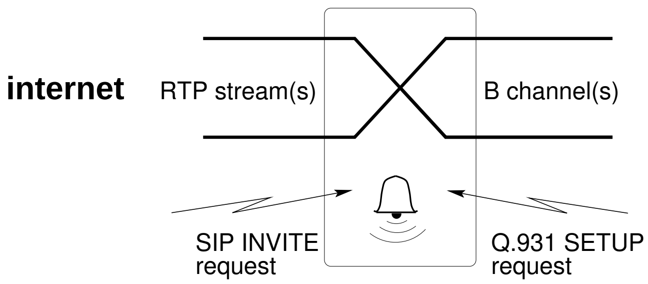

Abb. 3.1: Internet-ISDN Kopplung allgemein

3.2 Internet Multimediakonferenz

Zur Übertragung von Multimedia-Konferenzen im Internet findet inzwischen vorwiegend das Real-Time Transport Protocol RTP [SCFJ96] Anwendung. Dabei können in einem RTP-Strom mehrere Mediendatenströme (z.B. Audio und Video) transportiert werden. Da auch im ISDN die Möglichkeit besteht, verschiedene Mediendatenströme über eine Verbindung zu transportieren, ist eine Teilnahme per Telefon auch an solchen Internet-Konferenzen sinnvoll, in denen nicht nur Audio sondern auch Video benutzt wird. Lediglich die Formate von RTP-Strom und B-Kanal müssen angepaßt werden, sofern sie unterschiedlich sind. Während im europäischen ISDN Audiodaten A-law codiert (in den U.S.A. μ-law) mit einer Abtastrate von 8kHz übertragen werden, findet für die Audiodatenübertragung im MBONE eine breite Palette von Formaten wie GSM, lineare PCM, A-law, μ-law, DVI oder LPC bei unterschiedlichen Abtastraten Anwendung.

Wenngleich derzeit die Mehrheit der Telefonanschlüsse noch analoger Natur sind und die Preise für

Bildtelefonie-Endgeräte ihrer flächendeckenden Verbreitung noch im Wege stehen, sollte die zu realisierende Architektur die Konferenzteilnahme mit mehreren Mediendatenströmen prinzipiell unterstützen. Alle weiteren Betrachtungen in dieser Arbeit beschränken sich jedoch nur auf Audiodatenströme.

3.3 Telefonische Konferenzteilnahme

Während also die Kopplung der Mediendatenströme von ISDN und Internet lediglich eine Frage der Formatwandlung ist, stellt sich die Frage, welche Signalisierungsmöglichkeiten nötig sind, um telefonisch an einer Internet-Konferenz teilzunehmen bzw. was sonst noch benötigt wird.

Generell (also nicht nur telefonisch) erfordert die passive Teilnahme an einer unverschlüsselt per RTP übertragenen Konferenz, die Kenntnis von IP-Gruppen-Adresse (bei einem Multicast-Datenstrom) sowie der UDP-Portnummer. Passive Teilnahme soll bedeuten, daß der Anrufer selbst keine Teilnehmer einlädt, sondern von sich aus oder aufgrund einer Einladung der Konferenz beitritt. IP-Adresse und Portnummer können prinzipiell erlangt werden durch:

1. Einen anderen Teilnehmer, der in einem Einladungswunsch die Konferenzparameter überträgt, z.B. in einem SIP-INVITE-Request,
2. durch ein Session Directory, ein Dienst, der Informationen über aktuelle oder angekündigte Konferenzen ähnlich einer Programmzeitschrift sammelt, oder aber
3. über andere Kommunikationswege, wie Telefon oder Fax.

Im ersten Fall ist das Problem bereits gelöst, da der Anrufer aus dem Internet die Kopplung von RTP-Strom und ISDN-Strom bereits vollzogen hat; dem Anrufer kann es egal sein, mit welchen Parametern der Datenstrom in irgendeinem Teil des Übertragungsweges geführt wird.

3.4 Zugang zum Session Directory (SD)

Session Directories sind Programme, die im Internet übertragene Konferenzankündigungen sammeln und über eine entsprechende Schnittstelle einem Netzwerkbenutzer anbieten. Einem Anrufer steht dieser Dienst also zunächst nicht zur Verfügung. Da aber außer dem dritten Weg, keine anderen Möglichkeiten existieren, die Konferenz-Parameter zu erlangen, wäre es für den Anrufer sehr hilfreich, auf diese Informationen zugreifen zu können.

Konferenzankündigungen werden mit kommerziellen Programmen wie IP/TV[1] oder frei erhältlichen wie SDR[2] via IP-Multicast [Dee89] periodisch[3] im MBONE verschickt. Als Protokoll hat sich hierfür SDP durchgesetzt. Im folgenden SDP-Paket wird durch den Benutzer „Andrew Patrick“ eine Session mit Audio und Video angekündigt:

[1] http://www.precept.com
[2] http://ugwww.ucs.ed.ac.uk/mice/archive/sdr.html
[3] bei meinen Messungen in der GMD alle 7 bis 13 Minuten

```
v=0
o=andrew 3059744132 3059744250 IN IP4 131.136.234.103
s=CBC Newsworld Online
i=NEWSWORLD is Canada's 24-hour all news and information
specialty network. If you view this session, PLEASE
complete the questionnaire at
http://debra.dgbt.doc.ca/mbone/newsworldq.html.
NEWSWORLD is a non-profit service dedicated wholly
to news and inf...
u=http://debra.dgbt.doc.ca/mbone/newsworld.html
e=Andrew Patrick <andrew@spiff.crc-mcast.drenet.dnd.ca>
p=Andrew Patrick +1 613 990-4675
c=IN IP4 224.2.222.241/127
t=3059742600 3062161800
a=tool:sdr v2.2a22
a=type:broadcast
m=audio 24992 RTP/AVP 0
c=IN IP4 224.2.222.241/127
a=ptime:40
m=video 64512 RTP/AVP 31
c=IN IP4 224.2.221.77/127
```

Auskunft über den Inhalt der angekündigten Session geben die Felder s= (session name) und i= (session information). Das Feld m= (media name and transport address) enthält die für die Konferenzteilnahme erforderlichen Medienparameter, in dem gezeigten Beispiel also Audio (m=audio) und Video (m=video), das Feld t= gibt Start- und Stoppzeit der Session gemäß [Mil92] an.

Aufgrund der Tatsache, daß sich die Session-Informationen ständig ändern, bleibt die Sprachsynthese der einzige Weg, einem Anrufer diese Informationen *aktuell* zu präsentieren. Da die Ankündigungen in relativ großen Abständen (mehrere Minuten) ausgeschickt werden, müssen diese für einen potentiellen Anrufer in einer Art Cache gesammelt werden, sofern dieser nicht minutenlang am Telefon warten will.

Bei Messungen bei FOKUS GMD konnte ich an einem Tag zwischen 09^{48} und 16^{20} Uhr also innerhalb von $6^1/_2$h 1364 Session-Announcements empfangen. In diesen wurden 24 verschiedene Konferenzen angekündigt, im Mittel wurde also jede Konferenz $\frac{1364}{24} \approx 56$ mal, also $\frac{390min}{56} \approx 7$ mal pro Stunde, d.h. alle 8 bis 9 Minuten angekündigt.

Angesichts der relativ geringen Zahl von 24 Konferenzen könnte man meinen, daß einem Anrufer einfach alle aufgefangenen Ankündigungen vorgelesen werden. Mittels Telefontastatur (siehe 3.5.1) kann er dann die gewünschte Konferenz wählen. Natürlich sollte sich das Vorlesen auf den Inhalt der s= Felder beschränken. Nur auf Wunsch (Tastendruck) sollten i=, t= und m= Felder vorgelesen werden. An anderen Orten, vor allem solchen, an denen das MBONE besser ausgebaut ist als in Deutschland, sind größere Zahlen zu erwarten. Geht man aufgrund des Ausbaus von Internet und MBONE auch bei uns von einer Zunahme der Announcements aus, so scheint das Vorlesen aller Ankündigungen nicht zweckmäßig.

Die Menge der Konferenzankündigungen kann reduziert werden, indem nur solche Sessions vorgelesen werden, an denen eine telefonische Teilnahme sinnvoll möglich ist. Also reine Video-Sessions (ohne Audio) sollten nicht vorgelesen werden, wenn der Anrufer nur einen Audiokanal benutzt. Auch sollten nur gegenwärtig oder in naher Zukunft (z.B. nur am gleichen Tage) stattfindende Sessions vorgelesen werden.

Eine weitere Selektionsmöglichkeit bietet das TTL-Feld in der c= Zeile, mit dem der Verbreitungsbereich der Konferenz festgelegt wird. So könnte man sich vorstellen, bei einem Anruf in der Uni nur die Teilnahme an internen Konferenzen (TTL < 16) zuzulassen.

Eine weitere Möglichkeit, an diese Informationen zu gelangen besteht darin, sie sich zufaxen zu lassen. Auf dem Fax sind die angekündigten Sessions mit einer zwei- oder dreistelligen Nummer versehen, die der Anrufer eingeben muß, wenn er verbunden werden will.

3.5 Signalisierungsmöglichkeiten

Für eine telefonische Konferenzteilnahme benötigt der Anrufer also:

1. Einen Telefonanschluß und ein Endgerät, das die Teilnahme an dem gewünschten Mediendatenstrom unterstützt.
2. Die Telefonnummer eines Service-Providers, der die zu realisierende Internet-Anbindung möglichst *kostengünstig* bereitstellt.
3. Die IP-Adresse sowie die Portnummer der gewünschten Konferenz.
4. Eine Möglichkeit, beide Parameter am Telefon einzugeben.

Letzteres wird hier als Signalisierung bezeichnet und wird z.B. auch dann gebraucht, um während des Gespräches weitere Teilnehmer hinzuzubeten oder die Konferenz zu wechseln. Für die in 2.2.5 geforderte Realisierung allgemeiner Telefonieanwendungen wird eine generelle Möglichkeit benötigt, mit der ein Anrufer Eingaben am Telefonapparat vornehmen kann, z.B. um aus einer Liste von Möglichkeiten eine Auswahl zu treffen.

3.5.1 DTMF-Ton-Erkennung

Dem Anrufer, der telefonisch an einer Multimedia-Konferenz teilnehmen will, steht für die Signalisierung letztendlich nur die Tastatur des Telefons sowie die Sprache zur Verfügung. An modernen Telefonapparaten werden die während einer Verbindung gedrückten Tasten in Mehrfrequenztöne nach ITU-T Q.23/Q.24 [ITU89b] [ITU89a] umgewandelt und über den Sprachkanal zur Gegenstelle geschickt. Diese muß dann die Töne aus dem Datenstrom des Sprachkanals herausfiltern bzw. erkennen. Es handelt sich hierbei also um eine In-Band-Signalisierung. Um eine hohe Erkennungswahrscheinlichkeit zu gewährleisten, sind die Frequenzen so gewählt, daß sie möglichst nicht durch Sprache o.ä. imitiert werden können. Ein Ton wird dabei aus zwei Frequenzen gebildet, die jeweils aus einer Gruppe zu je vier Frequenzen entstammen (daher auch Dual-Tone Multiple-Frequency, kurz DTMF). Nach Q.23 muß ein Ton eine Mindestlänge von 40 ms haben, der Abstand zwischen zwei Tönen muß mindestens 20 ms betragen. Tabelle 3.1 zeigt die Frequenzen für die 16 definierten Tasten. Neben den zehn Zifferntasten sowie Stern und Raute gibt es vier weitere Tasten $\boxed{\texttt{A}}$ bis $\boxed{\texttt{D}}$, die für allgemeine Steuerungszwecke gedacht, auf den meisten Telefontastaturen jedoch nicht zu finden sind.

Für die Erkennung der Töne beim Empfänger wird aus Effizienzgründen oft Hardware basierend auf digitalen Signalprozessoren (DSPs) eingesetzt. Für den Einsatz in einem reinen Software-System ist dies wenig geeignet. In [AESP96][4] werden verschiedene Algorithmen zur Erkennung von DTMF-Tönen vorgestellt und in Hinblick auf Implementierungsaufwand und Normkonformität untersucht. Einer der

[4] http://ptolemy.eecs.berkeley.edu/papers/96/dtmf_ict/www/paper.html

	1209 Hz	1336 Hz	1477Hz	1633 Hz
697 Hz	1	2	3	A
770 Hz	4	5	6	B
852 Hz	7	8	9	C
941 Hz	*	0	#	D

Tabelle 3.1: Frequenz-Tastenzuordnung nach ITU-T Q.23

untersuchten Algorithmen ist der Goertzel-Algorithmus, der aufgrund seiner einfachen Implementierung in dieser Arbeit in einem DTMF-Dekoder zum Einsatz kommt.

3.5.2 Spracherkennung

Die sicher natürlichste Art der Signalisierung bietet die Spracherkennung, welche in zahlreichen kommerziellen Produktion heute bereits Anwendung findet. Ein Beispiel hierfür ist das Produkt VoiceType von IBM, das u.a. auch in dem Betriebssystem OS/2 der gleichen Firma zum Einsatz kommt. Während die Erkennungsgenauigkeit und -Geschwindigkeit als ausreichend bezeichnet werden kann, verhindert die bei diesem Produkt erforderliche Lernphase der Software den Einsatz in einer Telefonieanwendung. Bei der Suche nach frei verfügbarer Spracherkennungs-Software bin ich leider nur auf kommerzielle Produkte gestoßen, wie den folgenden:

- IBM VoiceType (http://www.software.ibm.com/is/voicetype/)
- VoiceAssist from Creative Labs (http://www.creaf.com/)
- Kurzweil Voice for Windows (http://www.kurzweil.com/)
- PureSpeech 2.0 Recognition Engine (http://www.speech.com/)
- DragonDictate (http://www.dragonsys.com/)

Eine auf einem DSP basierende Hardware-Lösung namens Antares, wird von Dialogic[5] angeboten. Auf dieser Plattform können mit Hilfe verschiedener Toolkits Telekommunikationslösungen realisiert werden. Zur Sprachsynthese wird hierfür beispielsweise von PureSpeech die Entwicklungsumgebung ReCite[6] angeboten.

Da mir kommerzielle Produkte jedoch nicht zur Verfügung standen und die Komplexität eine eigene Implementierung ausschließt, findet Spracherkennung in dieser Arbeit keine praktische Anwendung. Ihr Einsatz wird jedoch in meiner Implementierung berücksichtigt.

3.6 Sprachsynthese

Da für den Dialog mit einem Anrufer aus dem Telefonnetz die Audioverbindung als einziger Kommunikationskanal existiert, müssen alle Informationen verbal ausgegeben werden. Soll der Anrufer z.B.

[5] http://www.dialogic.com/
[6] http://www.speech.com./recite.html

nach der Konferenz gefragt werden, an der er teilnehmen will, so muß ein entsprechender vorab aufgenommener Text abgespielt werden. Während dies für einfache Dialoge wie die bei einem Anrufbeantworter ausreicht, so genügt dies für die Realisierung komplexer Informationsdienste, wie dem geplanten MBONE-Informationsservice nicht mehr. Die Aktualität der dem Anrufer gebotenen Informationen kann nur gewahrt werden, wenn die Sprachausgabe spontan, also mittels Sprachsynthese erzeugt wird. Hierfür gibt es zahlreiche kommerzielle und nicht-kommerzielle Produkte, von denen stellvertretend einige aufgezählt werden sollen:

- Rsynth (http://wwwtios.cs.utwente.nl/say)
- ORATOR by Bellcore (http://www.bellcore.com/ORATOR/)
- SENSYN by Sensimetrics Corporation (http://www.sens.com/)
- TruVoice from Centigram (http://www.centigram.com/)

Das Angebot an frei erhältlicher Software ist im Gegensatz zur Spracherkennungs-Software besser, und so konnte ich einige Programme ausprobieren. In der Sprachqualität zeigen sich sehr große Unterschiede, die besten Ergebnisse erzielte ORATOR II eine Entwicklung von Bellcore, die allerdings nicht frei erhältlich ist. Die freie Verfügbarkeit des Quellcodes des Rsynth-Paketes gestattete eine Portierung auf die zur Entwicklung benutzte AIX-Umgebung und kam somit versuchsweise zum Einsatz. Die Sprachqualität dieses Paketes ist erwies sich jedoch als ungenügend. Der Einsatz anderer Pakete scheiterte an der Nichtverfügbarkeit des Sourcecodes. Meine Implementierung unterstützt jedoch den Einsatz beliebiger Sprachsynthese-Software, da für die Synthese lediglich eine Kommandozeile ausgeführt wird, in der letztendlich jedes beliebige Programm oder Shell gestartet werden kann.

3.7 Eingabe von e-Mail-Adressen per Telefon

Um wie in 2.2.2 gefordert, Netzwerkbenutzer per Telefon anrufen, also zu einer Konferenz einladen zu können, bedarf es einer entsprechenden Teilnehmeradressierung. Für Einladungen soll das bereits erwähnte Session Invitation Protocol SIP zum Einsatz kommen. Im Gegensatz zum Telefonnetz unterstützt SIP eine personenbezogene Adressierung bei der eine Kombination aus User-Id und Rechner- bzw. Domain-Name [Moc87], [Moc87] verwendet wird, so wie sie in Form von e-Mail-Adressen im Internet seit langem in Gebrauch ist [Cro82]. Mittels eines SIP-INVITE-Requests kann ein Benutzer also durch Angabe seiner e-Mail-Adresse zu einer Konferenz eingeladen werden. Die e-Mail-Adresse kann sowohl in der Form $username@host.domain$ als auch $username@domain$ angegeben werden.

Dem Anrufer, der mittels SIP einen Netzwerkbenutzer anrufen will, muß also eine Möglichkeit erhalten, die e-Mail-Adresse des Teilnehmers per Telefon einzugeben. Soll dies nicht per Spracherkennung sondern per Tastatur geschehen, so muß die bereits erwähnte DTMF-Tonerkennung zum Einsatz kommen, welche jedoch nur 16 Zeichen unterstützt. Wie e-Mail-Adressen trotz des wesentlich größeren Zeichenvorrates mittels DTMF-Tasten am Telefon eingegeben werden können, sollen die folgenden Überlegungen zeigen.

RFC 822 [Cro82] erlaubt folgende Zeichen in einem Adressfeld:

```
atom     ::= 1*<any CHAR except specials, SPACE and CTLs>
char     ::= <any ASCII character>
specials ::= "(" | ")" | "<" | ">" | "@" | "," | ";" |
             ":" | """ | "." | "[" | "]"
ctl      ::= <any ASCII control>
```

Von den 128 im ASCII-Zeichensatz definierten Zeichen bleiben also 84 Zeichen übrig (127 - 31 controls - 1 space - 12 specials). Demnach wird in einer e-Mail-Adresse also auch zwischen Groß- und Kleinschreibung unterschieden[7], außerdem sind auch so ungewöhnliche Zeichen wie { zugelassen. Die 84 Zeichen müssen nun auf die 16 in ITU Q.23 definierten DTMF-Tasten gemapped werden. Da die meisten Telefone nur über die Tasten $\boxed{0}$ bis $\boxed{9}$ sowie $\boxed{*}$ und $\boxed{\#}$, also nur 12 Tasten verfügen, sollten nur diese für die Umsetzung herangezogen werden. Rechnerisch ergibt sich, daß jede der 12 Tasten des Telefonapparates neben der jeweiligen Ziffer mit sieben (!) weiteren Zeichen belegt sein müßte (84 / 12).

Würde man also im trivialen Fall auf jede Taste sieben Zeichen legen, so wären für die Eingabe einer üblichen Adresse wie „foertel@ibm.net" bei 15 Zeichen und 3.5 durchschnittlichen Tastendrücken insgesamt 52.5 Tastendrücke nötig, was offensichtlich nicht praktikabel ist. Neben der Spracheingabe bleiben folgende Alternativen:

Automatisierung des Wählvorgangs Mit einer einfach handhabbaren Hardware, etwa im Format eines Touch-Tone-Dialers, gibt der Benutzer auf einer miniaturisierten ASCII-Tastatur die gewünschte e-Mail-Adresse wie gewohnt ein. Das Gerät nimmt dann die Umsetzung in DTMF-Tones vor. Da eine umfangreichere Tastatur zwangsläufig zu größeren Geräteabmessungen führt, verringert sich allerdings auch hier die praktische Einsetzbarkeit. Außerdem dürfte eine flächendeckende Verbreitung derartiger Geräte schwer sein. Eine Alternative stellen Geräte dar, die bereits verbreitet sind, wie etwa Notebook-Rechner oder noch besser (weil kleiner) sog. Personal Digital Assistants (PDAs). Neben einer ASCII-Tastatur verfügen diese meist auch über eingebaute Lautsprecher und sind programmierbar, so daß lediglich eine Umsetzungs-Software gebraucht wird.

Reduzierung des Zeichenvorrates Eine Reduktion des erlaubten Zeichenvorrates, wie sie z.B. durch eine Gleichstellung von Groß- und Kleinbuchstaben erreicht werden könnte, würde die Mehrfachbelegung der 12 Telefontasten deutlich verringern.

Der ersten Möglichkeit weiter nachzugehen, scheint mir angesichts der sich auf dem Telekommunikationsmarkt abzeichnenden Entwicklungen wenig sinnvoll: Von Nokia wird beispielsweise ein Mobilfunktelefon (*Nokia Communicator 9000*[8]) mit eingebautem Fax-Gerät, WWW-Browser, e-Mail-Client und Telnet-Emulation angeboten. Es dürfte wohl nur eine Frage sehr kurzer Zeit sein, bis man mittels eines solchen Gerätes auch direkt Internet-Telefonie betreiben kann.

Die zweite Möglichkeit, den Zeichenvorrat zu reduzieren, scheint mir der sinnvollere Weg. Beschränkt man sich beispielsweise auf die ausschließliche Verwendung von Buchstaben (ohne Unterscheidung von Groß- und Kleinbuchstaben) in e-Mail-Adressen,

[7] Wovon in der Praxis allerdings selten Gebrauch gemacht wird, da z.B. UNIX-User-Ids grundsätzlich aus Kleinbuchstaben bestehen

[8] http://www.nokia.com/com9000/n9000.html

Option A			Option B		
1	2 ABC	3 DEF	1 QZ	2 ABC	3 DEF
4 GHI	5 JKL	6 MNO	4 GHI	5 JKL	6 MNO
7 PQRS	8 TUV	9 WXYZ	7 PRS	8 TUV	9 WXY
*	0	#	*	0	#

Tabelle 3.2: Telefontastaturen nach ITU-T E.161

so böte sich eine der in ITU-T E.161 [ITU93a] vorgeschlagenen Tastenbelegungen an (siehe auch 3.2). Dieser Standard ist seit langem etabliert und dementsprechend auch auf den Tastaturen zahlreicher Endgeräte aufgedruckt, wobei sich die Option A der vorgeschlagenen Belegung durchgesetzt hat. Ein Buchstabe wird hier durch mehrmaliges Drücken der betreffenden Taste erzeugt. Soll beispielsweise ein E eingegeben werden, so muß die 3-Taste zweimal gedrückt werden. Natürlich setzt dies voraus, das sich das Endgerät in einem alphanumerischen Modus befindet, ansonsten erzeugt das Drücken der Zifferntasten die jeweilige Ziffer. Auf die nächste Zeichenposition wird jeweils geschaltet wenn:

1. Automatisch nach ca. 1 s nachdem eine der Tasten 2 bis 9 das erstemal gedrückt wurde.
2. Nachdem eine andere Taste gedrückt wurde.

Werden also in kurzer Folge die Tasten 2 2 8 eingegeben, so wird sofort nach dem Drücken der 8 der Buchstabe B erzeugt und auf die nächste Zeichenposition geschaltet (also ohne das Timeout von ca. 1 s abzuwarten). Mit dem Drücken der 8 beginnt das Timeout von 1 s erneut, so daß nun ein T erzeugt wird, wenn innerhalb des Timeouts keine weitere Taste gedrückt wurde. Für die Eingabe von „foertel@ibm.net" wären dann

Zeichen:	f	o	e	r	t	e	l	@	i	b	m	.	n	e	t
Taste:	3	6	3	7	8	3	5	#	4	2	6	*	6	3	8
Anzahl:	3	3	2	3	1	2	3	1	3	2	1	1	2	2	1

immerhin noch 30 Tastendrücke erforderlich. Die Punkte werden hierbei mit der *-Taste und das at-Sign mit der #-Taste eingegeben. Zusätzlich müßte die Eingabe mit irgendeiner Taste z.B. # abgeschlossen werden.

Aufgrund der fehlenden Anzeige der eingetippten Zeichen am Telefonapparat ist eine Kontrolle der Eingabe durch den Benutzer nur schwer möglich, so daß auch diese Methode bei längeren Zeichenketten unbefriedigend ist. Zudem fehlt noch die Definition einer Korrekturtaste. Spätestens bei längeren Adressen wie „schulzrinne@cs.columbia.edu" (27 Zeichen) wird die Unbrauchbarkeit dieses Verfahrens deutlich.

Weitere Überlegungen führen zu der Frage, ob überhaupt alle Zeichen der e-Mail-Adresse erforderlich sind um den Teilnehmer eindeutig zu adressieren. So ist denkbar, daß bei einem Anruf z.B. bei FOKUS das Weglassen des Host-/Domain-Teils einen Default-Wert (z.B. fokus.gmd.de) impliziert. In diesem Fall wäre also die ausschließliche Angabe der User-Id eindeutig. So wären für „foertel" nur noch 14 und für

„schulzrinne“ immerhin noch 31 Tastendrücke erforderlich (bei Anwendung von Option A der ITU-T E.161 Empfehlung).

Diese Zahlen scheinen mir gerade noch akzeptabel, vor allem angesichts der fehlenden Eingabekontrolle. Natürlich würde man jedes auf die oben beschriebene Art erkannte Zeichen per Sprachsynthese o.ä. quittieren, aber dennoch würde der Anrufer bei langen Zeichenketten ($>$ 25 Zeichen) schnell den Überblick verlieren. Aus diesem Grunde scheint mir eine manuelle Eingabe von e-Mail-Adressen am Telefon nur in Ausnahmefällen sinnvoll, eben in solchen, wo die alleinige Angabe der User-Id eindeutig ist.

Da die Eingabe von Zeichenketten auch für andere Dinge als e-Mail-Adressen sinnvoll sein kann, unterstützt die in dieser Arbeit vorgestellte Implementierung die Eingabe von Zeichenketten nach ITU-T E.161 in der beschriebenen Art und Weise.

3.8 Computer Telephony Standards

Da die angestrebte Integration von Telefonteilnehmern in Internet-Konferenzen mit dem Thema Computer-Telephony-Integration (CTI) zu tun hat, liegt der Gedanke nahe, einen Blick auf existierende CTI-Standards zu werfen, siehe auch [Str96][9] und [Mes96]. Hier spielen vor allem Microsofts Windows Telephony Application Programming Interface (TAPI) und TSAPI [Cro96], das u.a. von Novell entwickelt wurde, eine Rolle. Weiterhin gibt es von der European Computer Manufacturers Association (ECMA) Computer-Supported Telecommunications Applications (CSTA) Standards, in denen die hardware-unabhängige Steuerung von Telefoniegeräten (Telefone, Tk-Anlagen) behandelt wird. Im TSAPI sind hierfür entsprechende Primitiven realisiert. So gibt es Funktionen zum Auf- und Abbau von Verbindungen zwischen Geräten (*cstaMakeCall()*, *cstaClearCall()*) sowie Zusatzfunktionen wie *cstaHoldCall()* und *cstaTransferCall()* um Merkmale wie Makeln oder Rufweiterleitung zu realisieren.

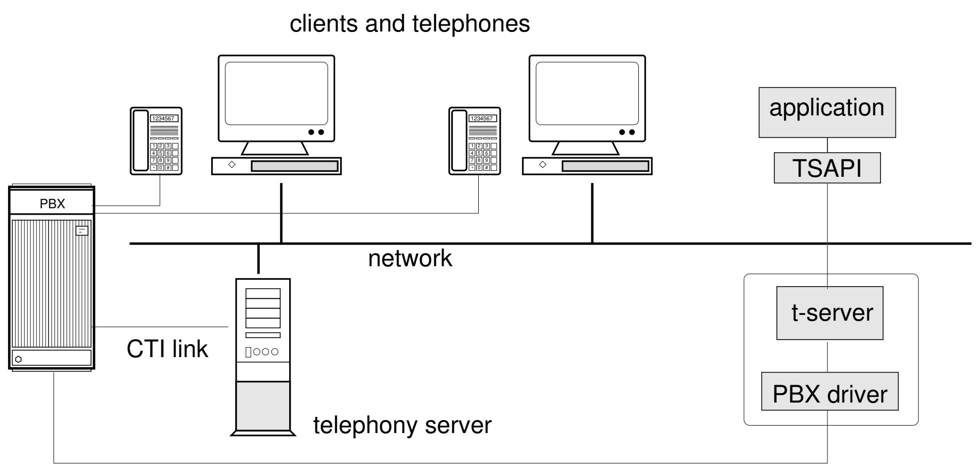

Abb. 3.2: Network telephony system architecture

Die genannten Standards gehen von einer Client-Server Architektur aus, in der sog. Telephony-Server über einen CTI-Link mit Tk-Anlagen (PBX) verbunden sind und deren Steuerung übernehmen (siehe

[9] http://www.dialogic.com/company/whitepap/carlieee.htm

Abb 3.2). Ein Client kann über eines der o.g. APIs die Dienste des Servers in Anspruch nehmen. Hierbei wird jedoch nach wie vor der Telefonapparat zur Sprachübertragung benutzt, lediglich seine Steuerung sowie die der Tk-Anlage erfolgt mittels des APIs. Sprach- oder Bilddatenübertragung bleiben von diesen Standards unberücksichtigt. Ein typisches kommerzielles Produkt, in dem diese Schnittstellen zum Einsatz kommen ist CT-Connect[10] von Dialogic. Dabei handelt es sich um eine Call-Control-Server-Software mit deren Hilfe sich Telefonanlagen, die über die oben genannten Schnittstellen verfügen, in Datennetze einbinden lassen. Die Software unterstützt „...call control and monitoring through links to many popular telephone switches“, so der Hersteller. Vereinfacht kann man sagen, daß mit Hilfe dieser APIs der PC bzw. die Workstation zum Komforttelefon werden. Da dies nicht Ziel dieser Arbeit ist, kommen CTI-Standards hier nicht zur Anwendung.

[10]http://www.dialogic.com/PRODUCTS/D_SHEETS/1741WEB.HTM

Kapitel 4

Call-Server

4.1 Einleitung

Um die erforderliche Instanz zur Kopplung zwischen ISDN und Internet realisieren zu können, wird eine entsprechende Hardware-Umgebung benötigt. Zur Durchführung dieser Arbeit stand eine SUN-Sparcstation IPX mit dem Betriebssystem SOLARIS und eingebauter ISDN-Karte (ebenfalls von SUN) zur Verfügung. Durch den standardmäßig vorhandenen Netzwerkanschluß sind die Hardware-Voraussetzungen somit gegeben.

4.2 Software-Schnittstelle zum ISDN

Für den software-seitigen Zugang zum ISDN via BRI oder PRI hat sich im Bereich der Personal-Computer seit Jahren das Common ISDN API (CAPI) etabliert [CAP94], das inzwischen auch Schnittstellen zu den Betriebssystemen OS/2, UNIX und NetWare festlegt. Der CAPI-Standard definiert Messages, die zwischen einer Anwendung und der CAPI-Library über Queues ausgetauscht werden. Die Abwicklung des ITU-T Q.931 [ITU93b] D-Kanal-Protokolls (auch als DSS1 bekannt) wird vollständig vom CAPI wahrgenommen.

In Ermangelung einer CAPI-Implementierung für SOLARIS wurde für die Durchführung dieser Arbeit auf eine ISDN-Library von Zahl [Zah97] zurückgegriffen. Anstelle einer nachrichtenorientierten Kommunikation findet hier eine ereignisorientierte Kommunikation statt: Requests werden über normale Funktionsaufrufe an die Library gerichtet. Für die Behandlung von Indications registriert die Anwendung bei der Library Event-Handler-Funktionen, die bei Eintreten der entsprechenden Events von der Library aufgerufen werden. Dazu muß die Anwendung ähnlich wie beim X11-Window-System permanent in einer Main-Loop der ISDN-Library laufen. Die Behandlung von Events durch die Anwendung muß in so kurzer Zeit wie möglich erfolgen, damit die Library vom ISDN eingehende Indications schnell genug behandeln kann. Ansonsten würde das Q.931-Protokoll verletzt werden, falls Timeouts ablaufen und nicht rechtzeitig behandelt werden können.

4.2.1 Initiierung eines ausgehenden Anrufes

Abbildung 4.1 zeigt den Ablauf eines normalen ausgehenden Anrufes unter Verwendung der ISDN-Library. Die Verbindung wird durch den Angerufenen beendet. Zunächst initiiert die Anwendung mit Hilfe der Library-Funktion *IsdnCallReq()* einen Anruf. Daraufhin schickt die Library eine Bestätigung (ISDN_CALL_IND), in der u.a. eine eindeutige Verbindungs-Id enthalten ist. Anschließend erhält die Anwendung einen B-Kanal zugewiesen (ISDN_CHANNEL_IND) sowie ein ISDN_ALERTING_IND, mit dem die Gegenstelle dem Anrufer die Signalisierung des Anrufes mitteilt (mit anderen Worten: Beim Angerufenen klingelt es jetzt). Das daraufhin eintreffende ISDN_CONNECT_IND signalisiert, daß der Angerufene den Anruf entgegengenommen hat; der B-Kanal zwischen beiden Sprechstellen ist somit aufgebaut.

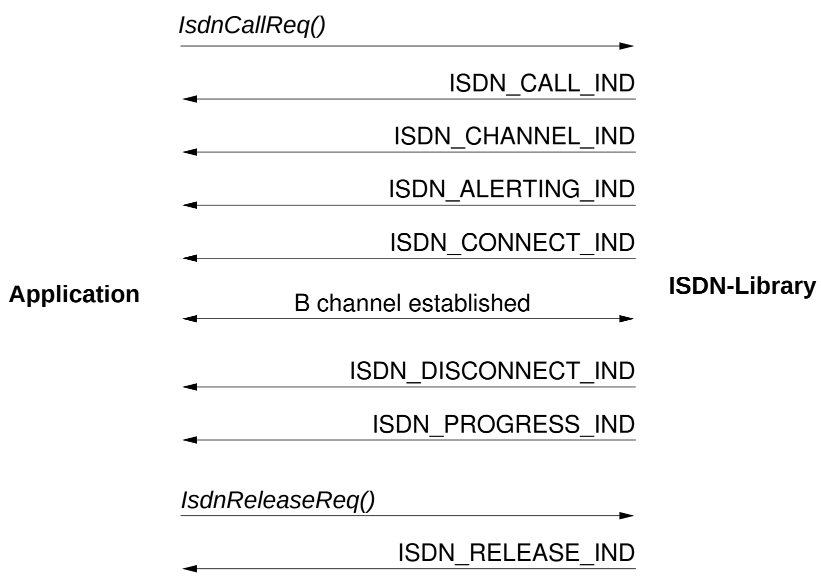

Abb. 4.1: Outgoing call using the ISDN library

Die Verbindung wird durch den Angerufenen beendet, was die Library mit einem ISDN_DISCONNECT_IND quittiert. Da der B-Kanal zwischen lokaler Vermittlungsstelle (VStO) und dem Anrufer immer noch aufgebaut ist, schickt die Library ein ISDN_PROGRESS_IND. In aller Regel wird jetzt von der VStO ein Besetzt-Zeichen auf dem B-Kanal gesendet. Die Anwendung beendet die Verbindung schließlich durch Aufruf der Library-Funktion *IsdnReleaseReq()* und erhält als finale Bestätigung ein ISDN_RELEASE_IND.

Wie man sieht, ist für die Initiierung eines Anrufes nur ein Funktionsaufruf erforderlich, von den eingehenden Indications brauchen lediglich ISDN_CHANNEL_IND und ISDN_RELEASE_IND behandelt zu werden. Ersteres deshalb, weil dort der Filedescriptor für den B-Kanal mitgeteilt wird und letzteres, weil damit das Ende der Verbindung erkannt wird. Ein ISDN_DISCONNECT_IND würde die Anwendung z.B. nicht erhalten, wenn sie die Verbindung selbst mit *IsdnReleaseReq()* beendet. Das ISDN_RELEASE_IND signalisiert in jedem Fall das Ende der Verbindung zwischen Anwendung und VStO.

Letztendlich spiegelt diese Dialogfolge den im Q.931 definierten Ablauf wieder, auch die dort verwendeten Namen finden sich in den Library-Funktionen und Namen der Indications wieder. Eine der wesentlichen Aufgaben der Library ist die Verwaltung der zahlreichen Timer, die im Q.931 definiert sind (allein 16 auf der Benutzerseite) sowie die Kommunikation mit dem darunterliegenden Q.921-Modul.

Während Requests per Funktionsaufrufe abgeschickt und Indications von Event-Handlern behandelt werden, erfolgt der Austausch der B-Kanal-Daten über direkten Zugriff auf das ISDN-Device. Mit einem ISDN_CHANNEL_IND teilt die Library der Anwendung einen Filedescriptor mit, auf dem der B-Kanal bereits geöffnet ist. Der Datenaustausch für den B-Kanal erfolgt also wie unter UNIX üblich per *read()* und *write()* System-Calls.

4.2.2 Eingehende Anrufe

Abbildung 4.2 zeigt die Behandlung eines eingehenden Anrufes mit Beendigung durch den Angerufenen bzw. die Applikation.

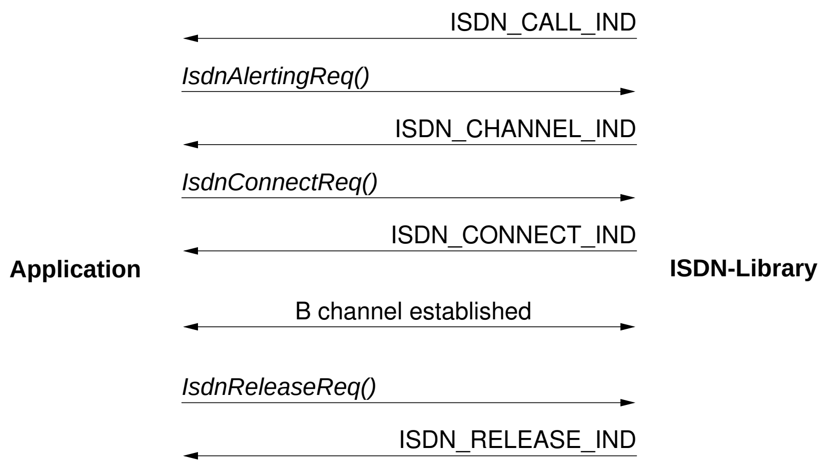

Abb. 4.2: Incoming call handling using the ISDN library

Der eingehende Anruf wird der Anwendung mit einem ISDN_CALL_IND signalisiert, durch welchen der Anwendung gleichzeitig eine eindeutige Verbindungs-Id mitgeteilt wird. Durch Aufruf von *IsdnAlertingReq()* wird der Gegenstelle die Signalisierung (also das Klingeln) angezeigt (der Anrufer erhält das Klingelzeichen). Anschließend stellt die Library den Filedescriptor für den B-Kanal zur Verfügung (ISDN_CHANNEL_IND). Schließlich nimmt die Anwendung mittels *IsdnConnectReq()* den Anruf entgegen; der B-Kanal zwischen beiden Sprechstellen gilt mit dem Eintreffen des ISDN_CONNECT_IND als aufgebaut. Der Angerufene beendet den Anruf durch Aufruf von *IsdnReleaseReq()*, was entsprechend mit ISDN_RELEASE_IND bestätigt wird.

Durch die Vergabe der Verbindungs-Ids kann eine Anwendung durchaus mehrere eingehende oder ausgehende Verbindungen gleichzeitig verwalten.

4.3 Der Call-Server

Für die Verbindung von Telefonnetz (ISDN) und Internet wurde im Rahmen dieser Arbeit eine Software entwickelt, die Anrufe sowohl aus dem Telefonnetz als auch dem Internet annimmt und behandelt. Da die Behandlung von Anrufen die wesentliche Aufgabe der Software ist, wurde das entstandene und im folgenden beschriebene Produkt als Call-Server bezeichnet.

Um die in den letzten beiden Kapiteln beschriebenen Aufgaben möglichst flexibel realisieren zu können, lag der Gedanke nahe, die Behandlung von Anrufen durch Skripte, etwa wie UNIX-Shell-Skripte, ausführen zu lassen.

4.3.1 Anrufe aus dem ISDN

Bei einem Anruf aus dem Telefonnetz wird zunächst ein Skript gestartet, dessen erste Aufgabe die Annahme des Anrufes ist. Hierfür können z.B. Angaben wie Uhrzeit oder Nummer des Anrufers analysiert werden, etwa um eine Zugangsberechtigung zu prüfen. Nach der Rufannahme durch das Skript „kommuniziert“ der Anrufer mit diesem, das dann entsprechende Aktionen auslöst, wie das Abspielen von Ansagen etc. Der Ablauf des Skriptes kann durch die in 3.5.1 und 3.5.2 beschriebene DTMF- und Spracherkennung vom Anrufer beeinflußt werden.

Kopplung von B-Kanal und RTP-Strom

So kann beispielsweise zwecks Teilnahme an einer MBONE-Konferenz eine Rufumleitung ins Internet realisiert werden, wie sie in Abbildung 4.3 skizziert ist.

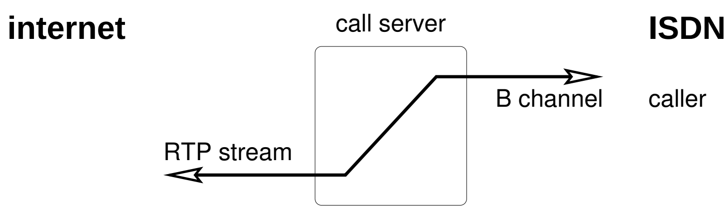

Abb. 4.3: ISDN-Verbindung mit RTP-Strom

Ein entsprechender Skript-Befehl **redirect** bewerkstelligt die Umleitung. Bei dieser Art der Umleitung, die man als passiv bezeichnen könnte, wird lediglich der Audiostrom umgeleitet, nicht aber der Anruf selbst. Es erfolgt also keine Rufsignalisierung ins Internet. Das Ende der Umleitung kann entweder durch das Skript (z.B. einem abgelaufenen Timer oder per **hangup**) oder aber durch den Anrufer (per Auflegen) aus dem ISDN erfolgen. Abbildung 4.4 zeigt den Ablauf, bei dem das Verbindungsende durch das Skript eingeleitet wird.

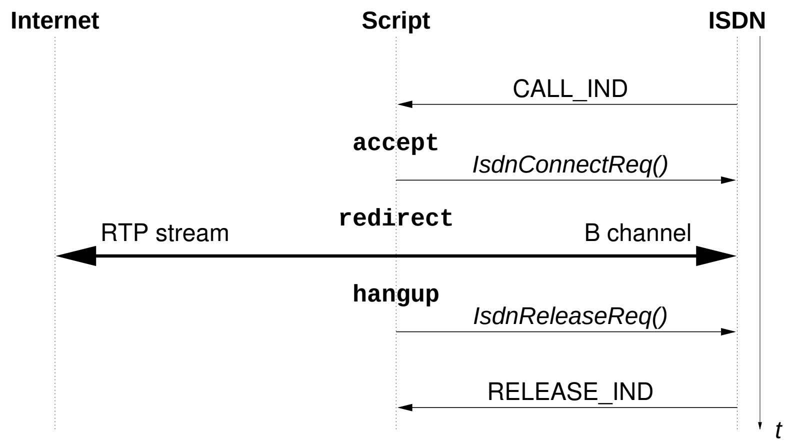

Abb. 4.4: Ablauf: ISDN-Verbindung mit RTP-Strom

Rufweiterleitung ins Internet

Um den Anruf wie in Abbildung 4.5 gezeigt aus dem ISDN im Internet per SIP zu signalisieren, sieht die Skriptsprache den Befehl **invite** vor. Zusammen mit diesem werden die SIP-spezifischen Parameter übergeben, unter anderem die e-Mail-Adresse des Gerufenen. Es ist in jedem Fall Aufgabe des Skriptes, das Rufziel bzw. dessen e-Mail-Adresse zu ermitteln. Dies kann natürlich auch durch den Anrufer per Telefontastatur geschehen, wie in 3.7 beschrieben.

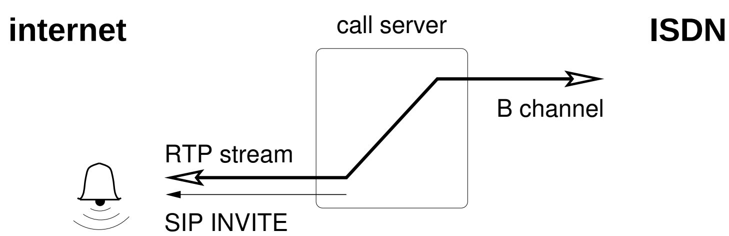

Abb. 4.5: Rufweiterleitung ins Internet

Die Ausführung der **redirect**-Anweisung ist also erst nach Rufannahme durch den Angerufenen sinnvoll (Abbildung 4.6).

Die Verbindung wird hier durch den Anrufer aus dem ISDN beendet (ISDN_RELEASE_IND). Eine u.a. kostensparende Variante wäre, die Verzögerung der ISDN-Rufannahme bis zu dem Zeitpunkt, an dem der positive Response Code (200) des SIP-Requests eintrifft. Lehnt der mit SIP gerufene Netzwerkbenutzer den Anruf nämlich ab, so wurde der ISDN-Ruf vergebens angenommen (außer für die Telefongesellschaft). Hierbei ist jedoch das ISDN-Timing zu beachten. ITU Q.931 sieht 120 s für die Rufsignalisierung vor, danach wird die Verbindung von der VStO beendet. Manche Telefonanlagen (z.B. die, die ich für die Entwicklung benutzt habe) tun dies scheinbar schon nach 90 s. Der SIP-Anruf muß also innerhalb dieser Zeit beantwortet werden, ansonsten wird die ISDN-Verbindung durch die VStO getrennt. Im allgemeinen wird ein Anrufer aus dem Telefonnetz, der nicht weiß, daß sein Anruf in ein völlig überlastetes Netzwerk umgeleitet wird, kaum länger als 20 bis 30 s warten. Daher scheint mir die sofortige Annahme des ISDN-Anrufes durchaus sinnvoll, um dann z.B. die Meldung „Bitte warten Sie, Ihr Anruf wird weitergeleitet“ abzuspielen.

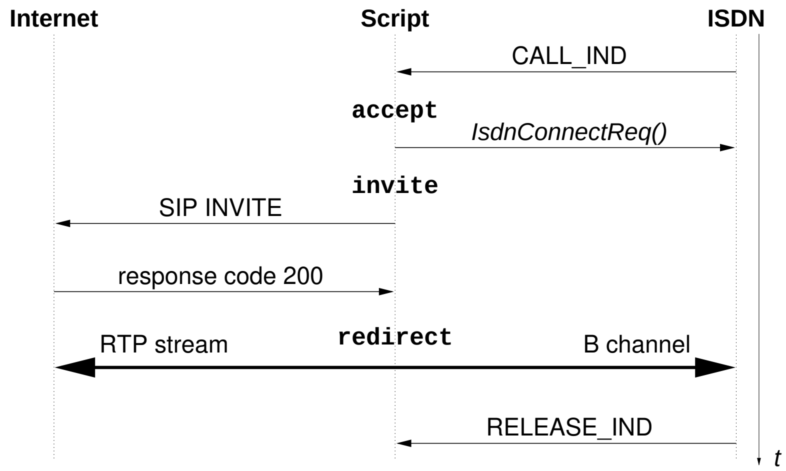

Abb. 4.6: Ablauf: Rufweiterleitung ins Internet

Rufumleitung ins ISDN

Das Ziel einer Rufumleitung kann wiederum ein Telefonanschluß sein, wenn beispielsweise der gewünschte Gesprächspartner derzeit nicht über das Internet erreichbar, sondern stattdessen nur telefonisch erreichbar ist (Abbildung 4.7).

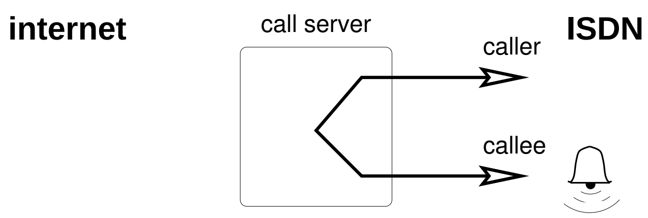

Abb. 4.7: ISDN-ISDN-Rufweiterleitung

Für diese Form der Rufumleitung wird ein zweiter B-Kanal benötigt. Im Gegensatz zur Umleitung ins Internet erfolgt hier durch die **redirect**-Anweisung gleichzeitig die Signalisierung des Anrufes. Die Benutzung von **invite** erübrigt sich somit. Abbildung 4.8 zeigt den Ablauf. Zur Vereinfachung sind in der Abbildung die Indications ISDN_CHANNEL_IND und ISDN_ALERTING_IND weggelassen. Auch hier wird die Verbindung durch das Skript beendet.

4.3.2 Anrufe aus dem Internet

Anrufe aus dem Internet werden grundsätzlich in Form von SIP-INVITE-Requests entgegengenommen und ebenfalls durch Skripte behandelt. Ein Anruf wird auch hier durch die Ausführung der **accept**-Anweisung angenommen. Der Anrufer erhält einen SIP-Response-Code 200. Ab diesem Zeitpunkt ist das Skript mit dem im INVITE-Request genannten RTP-Strom (Audio) verbunden.

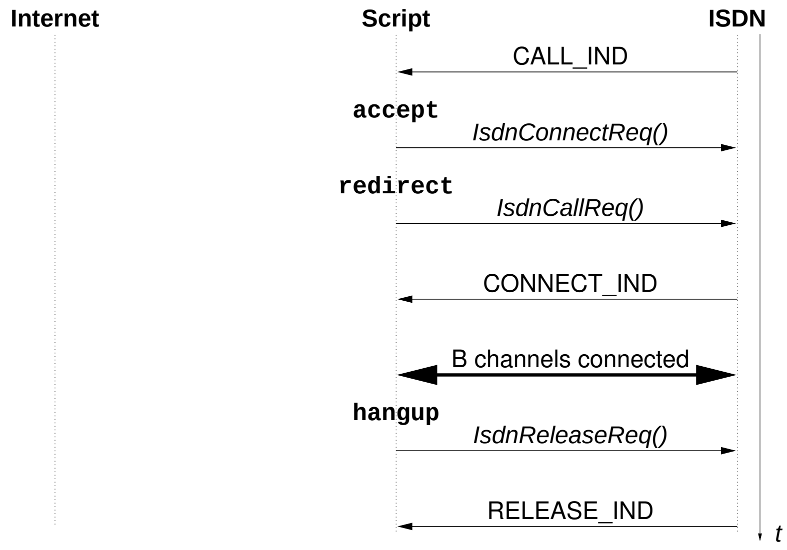

Abb. 4.8: Ablauf: ISDN-ISDN Rufumleitung

Rufumleitung ins ISDN

Mittels **redirect** kann nun eine Umleitung ins ISDN erfolgen. Den Ablauf zeigt Abbildung 4.9. Die Rufumleitung wird diesmal durch den gerufenen ISDN-Teilnehmer beendet (das Skript erhält ein ISDN_RELEASE_IND).

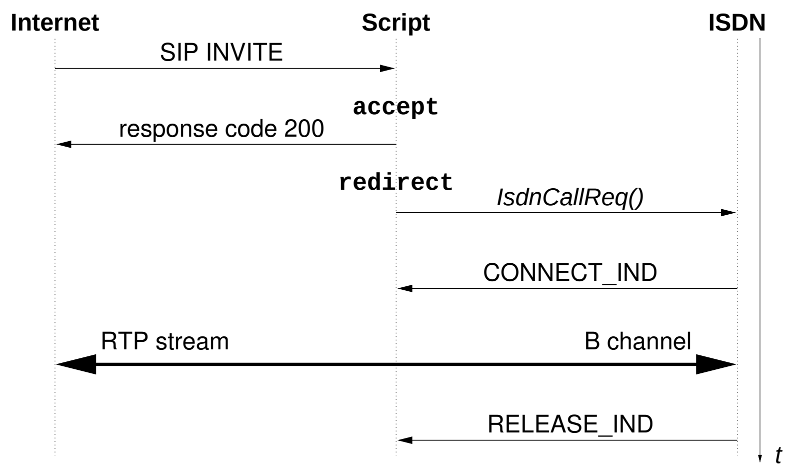

Abb. 4.9: Ablauf: Umleitung eines SIP-Anrufes ins ISDN

Da das Ende eines RTP-Stromes quasi nur am dauerhaften Ausbleiben der RTCP-Pakete erkannt werden kann, bleibt die Beendigung einer solchen Umleitung dem gerufenen ISDN-Teilnehmer oder dem Skript überlassen. Eventuell könnte man ein RTCP BYE der Person, die den SIP-Anruf initiiert hat als Indikator für ein Verbindungsende nehmen, aber das scheint mir sehr vage. Der Anrufer könnte die Konferenz initiiert und sich zwischenzeitlich verabschiedet haben (daher das BYE-Paket). Jedoch könnten inzwischen andere Benutzer der Konferenz beigetreten sein, mit denen der ISDN-Teilnehmer noch in Verbindung bleiben will. Ein definitives Ende der Verbindung kann also nur durch den Telefonteilnehmer (durch Auflegen) oder durch das Skript (z.B. Timeout) herbeigeführt werden.

Rufumleitung ins Internet

Die Umleitung eines SIP-Anrufes ins Internet erfolgt mit Hilfe der **redirect**-Anweisung, ohne daß der Anruf zuvor mit **accept** angenommen wurde. Das **redirect** als Antwort auf ein eingehendes SIP-Invite liefert einen Response-Code 3xx, mit dem Anrufer die neue Lokation der gerufenen Person mitgeteilt wird.

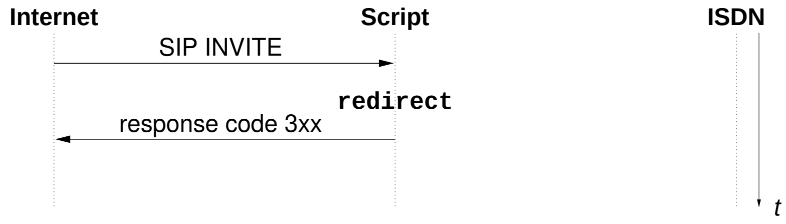

Abb. 4.10: SIP-Rufumleitung

Bei dieser Form der Umleitung, die eigentlich nur in der Rückweisung des INVITE-Requests besteht, wird also kein Mediendatenstrom behandelt.

Wie man sieht, werden die vorgestellten Formen der Rufumleitung alle mit Hilfe von **redirect** bewerkstelligt, das sich je nach Anrufquelle und Umleitungsziel anders verhält. Eigentlich sollte dies die Anrufbehandlung vereinheitlichen und somit auch vereinfachen. Mir ist nicht ganz klar, ob dies auch wirklich gelungen ist, zumal in dem Fall der Umleitung SIP→SIP zuvor kein **accept** aufgerufen werden darf. Genau dieser Fall bleibt noch offen, wenn nämlich ein SIP-Request angenommen und anschließend mit **redirect** ins Internet zurück umgeleitet wird. Man könnte sich hier durchaus eine Umleitung des RTP-Stroms vorstellen, die aber derzeit nicht realisiert ist.

4.3.3 Skript-Anrufer-Kommunikation

Anstatt sich zu einem anderen Gesprächspartner mittels **redirect** verbinden zu lassen, können Anrufer auch mit Skripten selbst „kommunizieren“: Das Skript kann Ansagen (Audiofiles/Sprachsynthese) abspielen oder Nachrichten aufzeichnen. In Verbindung mit der DTMF-Ton-Erkennung sowie der Möglichkeit, externe Programme zu starten, lassen sich so beliebige Informations- und Ansagedienste bis hin zu Call-Center-Anwendungen realisieren. Die bei SIP- und ISDN-Anrufen verfügbare Identifikation des Anrufers ermöglicht zudem eine anruferspezifische Behandlung von Anrufen. Einige Beispiele:

Fax-Service
: Der Anrufer wählt per Tastendruck eine bestimmte Information, gibt anschließend seine Faxnummer ein und erhält die gewünschten Informationen zugefaxt; ein Dienst, den sich bereits jetzt zahlreiche Firmen zunutze machen.

Erweiterter Anrufbeantworter
: Ein Anrufer kann nicht nur Nachrichten hinterlassen, sondern sich auch weiterverbinden lassen, falls er es besonders eilig hat. Hierfür kann er sich durch Eingabe einer PIN legitimieren.

Vorlesungs-Service

Unter einer bestimmten Rufnummer könnten Universitätsvorlesungen, die im MBONE übertragen werden, auch per Telefon zugänglich gemacht werden. Allerdings würde in Deutschland bei den gegenwärtigen Telekomtarifen eine 10^{15} Uhr Vorlesung von 90 Minuten Dauer 7.20 DM kosten (0.12 DM je 90 s).

Automatische Vermittlung

In einer Firma könnte ein (motivierter) Mitarbeiter Anrufe nach Feierabend und am Wochenende zu sich nach Hause stellen lassen, bei Anrufen aus der Chefetage wird „Falsch Verbunden!“ abgespielt. Zwar lassen sich mit fast jedem ISDN-Telefon Rufumleitungen programmieren, aber eben nicht zeit- oder anruferabhängig.

Mailbox-Abfrage

Eingegangene e-Mails können per Telefon abgefragt werden. Die Identifizierung des Anrufers erfolgt entweder durch Abfrage der Rufnummer des Anrufers (bei Anrufen aus dem ISDN) oder durch Abfrage des „From:“-Feldes bei SIP-Anrufen (unsicher weil manipulierbar). Alternativ kann die Eingabe einer PIN verlangt werden. E-Mails können dann auf Wunsch auch zugefaxt werden.

Vor allem in Richtung Computer-Telephony-Integration (CTI) lassen sich beliebig weitere Beispiele ausdenken, von denen jedoch die meisten mit bereits existierenden Produkten realisierbar sind.

4.4 Serverkonfiguration

4.4.1 ISDN-Konfiguration

Im ISDN können einem Anschluß grundsätzlich mehrere Rufnummern zugeordnet sein (Multiple Subscriber Number, kurz: MSN). Beim Euro-ISDN sind dies bis zu 10 Nummern je Basisanschluß. Im SETUP-Paket eines Anrufes werden sowohl die Rufnummer des Anrufers (calling party number) als auch die gerufene Nummer übertragen (called party number). Damit nicht bei jedem Anruf auf jeder MSN des benutzten ISDN-Anschlusses sofort ein Skript gestartet wird, kann man in einer Konfigurationsdatei `isdn.cfg` das Verhalten des Servers bei Anrufen aus dem ISDN festlegen. Für beliebige Kombinationen von „called party“, „calling party“ und benutzten Medien können bestimmte Dienste festgelegt werden, die bei einem Anruf angestoßen werden. So wäre denkbar, daß Anrufe zu einer bestimmten MSN grundsätzlich von einer Fax-Software behandelt werden, das Starten eines Skriptes wäre hier unerwünscht.

Bei leerer Konfigurationsdatei werden alle Anrufe ignoriert. Jede Zeile in dieser Datei definiert ein Tripel von rufender und gerufener MSN, einer Medien-Kennung sowie die dazugehörige Aktion, die der Server bei einem Anruf auslösen soll. Die einzelnen Einträge werden durch Leerzeichen getrennt. Rufende und gerufene MSN können mittels * rechtsmaskiert werden, d.h. es können Nummernbereiche mit gleichen Anfangsziffern bezeichnet werden (siehe Abbildung 4.11). Kommentarzeilen beginnen mit einem #.

```
# isdn.cfg -- isdn server configuration file
#
# calling     called
# party no.   party no. media         action

  *           41        audio         iPhone
  30314*      44        audio         iPlay KeinAnschluss-8k.au loop
  *           45        audio&video   iScript foertel staff test.cpl
  *           45        audio|video   iScript foertel staff test1.cpl
  <unknown>   *         audio         iPlay NoServiceAvailable.au
```

Abb. 4.11: Example for the `isdn.cfg` configuration file

Die Medienkennung legt die Medien fest, die behandelt werden können. Derzeit sind „audio" und „video" definiert. In den ersten beiden Zeilen des obigen Beispiel werden z.B. nur Anrufe mit Audiodatenstrom entgegengenommen. Ruft also jemand mit einem Fax die 41 an, so wird der genannte Service nicht ausgeführt. Die Medienangaben können mit & und | UND- bzw. ODER-verknüpft werden, wie in der dritten und vierten Zeile.

Die erste Zeile definiert, daß alle Anrufe zur MSN 41 mit einem Service namens iPhone behandelt werden. Anrufern aus der TU-Berlin (30314*) wird unabhängig von der gewählten Nummer das angegebene Audiofile endlos (loop Option) vorgespielt (zweite Zeile). Die restlichen Zeilen aktivieren bei Anrufen die Ausführung des angegebenen Skriptes unter der User-Id `foertel` und der Group-Id `staff`. Während die erste Zeile für die MSN 45 das Skript `test.cpl` nur bei Anrufen mit *Audio und Video* ausführt wird, wird `test1.cpl` bei Anrufen mit Audio *und/oder* Video gestartet. Da die Zeilen der Konfigurationsdatei von oben nach unten interpretiert werden, wird bei einem Anruf der Dienst aufgerufen, bei der die drei Parameter als erstes passen. Bei einem Anruf mit Audio *und* Video auf der 45 wird also *test.cpl* ausgeführt.

Für den Fall, daß die Nummer des Anrufers/Gerufenen nicht bekannt ist, wie etwa bei Anrufen aus dem analogen Telefonnetz, sind die entsprechenden Felder mit <unknown> belegt. Folgende Dienste sind vorgesehen:

iPhone

Einfacher Telefonservice mit dem über einen Client Telefongespräche von einer Workstation aus ins ISDN und umgekehrt geführt werden können. Bei einem Anruf aus dem ISDN werden alle mit dem Server verbundenen Clients informiert, die den Anruf dann optisch und akustisch signalisieren. Der erste Client, der den Anruf entgegennimmt, erhält das Gespräch. Über eine UDP-Verbindung werden B-Kanal und Audio-Hardware der Workstation via Unicast miteinander verbunden. Dieser Dienst wurde nur zu Testzwecken implementiert.

iPlay

Schreibt den Inhalt der angegebenen Audiodatei in den Audiodatenstrom des Anrufers.

iScript

Führt eine Skriptdatei unter der angegebenen User-Id und Group-Id aus.

iAnswer

Einfacher Anrufbeantworter: Nach dem Abspielen der angegebenen Audiodatei wird ein Piepton generiert. Im Anschluß daran wird der eingehende Audiostrom in einer Datei aufgezeichnet. Dieser Dienst ist nur angedacht nicht aber implementiert.

iFwd

Eingehende Anrufe werden zum angegebenen Umleitungsziel weitergeleitet. Dieser Dienst ist nur angedacht nicht aber implementiert.

Wie sich später zeigen wird, können bis auf den iPhone-Dienst alle anderen auch durch Skripte realisiert werden. Dies hat sich jedoch erst im Laufe der Arbeit ergeben und so ist die Konfigurationsdatei `isdn.cfg` erhalten geblieben. Der iPhone-Dienst war ohnehin nur zum Testen gedacht, denn im Rahmen dieser Arbeit ging es nicht um die Neuerfindung des Telefons; auf ihn kann also verzichtet werden.

Somit könnte bei einen Anruf sofort ein bestimmtes Skript gestartet werden, z.B. `icc.cpl` für Incoming Call, in dem dann entschieden wird, ob der Anruf angenommen wird oder nicht. Für verschiedene Anrufer und Medien könnten dann von `icc.cpl` aus andere Skripte aufgerufen werden. Der Hauptteil eines solchen Skriptes könnte in Anlehnung an die im nächsten Kapitel entwickelte Sprache wie folgt aussehen:

```
...
if [string match 30314* $sysCallingParty ] {
  accept
  exec "iAuPlay KeinAnschluss.au"
  exit
}
switch $sysCalledParty {
  "44" { accept
         enter AnsweringMachine
       }
  "45" { accept
         source  foertel/test.cpl
       }
  ...
}
```

Die Variable $sysCalledParty$ enthält die vom Anrufer gewählte Nummer, $sysCallingParty$ enthält die Nummer des Anrufers, sofern vohanden. Der **source**-Befehl liest und interpretiert das angegebene Skript.

Im Gegensatz zur tabellarischen Darstellung in `isdn.cfg` kann diese Lösung bei größeren Skripten schnell unübersichtlich und fehleranfällig werden, wie sich in dem Beispiel bereits andeutet. Außerdem ist unklar, unter welcher User-Id die Skripte ausgeführt werden. Derzeit läuft der Call-Server als Benutzer `root`, um auf das ISDN-Device zugreifen zu können. Aus Sicherheitsgründen ist dies problematisch, wenn Benutzer die mit **source** gelesenen Skripte selbst erstellen dürfen. Zur Lösung dieses Problems könnte man jedoch einen **setuid**-Befehl implementieren, der vor Ausführung des **source**-Befehls aufgerufen wird.

4.4.2 SIP-Konfiguration

Für SIP-Anrufe müßte eine ähnliche Behandlung stattfinden, wir bei ISDN-Anrufen. Aus zeitlichen Gründen ist jedoch die Behandlung von SIP-Anrufen nicht fertiggeworden. Vorgesehen ist jedoch, daß beim Eintreffen eines SIP-INVITE-Requests die Skriptdatei `icc-sip.cpl` ausgeführt wird. Auch hier stellt sich die Frage, unter welcher User-Id die Ausführung stattfindet und ob die Skriptdarstellung sehr übersichtlich ist.

4.5 Der Ablauf bei einem Anruf

Bei einem Anruf aus dem ISDN wird zunächst anhand der Konfigurationsdatei geprüft, welches Skript bzw. welcher Dienst für die Behandlung des Anrufes zuständig ist. Wurde kein passender Eintrag gefunden, so wird der Anruf ignoriert, anderenfalls wird der betreffende Dienst gestartet. Da die Dienste bzw. Skripte in einem eigenen Adressraum laufen, führt jeder weitere Anruf zum Start einer neuen Instanz des jeweiligen Dienstes bzw. Skriptes. Die von der ISDN-Library für jede Verbindung vergebene Id (Call-Id) dient gleichzeitig zur eindeutigen Identifizierung der gestarteten Instanzen, so daß beim Eintreffen weiterer Indications diese einer laufenden Instanz eindeutig zugeordnet werden können.

Die Behandlung des ISDN-Protokolls (ISDN-Library), das Starten der Dienste bzw. Skripte sowie die Behandlung der RTP- und B-Kanal-Datenströme erfolgt also durch den Call-Server, der derzeit als einzelner Prozeß realisiert ist. In den folgenden Kapiteln wird sich zeigen, daß für bestimmte Aktivitäten des Servers weitere Prozesse gestartet werden müssen.

Kapitel 5

Call Processing Language

5.1 Einleitung

Zur Behandlung eingehender Anrufe wird eine Skriptsprache benötigt, die folgenden Kriterien genügt:

- Die Sprache sollte einfach, d.h. leicht erlernbar sein und sich daher an eine bekannte Sprache anlehnen.
- Die Sprache sollte auf vielen/allen Plattformen verfügbar bzw. leicht dorthin portierbar sein.
- ISDN- und SIP-Anrufe sollen möglichst gleichartig behandelt werden. So soll z.B. in einer Anweisung zur Umleitung eines eingehenden Anrufes nicht unterschieden werden müssen, ob der Anruf aus dem ISDN oder dem Internet kam.
- Die Sprache muß in der Lage sein, asynchron auftretende Ereignisse (Events) wie Timeouts oder das Auflegen des Anrufers zu behandeln.

Den ersten beiden Forderungen würde die Verwendung von Tcl [Ous94] gerecht werden. Zudem bietet Tcl einen ausreichend großen Sprachumfang um auch komplexe Anwendungen zu erstellen. Da Tcl keine Möglichkeiten besitzt, asynchronen Ereignisse aufzufangen und zu behandeln, müßten hierfür sowie für die telefoniespezifischen Funktionen entsprechende Erweiterungen vorgenommen werden.

Dieser Weg wird in einer Arbeit von Uhler [Uhl93] beschritten. Dort bildet ein um einige Befehle erweiterter Tcl-Interpreter die Skriptsprache *PhoneScript*, mit der Telefonieanwendungen erstellt werden können. Ein (analoges) Telefon und der Computer werden über eine spezielle Hardware miteinander verbunden. Mit Befehlen wie **call** und **play** können Anrufe initiiert und Audiofiles abgespielt werden. Den speziellen ISDN-Merkmalen wie mehrere Mediendatenströme/Kanäle sowie dem Zugang zu Multimedia-Konferenzen wird allerdings nicht Rechnung getragen.

Da bei einem Anruf verschiedene Ereignisse asynchron auftreten können, liegt der Gedanke nahe, eine Telefonieanwendung als Finite-State-Machine zu betrachten und als solche zu programmieren. In *PhoneScript* ist Tcl zwar um ein Eventhandling erweitert worden, jedoch gibt es keine Zustände bzw. nur einen einzigen, in dem die definierten Events gelten. Beispielsweise bei der Realisierung verschachtelter Auswahlmenüs, die der Anrufer mittels Telefontastatur durchlaufen kann, wäre es mit *PhoneScript* recht umständlich, die verschiedenen Menüebenen (Zustände) zu unterscheiden, um eine gedrückte Taste zu behandeln. Entsprechende Verschachtelungen von if-then-else oder switch-Anweisungen würden die Lesbarkeit der Programme erschweren.

Andererseits ist es unnötig, sequentiell ablaufende Anwendungen wie einen einfachen Ansagedienst oder gar einen Anrufbeantworter in verschiedene Zustände zu zerlegen, da die einzigen Events, die auftreten können (Auflegen durch den Anrufer oder Ablauf der Sprechzeit) zum Ende der Anwendung führen. Eine Unterscheidung mehrerer Zustände würde hier also zu unnötig aufgeblähten Programmen führen.

5.2 Prozedurale vs. ereignisorientierte Sprache

Günstig scheint somit eine Sprache, die beides unterstützt: Prozedurale Programmierung für überwiegend sequentiell ablaufende Anwendungen und ereignisorientierte Programmierung für Anwendungen, bei denen viele gleiche Events unter verschiedenen Umständen, d.h. Zuständen auftreten können.

Am Beispiel eines einfachen Anrufbeantworters sollen die angesprochenen Probleme kurz verdeutlicht werden. Eine rein sequentielle Darstellung zeigt folgendes Beispiel in einer fiktiven Sprache ($sysCalls$ ist eine Variable, die die Anzahl der eingehenden Anrufe zählt).

```
while (1) {
  play "ogm/welcome.au"              # Play a welcome message
  getkey $in                         # Get user selection
  switch $in {
    case "1":
      play "icm/*.au"                # Play all recorded messages
      continue
    case "2":
      record "icm/m$sysCalls.au"     # Record a new message
      continue
    case "#":
      play "ogm/bye.au"
      hangup
      exit                           # terminate script
    default:
      play "invalid input"
      continue
  }
}
```

Der Anrufer kann hier mit drei Tasten wählen, ob er eingegangene Nachrichten abhören will (durch Drücken der $\boxed{1}$) oder eine neue Nachricht aufnehmen will (durch Drücken der $\boxed{2}$). Mit der $\boxed{\#}$-Taste kann er die Anwendung beenden, was natürlich auch durch Auflegen geht. Für den Fall, in dem bereits Nachrichten eingegangen sind, stellt sich die Frage, wie das Abspielen ohne Auflegen vorzeitig beendet werden kann. Die $\boxed{\#}$-Taste hat hier natürlich keine Wirkung, da das Skript sequentiell abgearbeitet wird. Eine zudem um einige Zeilen kürzere Lösung zeigt folgende ereignisorientierte Variante:

```
state AnsweringMachine {
  on start
    play "ogm/welcome.au"            # Play a welcome message
  on key "1"
    play "icm/*.au"                  # Play all recorded messages
```

```
      enter welcome
    on key "2"
      record "icm/m$sysCalls.au"      # Record a new message
      enter welcome
    on key "#"
      play "ogm/bye.au"               # say "good bye"
      hangup                          # disconnect and
      exit                            # terminate script
  }
```

Eine explizite Eingabeanweisung wie **getkey** im ersten Beispiel entfällt hier, da die Tastendrücke als Ereignisse definiert sind. Ihre Erkennung löst automatisch die Ausführung der Anweisungen hinter der jeweiligen **on key**-Klausel aus. Wird während des Abspielens der Nachrichten die # -Taste gedrückt, wird die Ausführung der **play**-Anweisung unterbrochen und hinter dem **on key '#'** fortgesetzt. So kann hier außerdem ein Anrufer, der die Funktionsweise der Anwendung bereits kennt, die Begrüßungsnachricht durch vorzeitiges Drücken einer der drei Tasten überspringen.

5.3 Vergleiche mit anderen Sprachen

5.3.1 Modemsprachen

Bei der Wahl einer geeigneten Sprache liegt es aufgrund der ähnlichen Einsatzgebiete (Behandlung von Anrufen) nahe, einen Blick auf sog. Modemsprachen zu werfen. Dies sind Sprachen, die meist ein Verbindung mit Terminal-Emulationsprogrammen angeboten werden und die Aufgabe haben, den Verbindungsaufbau zu anderen Rechnern über serielle Leitungen zu automatisieren. Folgende Sprachen habe ich mir näher angeschaut:

- Microsoft's DFÜ-Skriptsprache
- Trumpet Scripting Language
- Reflection Command Language (RCL) von Walker Richer & Qinn Inc. (WRQ) [1]
- Connect Control Language (CCL)[2] von InterCon

Tabelle 5.1 stellt die wesentlichen Eigenschaften der Sprachen gegenüber.

Wie man sieht, besitzen die Sprachen stark ähnliche Merkmale, was angesichts ihres Einsatzgebietes kaum wundert. Die Sprachen von Microsoft, Trumpet und InterCon dienen zum Modem-Setup bzw. Setup des lokalen Netzwerk-Stacks bei SLIP oder PPP-Verbindungen. Die fehlende Fähigkeit, eingehende Anrufe zu beantworten ist scheinbar darauf zurückzuführen.

Den weitaus größten Sprachumfang besitzt die Sprache von WRQ, die inzwischen von einem Visual Basic ähnlichen Reflection-Basic ersetzt worden ist, mit dem man (unter MacOS und Windows) auch Anwendungen mit grafischer Oberfläche erstellen kann. Jedoch ist auch hier das Haupteinsatzgebiet das automatisierte Login auf fremden Rechnern. Anrufe im Sinne von Voice-Calls werden von keiner der Sprachen unterstützt.

[1] http://www.wrq.com
[2] http://www.intercon.com/techsupport/classes/ccl.html

Merkmal	**Microsoft**	**Trumpet**	**RCL**	**CCL**
supported data types	int, string, bool	int, string	int, string	string
supported loop statements	while	while, repeat	while	none
conditional branch statements	if-then	if-then	if-then	iftries
incoming call handling supported?	no	no	no	no
support for outgoing calls	yes	yes	yes	yes
supports procedures/functions?	yes	no	yes	no
event handling supported?	no	no	no	no

Tabelle 5.1: Modem control languages

5.3.2 Ereignisorientierte Sprachen

Die Suche nach ereignisorientierten Sprachen hatte die folgenden Sprachen zum Ergebnis:

- LOTOS
- SDL (ITU-T Z.100)
- Tcl-expect

LOTOS ist eine Formale Sprache zur Beschreibung verteilter Systeme, die im ISO-Standard IS 8807 beschrieben ist. SDL (CCITT Specification and Description Language) ist eine in der ITU-T Norm Z.100 [ITU93c] definierte Sprache zur Beschreibung von Telekommunikationssystemen. Die Sprachen verfügen u.a. über Möglichkeiten zur Modellierung von Daten und Prozessen sowie der Kommunikation zwischen Prozessen. SDL definiert neben einer textuellen auch eine grafische Notation. Da die Sprachen Systeme auf eine sehr abstrakte Art beschreiben, sind die mit ihnen erstellten Beschreibungen keineswegs als kompakt und leicht lesbar zu bezeichnen. Jeder benutzte Datentyp muß in aufwendigen Definitionen erst einmal erzeugt werden. Allein aufgrund ihrer Komplexität und des großen Lernaufwandes scheinen mir diese Sprachen oder Varianten davon ungeeignet für den Einsatz im Call-Server.

Tcl-expect ist eine in Tcl geschriebene Anwendung, mit der sich andere Applikationen über Skripte steuern lassen. Wie mit den oben beschriebenen Modemsprachen kann z.B. die Anmeldeprozedur auf einem entfernten Rechner automatisiert werden. Hierfür wird mit einem Skriptbefehl `spawn` z.B. ein `rlogin`-Kommando abgesetzt. Das Skript kann nun unter Einsatz von Pattern-Matching auf bestimmte Ausgaben des Rechners wie „Password:“ warten und mit einem `send`-Befehl eine entsprechende Eingabe zum entfernten Rechner schicken.

Da das Erkennen von Strings sozusagen die einzige Art ist, Events auszulösen, ist auch `expect` für den Einsatz im Call-Server ungeeignet.

5.4 Call Processing Language (CPL)

Die gemachten Beobachtungen brachten mich dazu, mir über eine neue Sprache bzw. über eine Erweiterung der *PhoneScript*-Idee nachzudenken. Der Einsatz von Tcl erfüllt immerhin die ersten der beiden eingangs gestellten Forderungen nach einer möglichst einfachen, bereits bekannten bzw. leicht portierbaren

Sprache. Weiterhin sind bei Integration eines Tcl-Interpreters auch in Zukunft neue Features automatisch verfügbar, der Call-Server muß lediglich mit der neuesten Version der Tcl-Library gelinkt werden.

Es bietet sich also die Erweiterung von Tcl um die benötigten Anrufbehandlungs-Routinen an. Diese sollen es ermöglichen, eine Finite-State-Machine abzubilden. Darüberhinaus soll die Möglichkeit der prozeduralen Programmierung erhalten bleiben. Vor allem aber soll die Behandlung *eingehender* Anrufe mit dem Ziel der Teilnahme an Internet-Konferenzen unterstützt werden.

5.4.1 Finite-State-Machine

Da Tcl selbst keine Sprachkonstrukte für die Behandlung asynchroner Ereignisse besitzt, liegt es nahe, neben der Registrierung der neuen Anweisungen für das Call-Handling, aus einer Art Finite-State-Machine-Beschreibungssprache heraus einen Tcl-Interpreter zur Ausführung der eigentlichen Anweisungen aufzurufen. Diese Beschreibungssprache definiert alle Zustände der State-Machine in einer Datei (Skript) in beliebiger Reihenfolge. Anfangszustand ist dabei derjenige, der am Beginn des Skripts steht. Im folgenden wird exemplarisch ein Zustand in der Notation dieser Sprache, die ich Call Processing Language kurz CPL nennen will, dargestellt:

```
state statename {
    on <eventtype> {          # local event definition
      <tcl-script> }
}

on <eventtype> {              # global event definition
  <tcl-script>
}

proc <procname> <args> {      # creation of a new Tcl procedure
  <procbody>
}
```

Ein Zustand (state) wird durch die Klausel **state** $statename$ $\{\ldots\}$ beschrieben. Die Events, die innerhalb des Zustandes behandelt werden sollen, werden mit **on** $eventtype$ definiert. Die bei Eintritt des Events auszuführenden Anweisungen werden in dem Anweisungsblock { `<tcl-script>` } angegeben. Mit der als Tcl-Erweiterung implementierten **enter**-Anweisung, kann aus einem Anweisungsblock heraus in einen anderen Zustand gewechselt werden. Der Endzustand ist erreicht, wenn das Skript mittels **exit** verlassen wird. Um gleiche Events, die in mehreren oder allen Zuständen auftreten können, nicht in jedem Zustand erneut definieren zu müssen, können diese global definiert werden. Das Disconnect-Event ist ein solches Beispiel. Da der Anrufer jederzeit, also in jedem Zustand der State-Machine, die Verbindung beenden kann, und die Reaktion des Call-Servers darauf sehr wahrscheinlich immer die Gleiche sein wird, bietet sich die globale Definition dieses Events an.

Globale Event-Definition sind *außerhalb* eines **state**-Blocks, der Übersichtlichkeit wegen möglichst zentral an einer Stelle im Skript anzuordnen.

Mit der Library-Funktion *Tcl_Eval()* kann ein C-Programm einzelne Tcl-Anweisungen oder Anweisungsblöcke (sog. Tcl-Skripte) ausführen, so daß der Ausführung der Anweisungsblöcke der State-

Machine mit *Tcl_Eval()* scheinbar nichts im Wege steht. Durch die von mir gewählte Architektur des Call-Servers entsteht jedoch ein Problem, das im folgenden Abschnitt behandelt wird.

5.4.2 ISDN-Timing

CPL-Skripte werden vom Call-Server ausgeführt, welcher nebenbei das an einigen Punkten zeitkritische ISDN-Protokoll abwickeln muß. Beim Ausführen von Tcl-Skripten mit *Tcl_Eval()* hat der Server keine Kontrolle über die Ausführungszeit der Tcl-Skripte und so kann es vorkommen, daß in der ISDN-Schicht Timer ablaufen, ohne rechtzeitg behandelt werden zu können. Im schlimmsten Fall kann eine Endlosschleife oder eine *sleep* Anweisung den gesamten Server blockieren. Auch innerhalb der Skriptsprache definierte Timer müssen jederzeit die Ausführung eines Anweisungsblockes unterbrechen können. Alternativ könnte man natürlich das Tcl-Skript parsen und jedes Statement einzeln mit *Tcl_Eval()* ausführen, in der Hoffnung, daß keine Anweisung eine bestimmte Zeit überschreitet. Hierfür benötigt man jedoch einen vollständigen Tcl-Parser und das wäre schon der halbe Implementierungsaufwand für eine neue Sprache.

Eine bessere Alternative scheint mir in der Abarbeitung der Anweisungsblöcke bzw Tcl-Skripte durch eigene Prozesse bzw. Threads zu bestehen. Leider hat sich diese Erkenntnis erst relativ spät im Verlauf der Arbeit durchgesetzt, so daß die gegenwärtige Implementierung auf den Einsatz von *Tcl_Eval()* verzichtet. In 6.4 wird hierauf genauer eingegangen.

Der von mir eingeschlagene Weg besteht nun darin, die Anweisungsblöcke in den Event-Definitionen nicht durch einen Tcl-Interpreter sondern, einen eigenen Interpreter abarbeiten zu lassen. Um den Aufwand in Grenzen zu halten, habe ich nur einen Teil des Tcl-Sprachumfangs implementiert. Um den Einsatz eines Tcl-Interpreters zu einem späteren Zeitpunkt zu unterstützten, habe ich darauf geachtet, daß innerhalb der CPL-Anweisungsblöcke die Syntax mit Tcl identisch ist.

5.5 Sprachbeschreibung

Die möglichen Zustände, die bei der Behandlung eines Anrufes auftreten, können in CPL in Form einer Finite-State-Machine abgebildet werden. Es bleibt dem Anwender überlassen, ob er hiervon Gebrauch macht, oder die gesamte Anwendung in einem einzigen Zustand beschreibt. Ein CPL-Programm besteht somit aus mindestens einem Zustand. Für jeden Zustand werden die möglichen zu behandelnden Events definiert. Eintretende Events, für die keine Definition existiert, werden mit Ausnahme des Disconnect-Events ignoriert. Events können prinzipiell sein:

- Per Spracherkennung erkannte Wörter des Anrufers (**on speech**)[3]
- Erkannte Folgen von DTMF-Zeichen (**on dtmf**)
- Beenden der Verbindung durch den Anrufer (**on disconnect**)
- Ablauf eines Timers (**on timer**)
- Eintritt in einen neuen Zustand (**on enter**)

[3] Spracherkennung findet in der gegenwärtigen Implementierung keine Anwendung

- Fehler, die während der Skriptausführung auftreten[4]

Eine CPL Applikation besteht aus Zustandsbeschreibungen und Prozeduren. Letztere entsprechen den procs in Tcl. In den Zustandsbeschreibungen werden die möglichen Ereignisse und die entsprechenden Reaktionen in Form prozeduraler Anweisungsblöcke definiert. Innerhalb einer Zustandsdefinition vorgenommene Event-Definition gelten lokal, also nur innerhalb dieses Zustandes. Events, die außerhalb von Zuständen vereinbart sind, gelten global, wenn sie nicht durch eine gleiche Definition innerhalb des aktuellen Zustandes überdeckt werden.

Anhang B zeigt den vollständigen Sprachumfang von CPL. Alle Anweisungen sind case-sensitiv. Kommentare können durch # an beliebigen Stellen im Sourcecode eingeleitet werden und gelten immer bis zum Zeilenende (wie in Tcl).

5.5.1 Datentypen

Wie in Tcl so ist auch in CPL String der einzig unterstützte Datentyp. Für numerische Berechnungen, etwa mit der **incr**-Anweisung, erfolgt intern eine automatische Typumwandlung, ebenso bei Vergleichen zweier Variablen mit numerischem Inhalt.

5.5.2 Pattern-Matching

Mittels **on dtmf** können beliebige Folgen von DTMF Tönen im ankommenden Audiostrom einer ISDN-Verbindung erkannt werden. Die Erkennung einer Folge löst das entsprechende Event aus, deren weitere Behandlung analog den anderen Events erfolgt. Die erkannte Zeichenfolge steht unmittelbar nach Auslösung des Events in der Systemvariablen *sysDtmf* zur Verfügung. Die zu erkennenden Folgen werden in der Event-Definition durch einen Matchstring definiert. Neben den eigentlichen DTMF-Zeichen 0123456789ABCD*# kann der Matchstring die in Tabelle 5.2 aufgelisteten Metazeichen enthalten.

char	**meaning**
d	one digit (0..9)
D	one or more digits
a	one DTMF alpha character (A, B, C, D)
A	one or more alpha character
$*$	one or more DTMF character
?	one DTMF character

Tabelle 5.2: Pattern matching meta characters

Um Metazeichen von den DTMF-Zeichen zu unterscheiden, müssen letztere in einfache Hochkommata eingeschlossen werden. Einige Beispiele zeigt Tabelle 5.3

Um bei Eingaben variabler Länge die Erkennung abzuschließen, muß im Matchstring ein Schlußzeichen angegeben werden, z.B. das „#“. Nachteilig ist hierbei, daß dieses Zeichen ebenfalls in *sysDtmf* enthal-

[4]Laufzeitfehler lösen in der gegenwärtigen Implementierung keine Events aus

Anwendungsgebiet	pattern string
Eingabe einer IP-Adresse in dotted decimal notation (Terminierung durch #).	D'*'D'*'D'*'D'#'
Eingabe einer vierstelligen PIN	dddd
Eingabe einer Faxnummer (Terminierung durch #)	D'#'
Eingabe einer amex Kartennummer	'37'ddddddddddddd

Tabelle 5.3: Beispiele für Matchstrings zur DTMF-Erkennung

ten ist, obwohl es u.U. nicht Bestandteil der gewünschten Zeichenfolge ist. Um solche unerwünschten Zeichen herauszufiltern, stellt CPL die **cvt** Anweisung zur Verfügung.

Die Erkennung einer IP-Adresse wie in Tabelle 5.3 ist ein solches Beispiel. In Ermangelung einer .-Taste auf der Telefontastatur soll der Benutzer hier die *-Taste benutzen. Da die Zahlenblöcke zwischen den Punkten ein, zwei oder drei Ziffern lang sein können, wird im Matchstring als Metazeichen das 'D' benutzt. Abgesehen davon, daß hierdurch auch vier und mehr Ziffern zugelassen werden, müssen noch das abschließende # entfernt und die Sterne durch Punkte ersetzt werden, um eine gültige IP-Adresse zu erhalten. Dies sowie die Überprüfung auf eine korrekte IP-Adresse geschieht durch die **cvt**-Anweisung.

Um die Eingabe von Buchstaben per Telefontastatur nach ITU-T E.161 zu unterstützten, dient die **on ascii dtmf**-Variante der **on dtmf**-Definition. Da hierbei erkannte DTMF-Zeichen anders interpretiert werden, können **on dtmf** und **on ascii dtmf** nicht innerhalb desselben Zustands verwendet werden. Auch hier steht eine erkannte Zeichenfolge anschließend in $sysDtmf$. Im Gegensatz zur numerischen Tastenerkennung sind im Matchstring nur die Zeichen ? und * verfügbar. Soll also ein Unix-Username erkannt werden, so geschieht dies mit folgender Definition:

```
on ascii dtmf "*'#'" {
  ....
}
```

Beachte, daß auch hier die Eingabe mit einer bestimmten Taste (hier #) abgeschlossen werden muß, wenn man sich nicht auf eine feste Anzahl von Zeichen festlegen will. Folgende Variante kommt ohne eine solche Taste aus und löst nach genau dem dritten erkannten Buchstaben das Event aus:

```
on ascii dtmf "???" {
  ....
}
```

5.5.3 Event-Handling

Der erste im Skript definierte Zustand stellt zugleich den Anfangszustand dar. Bei Eingang eines Anrufes beginnt die Ausführung in dem Anweisungsblock der **on enter** Event-Definition. Daher wird diese Event-Definition im Anfangszustand erzwungen, in allen anderen Zuständen hingegen nicht, d.h. beim Eintritt in einen solchen Zustand werden keine Aktionen ausgeführt.

Übergänge in einen anderen Zustand erfolgen durch die Anweisung **enter**. Beim Auflegen durch den Anrufer (bei Anrufen aus dem ISDN) wird das Disconnect-Event erzeugt. Fehlt eine entsprechende Definition im aktuellen Zustand, so wird Skript sofort beendet. Wird das Disconnect-Event mittels **on disconnect** abgefangen, so kann das Skript beispielsweise Aufräumarbeiten erledigen etc. Der Aufruf von Programmen, die Audiodaten zum ISDN schicken, bleibt natürlich wirkungslos.

Die Reihenfolge der **on**-Anweisungen innerhalb einer Zustandsdefinition ist nur für das Erkennen von DTMF-Sequenzen relevant. Alle innerhalb eines Zustandes erkannten DTMF-Zeichen werden gesammelt und mit allen **on dtmf** bzw. **on ascii dtmf** Definition des Zustandes verglichen. Die Reihenfolge der Vergleiche erfolgt in der Reihenfolge der Definitionen. So würde in folgendem Beispiel die „10“ nie erkannt werden, weil zuvor bereits die Erkennung der „1“ ein Event auslöst.

```
on dtmf "1" {
  ....
}
on dtmf "10" {
  ....
}
```

Die Reihenfolge von Event-Definitionen ist also nur für die Erkennung der Matchstrings von Bedeutung.

Die Befehlsausführung in Folge eines Events terminiert ein evtl. noch laufendes Kommando. Wird beispielsweise ein Befehl mittels **exec** ausgeführt (z.B. Abspielen einer Ansage), so wird dieser unterbrochen, wenn z.B. eine DTMF-Sequenz erkannt wurde. Beachte, daß der alleinige Eintritt des Events nicht ausreicht; erst die Ausführung des ersten Befehls im Anweisungsblock einer Eventdefinition beendet einen laufenden Befehl. Wird beispielsweise ein Timer gestartet, das Timeout-Event aber nicht mit **on timer** abgefangen, so führt dies nicht zum Abbruch laufender Befehle, etwa eines **audiorec**!

Beachte, daß beim Abfangen des Disconnect-Events mit **on disconnect** das Skript für seine Beendigung selbst verantwortlich ist, d.h. bei fehlender **exit**-Anweisung läuft das Skript weiter.

5.5.4 CPL-Anweisungen

Anhang E enthält eine Liste aller CPL-Anweisungen, die in einem Anweisungsblock verwendet werden können. Die Liste ist unterteilt in Tcl-Anweisungen und CPL-Anweisungen. Bei der Verwendung eines Tcl-Interpreters zur Abarbeitung der Anweisungsblöcke müßten also die CPL-Anweisungen als neue Tcl-Befehle beim Tcl-Interpreter registriert werden. Nach Ausführung eines Tcl-Befehls enthält die Variable $sysStatus$ den Wert 0, wenn die Ausführung des Kommandos erfolgreich war. Werte größer 0 zeigen einen Fehler an. In der Datei `iSrvCpl.h` sind die möglichen Fehlerwerte als Macros definiert. Der Befehl **exec** liefert in $sysStatus$ den Exitstatus des ausgeführten Programms. Die **invite**-Anweisung liefert hingegen den SIP-Response-Code.

Die redirect-Anweisung

Die **redirect**-Anweisung dient der Umleitung des eingehenden Anrufes, also des Anrufes, der die Skript-Ausführung initiiert hat. Das Umleitungsziel wird in Form eines Uniform Resource Locators URL RFC

1738 [BLMM94] angegeben. Für Umleitungsziele ins Telefonnetz wird hierbei auf einen Vorschlag aus [Zig96] zurückgegriffen. Hiernach wird ein Telefonanschluß wie folgt als URL dargestellt:

```
phone://[+<country-code>-]<phone-number>
```

Die Umleitung eines bereits mit **accept** angenommenen Anrufes zum Anschluß 4711 würde also durch die Anweisung

```
redirect "phone://4711"
```

erfolgen. Kommt der eingehende Anruf bereits aus dem ISDN, so wird für die Rufumleitung ein zweiter B-Kanal verwendet. Steht kein freier Kanal zur Verfügung, so terminiert **redirect** und $sysStatus$ enthält den Wert ERR_CALL_FAILED (definiert in `iSrvCpl.h`). Die Rufumleitung ins ISDN hat zur Folge, daß es beim Angerufenen klingelt. Die Rufumleitung *ins* Internet mittels **redirect** sieht keine Rufsignalisierung vor. Vielmehr wird der eingehende Audiostrom mit dem angegebenen RTP-Audiostrom verbunden. Um die Parametrisierung der **redirect**-Anweisung einheitlich zu gestalten, wurde hierfür ebenfalls eine URL-Darstellung gewählt. Für das Verbinden mit einem RTP-Strom wird ein neuer URL-Typ benutzt, der rtp://-URL, dessen Syntax wie folgt lautet:

```
rtp://<host>:<port>
```

Die Art der Angabe von Hostadresse und Portnummer entspricht RFC 1738 [BLMM94]. Die Anweisung

```
redirect "rtp://224.2.58.17:23711"
```

verbindet den Anrufer also mit einem entsprechenden Audiostrom. Bei dieser Darstellung fehlt derzeit noch die Angabe des TTL, in der Implementierung wird hierfür mit einem Default-Wert gearbeitet.

Soll bei einer Rufumleitung *ins* Internet ein Anrufer gezielt eingeladen werden, so geschieht dies mit Hilfe der **invite**-Anweisung, welche im folgenden Abschnitt beschrieben wird.

Die invite-Anweisung

Die **invite**-Anweisung dient wie in 4.3.1 beschrieben, der Erzeugung eines SIP-INVITE-Requests, also zur Einladung eines Netzwerkbenutzers. Der gegenwärtige Stand[5] sieht folgende Syntax vor:

```
invitecmd ::= "invite" <to> <info> <media>
to        ::= "<to-user>@<to-host>"
info      ::= stringexpr
media     ::= "<media-addr>:<media-port>"
```

Die Felder „To:“, „i=“, „c=“ und „m=“ des SIP-Requests werden durch die Parameter der **invite**-Anweisung versorgt. Am folgenden Beispiel eines INVITE-Requests soll die Zuordnung der Feldinhalte gezeigt werden.

[5]Die Implementierung dieses Befehls ist noch nicht abgeschlossen

```
INVITE <request-id> SIP/2.0
From: <from-user>@<from-host>
To: <to-user>@<to-host>

v=0
o=<o-user> <session-id> <version> IN IP4 <my-ip>
s= MBONE audio
i= <info>
c= IN IP4 <media-addr>/<ttl>
t= 0 0
m= audio <media-port> RTP 0
```

Alle anderen Felder wie „Path:“ oder „e= “ werden nicht erzeugt. Die in spitzen Klammern geschriebenen Feldinhalte werden gemäß Tabelle 5.4 belegt. Die restlichen werden durch den Call-Server fest belegt.

`request-id`	wird vom Call-Server erzeugt
`from-user`	User-Id, unter der das aktuelle Script ausgeführt wird
`from-host`	Rechnername, auf dem der Call-Server läuft
`to-user`	Teil des ersten Parameters der **invite**-Anweisung
`to-host`	Teil des ersten Parameters der **invite**-Anweisung
`o-user`	wie from-user
`session-id`	wird vom Call-Server erzeugt
`version`	wird vom Call-Server erzeugt
`my-ip`	Adresse des Rechners, auf dem der Call-Server läuft
`info`	zweiter Parameter der **invite**-Anweisung
`media-addr`	Teil des dritten Parameters der **invite**-Anweisung
`ttl`	Teil des dritten Parameters der **invite**-Anweisung
`media-port`	Teil des dritten Parameters der **invite**-Anweisung

Tabelle 5.4: Belegung der Felder im SIP-INVITE-Request

Folgender CPL-Code verschickt einen SIP-INVITE-Request, die benutzten Variablen müssen zuvor vom Skript entsprechend belegt werden.

```
invite "$user@$hostname" "Call from $sysCalledParty" "$ip:$port"
if {$sysStatus == 200} then {
  redirect "rtp://$ip:$port"
  ...
}
```

Es ist vorgesehen, daß die Ausführung der **invite**-Anweisung bis zum Eintreffen des Response-Codes bzw. bis zum Ablauf eines Timeouts blockiert. Als Returncode wird entweder der dreistellige Response-Code oder aber beim Timeout ein vierstelliger Wert zurückgeliefert. Das Blockieren des Skriptes scheint mir allerdings ungünstig, wenn dem Anrufer während des Wartens auf die Antwort z.B. eine Ansage wie „Bitte warten Sie...“ vorgespielt werden soll. Günstiger wäre den Eingang des Response-Codes als Event zu realisieren.

5.5.5 Ausführung externer Programme

Beliebige Programme/Shell-Skripte können mit dem **exec**-Kommando ausgeführt werden. Da die Ausführungszeit der gestarteten Programme variabel bzw. dem CPL-Interpreter nicht bekannt ist, wird für das ausführende Kommando einen Prozeß erzeugt, der über Pipes mit dem Server-Prozeß verbunden ist, Abbildung 5.1 veranschaulicht dies am Beispiel der Ausführung des Programms `iAuCat`, welches den Inhalt einer Audiodatei einliest und nach stdout schreibt.

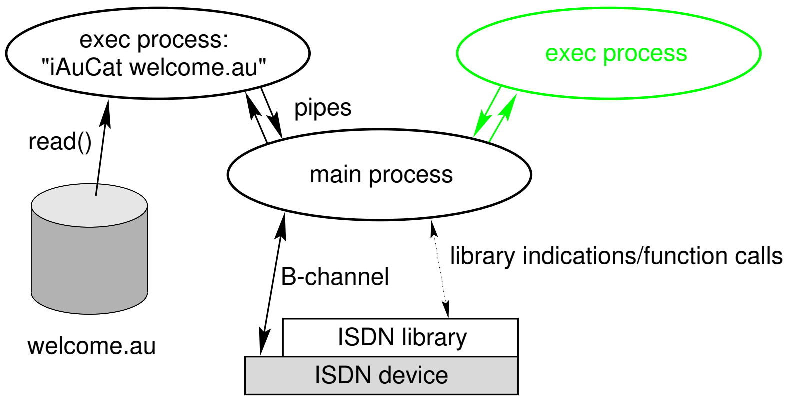

Abb. 5.1: Process communication during execution of external commands

Durch dieses Verfahren werden die in 5.4.2 beschriebenen Timing-Probleme umgangen. Um das Prozeß-Handling und die damit verbundenen Probleme nur einmal zu bewältigen, habe ich CPL von allen Kommandos befreit, die ebenfalls eine variable bzw. unbekannte Ausführungszeit benötigen. So waren ursprünglich eigene CPL-Kommandos für das Abspielen von Audiofiles und das Generieren von Tönen vorgesehen. Diese Kommandos sind nun als externe Kommandos realisiert (Tabelle 5.5).

Programm	**Bedeutung**
iAuCat	Liest den Inhalt einer Audiodatei, und schreibt diesen nach stdout
iRTP	Verbindet den Anrufer mit einem RTP-Strom
iTone	Erzeugt einen Ton der angegebenen Länge und Frequenz

Tabelle 5.5: Externe Programme

Bei der Wahl der mit **exec** zu startenden Programme ist zu beachten, daß stdout und stdin jeweils mit dem ISDN-B-Kanal verbunden sind. Ausgaben nach stderr werden ins Logfile des Skriptes geschrieben. Die benutzten Programme müssen also je nach Land einen 8kHz A-law oder μ-law Audiostrom erzeugen. Da dies leider die wenigsten Programme tun, habe ich die in Tabelle 5.5 aufgezählten geschrieben.

Wie bereits erwähnt, werden mit **exec** gestartete Prozesse beim Eintreffen von Events abgebrochen. Wird beispielsweise ein Audiofile abgespielt, so wird der Prozeß unterbrochen sobald eine mittels **on dtmf** bzw. **on ascii dtmf** definierte Tastenfolge erkannt wurde. Da die Ausführung der **exec**-Anweisung die Skriptausführung blockiert, gibt es keine andere Abbruchmöglichkeit, als durch Events. So kann die Ausführung eines Kommandos beispielsweise mit Hilfe eines Timers zeitlich begrenzt werden. Das durch den Timeout ausgelöste Event beendet die Befehlsausführung.

Als Returncode liefet **exec** den Exitstatus des gestarteten Programms, wie folgendes Beispiel zeigt:

```
...
exec "iAuCat welcom.au"            # play greeting message
if { $sysStatus != 0 } {           # error occured?
  exec "iAuCat error.au"           # error message
  exit                             # terminate script
}
...
```

5.5.6 Variablen

CPL unterstützt derzeit nur globale Variablen, welche mit der **set**-Anweisung erzeugt und initialisiert werden. Variablennamen können maximal 256 Zeichen lang sein, müssen mit einem Buchstaben beginnen und können Buchstaben und Ziffern enthalten. Zustands- oder prozedurlokale Variablen sind derzeit nicht vorgesehen. Anhang D enthält eine Liste aller in CPL vordefinierten Variablen. Die mit „read only" gekennzeichneten Variablen können nicht mittels **set** Kommando verändert werden; etwaige Zuweisungen führen zu einer Fehlermeldung (ERR117) und zum Abbruch des Skriptes. Bis auf $sysCalls$ und $sysConnections$ sind alle Variablen vollständig implementiert.

Bei der späteren Verwendung von *Tcl_Eval()* ist zu beachten, daß die Anweisungen im globalen Kontext des Tcl-Interpreters ausgeführt werden. In den dann als normale Tcl-Prozeduren realisierten procs stehen, wie in Tcl üblich, lokale Variablen zur Verfügung.

5.5.7 Timer

Mit der **timer**-Anweisung können Timer beliebig gestartet und angehalten werden. Die Syntax der Anweisung lautet:

```
timercmd      ::= "timer" timername { timerstart | "stop" }
timerstart    ::= { "local" | "global" } timeout
timeout       ::= hours":"minutes":"seconds
```

In Abhängigkeit vom Geltungsbereich werden lokal und globale Timer unterschieden. Ein lokaler Timer wird mit **timer** $timername$ **local** $timeout$ und ein globaler mit **timer** $timername$ **global** $timeout$ gestartet. Dabei ist $timername$ ein beliebiger Name und $timeout$ die Zeit, nach deren Ablauf das Timer-Event ausgelöst wird. Letzteres kann mittels **on timer** aufgefangen und behandelt werden.

Im folgenden Beispiel eines Auswahlmenüs wird der lokale Timer $t1$ auf 8 Sekunden gesetzt. Nach seinem Ablauf wird eine Meldung ausgegeben und der Timer durch Wiedereintritt in den selben Zustand (**enter Menu**) erneut gestartet.

```
state Menu {

    on enter {
      exec "iAuCat sounds/Menu.au"         # tell user the menu
      timer t1 local 00:00:08              # allow 8 secs for input
    }
    on dtmf "1" { enter GetIPaddr           }
```

```
        on dtmf "2" { enter RecordMessage       }
        on dtmf "#" { enter Bye                 }
        on timer t1 {
          exec "iAuCat sounds/InvalidOrNoInput.au"
          enter Menu
        }
    }
```

Ein lokaler Timer gilt immer nur innerhalb des Zustandes, in dem er gestartet wurde. Beim Verlassen dieses Zustandes werden automatisch werden alle lokalen Timer beendet, ohne daß ein Timer-Event generiert wird. Aus diesem Grunde ist in obigen Beispiel das Stoppen des Timers $t1$ nicht erforderlich, da nach dem Drücken einer der Tasten $\boxed{1}$, $\boxed{2}$ oder $\boxed{\#}$ der aktuelle Zustand verlassen wird.

Globale Timer gelten innerhalb des gesamten Skriptes, werden also beim Verlassen eines Zustandes nicht automatisch beendet. So kann z.B. durch Starten eines globalen Timers die Gesamtdauer einer Verbindung limitiert werden, auch dann wenn das Skript aus mehreren Zuständen besteht. Globale Timer dürfen nicht durch einen lokalen Timer gleichen Namens überdeckt werden.

Im folgenden Beispiel werden zwei Timer parallel gestartet, der globale *connectTime* mit 30 Minuten und der lokale Timer *t1* mit 4 Sekunden.

```
    state s1 {

        on enter {
          timer connectTime global 00:30:00
          timer t1 local 00:00:04
          accept
        }
        on dtmf "#" {
          enter s2
        }
        on timer t1 {
          log "t1 timed out!"
        }
    }

    state s2 {

        on timer connectTime {        # local event definition
          log "global timer 'connectTime' timed out!"
          enter s1
        }
    }

    on timer connectTime {            # global event definition
        log "timeout fetched by global definition"
        exit
    }
```

Sofern in *s1* nicht innerhalb von 4 Sekunden die $\boxed{\#}$-Taste erkannt wurde, läuft *t1* ab und es wird die Meldung „t1 timed out!“ ausgegeben. Wird nicht innerhalb der restlichen 3 Sekunden von *connectTime* die Taste $\boxed{\#}$ erkannt, so läuft auch *connectTime* ab und durch die globale Event-Definition am Ende des Skriptes, wird „timeout fetched by global definition“ ausgegeben und das Skript beendet.

Wird in *s1* die $\boxed{\#}$-Taste innerhalb von 4 Sekunden gedrückt, so wird in den Zustand *s2* verzweigt. Da dort keine anderen Events definiert sind, führt lediglich der Ablauf des globalen Timers *connectTime* zur Ausgabe von „global timer 'connectTime' timed out!“. Die globale Event-Definition am Skript-Ende ist hier also durch eine lokale Definition überlagert worden.

Ein gestarteter Timer kann jederzeit mit **timer** $timername$ **stop** gelöscht werden. Wird der Name eines nicht existierenden oder schon abgelaufenen Timers angegeben, so wird der Befehl ignoriert. Ein laufender Timer kann zurückgesetzt werden, indem er erneut gestartet wird.

5.6 CPL Fehlermeldungen

Anhang F enthält eine Liste aller Fehlermeldungen, die während der Skriptausführung auftreten können. Meldungen, die mit 1 beginnen, werden vom Skript-Compiler, Meldungen, die mit 2 beginnen von Skript-Interpreter generiert. Alle Meldungen werden nach stderr geschrieben und führen zum sofortigen Abbruch des Skriptes (nicht des Call-Servers). Bei der Abarbeitung eines CPL-Skriptes wird ein Logfile erstellt, in dem alle mit der **log**-Anweisung erzeugten Meldungen abgelegt werden. Das Logfile trägt den Namen des Skriptes plus angehängtem `.log`.

Kapitel 6

Implementierung

6.1 Einleitung

Die Implementierung des Call-Servers wurde von Anfang an möglichst modular gehalten, was trotz zahlreicher Änderungen erhalten blieb. Anfangs sollte der Server in der Lage sein, verschiedene Dienste zu realisieren bzw. sollte um solche leicht erweiterbar sein. Anhang H enthält eine Liste aller zur Implementierung gehörenden Files. Neben dem Hauptprogramm des Servers (`iSrv.c`) ist dort für jeden der realisierten Dienste ein eigenes File aufgeführt.

Durch die im Laufe der Arbeit zunehmende Mächtigkeit der Skriptsprache CPL, hat sich gezeigt, daß die angedachten und teilweise realisierten Dienste viel flexibler mit CPL realisierbar sind. Deshalb soll hier nur noch der Skript-Service (`iSrvScript.c`) beschrieben werden.

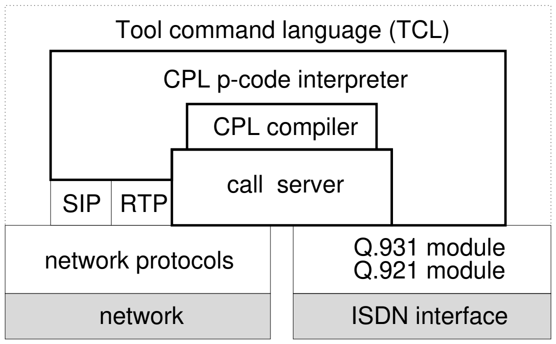

Abb. 6.1: Call-Server architecture

Abbildung 6.1 zeigt die Server-Architektur, einschließlich des geplanten Tcl-Einsatzes (gestrichelte Linie). Die dick umrandeten Linien kennzeichnen die von mir implementierten Komponenten. Die Implementierung umfaßt im wesentlichen:

- Behandlung von ISDN-Anrufen
- Ausführung von CPL-Skripten
- Erkennung von DTMF-Tönen mit Behandlung durch CPL-Skripte
- RTP-Mixer zur Kopplung des B-Kanaldatenstrom mit einem RTP-Audiostrom

6.2 Der Skript-Compiler

Zu Beginn der Arbeiten entstand die Frage, ob CPL-Skripte durch einen Interpreter abgearbeitet werden sollen oder erst compiliert und dann interpretiert werden sollen. Für letzteres sprächen geringfügige Laufzeitvorteile (durch den entfallenden Parser- und Übersetzungsvorgang), sowie erhöhte Sicherheit vor Laufzeitfehlern. Meines Erachtens ist einer der größten Nachteile beim Einsatz interpretierender Sprachen (vor allem in Produktionsumgebungen), die Gefahr von Laufzeitfehlern.

Der Laufzeitgewinn durch den Wegfall des Übersetzungsvorgang ist im Verhältnis zur Gesamtzeit für das Abarbeiten eines Skriptes jedoch so gering, daß er vernachlässigt werden kann. Um Skripte sowohl auf syntaktische Richtigkeit als auch deren Ablauf zu testen, gibt es das Programm `cpldebug` (siehe 6.5).

So werden CPL-Skripte interpretiert, was beim geplanten Einsatz des Tcl-Interpreters ohnehin unumgänglich ist. Jedoch erfolgt die Interpretation nicht, wie in Kommando-Shells üblich, zeilenorientiert, sondern das CPL-Skript wird zunächst in einen Pseudo-Code überführt, der in einem zweiten Schritt abgearbeitet wird. Hierfür zerlegt ein Scanner die Zeichen der Skriptdatei in einen Symbol- oder Token-Strom (lexikalische Analyse). Dieser wird einem Parser zugeführt, der die Syntaxanalyse vornimmt (siehe auch [ASU88]). Nach bestandener Syntaxprüfung wird ein Pseudo-Code (P-Code) erzeugt, der anschließend von einem Interpreter ausgeführt (siehe 6.3). Für den Übersetzungsvorgang wird die Skriptdatei nur einmal durchlaufen (1-Pass-Compiler).

Parser und Scanner sind in `iSrvCplComp.c` enthalten. Folgender Ausschnitt zeigt ein rudimentäres C-Programm, das das Skript `test.cpl` mit Hilfe der in `iSrvCplComp.c` exportierten Funktionen in P-Code überführt:

```
#include "iSrvCplComp.h"

main()
{
    FILE*           logf;
    t_script*       script;
    char*           fn = "test.cpl";

    ...
    if (Cpl_NewScript (script, fn, logf) == CPL_ERROR)
      exit (1);
    if (Cpl_ParseMain (script) != CPL_OK)
      exit (1);
    ...
}
```

Zunächst wird mit *Cpl_NewScript()* eine neue `script`-Struktur erzeugt. Außer dem Namen des Skriptes muß ein Pointer auf ein bereits geöffnetes Logfile übergeben werden. Der anschließende Aufruf von *Cpl_ParseMain()* öffnet die Skriptdatei und startet die Übersetzung. Im Falle von Fehlern werden diese auf stderr ausgegeben, und *Cpl_ParseMain()* terminiert mit CPL_ERROR.

Während des Übersetzungsvorganges wird eine Symboltabelle aufgebaut, auf die über die `script`-Struktur zugegriffen wird. Diese Tabelle enthält Einträge für Variablen, Zustandsnamen, Prozedurnamen und Konstanten und wird in `Cpl_NewScript()` mit den `$sys`-Variablen vorbelegt. In einer weiteren Tabelle werden alle im Skript definierten Zustände sowie deren Startadressen (1. Op-Code im Anweisungsblock hinter **on enter**) gesammelt. In jedem Zustand werden die definierten Events mit den Startadressen der Anweisungsblöcke festgehalten. Alle Datenstrukturen, die CPL betreffen, sind in `iSrvCpl.h` vereinbart. Zusammen mit einer Variablen, die den aktuellen Zustand kennzeichnet, bildet die Zustandstabelle die State-Machine, die nun vom Interpreter ausgeführt wird.

Neben dem scheinbar höheren Aufwand für das Erzeugen des P-Codes, hat diese Form der Skriptabarbeitung mehrere Vorteile:

- Zur P-Code-Erzeugung wird vor der eigentlichen Abarbeitung das gesamte Skript syntaktisch geprüft, so daß Fehler schnell entdeckt werden. Bei einer zeilenortientierten Interpretation, wie sie auch in Tcl stattfindet, bleiben Fehler wie falsch geschriebene Variablennamen solange unentdeckt, bis die betreffende Stelle ausgeführt wird.
- Das Skript wird nur einmal interpretiert, was einen geringfügigen Laufzeitvorteil bringt. Bei zeilenorientierter Skriptabarbeitung wird das Skript permanent interpretiert. Wird z.B. eine Schleife 100 mal durchlaufen, so werden Abbruchbedingung und Anweisungsteil entsprechend oft interpretiert.
- Die Realisierung von Sprüngen, also die Fortführung der Abarbeitung an beliebigen Stellen im Skript, z.B. nach Eintritt eines Events, ist schwieriger zu realisieren. Entweder müßte bei einem Sprung jedesmal das gesamte Skript nach dem Sprungziel durchsucht werden, oder aber alle Sprungziele müßten in einem separaten Durchlauf zu Beginn der Abarbeitung in einer Tabelle festgehalten werden.

Vor allem der letzte Punkt hat mich dazu gebracht, CPL-Skripte nicht zeilenorientiert abzuarbeiten.

6.3 Der P-Code-Interpreter

Nach erfolgreicher Übersetzung steht der P-Code in einer Tabelle auf die über die `script`-Struktur zugegriffen wird. Letztere enthält zusätzlich einen Programcounter, ein Index, der auf den jeweils nächsten auszuführenden Op-Code in die Code-Tabelle zeigt. Ein Op-Code ist lediglich eine Struktur mit einem Op-Code und drei optionalen Operanden. Dies können Offsets in die Symboltabelle oder Sprungadressen sein. Tabelle 6.1 zeigt die Umsetzung von CPL Anweisungen in Op-Codes, dabei kennzeichnet sym Operanden, die in der Symboltabelle liegen. Sprungadressen sind mit PC gekennzeichnet.

6.4 Scheduling und Timing

Nach dem Compilieren des Skriptes ist die State-Machine bereits initialisiert, so kann nun die Interpretation des P-Codes mit *Cpl_Schedule()* gestartet werden. Als Parameter wird die `script`-Struktur diesmal in einer übergeordneten `call`-Struktur übergeben, welche die Kontextinformationen des aktuellen Anrufes enthält.

```
main ()
{
    ...
    Cpl_Schedule (&call, evEnter, NULL);
    IsdnMainLoopp ();

} /* main() */
```

Zusätzlich erhält `Cpl_Schedule()` einen Event-Typ und einen optionalen String-Parameter übergeben. Nach der Ausführung des ersten Op-Codes setzt *Cpl_Schedule* einen Timer mit 0 und terminiert. Dadurch, daß nur ein Befehl ausgeführt wird, ist sichergestellt, daß die Kontrolle so schnell wie möglich an die Isdn-Main-Loop zurückgegeben wird, damit ggf. anstehende ISDN-Events behandelt werden können. Da der mit 0 gesetzte Timer sofort abgelaufen ist, wird durch die Handler-Routine erneut *Cpl_Schedule* aufgerufen. Die CPL-Anweisungen, bei denen so verfahren wird, sind in Tabelle 6.1 ohne $\star$ aufgeführt.

CPL statement	**generated op codes**	**CPL statement**	**generated op codes**
accept $\star$	opAccept sym	exec $\star$	opExec sym
redirect $\star$	opRedirect sym	audiorec	opAudiorec sym
<proc-call>	opCall sym	cvt	cvt sym sym sym
enter	enter sym	exit	exit
hangup	opHangup sym	if	opCond PC
	opGoto PC	incr	opIncr sym sym
invite	opInvite	proc	opReturn
timer local/global	opStartTimer sym sym	timer stop	opStopTimer sym

Tabelle 6.1: Op-Codes generated by the script compiler

Die mit $\star$ gekennzeichneten Befehle gelten hingegen als kritisch, d.h. ihre Ausführungszeit kann nicht vorhergesagt werden. Aus diesem Grunde wird für **exec** und **redirect** ein eigener Prozeß erzeugt, dessen stdout/stdin jeweils über eine Pipe mit dem Hauptprozeß verbunden ist. Bei der Ausführung dieser beiden Befehle ist der Hauptprozeß also nur für die Zeit der Prozeßerzeugung blockiert.

6.4.1 Eventbehandlung mit *Cpl_Schedule()*

Wie schon angedeutet, werden eintretende Events mit *Cpl_Schedule()* behandelt. Die möglichen Events sind in `iSrvCpl.h` durch den Aufzählungstyp `t_events` vereinbart. Beim Eintreten eines Events ruft also der betreffende Programmteil *Cpl_Schedule()* mit dem Eventtyp auf. Bei Timeouts und erkannten DTMF-Zeichenfolgen wird in dem dritten Parameter der Timername bzw. die erkannte Zeichenfolge übergeben. *Cpl_Schedule()* sucht in der Event-Tabelle des aktuellen Zustandes nach einer entsprechenden Eventdefinition und setzt die Code-Ausführung an der entsprechenden Stelle fort. Wurde in der Tabelle keine passende Event-Vereinbarung gefunden, so wird das Event ignoriert.

Neue Events, wie beispielsweise das Auftreten von Laufzeitfehlern, bei denen $sysStatus \neq 0$ wird, können also eingeführt werden, indem

1. in `iSrvCpl.h` der neue Typname eingetragen wird,

2. im Skript-Compiler in der Funktion *Cpl_ParseEventDef()* die entsprechende Syntax geparsed wird. *Cpl_EventAdd()* macht den Rest, d.h. die Event-Definition wird in der Tabelle des aktuellen Zustandes zusammen mit dem aktuellen Programcounter abgelegt.
3. An einer entsprechenden Stelle im Interpreter `iSrvCplInt.c` oder `iSrvCplExec.c` das Ereignis abgefragt und *Cpl_Schedule()* aufgerufen wird.

6.5 Testen von CPL-Skripts

Da es beim ernsthaften Einsatz des Call-Servers nicht zumutbar ist, daß während der Behandlung eines Anrufes Fehler im Skript zu seinem Abbruch führen, wird ein Programm benötigt, mit dem Skripte getestest werden können. Das Programm `cpldebug` ist hierfür gedacht. Nach einer Syntaxprüfung wird das Skript abgearbeitet. Die möglichen Events können per Tastatur simuliert werden.

Da `cpldebug` ohne ISDN und Netzwerkzugang arbeitet, beschränkt sich die Ausführung der einzelnen Kommandos auf deren Ausgabe am Bildschirm. Folgender Ausschnitt zeigt die Abarbeitung des Skriptes `red.cpl` (Benutzereingaben sind unterstrichen):

```
cpldebug red.cpl
BEGIN CODE EXECUTION:
 resuming...
 PC=0000  call 'SayHello': branching 0008
 PC=0008  log 'HELLO'
 PC=0009  return
 PC=0001  redirect 'rtp://becks:9090'

>>>S000: Enter event id
   (0=evEnter, 1=Disc, 2=Timeout, 3=DTMF, 4=speech, 5=Reschedule):
```

Während der Code-Ausführung wird der Program-Counter (PC=xxxx) mit dem dazugehörigen Befehl ausgegeben. Nach dem Abarbeiten des **on enter** Blocks des ersten Zustands, bleibt das Programm stehen und wartet auf Eingabe eines Events. S000 ist der augenblickliche Zustand, in dem sich die State-Machine befindet. Alle Zustände eines Skriptes sind beginnend bei 0000 fortlaufend durchnumeriert. Das Skript `red.cpl` sieht wie folgt aus:

```
state FirstState {
    on enter {
      SayHello
      redirect "rtp://becks:9090"
    }
    on disconnect {
      enter FinalState
    }
}

state FinalState {
    on enter {
      hangup
      exit
    }
}
```

```
proc SayHello {
    log "HELLO"
}
```

Da in diesem Skript außer Enter keine anderen Events definiert sind, führt lediglich die Eingabe von 0 bzw. 1, für die Simulation des Enter- bzw. Disconnect-Events zu einer Reaktion des Programms. Durch Angabe eines optionalen Parameters (Integer > 0) beim Aufruf von `cpldebug`, können Symbol- und Zustandstabelle des Interpreters ausgegeben werden:

```
SYMBOL TABLE:
   ind  line  type  current value
   000  00001 var   'sysVersion'
   001  00001 var   'sysCallDuration'
   002  00001 var   'sysDate'
   003  00001 var   'sysDay'
   004  00001 var   'sysTime'
   005  00001 var   'sysStatus'
   006  00001 var   'sysCallingParty'
   007  00001 var   'sysCalledParty'
   008  00001 var   'sysDtmf'
   009  00001 var   'sysHostname'
   010  00001 var   'sysTrace'
   011  00001 var   'sysUsername'
   012  00002 state 'FirstState' StateId = 0
   013  00005 proc  'SayHello' PC = 8
   014  00005 const 'rtp://becks:9090'
   015  00009 state 'FinalState' StateId = 1
   016  00021 const 'HELLO'

STATE TABLE:
  FirstState
    ON ENTER      PC=0000
    ON DISCONNECT PC=0003
  FinalState
    ON ENTER      PC=0005

CODE TABLE:
      0000  CALL SayHello (PC=0008)
      0001  REDIRECT @14
      0002  STOP
      0003  ENTER FinalState
      0004  STOP
      0005  HANGUP
      0006  EXIT
      0007  STOP
      0008  LOG @16
      0009  RET
```

Die Zustandstabelle enthält neben dem Zustandsnamen sowie den definierten Events deren Einstiegspunkte in die Code-Tabelle. Letztere wird am Ende ebenfalls ausgegeben. Die mit `@` versehenen Argumente sind Referenzen in die Symboltabelle.

Kapitel 7

Beispielanwendungen

In diesem Kapitel sollen anhand einiger Beispiel die Möglichkeiten von CPL gezeigt werden. Die Beispiele orientieren sich an der gegenwärtigen Implementierung, d.h. die Möglichkeiten, die sich durch den Einsatz eines Tcl-Interpreters ergeben sind hier nicht berücksichtigt.

7.1 Einfacher Anrufbeantworter

Das folgende CPL-Skript realisiert einen trivialen Anrufbeantworter: Nach Eingang eines Anrufes startet das Skript im Anweisungsblock der **on enter** Definition. Dort wird nach einem Eintrag ins Logfile der Anruf mittels **accept** entgegengenommen. Anschließend wird eine Ansage abgespielt, gefolgt von einem kurzen 1,3 kHz Ton. Nach dem Starten eines Timers wird der eingehende Audiostrom in der Datei `icm.au` maximal 1 Minute lang aufgezeichnet. Bei Ablauf des Timers wird die Verbindung beendet (**hangup**), anschließend terminiert das Skript (**exit**).

```
state IncomingCall {

    on enter {
      log "$sysDate $sysTime: incoming call from $sysCallingParty\n"
      accept
      exec "iAuCat Welcome.au"       # play outgoing message
      exec "iAuTone 1300 100 50"     # generate 1.3 kHz tone for 100 ms
      timer recTimer local 00:01:00  # allow 1 minute recording time
      audiorec "icm.au"              # record incoming msg
      }
    on timer recTimer {
      hangup
      exit
    }
}
```

Soll der Anrufbeantworter erst nach 10 Sekunden aktiviert werden, damit der Anruf beispielsweise an einem anderen Telefon entgegengenommen werden kann, so wäre das Beispiel wie folgt abzuändern:

```
state IncomingCall {

    on enter {
```

```
      timer pickupDelay local 00:00:10
    }
    on timer pickupDelay {
      log "$sysDate $sysTime: incoming call from $sysCallingParty\n"
      accept
      exec "iAuCat Welcome.au"       # play outgoing message
      exec "iAuTone 300 100 50"      # generate 1.3 kHz tone for 100 ms
      timer recTimer local 00:01:00  # allow 1 minute recording time
      audiorec "icm.au"              # record incoming msg
      }
    on timer recTimer {
      hangup
      exit
    }
}
```

Die **accept**-Anweisung, die den Anruf entgegennimmt, wird hier also um 10 Sekunden verzögert. Wurde (an einem Mehrgeräteanschluß) der Anruf an einem anderen Apparat angenommen, so wird ein Disconnect-Event ausgelöst, durch das das Skript terminiert wird.

7.2 Teilnahme an einem RTP-Strom

Das folgende Beispiel beschreibt eine State-Machine bestehend aus vier Zuständen, mit der der Benutzer an einem RTP-Datenstrom teilnehmen kann. Nach Eingabe von IP-Adresse und UDP-Portnummer wird er mittels **redirect** mit dem RTP-Audiostrom verbunden. An diesem Beispiel werden u.a. der Gebrauch der **cvt**-Anweisung sowie die Fehlerbehandlung mittels $sysStatus$ deutlich.

```
state Welcome {

    on enter {
      accept
      exec "sun5/iAuCat ounds/GutenTag.au"  # greeting message
      if { $sysStatus != 0 } {
        exec "sun5/iAuCat sounds/Fehler.au" # error message
        hangup
        exit
      }
      enter GetIPaddr
    }
}

state GetIPaddr {

    on enter {
      exec "sun5/iAuCat sounds/ip.au"
      timer t1 local 00:00:20               # allow 20 sec for IP addr.
    }
    on dtmf "D'*'D'*'D'*'D'#'" {
      cvt ip $sysDtmf "DTMF2IP"             # convert DTMF into IP addr.
      if { $sysStatus != 0 } {
        exec "sun5/iAuCat sounds/ipErr.au"
        enter GetIPaddr }
      else {
```

```
        enter GetPortNo
      }
    }
    on timer t1 {
      exec "sun5/iAuCat sounds/Eingeschlafen.au"
      enter GetIPaddr
    }
}

state GetPortNo {

    on enter {
      exec "sun5/iAuCat sounds/PortNr.au"
      timer t1 local 0:0:05 }              # allow 5 sec. for input
    on dtmf "ddddd" {
      set port $sysDtmf
      if {$port < 1024} {
        exec "sun5/iAuCat sounds/IllPortNo.au"
        enter GetPortNo
      }
      if {$port > 65535} {
        exec "sun5/iAuCat sounds/IllPortNo.au"
        enter GetPortNo
      }
      enter ListenToMbone }
    on timer t1 {
      exec "sun5/iAuCat sounds/Eingeschlafen.au"
      enter GetPortNo
    }
}

state ListenToMbone {

    on enter {
      exec "sun5/iAuCat sounds/BeendenMit.au"
      redirect "rtp://$ip:$port"
      if { $sysStatus != 0 } {
        exec "sun5/iAuCat sounds/Fehler.au" # error message
        hangup
        exit
      }
    }
    on dtmf "#" {
      enter Welcome
    }
}
```

Zur Veranschaulichung ist die State-Machine in Abbildung 7.1 grafisch dargestellt. Die in der folgenden Beschreibung in Klammern gesetzten Zahlen entsprechen den in der Abbildung schwarz hinterlegten Event-Nummern. Nach dem Abspielen einer Begrüßungsnachricht im Startzustand *Welcome* wird der Zustand *GetIPaddr* betreten (2), wo der Benutzer zur Eingabe einer IP-Adresse in Dotted-Decimal-Notation aufgefordert wird, deren Eingabe er mit der $\boxed{\#}$-Taste abschließen muß. Für die Eingabe bleiben ihm 20 s Zeit, danach (2a) wird eine Fehlermeldung ausgegeben und der Zustand erneut betreten, d.h. Eingabeaufforderung und Timerstart werden wiederholt. Nach Erkennung der Eingabe wird diese mit **cvt** umgewandelt und der Zustand *GetPortNo* zur Eingabe der Portnummer betreten (3). Diese wird als fünfstellige Zahl erwartet (**on dtmf 'ddddd'**). Nach Erkennung der Portnummer wird der Zustand *ListenToMbone* betreten (4), in dem der Anrufer mittels **redirect** mit der gewünschten Audiokonferenz verbunden wird.

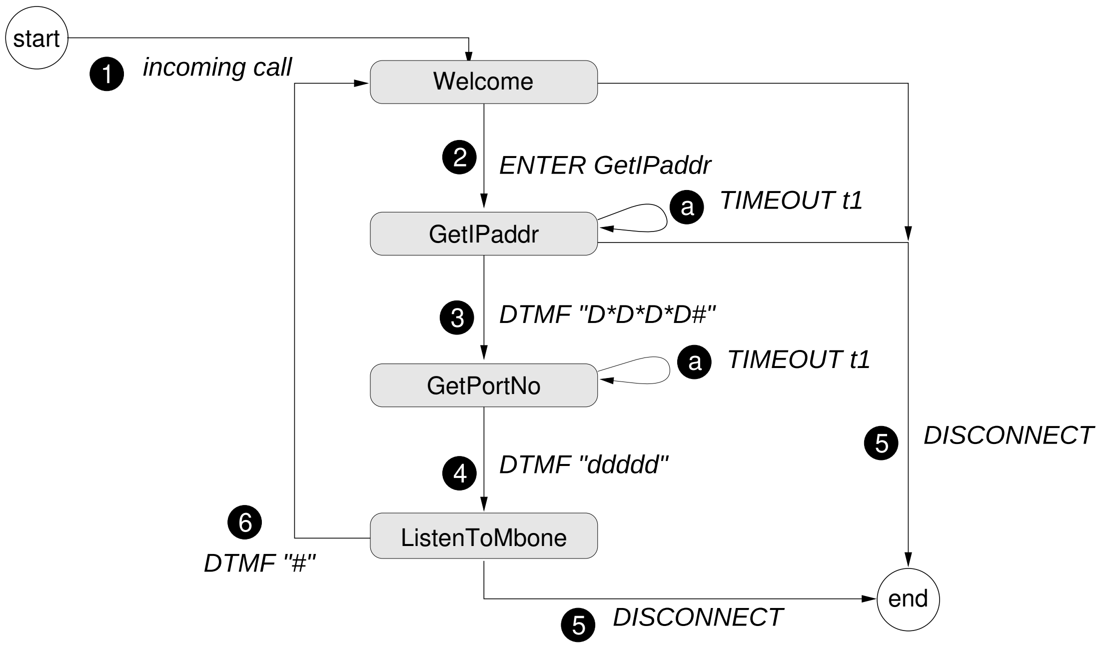

Abb. 7.1: Finite-State-Machine for the MBONE example

Die Konferenzteilnahme kann mit Hilfe der $\boxed{\#}$-Taste beendet werden (6), ohne gleichzeitig die Telefonverbindung zu beenden. Vielmehr gelangt der Anrufer wieder in den Ausgangszustand zurück.

7.3 Erweiterter Anrufbeantworter

Im folgenden Beispiel wird ein erweiterter Anrufbeantworter realisiert. Hier kann der Anrufer durch Eingabe einer Geheimnummer aufgezeichnete Nachrichten abhören. Ungeduldige Anrufer, die wiederum in Besitz einer Geheimnummer sind, können sich mit der gewünschten Person sofort verbinden lassen (**redirect**).

```
# ans.cpl   --  "Enhanced" answering machine
#
# (c) 1997 F. Oertel
#

state Welcome {

    on enter {
      set myPIN 4711
      set sysTrace 1
      accept
      if { $sysTime < "17:00:00" } {
        exec "sun5/iAuCat sounds/GutenTag.au" }     # greeting message
      else {
        exec "sun5/iAuCat sounds/GutenAbend.au" }   # greeting message
      enter Selection
```

```
    }
}

state Selection {

    on enter {
      exec "sun5/iAuCat sounds/Inhalt.au"   # tell user the menu
      timer t local 0:0:08                  # allow 8 secs for input
    }
    on dtmf "1" { enter RecordMessage       }
    on dtmf "2" { enter PlayMessages        }
    on dtmf "3" { enter Redirect            }
    on dtmf "#" { exit                      }
    on timer t {
      exec "sun5/iAuCat sounds/Eingeschlafen.au"
      enter Selection
    }
}

state RecordMessage {

    on enter {
      exec "sun5/iAuCat sounds/Sprechen.au"
      exec "sun5/iTone 1350 200 60"         # beep
      timer icm local 00:01:00              # allow 1 min for icm
      audiorec "icm/i$sysCalls.au"
    }
    on dtmf "#" {                           # terminate recording
      enter Selection }
    on timer icm {
      exec "sun5/iAuCat sounds/Genug.au"    # "thank you for ..."
      exit
    }
}

state PlayMessages {

    on enter {
      AskForPIN
    }
    on dtmf "4711" {                        # we've got a PIN!
      exec "sun5/iAuCat icm/*.au"           # play all recorded messages
     enter Selection
    }
    on timer pin {
      exec "sun5/iAuCat sounds/Eingeschlafen.au"
      enter Selection
    }
}

state Redirect {

    on enter {
      AskForPIN
    }
    on dtmf 4711 {
      timer pin stop
      redirect "phone://35"                 # redirect to my handy
```

```
      enter Selection
    }
    on dtmf "#" {
      enter Selection
    }
    on timer pin {
      exec "sun5/iAuCat sounds/Eingeschlafen.au"
      enter Selection
    }
}

proc AskForPIN {

    exec "sun5/iAuCat sounds/PIN.au"
    timer pin local 0:0:6                   # wait 6 secs for PIN
}
```

Dieses Beispiel zeigt u.a. den Einsatz von Prozeduren. In *AskForPIN* werden ein Audiofile abgespielt sowie ein Timer gestartet. Da die Prozedur aus den Zuständen *Redirect* und *PlayMessages* aufgerufen wird, gilt der Timer jeweils lokal in diesen Zuständen.

7.4 Preiswertes Telefonieren

Ein Dienstanbieter kann teure Auslandsgespräche oder ähnliches zum Ortstarif anbieten: Der Anrufer ruft zunächst den Dienstanbieter an, legitimiert sich durch eine Geheimzahl (oder auch nicht) und wählt dann die gewünschte Nummer, mit der er dann per **redirect** verbunden wird. Folgendes Programm realisiert diesen Dienst (ohne Anruferlegitimation):

```
state Welcome {

    on enter {
      accept
      exec "sun5/iAuCat sounds/GutenTag.au"
      timer DialTimeout local 0:00:20
    }
    on dtmf "D'#'" {
      timer DialTimeout stop
      timer ConnectTimeout local 00:30:00
      redirect "phone://$sysDtmf"
    }
    on timer DialTimeout {
      exec "sun5/iAuCat sounds/att.au"
      exit
    }
}
```

Nach der Ansage hat der Benutzer 20 s lang Zeit eine beliebige Nummer einzugeben. Nachdem er die Eingabe mit der $\boxed{\#}$-Taste beendet hat, wird er sofort mit der gewünschten Nummer verbunden. Die Verbindungsdauer wird durch den Timer *ConnectTimeout* auf 30 Minuten begrenzt. Zu beachten ist, daß während des Gespräches eine weitere Nummer eingegeben und mit $\boxed{\#}$ quittiert werden kann. Das hierbei erneut ausgelöste DTMF-Event beendet die aktuelle Umleitung automatisch, so daß der zweite B-Kanal für das erneute **redirect** sofort wieder zur Verfügung steht. Wichtig ist hier außerdem den Timer für die Nummerneingabe zu beenden, da sonst nach Ablauf der 20 s die Verbindung beendet wird.

Kapitel 8

Bewertung

8.1 Einleitung

In diesem Kapitel folgt abschließend der Versuch einer kritischen Bewertung der erreichten Ergebnisse. Nach einer Aufzählung dessen, was fehlt bzw. nicht rechtzeitig fertig geworden ist, sollen Call-Server und CPL in Hinblick auf ihren praktischen Nutzen betrachtet werden. Der abschließende Abschnitt 8.4 soll Richtungen für eine Weiterentwicklung aufzeigen.

8.2 Was fehlt?

Während die Behandlung von ISDN-Anrufen problemlos funktioniert, wie die Beispiele im letzten Kapitel zeigen, so ist die Behandlung von SIP-Anrufen nicht fertig geworden. Dies lag daran, daß die Behandlung von SIP-Anrufen durch CPL-Skripte ursprünglich nicht vorgesehen war, und die Implementierung der Anrufbehandlung um die ISDN-Library herum implementiert wurde. Dies hat zur Folge, daß die anrufbehandelnden Datenstrukturen um die entsprechenden Kontextinformationen für SIP-Anrufe erweitert werden müssen.

Weiterhin gibt es durch einen Fehler in dem der ISDN-Library zugrundeliegenden Q.921-Modul Probleme mit der ISDN-Rufumleitung. Beim Verbindungsabbau des zweiten B-Kanals terminiert der Call-Server abnormal. Dieser Fehler sollte jedoch relativ leicht zu finden und beseitigen sein. Die empfohlene Verwendung eines CAPI-Interfaces würde dieses Problem umgehen.

Sehr rudimentär ist die implemenierte Behandlung der verschiedenen Audioformate: Auf der RTP-Seite werden lediglich 8kHz A-law oder μ-law Audioströme unterstützt. Auf der ISDN-Seite wird ein A-law Audiostrom erwartet.

8.3 Bewertung

Die Beispiele des letzten Kapitels zeigen, daß mit der Implementierung durchaus sinnvolle und funktionierende Anwendungen realisiert werden können. Die Tatsache, daß Anrufe flexibel mit Hilfe einer Skriptsprache behandelt werden können, eröffnet vor allem im Bereich CTI vielfältige Möglichkeiten, wenngleich diese bereits in zahlreichen kommerziellen Produkten verfügbar sind. Im Gegensatz zu diesen bietet die vorgestellte Implementierung mehrere Vorteile:

- Durch die Verfügbarkeit des Source-Codes und die Einbeziehung von Tcl ist sie leicht erweiterbar und auf andere Plattformen portierbar.
- Der Sprachumfang von Tcl gestattet wesentlich flexiblere Anwendungen, da er nicht auf einen bestimmten Anwendungsbereich, wie beispielsweise Call-Center-Anwendungen spezialisiert ist.
- Es werden nicht nur Telefonanrufe sondern auch solche aus dem Internet berücksichtigt, wodurch sich vielfältige Kopplungsmöglichkeiten beider Netze ergeben. So könnte durch eine kleine Erweiterung des Call-Servers auch das Eintreffen neuer e-Mails einen Anruf zum Empfänger auslösen, falls dieser nicht an seiner Workstation anwesend ist.

So kann die geschaffene Lösung in verschiedenen Bereichen eingesetzt werden:

Endanwender
: Der Endanwender kann sich im Heimbereich quasi sein eigenes elektronisches Sekretariat mit intelligentem Anrufbeantworter etc. realisieren.

Informationsdienste
: Ein Diensteanbieter kann Informationsdienste mit Faxabruf, Rufumleitung, Ansagenaufzeichnung u.s.w. realisieren.

Internet-Provider
: Ein Internet-Provider kann allgemein Zugang zu MBONE-Konferenzen anbieten, oder aber Firmen, Institutionen bieten Zugang zu internen Konferenzen an.

8.3.1 Der Call-Server

Die Verwendung einer CAPI-Schnittstelle wäre für eine Portierung auf andere Plattformen sehr hilfreich, da CAPI für die wichtigsten Plattformen wie OS/2, MS/Windows und UNIX definiert ist. Die gegenwärtig auf der SUN-Plattform eingesetzte ISDN-Library kann relativ leicht mit einer solchen Schnittstelle versehen werden: Anstelle der ISDN-Main-Loop kommuniziert ein eigener CAPI-Prozeß mit dem ISDN-Layer und schreibt alle Events in die vom CAPI definierten Queues. Anwendungen kommunizieren via dieser Queues, derer CAPI eine je registrierter Applikation und eine für das CAPI definiert.

Wie schon in Kapitel 5 erwähnt, ist die gewählte Server-Architektur für die parallele Behandlung mehrerer Anrufe mit CPL-Skripten ungünstig gewählt: Es sollte vielmehr eine Trennung von ISDN-Layer und Skript-Interpretation erfolgen, wie in Abbildung 8.1 gezeigt.

Hierbei gibt es einen Hauptprozeß, der im wesentlichen die Aufgabe hat, bei eingehenden Anrufen aus ISDN und Internet für den Tcl-Anweisungsblock des betreffenden Events einen Thread zu erzeugen. Beim Eintreffen neuer Events oder bei abgelaufenen Timern kann der Hauptprozeß den Thread abbrechen. Bei dieser Architektur muß der Hauptprozeß zu Beginn zwar immer noch das Skript parsen, jedoch kann dieser Aufwand vernachlässigt werden. Meine Versuche, in der gegenwärtigen Implementierung die

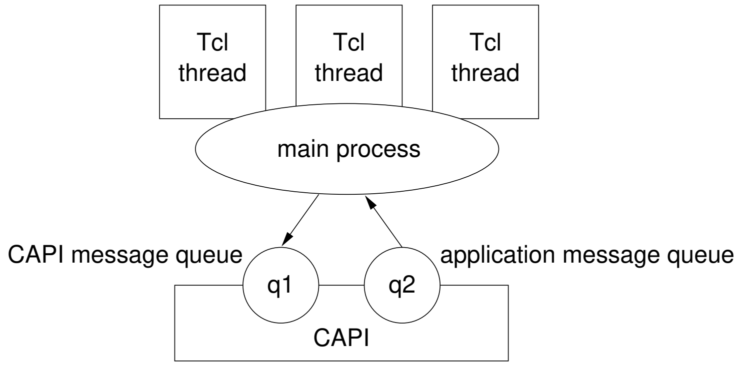

Abb. 8.1: Proposed call server architecture

Zeit für das Compilieren (Erzeugung des P-Codes) eines CPL-Skriptes mit Hilfe des `time`-Kommandos sowie dem Profiler zu messen, scheiterten, da die Meßwerte im Bereich der Timer-Auflösung lagen.

Die Realisierung dieser Architektur setzt jedoch einen thread-tauglichen Tcl-Interpreter voraus.

Ein weiteres Problem ist die erforderliche Rechenleistung, die insbesondere durch folgende Dinge recht hoch ist:

- DTMF-Erkennung
- Spracherkennung
- Wandlung der Audioformate

Die für die Implementierung benutzte IPX Sparcstation war laut `vmstat` mit der Behandlung eines Anrufes, eingeschalteter DTMF-Erkennung und bei gleichzeitigem Abspielen eines Audiofiles schon zu 50% ausgelastet, wobei die DTMF-Erkennung bereits etwa 20% der Auslastung ausmachten. Aus diesem Grunde wird die DTMF-Erkennung nur in solchen Zuständen eingeschaltet, in denen **on dtmf** bzw. **on ascii dtmf** benutzt werden. Bei der hier nicht praktisch erprobten Spracherkennung dürfte der Rechenaufwand um ein Vielfaches höher liegen. Auch sollte nur bei vorhandenen **on speech** Definitionen eingeschaltet werden. Die Wandlung der Audioformate wird vor allem bei besonders rechenintensiven Algorithmen wie die für ADPCM [ITU90] oder LPC-10E [ITU92] ein Problem, da sich der Rechenaufwand mit der Anzahl der parallel behandelten Audioströme multipliziert. Für die gleichzeitige Behandlung von 30 Anrufen, wie sie beim S_{2M}-Anschluß (PRI) vorkommt, wäre dann eine entsprechende Rechenleistung vorzusehen.

8.3.2 Die Skriptsprache

Die Skriptsprache scheint mir bei Einsatz des Tcl-Interpreters nicht unbedingt verbesserungsbedürftig. Es sind alle sprachlichen Konstrukte zur Realisierung von Anwendungen sowohl als State-Machine als auch in Form sequentiell ablaufender Programme vorhanden.

Mehrere Mediendatenströme

Für die Einbeziehung mehrerer Mediendatenströme wie Audio und Video, die ja für ISDN-Teilnehmer durchaus realistisch sind, müßte allerdings noch eine Unterstützung eingebaut werden. Für Anrufe aus dem ISDN stelle ich mir vor, dem Skript beim Aufruf die entsprechenden Kanäle in Form von Filedescriptoren als Parameter mitzugeben. In einem Skript, in dem nur Video behandelt wird, wäre es z.B. unsinning stdout immer fest mit dem Audiostrom des eingehenden Anrufes zu verbinden. Dies erhöht die Existenzberechtigung der schon infrage gestellten `isdn.cfg` Konfigurationsdatei.

Zeitangaben bei Timern

Die beim Starten von Timern ursprünglich in Millisekunden angegebenen Zeiten werden nun, wie in Anhang B beschrieben in der Notation HH:MM:SS angegeben, was bei großen Zeitangaben wesentlich lesbarer ist. Aufgrund der Implementierung können Zeiten im Millisekundenbereich ohnehin nicht exakt eingehalten werden. In der Implementierung ist diese Syntax bis zum Ende der Arbeit möglicherweise noch nicht berücksichtigt worden.

Globale Fehlerbehandlung

Die Behandlung von Fehlern könnte durch eine ereignisorientierte Fehlerbehandlung bei komplexen Anwendungen wesentlich vereinfacht werden. Bisher muß hinter jeder CPL-Anweisung, bei der ein Fehler auftreten könnte, eine Abfrage mit $sysStatus$ stehen. Ein neues Event **on error** könnte global definiert werden und Fehler behandeln. Spezialfälle in bestimmten Zuständen könnten wie bei Timern mit einer lokalen Definition überdeckt werden.

8.4 Ausblick

Neben der zukünftigen Verwendung eines Tcl-Interpreters wird sicher die Anrufbehandlung Verbesserungen erfahren müssen. So wäre es denkbar, daß bereits laufende Skripte weitere eingehende Anrufe behandeln. Hierfür sollte dann jeder Befehl, der auf einem Mediendatenstrom operiert mit einer Anruf-Id parametrisiert werden. Generell sollten diese Befehle flexibler einsetzbar sein. Derzeit sind Befehle wie **hangup** oder **exec** auf den Anruferdatenstrom bezogen. So wird z.B. bei einer bestehenden ISDN-ISDN-Rufumleitung durch **hangup** nicht nur die Umleitung sondern der eingehende Anruf beendet. Eingehende und ausgehende Datenströme sollten jedoch getrennt und gleichbehandelt werden. CPL-Anweisungen müßten hierfür mit einem Parameter versehen werden, der den entsprechenden Datenstrom identifiziert. So kann dann z.B. ein Audiofile nicht nur dem Anrufer sondern auch anderen Konferenzteilnehmern abgespielt werden.

Weiterhin wäre auf der Netzwerkseite die Einbeziehung entsprechender Session-Control Protokolle sinnvoll, da mit SIP derzeit lediglich der Konferenzaufbau unterstützt wird[1]. Der Netzwerkbenutzer sollte jedoch mindestens über die gleichen Möglichkeiten verfügen, wie der Telefonanrufer, d.h. auch er sollte

[1]Nach Aussagen der Autoren soll SIP in Zukunft auch den Verbindungsabbau signalisieren können

eine Verbindung beenden können und eine Signalisierungsmöglichkeit analog der DTMF-Zeichengabe besitzen.

So gesehen stellt die gegenwärtige Implementierng von Call-Server und CPL nur den Anfang einer universellen Lösung zur Integration von Telefonnetzteilnehmern in MBONE-Konferenzen dar.

Anhang A

Verwendete Abkürzungen

BRI	Basic Rate Interface, dt.: ISDN-Basisanschluß
CAPI	Common ISDN API
CSTA	Computer-Supported Telecommunications-Applications
CTI	Computer Telephony Integration
DTMF	Dual-Tone Multiple-Frequency
ECMA	European Computer Manufactures Association
FOKUS	Forschungsgruppe Offene Kommunikationssyteme der GMD
GMD	Gesellschaft für Mathematik und Datenverarbeitung
IP	Internet Protocol (RFC 791)
ISDN	Integrated Services Digital Network, dt.: diensteintegrierendes digitales Netzwerk
ITU	International Telecommunication Union
MBONE	Multimedia Backbone, virtuelles Subnetz im Internet, innerhalb dessen IP-Multicast-Datenströme übertragen werden
MSN	Multiple Subscriber Number, Mehrfachrufnummer im ISDN
PPP	Point-to-Point Protocol (RFC 1661)
PRI	Primary Rate Interface, dt.: ISDN-Primärmultiplexanschluß, kurz: S_{2M}
PSTN	Public Switched Telephone Network, dt.: leitungsvermitteltes Telefonnetz
RTP	Real-Time Transport Protocol (RFC 1889)
SAP	Session Announcement Protocol
SDP	Session Description Protocol
SIP	Session Initiation Protocol
SLIP	Serial Line IP (RFC 1055), über serielle Leitungen geführtes IP
TAPI	Telephony API (Intel/Microsoft)
Tcl	Tool Command Language
TSAPI	Telephony Services API (Novell)
TTL	time-to-live
UDP	User Datagram Protocol (RFC 768)
URL	Uniform Resource Locator (RFC 1738)
VStO	Vermittlungsstelle im Ortsnetz, Ortsvermittlungsstelle
WWW	World Wide Web

Anhang B

CPL Grammar

```
application   ::= statedef { statedef | procdef | eventdef }*
statedef      ::= stateheader statebody
stateheader   ::= "state" statename
statename     ::= string
statebody     ::= "{" {eventdef}+ "}"
eventdef      ::= "on" eventspec procbody
eventspec     ::= { [ "ascii" ] "dtmf" matchpattern |
                    "speech" matchpattern |
                    "timer"  timername | "enter" | "disconnect" }
matchpattern  ::= stringexpr
timername     ::= string
procdef       ::= procheader procbody
procheader    ::= "proc" procname
procname      ::= string
procbody      ::= "{" {statement}* "}"
statement     ::= { "accept" | hangupcmd | invitecmd |
                     reccmd | statetrans | logcmd |
                     fwdcmd | timercmd | cvtcmd | tclcmd }
hangupcmd     ::= "hangup" [ reasoncode ]
reasoncode    ::= stringexpr
statetrans    ::= "enter" statename
reccmd        ::= "audiorec" filename
filename      ::= stringexpr
logcmd        ::= "log" stringexpr
fwdcmd        ::= "redirect" destination
destination   ::= phone-url | rtp-url
phone-url     ::= "phone://" [ "+" countrycode "-" ] phoneno
countrycode   ::= {digit}+
phoneno       ::= digit { digit | '-' }*
rtp-url       ::= "rtp://" hostspec [ ":" portnumber ]
hostspec      ::= string
portnumber    ::= {digit}+
cvtcmd        ::= "cvt"  varname  stringexpr  stringexpr
invitecmd     ::= "invite" from info media
from          ::= stringexpr
info          ::= stringexpr
media         ::= stringexpr
```

```
timercmd      ::= "timer" timername { timerstart | "stop" }
timerstart    ::= { "local" | "global" } timeout
timeout       ::= hours":"minutes":"seconds
hours         ::= integer
minutes       ::= integer
seconds       ::= seconds
tclcmd        ::= branchcmd | execcmd | "exit" | incr |
                  procname | setcmd | whilecmd }
setcmd        ::= "set" varname stringexpr
execcmd       ::= "exec" command
branchcmd     ::= "if" condition procbody [ "else" procbody ]
condition     ::= "{" stringexpr operator stringexpr "}"
operator      ::= ">", "<", "==", ">=", "<=", "!="
incr          ::= "incr" varname [ increment ]
increment     ::= stringexpr
whilecmd      ::= "while" condition procbody

varname       ::= string
stringexpr    ::= """ string """ | "$" varname | integer
varname       ::= string
string        ::= {alphadigit}+
alphadigit    ::= digit | alpha
integer       ::= [ "-" ] {digit}+
digit         ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
```

Repetition of curly bracketed groups are indicated by an asterisk (*) or plus sign (+). An asterisk indicates that the group is optional and can be further repeated any number of times; a plus sign indicates that the group must by present and can by further repeated any number of times. Optional elements are grouped using square brackets ([]).

Anhang C

Program Files

C.1 Call Server Components

Filename	Meaning
iSrv.c	main program of server
iSrvCfg.c	configuration file handling
iSrvStates.c	ISDN call state handling
iSrvCplComp.c	CPL script compiler
iSrvCplInt.c	CPL script interpreter
iSrvCplExec.c	CPL script execution
iSrvDTMF.c	DTMF detection
iSrvUtils.c	misc. utilities
iSrvAudio.c	au file playing
iSrvPlay.c	iPlay service
iSrvScript.c	iScript service
iSrvPhone.c	iPhone service
iSrvRTP.c	iRTP service

C.2 Other Files

cpldebug.c	CPL script file debugger
iAuCat.c	write audio file payload to stdout
iPhone.c/tcl	iPhone client C and Tcl/Tk source
iSpell.c	spells a given number (using audio files)
iTone.c	generates tones of specified legnth and frequency

Anhang D

CPL system variables

sysCallDuration (read only)

Returns the duration of the current connection, in the format HH:MM:SS. Prior to execution of an **accept** command this value is undefined.

sysCalledParty (read only)

If the current call was originated by an ISDN subscriber, this variable contains the called party number, otherwise „<unknown>“. If the call is a SIP call the content of the To: header field of the INVITE request is returned, e.g. „foertel@ibm.net“.

sysCallingParty (read only)

If the current call was originated by an ISDN subscriber, this variable contains the calling party number, otherwise „<unknown>“ is returned. If the call is a SIP call the content of the From: header field of the INVITE request is returned.

sysCalls (read only)

Number of times this script was called by the server (not the number of actually accepted calls). Currently not implemented.

sysConnections (read only)

Number of calls actually accepted by this script (since server startup). Currently not implemented.

sysDate (read only)

Returns the current date in the format YYYY/MM/DD

sysDay (read only)

Returns the current day of week (1=monday, ..., 7)

sysDtmf

Contains the last DTMF key sequence recognized by an **on dtmf** or **on ascii dtmf** trigger.

sysHostname (read only)

Returns the current hostname.

sysOrigin

Returns „phone“ if the current call is a phone call. If a SIP INVITE request is being processed, „sip“ is returned.

sysStatus (read only)

Contains 0, if the last CPL command was executed successfully. Values greater than 0 are indicating errors.

sysTime (read only)

Returns the current time in 24 hour format HH:MM:SS

sysTrace

Returns 1 if command tracing is on, otherwise 0. Tracing may be enabled or disabled by setting this variable. Tracing informations are written to stderr. Command tracing is turned off by default.

sysUsername (read only)

Returns the current user name.

Anhang E

CPL Statements

E.1 CPL statements

accept

Accepts the current call (the one which invoked the current script). If the call was originated by an ISDN subscriber, a connect indication is issued. If the call was a SIP INVITE request, a SIP reply is sent with response code 200.

audiorec $filename$

Writes the incoming media stream (audio) to the file $filename$. An au file header is written at the beginning of the file. Recording stops when an event occurs or when the current state is left. If you want to delimit recording time, you have to start a timer before. Recording stops upon reception of the timeout event.

cvt $varname\ stringexpr\ cvtfct$

Similar to the **set** command $stringexpr$ is assigned to the variable $varname$. In addition $cvtfct$ specifies a conversion function to be performed on $stringexpr$.

enter $statename$

Changes to the specified state. All running timers are stopped. If the new state has an **on enter** definition, execution resumes its statement block.

hangup [$reasoncode$]

Disconnects from the current call. After reception of the disconnect indication from the ISDN layer, a disconnect event is generated.

log $stringexpr$

Writes text to the script log file.

redirect $destination$

Redirects the current call to another destination, which may be either a telephone number or a conference address. The destination is specified by an URL. A phone URL redirects the call to another telephone number, which is a component of the URL. A RTP URL connects the call with the RTP media stream, whose IP address and port number is supplied within the URL. Redirection is suspended when an event occurs, such as a disconnect.

timer $timername$ [(**local** | **global**) $timeout$ | **stop**]

The **timer** command is used to start and stop timers. The command „**timer** $tname$ **start** $time$“ starts a timer named $tname$. When the specified time (in milliseconds) has been elapsed a timer event is triggered. This event may be trapped using an „**on timer** $timername$“ event definition. Use **timer** $timername$ **stop** to stop the specified timer.

E.2 Tcl statements

$procname$

Invokes the procedure $procname$. Note, that there is no way to pass parameters to the called procedure. Also note, that the procedure call does not change the current state.

exec $stringexpr$

Executes a system command, stdin and stdout are connectedto the B channel i.e. the RTP stream of the incoming call. All output on stderr is sent to the script log file.

exit

Terminates the current script. If a ISDN call is pending, it is disconnected before.

if $condition\ procbody1$ [**else** $procbody2$]

Evaluates $condition$ and continues execution in either $procbody1$ or $procbody2$.

incr $varname$ [$increment$]

Increments the integer value stored in $varname$ by *increment*. If the optional increment parameter is omitted, 1 is added to $varname$

set $varname\ stringexpr$

Assigns $stringexpr$ to the variable specified by $varname$.

while $condition\ procbody$

Executes the statements contained in $procbody$ if $condition$ evaluates to true. This process repeats until $condition$ evaluates to false.

Anhang F

CPL Run-Time-Errors

Message numbers starting with 1 are generated by the script compiler, so in most cases they are syntax errors. Those numbers starting with 2 are issued by the pseudo code interpreter.

101 **Message:** No state definitions found in source file

Cause: The script file contains no **state** clause. A script file must have at least one state definition.

102 **Message:** Initial state must have an **on enter** definition

Cause: The first state definition in a script must have an **on enter** event definition, because it serves as the entrypoint for the script.

103 **Message:** Multiple enter event definitions for the state `'x'`

Cause: More than one **on enter** clause was found in a state definition.

104 **Message:** String too long

Cause: The maximum number of characters for a string has been exceeded, which is defined by the MAX_STRING_LENGTH macro in `iSrvCpl.h`. Currently this value is set to 256 characters.

105 **Message:** Unterminated string

Cause: While parsing a string constant an opening quote was found but no closing one.

106 **Message:** Unexpected end of file

Cause: While parsing a command the end-of-file was detected prematurely.

107 **Message:** Symbol too long

Cause: The maximum number of characters per symbol has been exceeded, which is defined by the MAX_SYMBOL_LENGTH macro in `iSrvCpl.h`. Currently this value is set to 256 characters.

108 **Message:** Zero length variable name

Cause: A $ sign was found but no variable name.

109 **Message:** Unknown symbol 'X'

Cause: An unknown symbol was found by the parser.

110 **Message:** String expression expected for audio filename

Cause: The filename in an **audiorc** statement is missing.

112 **Message:** `'{'` expected

Cause: The { at the beginning of a statement block is missing.

113 **Message:** Operand expected

Cause: An error occured while parsing a condition.

114 **Message:** Operator $<$, $>$, $==$, $>=$, $<=$ or $!=$ expected

Cause: An error occured while parsing a condition.

115 **Message:** '}' expected

Cause: The closing } at the end of a statement block was not found.

116 **Message:** Variable name expected

Cause: The variablename for a **set**, **cvt** or **incr** statement was not supplied.

117 **Message:** '$x' is a read only variable

Cause: An attempt was made, to assign a value to a read-only variable using a **set** or **cvt** command.

118 **Message:** String expression expected

Cause: The 2. parameter of a **cvt** statement is not a string expression.

119 **Message:** String expression expected

Cause: The 3. parameter of a **cvt** statement is not a string expression.

120 **Message:** Expected statename

Cause: The state name after an **enter** statement is missing

121 **Message:** Eventdefition starting with 'on' expected

Cause: While parsing a statebody an other symbol then **on** was found.

122 **Message:** String expression expected for match pattern

Cause: After an **on dmtf** the match pattern string is missing.

123 **Message:** Timer name expected

Cause: In a **on timer** clause the timer name is invalid or missing.

124 **Message:** Expected event specification

Cause: After an **on** a symbol other than **disconnect**, **enter**, **timer**, **dtmf** or **speech** was found.

125 **Message:** Command string expression expected

Cause: The string expression in a **exec** statement is invalid or missing.

126 **Message:** String expression expected

Cause: The 1. parameter of an **invite** statement is not a string expression.

127 **Message:** String expression expected

Cause: The 2. parameter of an **invite** statement is not a string expression.

128 **Message:** String expression expected

Cause: The 3. parameter of an **invite** statement is not a string expression.

129 **Message:** String expression expected

Cause: The string argument for a **log** statement is invalid or missing.

130 **Message:** Procedure name expected

Cause: An invalid or no procedure name was found in a **proc** definition.

131 **Message:** Statement block beginning with '{' expected

Cause: The { at the beginning of a statement block is missing.

132 **Message:** String expression expected for destination

Cause: The string expression in an **redirect** statement is invalid or missing.

133 **Message:** String expression expected

Cause: The string expression in an **set** statement is invalid or missing.

134 **Message:** Statename expected

Cause: An invalid or no state name was found in a **state** definition.

135 **Message:** '{' expected

Cause: An other symbol than { was found after the state name in a state definition.

136 **Message:** The are no event definitions for state 'x'

Cause: An empty state definition was found. A state definition must have at least one event definition.

137 **Message:** Statement expected

Cause: An invalid statement has been found in a statement block.

138 **Message:** Timer name expected

Cause: An invalid timer name was found in a **timer** statement.

138 **Message:** String expression expected for timeout

Cause: In a timer statement the timeout string expression after the **local/global** was missing.

139 **Message:** Expected 'local', 'global' or 'stop'

Cause: In a **timer** statement a symbol other than **local**, **global** or **stop** was found after the timer name.

140 **Message:** Undefined state name

Cause: The state name used in an **enter** statement is not declared within the script file, i.e. there is no state definition using this name.

141 **Message:** Undefined proc name

Cause: The procedure name used in an **call** statement is not declared within the script file, i.e. there is no procedure definition using this name.

142 **Message:** Unexpected symbol

Cause: An other symbol than **state**, **proc** or **on** was found outside a **state** or **proc** block.

201 **Message:** Variable 'x' not declared

Cause: A variable has been referenced which has not been assigned a value previously. Use the **set** statement to assign a value to a variable.

202 **Message:** Illegal CPL instruction, PC=xxxx

Cause: An illegal op code was found by the pseudo code interpreter. The script execution stopped at program counter xxxx. Use the `cpldebug` program to determine the exact location.

203 **Message:** Can't open file 'x' for recording audio

Cause: The file named 'x' could not be opend for recording the incoming audio stream. Check permissions!

204 **Message:** Error forking in server process

Cause: The fork() system call for an **exec** command failed.

205 **Message:** Too many timers active at the same time
Cause: The maximum number of concurrently active timers has been exceeded. This value is defined by the MAX_TIMERS macro in the `iSrvCpl.h` file and is currently set to 10 concurrent timers.

206 **Message:** Unexpected URL type in destination specification
Cause: An invalid URL type was found in the destination address of a **redirect** statement.

207 **Message:** Syntax error in URL `'x'`
Cause: The URL you passed to a **redirect** statement has an invalid syntax.

Anhang G

Reserved Words in CPL

accept	ascii	audiorec	cvt
disconnect	dtmf	else	enter
exec	exit	global	hangup
if	incr	invite	local
log	on	proc	redirect
speech	set	state	stop
timer	while		

Anhang H

Source-Code

H.1 Call Server (main)

H.1.1 iSrv.h

```
/*
** client requests / indications to server
*/
#ifndef _ISRV_H
#define _ISRV_H

#ifndef TRUE
#define TRUE                    1
#endif
#ifndef FALSE
#define FALSE                   0
#endif

#define REQ_SHUTDOWN            1
#define REQ_CALL                2
#define REQ_DIAL                3
#define REQ_DISC                4
#define REQ_ALERTING            5
#define REQ_CONNECT             6
#define IND_PORTNUM             7

/*
** indications from server
*/
#define IND_ERROR               0
#define IND_CALL                10
#define IND_CONNECT             20
#define IND_CHANNEL             30
#define IND_ALERTING            40
#define IND_RELEASE             50
#define IND_CALLID              60
#define IND_DISCONNECT          70

typedef struct {
        int                     type;
        int                     callId;
        int                     errCode;
        char                    errMsg[132];
        char                    calledNumber[80];
        char                    callingNumber[80];
        unsigned short          portNum;
} t_req;


#define PORTNUM                 4711

#define ZERO(xxx)               memset (&xxx, '\0', sizeof (xxx));
```

```
#endif
```

H.1.2 iSrv.c

```
/* iSrv.c -- isdn server
**
** (c) Frank Oertel 1996
**
** 01.11.1996, FO, created
** 30.12.1996, FO, script service added
** 31.12.1996, FO, script service tested
** 06.01.1997, FO, MBONE service added
** 19.01.1997, FO, bug fixed
*/

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/signal.h>
#include <netinet/in.h>
#include <fcntl.h>
#include <time.h>
#include <netdb.h>
#include <sys/wait.h>

#include "iSrv.h"
#include "iSrvCfg.h"
#include "iSrvPhone.h"
#include "iSrvPlay.h"
#include "iSrvScript.h"
#include "iSrvUtils.h"
#include "iSrvMbone.h"

#include "IsdnLib.h"
#include "/u/czahl/ibm.paul/audio.h.paul"

#define DEBUG(x)                x

#define CFG_FILENAME            "isdn.cfg"
#define LOG_FILENAME            "srv.log"

static  void                    SigChild ();
static  void                    HandleIncomingCall (t_IsdnCallInd *ind);
static  int                     Init (void);
static  void                    IsdnCb (t_IsdnInd *ind);


/*
** internals:
*/

static void SigChild ()
{
        int                     status;
        t_callState*            state;
extern  void                    ExecStdOutCb (void *ClientData, int mask);

        DEBUG (fprintf (stderr, "SIGCHILD\n");)
        if ((state = CallStateFindByPid (wait (&status))) == NULL) {
          DEBUG (fprintf (stderr, "PID not found, script probably already terminated\n");)
          signal (SIGCHLD, SigChild);
          return;
        }

        signal (SIGCHLD, SigChild);

        DEBUG (fprintf (stderr, "script noch da, status=%d\n", WEXITSTATUS (status));)
        state->script->status = WEXITSTATUS (status);
```

```
        if ( WEXITSTATUS (status) != 0 ) {
          IsdnDeleteFileHandler (state->script->execStdOut);
          close (state->script->execStdOut);
          state->script->execStdOut = -1;
          Cpl_Schedule (state, evReschedule, NULL, FALSE); }
        else
          if (state->script->execStdOut > -1)
            IsdnCreateFileHandler (state->script->execStdOut, ISDN_READABLE, ExecStdOutCb, state);


} /* SigChild() */




static void SigInt ()
{

        Msg ("Got interrupt signal, terminating...");
        MsgClose ();

        IsdnFini ();

        exit (0);

} /* SigInt() */



static void HandleIncomingCall (t_IsdnCallInd *ind)
{
        char*                   p;
        char*                   srv;
        char*                   srvArgs;
        char*                   srvName;
        t_callState             *state;

        if ((state = CallStateNew ()) == NULL) {
          Msg ("ERROR: Too many ISDN connections");
          return;
        }

        state->state = SRV_S_CALLING;
        state->callId = ind->callId;
        strcpy (state->callingParty, ind->callingPartyNo.number);
        strcpy (state->calledParty, ind->calledPartyNo.number);

        if ((srv = FindServiceDefinition (state->callingParty, state->calledParty)) == NULL) {
          Msg ("%d: No service definition", state->callId);
          return;
        }

        srvName = malloc (strlen (srv));
        srvArgs = malloc (strlen (srv));
        p = srv;
        p = GetSym (p, srvName);
        p = GetSym (p, srvArgs);

        Msg ("%d: Servicing call with '%s' service", state->callId, srvName);
        if (strcmp (srvName, "iPhone") == 0) {
          state->ChannelIndHandler      = iPhoneHandleChannelInd;
          state->ConnectIndHandler      = iPhoneHandleConnectInd;
          state->DisconnectIndHandler   = iPhoneHandleDisconnectInd;
          state->ReleaseIndHandler      = iPhoneHandleReleaseInd;
          iPhoneIncCall (state);
          return;
        }

        if (strcmp (srvName, "iPlay") == 0) {
          state->ConnectIndHandler      = iPlayHandleConnectInd;
          state->ChannelIndHandler      = iPlayHandleChannelInd;
          state->DisconnectIndHandler   = iPlayHandleDisconnectInd;
          state->ReleaseIndHandler      = iPlayHandleReleaseInd;
```

```
          iPlayIncCall (state, srv);
          return;
        }

        if (strcmp (srvName, "iScript") == 0) {
          state->ConnectIndHandler      = iScriptHandleConnectInd;
          state->ChannelIndHandler      = iScriptHandleChannelInd;
          state->DisconnectIndHandler   = iScriptHandleDisconnectInd;
          state->ReleaseIndHandler      = iScriptHandleReleaseInd;
          if (iScriptIncCall (state, srv) == -1) {
            state->ConnectIndHandler    = NULL;
            state->ChannelIndHandler    = NULL;
            state->DisconnectIndHandler = NULL;
            state->ReleaseIndHandler    = NULL;
          }
          return;
        }

        if (strcmp (srvName, "iMbone") == 0) {
          state->ConnectIndHandler      = iMboneHandleConnectInd;
          state->ChannelIndHandler      = iMboneHandleChannelInd;
          state->DisconnectIndHandler   = iMboneHandleDisconnectInd;
          state->ReleaseIndHandler      = iMboneHandleReleaseInd;
          iMboneIncCall (state, srv);
          return;
        }

        Msg ("Unknown service '%s'", srv);

} /* HandleIncomingCall() */



static int Init (void)
{
        signal (SIGCHLD, SigChild);
        signal (SIGINT, SigInt);

        if (MsgOpen (LOG_FILENAME) == -1) {
          fprintf (stderr, "Error opening server log file '%s'\n", LOG_FILENAME);
          return -1;
        }

        if (ReadCfgFile (CFG_FILENAME) != 0) {
          Msg ("Error reading configuration file '%s'\n", CFG_FILENAME);
          return -1;
        }

        if (CallStateInit () == -1)
          return -1;
        if (IsdnInitialize (IsdnCb) == -1)
          return -1;
        if (iPhoneInit () == -1)
          return -1;
        Msg ("ISDN server started");

        return 0;

} /* Init() */



void IsdnCb (t_IsdnInd *ind)
/*
** We've got an event from the ISDN device, here we try handle it
*/
{
        t_callState*            state;

        if (ind->ind.indType != ISDN_CALL_IND)
          if ((state = CallStateFind (ind->ind.callId)) == NULL) {
            fprintf (stderr, "Non-existent callId %d received from ISDN!\n", ind->ind.callId);
            return;
```

```
    }

  switch (ind->ind.indType) {
    case ISDN_ALERT_IND:
      DEBUG (fprintf (stderr, "%d: Alert indication!\n", ind->ind.callId);)
      break;
    case ISDN_CALL_IND:
      Msg ("%d: Incoming call, calling # is '%s', called # is '%s'",
           ind->ind.callId, ind->callInd.callingPartyNo.number, ind->callInd.calledPartyNo.number);
      HandleIncomingCall (&ind->callInd);
      break;
    case ISDN_CHANNEL_IND:
      DEBUG (fprintf (stderr, "%d: Channel indication!\n", ind->ind.callId);)
      state->bChannelFd = ind->channelInd.fd;
      if (state->ChannelIndHandler == NULL)
        fprintf (stderr, "No handler for B-channel indications defined!\n");
      else
        state->ChannelIndHandler (state);
      break;
    case ISDN_CHARGE_IND:
      DEBUG (fprintf (stderr, "%d: Charge indication!\n", ind->ind.callId);)
      break;
    case ISDN_CONNECT_IND:
      DEBUG (fprintf (stderr, "%d: Connect indication!\n", ind->ind.callId);)
      state->state = SRV_S_CONNECTED;
      time (&(state->startTime));
      if (state->ConnectIndHandler == NULL)
        fprintf (stderr, "No handler for connect indications defined!\n");
      else
        state->ConnectIndHandler (state);
      break;
    case ISDN_DISCONNECT_IND:
      Msg ("%d: Disconnected", state->callId);
      if (state->DisconnectIndHandler == NULL)
        fprintf (stderr, "No handler for disconnect indications defined!\n");
      else
        state->DisconnectIndHandler (state);
      state->state = SRV_S_DISCONNECTED;
      break;
    case ISDN_PROGRESS_IND:
      DEBUG (fprintf (stderr, "%d: Progress indication!\n", ind->ind.callId);)
      break;
    case ISDN_RELEASE_IND:
      DEBUG (fprintf (stderr, "%d: Release indication!\n", state->callId);)
      IsdnDeleteFileHandler (state->bChannelFd);
      state->bChannelFd = -1;
      if (state->ReleaseIndHandler == NULL)
        fprintf (stderr, "No handler for release indications defined!\n");
      else
        state->ReleaseIndHandler (state);
      CallStateDelete (state);
      break;
  }

} /* IsdnCb() */


void main (int argc,  char *argv[])
{

  if (Init () == -1)
    exit (1);

  IsdnMainLoop ();

  IsdnFini ();


} /* main() */
```

H.2 Script Service

H.2.1 iSrvScript.h

```
/*
** iSrvScript.h
*/

#ifndef _ISRVSCRIPT_H
#define _ISRVSCRIPT_H

#include "iSrvStates.h"

extern  void                    iScriptHandleChannelInd (t_callState* state);
extern  void                    iScriptHandleConnectInd (t_callState* state);
extern  void                    iScriptHandleDisconnectInd (t_callState* state);
extern  void                    iScriptHandleReleaseInd (t_callState* state);
extern  int                     iScriptIncCall (t_callState* state, char* scriptFn);

#endif
```

H.2.2 iSrvScript.c

```
/*
** iSrvScript.c  --  Call Script Handling
**
** 31.12.1996, FO, created
** 05.01.1997, FO, some modifications
** 08.01.1997, FO, cosmetic changes
** 27.01.1997, FO, command tracing added
** 07.02.1997, FO, support for global CPL events
** 09.02.1997, FO, garbish removed
*/

#include <stdio.h>
#include <fcntl.h>
#include <signal.h>

#define DEBUG(x)                x
#include "iSrv.h"
#include "iSrvUtils.h"
#include "iSrvCplInt.h"
#include "iSrvCplComp.h"
#include "iSrvCplExec.h"
#include "iSrvScript.h"
#include "iSrvAudio.h"
#include "iSrvDTMF.h"
#include "IsdnLib.h"

#define LOGFILE_MODE            0644

/*
** forward declarations of local functions
*/

static  void                    iScriptBchannelCb (void *ClientData, int mask);


/*
** internal functions
*/

static void iScriptBchannelCb (void *ClientData, int mask)
{
        t_callState*            state = (t_callState*)ClientData;
        int                     bFree;
        char                    buf[512];
        int                     i;
        int                     n;
        int                     rPtr, wPtr;
```

```
        n = read (state->bChannelFd, buf, sizeof (buf));
        if (n < 0) {
          perror ("read (bChannel)");
          return;
        }
        state->bChannelBytes += n;

        if (DtmfDetect (state, buf, n, 1) && state->script->dtmfDetection) {
          DEBUG (fprintf (stderr, "DTMF='%s'\n", state->dtmf_state->keySeq);)
          Cpl_Schedule (state, evDtmf, state->dtmf_state->keySeq, FALSE);
          return;
        }

        if (state->audioLog > -1)                        /* audio logging? */
          write (state->audioLog, buf, n);
        if (state->script->execPid > -1) {               /* exec command in progress? */
          if (state->script->execStdOut == -1) {         /* pipe already closed? */
            if (state->bChannelBytes >= state->script->execBytes) {
              DEBUG (fprintf (stderr, "pipe is empty\n");)
              state->script->execPid = -1;
              Cpl_Schedule (state, evReschedule, NULL, FALSE);
              return;
            }
          }
          wPtr = state->script->execWptr;
          rPtr = state->script->execRptr;
          bFree = (wPtr >= rPtr ? R_BUF_SIZE - wPtr + rPtr : rPtr - wPtr);

          if (bFree > (R_BUF_SIZE / 2) && state->script->execStdOut > -1)
            IsdnCreateFileHandler (state->script->execStdOut, ISDN_READABLE, ExecStdOutCb, state);

          if (R_BUF_SIZE - bFree < n && state->script->execStdOut > -1) {
            for (i = 0; i < sizeof (buf)*2; i++) {
              state->script->execRbuf[wPtr++] = 0;
              if (wPtr >= R_BUF_SIZE)
                wPtr = 0;
            }
            state->script->execBytes += 2 * sizeof (buf);
            state->script->execWptr = wPtr;
          }
          for (i = 0; i < n; i++) {
            buf[i] = state->script->execRbuf[rPtr++];
            if (rPtr >= R_BUF_SIZE)
              rPtr = 0;
          }

          write (state->bChannelFd, buf, n);
          state->script->execRptr = rPtr;

        }

        if (state->fwdState != NULL && ((t_callState*)state->fwdState)->bChannelFd > -1) {
          if (write (((t_callState*)state->fwdState)->bChannelFd, buf, n) == -1) {
            perror ("write() to first B channel");
            return;
          }
        }

} /* iScriptBchannelCb() */



/*
** exported functions:
*/

void iScriptHandleChannelInd (t_callState *state)
{
        return;

} /* iScriptHandleChannelInd() */
```

```
void iScriptHandleConnectInd (t_callState *state)
{

fprintf (stderr, "connected, ");
        IsdnCreateFileHandler (state->bChannelFd, ISDN_READABLE, iScriptBchannelCb, state);
        Cpl_Schedule (state, evReschedule, NULL, FALSE);
        return;

} /* iScriptHandleConnectInd() */



void iScriptHandleDisconnectInd (t_callState *state)
{
        t_IsdnReleaseReq        iReq;

        IsdnDeleteFileHandler (state->bChannelFd);
        state->bChannelFd = -1;

        ZERO (iReq)
        iReq.callId = state->callId;
        if (IsdnReleaseReq (&iReq) < 0)
          IsdnError ("*** Error while RELEASE:");
        if (state->fwdState != NULL) {
          ZERO (iReq)
          iReq.callId = ((t_callState*)state->fwdState)->callId;
          if (IsdnReleaseReq (&iReq) < 0)
            IsdnError ("*** Error while RELEASE:");
        }

        return;

} /* iScriptHandleDisconnectInd() */



void iScriptHandleReleaseInd (t_callState *state)
{

        Cpl_Schedule (state, evDisconnect, NULL, FALSE);/* resume code execution */

        return;

} /* iScriptHandleReleaseInd() */



int iScriptIncCall (t_callState* state, char* action)
{
        t_IsdnAlertingReq       aReq;
        int                     i;
        char*                   p;
        char                    scriptFn[128];
        int                     slog;
        char                    logFn[128];
        char                    username[128];
        char                    groupname[128];

        p = GetSym (action, scriptFn);          /* skip service name */
        p = GetSym (p, username);
        p = GetSym (p, groupname);
        p = GetSym (p, scriptFn);
        if (strlen (scriptFn) == 0) {
          Msg ("Error in configuration file: Scriptname missing");
          return -1;
        }

        sprintf (logFn, "%s.log", scriptFn);
        if ((slog = open (logFn, O_CREAT | O_WRONLY | O_TRUNC | O_APPEND, LOGFILE_MODE)) == -1) {
          Msg ("Error opening script log file");
```

```
        return -1;
      }
      Msg ("Executing script '%s'", scriptFn);

      /*
      ** create new script structure
      */
      if (Cpl_NewScript (&(state->script), scriptFn, slog) != CPL_OK)
        return;

      /*
      ** parse script file
      */
      if (Cpl_ParseMain (state->script) != CPL_OK) {
        Msg ("Terminating %s due to errors", scriptFn);
        return -1;
      }

      ZERO (aReq);
      aReq.callId = state->callId;
      if (IsdnAlertingReq (&aReq)) {
        IsdnError ("*** Error while ALERTING:");
        return -1;
      }

      DtmfInit (state);

      /*
      ** set some global interpreter vars
      */
      i = Cpl_SymAdd (state->script, symVar, "sysCallingParty", TRUE);
      Cpl_SymSet (state->script, i, state->callingParty, NULL);
      i = Cpl_SymAdd (state->script, symVar, "sysCalledParty", TRUE);
      Cpl_SymSet (state->script, i, state->calledParty, NULL);

      state->audioLog = -1;
      state->script->trace = 0;

      state->script->dtmfDetection = state->script->states[0].dtmfDetection;

      Cpl_Schedule (state, evEnter, NULL, FALSE);

      return 0;

} /* iScriptIncCall() */
```

H.3 CPL Routines

H.3.1 General Header File

```
/*
** iSrvCpl.h
*/

#ifndef _ISRVCPL_H
#define _ISRVCPL_H

#ifndef TRUE
#define TRUE                    1
#endif
#ifndef FALSE
#define FALSE                   0
#endif

#define CPL_ERROR               1
#define CPL_OK                  2

#define TIMER_LOCAL             1
#define TIMER_GLOBAL            2
```

```
#define ERR_ALREADY_CONNECTED   1001
#define ERR_ILL_IP_ADDR         1101
#define ERR_ILL_TYPE            1102
#define ERR_NOT_CONNECTED       1201
#define ERR_OPENING_FILE        1301
#define ERR_TOO_MANY_TIMERS     1401
#define ERR_NO_SUCH_TIMER       1402
#define ERR_MALLOC_FAILED       1403
#define ERR_CREATE_TIMER        1404
#define ERR_CALL_FAILED         1500

#define MAX_CODE_SP             10000
#define MAX_DTMF_LEN            100
#define MAX_STATES              100
#define MAX_STRING_LENGTH       256
#define MAX_SYMBOL_LENGTH       256
#define MAX_TIMERS              10
#define R_BUF_SIZE              10000
#define STACKSIZE               1000
#define SYM_TAB_SIZE            1000

typedef struct {
          enum                  {unknown_url, phone_url, rtp_url, file_url, ftp_url, http_url,
                                gopher_url, mailto_url, news_url, nntp_url, telnet_url,
                                wais_url} type;
          char                  host[256];
          int                   port;
          char                  phoneNo[256];
          char                  countryCode[256];
        } t_parsedUrl;
typedef struct {
          char*                 name;
          enum                  {symProc, symState, symVar, symConst} type;
          char*                 value;
          char*                 (*getValueFunc)();
          int                   lineNo;         /* first occurance in source file */
          int                   readOnly;
        } t_symTabEntry;
typedef enum {
          opAccept, opAudioRec, opCall, opCvt, opEnter, opExec, opExit, opGoto,
          opHangup, opIncr, opInvite, opCond, opLog, opRedirect,
          opReturn, opSet, opStartTimer, opStop, opStopTimer
        } t_opcodes;
typedef enum {
          opEQ, opLT, opGT, opLE, opGE, opNE
        } t_operands;
typedef enum {
          evEnter, evDisconnect, evTimeout, evDtmf, evSpeech, evReschedule
        } t_events;
typedef struct {
          t_opcodes             opcode;
          int                   op1;            /* index into symbol table */
          int                   op2;
          int                   op3;
          int                   addr1;          /* jump address for then block */
          int                   addr2;          /* jump address for else block */
        } t_code;
typedef struct {
          t_events              type;           /* type of event */
          void*                 next;           /* link to next event within this state */
          int                   codeStart;      /* PC for procbody */
          int                   argInd;         /* index into symbol table */
        } t_eventDef;
typedef struct {
          char*                 name;
          t_eventDef*           eventDefs;
          t_eventDef*           enterEvent;     /* points to the "enter" event definition */
          int                   dtmfDetection;  /* 0 = off, 1 = on */
        } t_stateDef;
typedef struct {
          char                  name[MAX_STRING_LENGTH];
          int                   scope;          /* local / global */
          int                   id;             /* returned by IsdnCreateTimerHandler() */
```

```
        int                  flag;
        void*                state;
      } t_timers;
typedef struct {
        /*
        ** compiler relevant data:
        */
        FILE*                srcFile;                /* the script file */
        char*                srcFilename;            /* its name */
        int                  lineCount;              /* current line while parsing */
        int                  sym;                    /* current symbol (token) */
        char                 symName[MAX_SYMBOL_LENGTH];     /* name of current symbol */
        char                 strValue[MAX_STRING_LENGTH];    /* the string itself, if current symbol is SYM_STRING */
        int                  intValue;                       /* the integer itself, if current symbol is SYM_INTEGER */
        /*
        ** structures filled up by the compiler and used by the interpreter:
        */
        t_stateDef           states[MAX_STATES];     /* list/count of states defined */
        int                  stateCount;             /* in script file */
        t_eventDef*          eventDefs;              /* list of global events */
        t_code               codeStack[MAX_CODE_SP]; /* contains the generated code */
        int                  codeSP;
        t_symTabEntry        symTab[SYM_TAB_SIZE];   /* our symbol table */
        int                  symInd;
        t_timers*            timers[MAX_TIMERS];
        /*
        ** interpreter relevant data:
        */
        int                  status;                 /* status of last CPL command */
        int                  trace;
        int                  stack[STACKSIZE];       /* stack to hold return addresses */
        int                  stackPtr;               /* while executing call statements */
        int                  logFile;
        char                 dtmfSeq[MAX_DTMF_LEN];
        int                  dtmfDetection;          /* 0 = off, 1 = on */
        int                  state;                  /* current state of state machine */
        int                  pc;                     /* program counter */
        int                  execPid;                /* process id for executing system command via CPL exec command */
        int                  execStdOut;             /* stdout from exec process */
        int                  execBytes;              /* bytes read from exec pipe */
        char                 execRbuf[R_BUF_SIZE];   /* ring buffer to read out the pipe */
        int                  execRptr;
        int                  execWptr;
      } t_script;

#endif
```

H.3.2 CPL Script Compiler

iSrvCplComp.h

```
/*
** iSrvCplComp.h
**
** (c) 1996 by F. Oertel
**
** 30.12.1996, FO, created
*/

#ifndef _I_SRVCPLCOMP_H
#define _I_SRVCPLCOMP_H

#include "iSrvCpl.h"

extern  int                  Cpl_Error (t_script* script, char* fmt, ...);
extern  void                 Cpl_FreeScript (t_script* script);
extern  int                  Cpl_NewScript (t_script** script, char* fn, int logf);
extern  int                  Cpl_ParseMain (t_script* script);
extern  int                  Cpl_SymAdd (t_script* script, int symType, char* symName, int readOnly);
extern  void                 Cpl_SymDump (t_script* script);
extern  int                  Cpl_SymLookup (t_script* script, char* symName, int symType);
```

```
extern  int                     Cpl_SymSet (t_script* script, int ind, char* value, char* (*getFunc)());

#endif
```

iSrvCplComp.c

```
/*
** iSrvCplComp.c -- CPL script compiler
**
** (c) 1996 by F. Oertel
**
** 23.12.1996, FO, created
** 27.12.1996, FO, code generation added
** 30.12.1996, FO, call / return statements added
** 31.12.1996, FO, accept statement added, cosmetic changes
** 01.01.1997, FO, interpreter functions moved to "iSrvCplInt.c"
** 03.01.1997, FO, Cpl_FreeScript() added
** 05.01.1997, FO, redirect command added
** 06.01.1997, FO, some debugging
** 07.01.1997, FO, audioplay removed
** 08.01.1997, FO, scip, mbone, say, tone commands removed
** 08.01.1997, FO, Start/StopTimer added
** 09.01.1997, FO, changecd syntax for timer commands
** 12.01.1997, FO, catch command removed (replaced by "on dtmf")
** 12.01.1997, FO, match pattern check added
** 14.01.1997, FO, if-then-else added
** 16.01.1997, FO, the forgotten != for conditions added
** 16.01.1997, FO, Cpl_PaseConvert() added
** 27.01.1997, FO, incr / while / invite added
** 27.01.1997, FO, integer constants may now be negative
** 27.01.1997, FO, the "then" phrase is no longer required in an if statement
** 29.01.1997, FO, sysHostname / sysUsername added
** 31.01.1997, FO, hangup reason added
** 02.02.1997, FO, call syntax changed
** 06.02.1997, FO, bug in ParseTimer() fixed
** 07.02.1997, FO, timer syntax changed for global timers
** 07.02.1997, FO, support for global event defintions added
*/

#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <malloc.h>

#define DEBUG(x)

#include "iSrv.h"
#include "iSrvCpl.h"
#include "iSrvCplInt.h"
#include "iSrvCplComp.h"
#include "iSrvUtils.h"

#define EOL                     '\x0a'

#define SYM_AUDIOREC            1
#define SYM_BEGIN               2
#define SYM_CALL                3
#define SYM_DISCONNECT          5
#define SYM_DTMF                6
#define SYM_END                 7
#define SYM_ENTER               8
#define SYM_EOF                 9
#define SYM_EXEC                10
#define SYM_EXIT                11
#define SYM_FLOAT               2002
#define SYM_HANGUP              12
#define SYM_ERROR               1000
#define SYM_INTEGER             2001
#define SYM_INTO                13
#define SYM_LOG                 14
#define SYM_NAME                3000
#define SYM_ON                  15
```

```
#define SYM_PROC                16
#define SYM_SPEECH              17
#define SYM_SET                 18
#define SYM_LOCAL               19
#define SYM_STATE               20
#define SYM_STOP                21
#define SYM_STRING              2000
#define SYM_TIMER               22
#define SYM_VARNAME             3001
#define SYM_ACCEPT              23
#define SYM_REDIRECT            24
#define SYM_IF                  25
#define SYM_ELSE                27
#define SYM_GT                  28
#define SYM_LT                  29
#define SYM_EQ                  30
#define SYM_GE                  31
#define SYM_LE                  32
#define SYM_NE                  33
#define SYM_CVT                 34
#define SYM_INVITE              35
#define SYM_INCR                36
#define SYM_WHILE               37
#define SYM_GLOBAL              38

        struct t_cmdInfo {
          int                   sym;
          char*                 cmd;
          void*                 func;
        } cmds[] = {
        {SYM_ACCEPT,    "accept",       NULL},
        {SYM_AUDIOREC,  "audiorec" ,    NULL},
        {SYM_CALL,      "call",         NULL},
        {SYM_DISCONNECT,"disconnect",   NULL},
        {SYM_DTMF,      "dtmf",         NULL},
        {SYM_ENTER,     "enter",        NULL},
        {SYM_EXEC,      "exec",         NULL},
        {SYM_EXIT,      "exit",         NULL},
        {SYM_REDIRECT,  "redirect",     NULL},
        {SYM_HANGUP,    "hangup",       NULL},
        {SYM_INTO,      "into",         NULL},
        {SYM_LOG,       "log",          NULL},
        {SYM_ON,        "on",           NULL},
        {SYM_PROC,      "proc",         NULL},
        {SYM_SPEECH,    "speech",       NULL},
        {SYM_SET,       "set",          NULL},
        {SYM_LOCAL,     "local",        NULL},
        {SYM_STATE,     "state",        NULL},
        {SYM_STOP,      "stop",         NULL},
        {SYM_TIMER,     "timer",        NULL},
        {SYM_IF,        "if",           NULL},
        {SYM_ELSE,      "else",         NULL},
        {SYM_CVT,       "cvt",          NULL},
        {SYM_INVITE,    "invite",       NULL},
        {SYM_WHILE,     "while",        NULL},
        {SYM_INCR,      "incr",         NULL},
        {SYM_GLOBAL,    "global",       NULL},
        {0,             NULL,           NULL}
        };

/*
** forward declarations of local functions:
*/
static  int                     Cpl_Check (t_script* script);
static  int                     Cpl_FindSymbol (t_script* script, char* token);
static  int                     Cpl_GetByte (t_script* script);
static  int                     Cpl_GetString (t_script* script);
static  int                     Cpl_GetSym (t_script* script);
static  int                     Cpl_ParseAccept (t_script* script);
static  int                     Cpl_ParseAddCode (t_script* script, t_opcodes opcode, int op1, int op2, int op3);
static  int                     Cpl_ParseAudioRec (t_script* script);
static  int                     Cpl_ParseCall (t_script* script);
static  int                     Cpl_ParseCondition (t_script* script);
```

```
static  int                     Cpl_ParseConvert (t_script* script);
static  int                     Cpl_ParseEnter (t_script* script);
static  int                     Cpl_ParseEventDef (t_script* script, int globalFlag);
static  int                     Cpl_ParseExit (t_script* script);
static  int                     Cpl_ParseExec (t_script* script);
static  int                     Cpl_ParseHangup (t_script* script);
static  int                     Cpl_ParseIfThenElse (t_script* script);
static  int                     Cpl_ParseIncr (t_script* script);
static  int                     Cpl_ParseInvite (t_script* script);
static  int                     Cpl_ParseLog (t_script* script);
static  int                     Cpl_ParseProc (t_script* script);
static  int                     Cpl_ParseProcBody (t_script* script);
static  int                     Cpl_ParseRedirect (t_script* script);
static  int                     Cpl_ParseSet (t_script* script);
static  int                     Cpl_ParseState (t_script* script);
static  int                     Cpl_ParseStatement (t_script* script);
static  int                     Cpl_ParseStringExpr (t_script* script);
static  int                     Cpl_ParseTimer (t_script* script);
static  int                     Cpl_ParseWhile (t_script* script);
static  int                     Cpl_StateAdd (t_script* script, char* name);
static  int                     Cpl_SymCheck (t_script* script);

/*
** Local functions:
*/

static int Cpl_AddCode (t_script* script, t_opcodes opcode, int op1, int op2, int op3)
{
        script->codeStack[script->codeSP].opcode = opcode;
        script->codeStack[script->codeSP].op1 = op1;
        script->codeStack[script->codeSP].op2 = op2;
        script->codeStack[script->codeSP].op3 = op3;
        script->codeStack[script->codeSP].addr1 = 0;
        script->codeStack[script->codeSP].addr2 = 0;
        script->codeSP++;
        return CPL_OK;

} /* Cpl_AddCode() */



static int Cpl_Check (t_script* script)
{
        int                     i;

        if (Cpl_SymCheck (script) != CPL_OK)
          return CPL_ERROR;

        if (script->stateCount == 0)
          return Cpl_Error (script, "No state definitions found in source file (ERR101)");

        if (script->states[0].enterEvent == NULL)
          return Cpl_Error (script, "Initial state must have an 'enter' definition (ERR102)");

        for (i = 0; i < script->stateCount -1; i++)
          if (script->states[i].enterEvent == NULL)
            fprintf (stderr, "WARNING: No enter event definition in state '%s'\n", script->states[i].name);

        return CPL_OK;

} /* Cpl_Check() */



static int Cpl_EventAdd (t_script *script, t_events eventType, int argInd, int globalFlag)
/*
** add the specified event to the current states list of events
*/
{
        t_eventDef**            ev;
        t_eventDef*             p;
        t_stateDef*             state;
```

```
        if (globalFlag)
          ev = &script->eventDefs;
        else {
          state = &script->states[script->stateCount -1];
          ev = &state->eventDefs;
        }

        if (*ev == NULL)
          p = *ev = malloc (sizeof (t_eventDef));
        else {
          p = *ev;
          while (p->next)
            p = p->next;
          p->next = malloc (sizeof (t_eventDef));
          p = p->next;
        }
        if (eventType == evEnter && !globalFlag)
          if (state->enterEvent == NULL)
            state->enterEvent = p;
          else
            return Cpl_Error (script, "Multiple enter event definitions for state '%s' (ERR103)", script->states[script->stateCount-1].name);
        p->type = eventType;
        p->next = NULL;
        p->codeStart = script->codeSP;
        p->argInd = argInd;

        return CPL_OK;

} /* Cpl_EventAdd() */



static int Cpl_FindSymbol (t_script* script, char* token)
{
        int                     i = 0;

        while (cmds[i].sym) {
          if (strcmp (cmds[i].cmd, token) == 0)
            break;
          i++;
        }
        if (!cmds[i].sym)
          return SYM_NAME;
        else
          return cmds[i].sym;

} /* Cpl_FindSymbol() */



static int Cpl_GetByte (t_script* script)
{
        int                     c;

        c = getc (script->srcFile);

        if (c == '#') {
          DEBUG (printf (" *** COMMENT: '");)
          while (((c = getc (script->srcFile)) != EOF) && c != EOL)
            DEBUG (putchar  (c))
            ;
          DEBUG (printf ("'\n");)
          if (c == EOL)
            script->lineCount++;
          return Cpl_GetByte (script);
        }

        if (c == EOL) {
          script->lineCount++;
        }

        return c;
```

```
} /* Cpl_GetByte() */



static int Cpl_GetInteger (t_script* script, char c, int negative)
{
        int                     i = 0;
        int                     val;

        if (negative)
          script->symName[i++] = '-';
        val = c - '0';
        script->symName[i++] = c;
        while (isdigit (c = Cpl_GetByte (script))) {
          script->symName[i++] = c;
          val = val * 10 + (c - '0');
        };
        if (negative)
          val *= -1;
        script->intValue = val;
        script->symName[i] = '\0';
        ungetc (c, script->srcFile);
        return (script->sym = SYM_INTEGER);

} /* Cpl_GetInteger() */



static int Cpl_GetString (t_script* script)
{
        int                     c;
        int                     esc = FALSE;
        int                     i = 0;

        DEBUG (fprintf (stderr, " *** STRING: '");)
        while ((c = getc (script->srcFile)) != EOF && c != '"' && c != EOL) {
          if (esc) {
            esc = FALSE;
            if (c == 'n')
              c = '\n';
          } else if (c == '\\') {
            esc = TRUE;
            continue;
          }
          script->strValue[i++] = c;
          DEBUG (putc (c, stderr);)
          if (i == MAX_STRING_LENGTH) {
            script->strValue[MAX_STRING_LENGTH -1] = '\0';
            Cpl_Error (script, "String too long '%s' (ERR104)", script->strValue);
            return CPL_ERROR;
          }
        }
        DEBUG (fprintf (stderr, "'\n");)
        if (c == EOL) {
          Cpl_Error (script, "Unterminated string (ERR105)");
          return CPL_ERROR;
        }
        if (c == EOF) {
          Cpl_Error (script, "Unexpected end of file (ERR106)");
          return CPL_ERROR;
        }
        script->strValue[i] = '\0';
        strcpy (script->symName, "<string>");
        return CPL_OK;

} /* Cpl_GetString() */



static int Cpl_GetSym (t_script* script)
{
        int                     c, c1;
        int                     i = 0;
```

```
while (((c = Cpl_GetByte (script)) != EOF) && isspace (c))
  ;
if (c == EOF) {
  strcpy (script->symName, "<end of file>");
  return (script->sym = SYM_EOF);
}

if (isalpha (c) || c == '$') {
  c1 = c;
  if (c1 == '$') {
    c = Cpl_GetByte (script);
    if (c == EOF) {
      strcpy (script->symName, "<end of file>");
      return (script->sym = SYM_EOF);
    }
  }
  while (isalpha (c) || isdigit (c) || c == '_') {
    script->symName[i++] = c;
    if (i == MAX_SYMBOL_LENGTH) {
      script->symName[MAX_SYMBOL_LENGTH-1] = '\0';
      Cpl_Error (script, "Symbol too long '%s' (ERR107)", script->symName);
      return (script->sym = SYM_ERROR);
    }
    c = Cpl_GetByte (script);
  } /* while */
  script->symName[i] = '\0';
  DEBUG (fprintf (stderr, " *** SYMBOL: '%s'\n", script->symName);)
  if (i == 0) {
    Cpl_Error (script, "Zero length variable name (ERR108)");
    return (script->sym = SYM_ERROR);
  }
  if (c1 == '$') {
    Cpl_SymAdd (script, symVar, script->symName, FALSE);
    return (script->sym = SYM_VARNAME);
  } else
    return (script->sym = Cpl_FindSymbol (script, script->symName));
} else if (isdigit (c))
  return (Cpl_GetInteger (script, c, 0));
else {
  script->symName[0] = c;
  script->symName[1] = '\0';
  switch (c) {
    case '-':
      if (isdigit (c = Cpl_GetByte (script)))
        return Cpl_GetInteger (script, c, 1);
      else {
        ungetc (c, script->srcFile);
        break;
      }
    case '"':
      if (Cpl_GetString (script) == CPL_OK)
        return (script->sym = SYM_STRING);
      else
        return (script->sym = SYM_ERROR);
    case '{':
      return (script->sym = SYM_BEGIN);
    case '}':
      return (script->sym = SYM_END);
    case '!':
      if ((c = Cpl_GetByte (script)) == '=')
        return (script->sym = SYM_NE);
      else {
        ungetc (c, script->srcFile);
        break;
      }
    case '>':
      if ((c = Cpl_GetByte (script)) == '=')
        return (script->sym = SYM_GE);
      else {
        ungetc (c, script->srcFile);
        return (script->sym = SYM_GT);
      }
```

```
            case '<':
              if ((c = Cpl_GetByte (script)) == '=')
                return (script->sym = SYM_LE);
              else {
                ungetc (c, script->srcFile);
                return (script->sym = SYM_LT);
              }
            case '=':
              if ((c = Cpl_GetByte (script)) == '=')
                return (script->sym = SYM_EQ);
              else {
                ungetc (c, script->srcFile);
                Cpl_Error (script, "Unknown symbol '=' (ERR109)");
                return (script->sym = SYM_ERROR);
              }

          }
        }

        Cpl_Error (script, "Unknown symbol '%s' (ERR109)", script->symName);
        return (script->sym = SYM_ERROR);

} /* Cpl_GetSym() */



static int Cpl_ParseAccept (t_script* script)
/*
** Parse accept command, syntax:
**
**      "accept"  :==  "accept"
*/
{
        DEBUG (fprintf (stderr, " accept");)
        Cpl_GetSym (script);
        return Cpl_AddCode (script, opAccept, 0, 0, 0);

} /* Cpl_ParseAccept() */



static int Cpl_ParseAudioRec (t_script* script)
{
        int                     sExpr;

        DEBUG (fprintf (stderr, " audiorec");)
        if ((sExpr = Cpl_ParseStringExpr (script)) == -1)
          return Cpl_Error (script, "String expression expected for audio filename, found '%s' (ERR110)", script->symName);

        Cpl_GetSym (script);
        return Cpl_AddCode (script, opAudioRec, sExpr, 0, 0);

} /* Cpl_ParseAudioRec() */



static int Cpl_ParseCall (t_script* script)
{
        int                     i;

        DEBUG (fprintf (stderr, " call");)
/*
        if (Cpl_GetSym (script) != SYM_NAME)
          return Cpl_Error (script, "Expected procedure name, found '%s' (ERR111)", script->symName);
*/
        i = Cpl_SymAdd (script, symProc, script->symName, TRUE);
        Cpl_GetSym (script);
        return Cpl_AddCode (script, opCall, i, 0, 0);

} /* Cpl_ParseCall() */


static int Cpl_ParseCondition (t_script* script)
```

```
/*
** Parse a (very simple) condition:
**
**      "{" stringexpr operator stringexpr "}"
**
** Operators are:
**
**      ">", "<", "==", ">=", "<=", "!="
*/
{
        int                     op1, oper, op2;

        if (Cpl_GetSym (script) != SYM_BEGIN)
          return Cpl_Error (script, "'{' expected, found '%s' (ERR112)", script->symName);

        if ((op1 = Cpl_ParseStringExpr (script)) == -1)
          return Cpl_Error (script, "Operand expected, found '%s' (ERR113)", script->symName);

        switch (Cpl_GetSym (script)) {
          case SYM_EQ: oper = opEQ; break;
          case SYM_LE: oper = opLE; break;
          case SYM_LT: oper = opLT; break;
          case SYM_GT: oper = opGT; break;
          case SYM_GE: oper = opGE; break;
          case SYM_NE: oper = opNE; break;
          default:
            return Cpl_Error (script, "Operator <, >, ==, >=, <= or != expected, found '%s' (ERR114)", script->symName);
        }

        if ((op2 = Cpl_ParseStringExpr (script)) == -1)
          return Cpl_Error (script, "Operand expected, found '%s' (ERR113)", script->symName);

        if (Cpl_GetSym (script) != SYM_END)
          return Cpl_Error (script, "'}' expected, found '%s' (ERR115)", script->symName);

        Cpl_AddCode (script, opCond, op1, oper, op2);

        return CPL_OK;

} /* Cpl_ParseCondition() */



static int Cpl_ParseConvert (t_script* script)
/*
** Parse the cvt command, syntax:
**
**      cvtcmd  :==  "cvt"  varname  stringexpr  stringexpr
*/
{
        int                     sourceOp;
        int                     targetOp;
        int                     typeOp;

        DEBUG (fprintf (stderr, " cvt");)

        if (Cpl_GetSym (script) != SYM_NAME)
          return Cpl_Error (script, "Variable name expected, found '%s' (ERR116)", script->symName);
        targetOp = Cpl_SymAdd (script, symVar, script->symName, FALSE);
        if (script->symTab[targetOp].readOnly)
          return Cpl_Error (script, "'$%s' is a read only variable (ERR117)", script->symName);

        DEBUG (fprintf (stderr, " %s", script->symName);)

        if ((sourceOp = Cpl_ParseStringExpr (script)) == -1)
          return Cpl_Error (script, "String expression expected, found '%s' (ERR118)", script->symName);

        if ((typeOp = Cpl_ParseStringExpr (script)) == -1)
          return Cpl_Error (script, "String expression expected, found '%s' (ERR119)", script->symName);

        Cpl_GetSym (script);
        return Cpl_AddCode (script, opCvt, targetOp, sourceOp, typeOp);
```

```
} /* Cpl_ParseConvert() */



static int Cpl_ParseEnter (t_script* script)
{
        int                     i;

        DEBUG (fprintf (stderr, " enter");)
        if (Cpl_GetSym (script) != SYM_NAME)
          return Cpl_Error (script, "Expected statename, found '%s' (ERR120)", script->symName);

        i = Cpl_SymAdd (script, symState, script->symName, TRUE);
        Cpl_GetSym (script);
        return Cpl_AddCode (script, opEnter, i, 0, 0);

} /* Cpl_ParseEnter() */



static int Cpl_ParseEventDef (t_script* script, int globalFlag)
/*
** syntax:  "on"  eventspec  procbody
*/
{
        int                     i;

        if (script->sym != SYM_ON)
          return Cpl_Error (script, "Eventdefition starting with 'on' expected, found '%s' (ERR121)", script->symName);

        Cpl_GetSym (script);
        DEBUG (fprintf (stderr, "    on %s", script->symName);)
        switch (script->sym) {
          case SYM_DTMF:
            if ((i = Cpl_ParseStringExpr (script)) == -1)
              return Cpl_Error (script, "String expression expected for match pattern, found '%s' (ERR122)", script->symName);
/*
fprintf (stderr, "trying to check  match pattern %d\n", i);
            if (Cpl_CompileMatchPattern (script, script->symTab[i].value) != CPL_OK)
              return CPL_ERROR;
*/
            if (Cpl_EventAdd (script, evDtmf, i, globalFlag) != CPL_OK)
              return CPL_ERROR;
            script->states[script->stateCount-1].dtmfDetection = TRUE;
            break;
          case SYM_SPEECH:
            break;
          case SYM_ENTER:
            if (Cpl_EventAdd (script, evEnter, 0, globalFlag) != CPL_OK)
              return CPL_ERROR;
            break;
          case SYM_DISCONNECT:
            if (Cpl_EventAdd (script, evDisconnect, 0, globalFlag) != CPL_OK)
              return CPL_ERROR;
            break;
          case SYM_TIMER:
            if (Cpl_GetSym (script) != SYM_NAME)
              return Cpl_Error (script, "Timer name expected, found '%s' (ERR123)", script->symName);
            i = Cpl_SymAdd (script, symConst, script->symName, TRUE);
            Cpl_SymSet (script, i, script->symName, NULL);
            if (Cpl_EventAdd (script, evTimeout, i, globalFlag) != CPL_OK)
              return CPL_ERROR;
            break;
          default:
            return Cpl_Error (script, "Expected event specification, found '%s' (ERR124)", script->symName);
        }

        if (Cpl_ParseProcBody (script) != CPL_OK)
          return CPL_ERROR;

        return Cpl_AddCode (script, opStop, 0, 0, 0);

} /* Cpl_ParseEventDef() */
```

```
static int Cpl_ParseExec (t_script* script)
{
        int                     sExpr;

        DEBUG (fprintf (stderr, " exec"););
        if ((sExpr = Cpl_ParseStringExpr (script)) == -1)
          return Cpl_Error (script, "Command string expression expected, found '%s' (ERR125)", script->symName);

        Cpl_GetSym (script);
        return Cpl_AddCode (script, opExec, sExpr, 0, 0);

} /* Cpl_ParseExec() */



static int Cpl_ParseExit (t_script* script)
{
        DEBUG (fprintf (stderr, " exit"););
        Cpl_GetSym (script);
        return Cpl_AddCode (script, opExit, (int)NULL, (int)NULL, (int)NULL);

} /* Cpl_ParseExit() */



static int Cpl_ParseHangup (t_script* script)
/*
** Parse hangup command, syntax:
**
**      "hangup"  :==  "hangup"  [ reasoncode ]
*/
{
        int                     reason;

        DEBUG (fprintf (stderr, " hangup"););

        if ((reason = Cpl_ParseStringExpr (script)) == -1)
          return Cpl_AddCode (script, opHangup, reason, -1, -1);
        else {
          Cpl_GetSym (script);
          return Cpl_AddCode (script, opHangup, -1, -1, -1);
        }

} /* Cpl_ParseHangup() */



static int Cpl_ParseIfThenElse (t_script* script)
/*
** Parse if-then-else statement, syntax:
**
**      "if" { condition } procbody [ "else" procbody ]
*/
{
        int                     savedSP1, savedSP2;


        savedSP1 = script->codeSP;

        if (Cpl_ParseCondition (script) != CPL_OK)
          return CPL_ERROR;

        if (Cpl_ParseProcBody (script) != CPL_OK)
          return CPL_ERROR;

        if (Cpl_GetSym (script) == SYM_ELSE) {
          savedSP2 = script->codeSP;
          Cpl_AddCode (script, opGoto, 0, 0, 0);
          script->codeStack[savedSP1].addr1 = script->codeSP;
          if (Cpl_ParseProcBody (script) != CPL_OK)
```

```
            return CPL_ERROR;
          Cpl_GetSym (script);
          script->codeStack[savedSP2].op1 = script->codeSP;
        } else
          script->codeStack[savedSP1].addr1 = script->codeSP;

        return CPL_OK;

} /* ParseIfThenElse() */



static int Cpl_ParseIncr (t_script* script)
/*
** Parse incr statement, syntax:
**
** "incr" varname [ increment ]
*/
{
        int                     incrOp;
        int                     targetOp;

        if (Cpl_GetSym (script) != SYM_NAME)
          return Cpl_Error (script, "Variable name expected, found '%s' (ERR116)", script->symName);
        targetOp = Cpl_SymAdd (script, symVar, script->symName, FALSE);
        if (script->symTab[targetOp].readOnly)
          return Cpl_Error (script, "'$%s' is a read only variable", script->symName);

        if ((incrOp = Cpl_ParseStringExpr (script)) == -1)
          return Cpl_AddCode (script, opIncr, targetOp, -1, -1);
        else {
          Cpl_GetSym (script);
          return Cpl_AddCode (script, opIncr, targetOp, incrOp, -1);
        }

} /* Cpl_ParseIncr() */



static int Cpl_ParseInvite (t_script* script)
/*
** Parse invite statement, syntax:
**
** "invite"  from  info  media-addr
*/
{
        int                     fromOp;
        int                     infoOp;
        int                     mediaOp;

        if ((fromOp = Cpl_ParseStringExpr (script)) == -1)
          return Cpl_Error (script, "String expression expected, found '%s' (ERR126)", script->symName);
        if ((infoOp = Cpl_ParseStringExpr (script)) == -1)
          return Cpl_Error (script, "String expression expected, found '%s' (ERR127)", script->symName);
        if ((mediaOp = Cpl_ParseStringExpr (script)) == -1)
          return Cpl_Error (script, "String expression expected, found '%s' (ERR128)", script->symName);

        Cpl_GetSym (script);

        return Cpl_AddCode (script, opInvite, fromOp, infoOp, mediaOp);

} /* Cpl_ParseInvite() */



static int Cpl_ParseLog (t_script* script)
/*
** Parse log command, syntax:
**
**      logcmd  :==  "log"  stringexpr
*/
{
        int                     sExpr;
```

```
        DEBUG (fprintf (stderr, " log");)

        if ((sExpr = Cpl_ParseStringExpr (script)) == -1)
          return Cpl_Error (script, "String expression expected, found '%s' (ERR129)", script->symName);

        Cpl_GetSym (script);
        return Cpl_AddCode (script, opLog, sExpr, 0, 0);

} /* Cpl_ParseLog() */



static int Cpl_ParseProc (t_script* script)
/*
** Parse CPL procedure, syntax:
**
**      procdef  :==  "proc"  procname  procbody
*/
{
        int                     i;
        char                    s[80];

        DEBUG (fprintf (stderr, "  proc");)

        if (Cpl_GetSym (script) != SYM_NAME)
          return Cpl_Error (script, "Procedure name expected, found '%s' (ERR130)", script->symName);
        DEBUG (fprintf (stderr, " %s", script->symName);)

        i = Cpl_SymAdd (script, symProc, script->symName, TRUE);/* put current program counter into symbol table */
        sprintf (s, "%d", script->codeSP);                      /* save current state id along with */
        Cpl_SymSet (script, i, s, NULL);                        /* state name in symbol table */
        DEBUG (fprintf (stderr, " %s", script->symName);)

        Cpl_ParseProcBody (script);

        DEBUG (fprintf (stderr, "\n");)
        return Cpl_AddCode (script, opReturn, 0, 0, 0);

} /* Cpl_ParseProc() */



static int Cpl_ParseProcBody (t_script* script)
/*
** Parse procedure body, syntax:
**
**      procbody  :==  "{"  statement+  "}"
*/
{

        if (Cpl_GetSym (script) != SYM_BEGIN)
          return Cpl_Error (script, "Statement block beginning with '{' expected, found '%s' (ERR131)", script->symName);
        DEBUG (fprintf (stderr, " {");)


        Cpl_GetSym (script);
        while (script->sym != SYM_END)
          if (Cpl_ParseStatement (script) != CPL_OK)
            return CPL_ERROR;

        DEBUG (fprintf (stderr, " }\n");)

        return CPL_OK;

} /* Cpl_ParseProcBody() */



static int Cpl_ParseRedirect (t_script* script)
/*
**      redirectcmd :==  "redirect"  stringexpr
*/
```

```
{
        int                     sExpr;

        DEBUG (fprintf (stderr, " redirect");)

        if ((sExpr = Cpl_ParseStringExpr (script)) == -1)
          return Cpl_Error (script, "String expression expected for destination, found '%s' (ERR132)", script->symName);

        Cpl_GetSym (script);
        return Cpl_AddCode (script, opRedirect, sExpr, 0, 0);

} /* Cpl_ParseRedirect() */



static int Cpl_ParseSet (t_script* script)
/*
** Parse the set command, syntax:
**
**      setcmd  :==  "set"  varname  stringexpr
*/
{
        int                     targetOp;
        int                     sourceOp;

        DEBUG (fprintf (stderr, " set");)
        if (Cpl_GetSym (script) != SYM_NAME)
          return Cpl_Error (script, "Variable name expected, found '%s' (ERR116)", script->symName);
        targetOp = Cpl_SymAdd (script, symVar, script->symName, FALSE);
        if (script->symTab[targetOp].readOnly)
          return Cpl_Error (script, "'$%s' is a read only variable (ERR117)", script->symName);

        DEBUG (fprintf (stderr, " %s", script->symName);)

        if ((sourceOp = Cpl_ParseStringExpr (script)) == -1)
          return Cpl_Error (script, "String expression expected, found '%s' (ERR133)", script->symName);

        Cpl_GetSym (script);
        return Cpl_AddCode (script, opSet, targetOp, sourceOp, (int)NULL);

} /* Cpl_ParseSet() */



static int Cpl_ParseState (t_script* script)
/*
**      statedef  :==  "state"  statename  "{"  eventdef+  "}"
*/
{
        int                     i;
        char                    s[80];

        DEBUG (fprintf (stderr, "  state");)
        if (Cpl_GetSym (script) != SYM_NAME)
          return Cpl_Error (script, "Statename expected, found '%s' (ERR134)", script->symName);

        Cpl_StateAdd (script, script->symName);                         /* add state to the scripts list of states */
        i = Cpl_SymAdd (script, symState, script->symName, TRUE);       /* put state name into symbol table */
        sprintf (s, "%d", script->stateCount -1);                       /* save current state id along with */
        Cpl_SymSet (script, i, s, NULL);                                /* state name in symbol table */
        DEBUG (fprintf (stderr, " %s", script->symName);)

        if (Cpl_GetSym (script) != SYM_BEGIN)
          return Cpl_Error (script, "'{' expected, found '%s' (ERR135)", script->symName);

        DEBUG (fprintf (stderr, " {\n");)
        while (Cpl_GetSym (script) != SYM_END) {
          if (Cpl_ParseEventDef (script, FALSE) != CPL_OK)
            return CPL_ERROR;
        }

        if ( script->states[script->stateCount-1].eventDefs == NULL )
          return Cpl_Error (script, "The are no event definitions for state '%s' (ERR136)",
```

```
                    script->states[script->stateCount-1].name);

        DEBUG (fprintf (stderr, "  }\n");)

        return CPL_OK;

} /* Cpl_ParseState() */



static int Cpl_ParseStatement (t_script* script)
{
          switch (script->sym) {
            case SYM_ACCEPT:
              return Cpl_ParseAccept (script);
            case SYM_AUDIOREC:
              return Cpl_ParseAudioRec (script);
            case SYM_CVT:
              return Cpl_ParseConvert (script);
            case SYM_ENTER:
              return Cpl_ParseEnter (script);
            case SYM_EXEC:
              return Cpl_ParseExec (script);
            case SYM_EXIT:
              return Cpl_ParseExit (script);
            case SYM_HANGUP:
              return Cpl_ParseHangup (script);
            case SYM_IF:
              return Cpl_ParseIfThenElse (script);
            case SYM_INCR:
              return Cpl_ParseIncr (script);
            case SYM_INVITE:
              return Cpl_ParseInvite (script);
            case SYM_LOG:
              return Cpl_ParseLog (script);
            case SYM_NAME:
              return Cpl_ParseCall (script);
            case SYM_REDIRECT:
              return Cpl_ParseRedirect (script);
            case SYM_SET:
              return Cpl_ParseSet (script);
            case SYM_TIMER:
              return Cpl_ParseTimer (script);
            case SYM_WHILE:
              return Cpl_ParseWhile (script);
            default:
              return Cpl_Error (script, "Statement expected, found '%s'", script->symName);
          } /* switch */


} /* Cpl_ParseStatement() */



static int Cpl_ParseStringExpr (t_script* script)
/*
** returns an index to the operand of an (simple) string expression
**
**      stringexpr  :==  quotedstring  |  varname  |  integer
*/
{
        int                     opInd;
        char                    s[40];

        if (Cpl_GetSym (script) == SYM_ERROR)
          return -1;

        switch (script->sym) {
          case SYM_STRING:
            opInd = Cpl_SymAdd (script, symConst, script->strValue, TRUE);
            Cpl_SymSet (script, opInd, script->strValue, NULL);
            break;
          case SYM_INTEGER:
```

```
            sprintf (s, "%d", script->intValue);
            opInd = Cpl_SymAdd (script, symConst, s, TRUE);
            Cpl_SymSet (script, opInd, s, NULL);
            break;
          case SYM_VARNAME:
            opInd = Cpl_SymAdd (script, symVar, script->symName, TRUE);
            break;
          default:
            return -1;
        }

        return opInd;

} /* Cpl_ParseStringExpr() */



static int Cpl_ParseTimer (t_script* script)
/*
**      timercmd :==  "timer"  timername { { "local" | "global" } timeout  |  "stop" }
*/
{
        int                     timerName;
        int                     timeoutExpr;

        DEBUG (fprintf (stderr, " timer");)

        Cpl_GetSym (script);
        if (script->sym != SYM_NAME)
          return Cpl_Error (script, "Timer name expected, found '%s' (ERR138)", script->symName);

        timerName = Cpl_SymAdd (script, symVar, script->symName, TRUE);
        Cpl_SymSet (script, timerName, script->symName, NULL);

        Cpl_GetSym (script);
        switch (script->sym) {
          case SYM_LOCAL:
            if ((timeoutExpr = Cpl_ParseStringExpr (script)) == -1)
              return Cpl_Error (script, "String expression expected for timeout, found '%s' (ERR139)", script->symName);
            else {
              Cpl_GetSym (script);
              return Cpl_AddCode (script, opStartTimer, timerName, timeoutExpr, TIMER_LOCAL);
            }
          case SYM_GLOBAL:
            if ((timeoutExpr = Cpl_ParseStringExpr (script)) == -1)
              return Cpl_Error (script, "String expression expected for timeout, found '%s' (ERR139)", script->symName);
            else {
              Cpl_GetSym (script);
              return Cpl_AddCode (script, opStartTimer, timerName, timeoutExpr, TIMER_GLOBAL);
            }
          case SYM_STOP:
            Cpl_GetSym (script);
            return Cpl_AddCode (script, opStopTimer, timerName, -1, -1);
          default:
            return Cpl_Error (script, "Expected 'local', 'global' or 'stop', found '%s' (ERR139)", script->symName);
        }

} /* Cpl_ParseTimer() */



static int Cpl_ParseWhile (t_script* script)
/*
** Parse while loop, syntax:
**
**      "while" { condition } procbody
*/
{
        int                     op1, op2, oper;
        int                     savedSP;

        savedSP = script->codeSP;
```

```
        if (Cpl_ParseCondition (script) != CPL_OK)
          return CPL_ERROR;

        if (Cpl_ParseProcBody (script) != CPL_OK)
          return CPL_ERROR;

        Cpl_AddCode (script, opGoto, savedSP, 0, 0);
        script->codeStack[savedSP].addr1 = script->codeSP;

        Cpl_GetSym (script);

        return CPL_OK;

} /* ParseIfThenElse() */


static int Cpl_StateAdd (t_script* script, char* name)
{
        int                     i = script->stateCount;

        DEBUG (fprintf (stderr, "CPL_STATEADD: S%03d %s\n", i, name);)
        script->states[i].enterEvent = NULL;
        script->states[i].name = malloc (strlen (name) +1);
        if ( script->states[i].name == NULL )
          return CPL_ERROR;
        strcpy (script->states[i].name, name);
        script->states[i].eventDefs = NULL;
        script->states[i].dtmfDetection = 0;            /* off */
        script->stateCount++;

        return CPL_OK;

} /* Cpl_StateAdd() */



static int Cpl_SymCheck (t_script* script)
/*
** Check symbol table for unsatisfied references
*/
{
        int                     err = CPL_OK;
        int                     i;


        for (i = 0; i <= script->symInd; i++) {
          if (script->symTab[i].type == symState || script->symTab[i].type == symProc)
            if (script->symTab[i].value == 0) {
              script->lineCount = script->symTab[i].lineNo;
              if (script->symTab[i].type == symState)
                Cpl_Error (script, "Undefined state name '%s' (ERR140)", script->symTab[i].name);
              else
                Cpl_Error (script, "Undefined proc name '%s' (ERR141)", script->symTab[i].name);
              err = CPL_ERROR;
            }
        } /* for */

        return err;

} /* Cpl_SymCheck() */


/*
** exported functions:
*/

int Cpl_Error (t_script* script, char* fmt, ...)
{
        va_list args;

        va_start (args, fmt);
        fprintf (stderr, "\n%s, line %d: ", script->srcFilename, script->lineCount);
        vfprintf (stderr, fmt, args);
```

```
        va_end (args);
        fprintf (stderr, "\n");

        return CPL_ERROR;

} /* Cpl_Error() */



void Cpl_FreeScript (t_script* script)
/*
** frees the script structure
*/
{
        int                     i;
        t_eventDef*             fp;
        t_eventDef*             p;

        if (script == NULL)
          return;

        for (i = 0; i < script->stateCount -1; i++) {
          p = script->states[i].eventDefs;
          while (p) {
            fp = p;
            p = p->next;
            free (fp);
          }
        }
        for (i = 0; i < script->symInd; i++) {
          if (script->symTab[i].name)
            free (script->symTab[i].name);
          if (script->symTab[i].value)
            free (script->symTab[i].value);
        }
        free (script);

} /* Cpl_FreeScript() */



int Cpl_NewScript (t_script** script, char* fn, int logf)
/*
** Opens a script file and returns the script structure
*/
{
        int                     i;

        if (fn == NULL) {
          fprintf (stderr, "No source file given!\n");
          return CPL_ERROR;
        }
        DEBUG (fprintf (stderr, "CPL_NEWSCRIPT: Creating new script\n");)
        if ((*script = malloc (sizeof (t_script))) == NULL) {
          fprintf (stderr, "Cannot allocate space for new script!\n");
          return CPL_ERROR;
        }
        if (((*script)->srcFile = fopen (fn, "r")) == NULL) {
          fprintf (stderr, "Can't open source file %s!\n", fn);
          return CPL_ERROR;
        }

        if (((*script)->srcFilename = malloc (strlen (fn) +1)) == NULL) {
          fprintf (stderr, "Cannot allocate space for filename!\n");
          return CPL_ERROR;
        }
        strcpy ((*script)->srcFilename, fn);
        (*script)->logFile      = logf;
        (*script)->lineCount    = 1;
        (*script)->stateCount   = 0;
        (*script)->symInd       = -1;
        (*script)->codeSP       = 0;
        (*script)->state        = 0;
```

```
        (*script)->pc           = 0;
        (*script)->execPid      = -1;
        (*script)->execStdOut   = -1;
        (*script)->status       = 0;
        (*script)->eventDefs    = NULL;
        for (i = 0; i < MAX_TIMERS; i++)
          (*script)->timers[i] = NULL;

        /*
        ** add predefined system variables to symbol table
        */
        i = Cpl_SymAdd (*script, symVar, "sysVersion", TRUE);
        Cpl_SymSet (*script, i, "V0.01", NULL);
        i = Cpl_SymAdd (*script, symVar, "sysCallDuration", TRUE);
        Cpl_SymSet (*script, i, NULL, Cpl_GetCallDuration);
        i = Cpl_SymAdd (*script, symVar, "sysDate", TRUE);
        Cpl_SymSet (*script, i, NULL, Cpl_GetDate);
        i = Cpl_SymAdd (*script, symVar, "sysDay", TRUE);
        Cpl_SymSet (*script, i, NULL, Cpl_GetDay);
        i = Cpl_SymAdd (*script, symVar, "sysTime", TRUE);
        Cpl_SymSet (*script, i, NULL, Cpl_GetTime);
        i = Cpl_SymAdd (*script, symVar, "sysStatus", TRUE);
        Cpl_SymSet (*script, i, NULL, Cpl_GetStatus);
        i = Cpl_SymAdd (*script, symVar, "sysCallingParty", TRUE);
        i = Cpl_SymAdd (*script, symVar, "sysCalledParty", TRUE);
        i = Cpl_SymAdd (*script, symVar, "sysDtmf", FALSE);
        i = Cpl_SymAdd (*script, symVar, "sysHostname", TRUE);
        Cpl_SymSet (*script, i, NULL, Cpl_GetHostname);
        i = Cpl_SymAdd (*script, symVar, "sysTrace", FALSE);
        Cpl_SymSet (*script, i, "0", NULL);
        i = Cpl_SymAdd (*script, symVar, "sysUsername", TRUE);
        Cpl_SymSet (*script, i, NULL, Cpl_GetUsername);

        return CPL_OK;

} /* Cpl_NewScript */



int Cpl_ParseMain (t_script* script)
/*
** Parse CPL script, syntax:
**
**      application  :==  {  statedef  |  procdef  |  eventdef }*
*/
{
        int                     eof = FALSE;
        int                     err = CPL_OK;

        DEBUG (fprintf (stderr, "CPL_PARSEMAIN:\n");)

        do {
          Cpl_GetSym (script);
          switch (script->sym) {
            case SYM_STATE:
              if (Cpl_ParseState (script) != CPL_OK)
                err = CPL_ERROR;
              break;
            case SYM_PROC:
              if (Cpl_ParseProc (script) != CPL_OK)
                err = CPL_ERROR;
              break;
            case SYM_ON:
              if (Cpl_ParseEventDef (script, TRUE) != CPL_OK)
                err = CPL_ERROR;
              break;
            case SYM_EOF:
              eof = TRUE;
              break;
            case SYM_ERROR:
              err = CPL_ERROR;
              break;
            default:
```

```
          err = Cpl_Error (script, "Unexpected symbol '%s' (ERR142)\n", script->symName);
        }
      } while (err == CPL_OK && !eof);
      fclose (script->srcFile);

      if (err != CPL_OK)
        return CPL_ERROR;

      return Cpl_Check (script);

} /* Cpl_ParseMain() */



int Cpl_SymAdd (t_script* script, int symType, char* symName, int readOnly)
{
        int                     i;

        if (symType == symState || symType == symVar || symType == symProc)
          if ((i = Cpl_SymLookup (script, symName, symType)) != -1)
            return i;

        if (symName == NULL) {
          fprintf (stderr, "INTERNAL ERROR: SymAdd(): Symbol name missing!\n");
          exit (1);
        }

        DEBUG (fprintf (stderr, "\nCpl_SymAdd: Adding %s to symbol table\n", symName);)
        /*
        ** not in symbol table: allocate new entry
        */
        i = ++script->symInd;
        script->symTab[i].type = symType;
        script->symTab[i].name = NULL;
        script->symTab[i].value = NULL;
        script->symTab[i].readOnly = readOnly;
        if (symType == symVar || symType == symState || symType == symProc) {
          script->symTab[i].name = malloc (strlen (symName) +1);
          if (script->symTab[i].name == NULL) {
            fprintf (stderr, "Cannot allocate memory for new symbol table entry\n");
            return -1;
          }
          strcpy (script->symTab[i].name, symName);
        }
        script->symTab[i].lineNo = script->lineCount;

        return i;

} /* Cpl_SymAdd() */



void Cpl_SymDump (t_script* script)
/*
** Dump symbol table for debugging purposes
*/
{
        int                     i;

        fprintf (stderr, "SYMBOL TABLE:\n");
        fprintf (stderr, "   ind  line  type  current value\n");
        for (i = 0; i <= script->symInd; i++) {
          fprintf (stderr, "   %03d  %05d ", i, script->symTab[i].lineNo);
          switch (script->symTab[i].type) {
            case symConst:
              fprintf (stderr, "const '%s'\n",
                      (script->symTab[i].value == NULL ? "<unknown>" : script->symTab[i].value));
              break;
            case symVar:
              fprintf (stderr, "var   '%s'\n",
                       (script->symTab[i].name == NULL ? "<unknown>" : script->symTab[i].name));
              break;
            case symState:
```

```
            fprintf (stderr, "state '%s' StateId = %s\n",
                     (script->symTab[i].name == NULL ? "<unknown>" : script->symTab[i].name),
                     (script->symTab[i].value == NULL ? "<unknown>" : script->symTab[i].value));
            break;
          case symProc:
            fprintf (stderr, "proc  '%s' PC = %s\n",
                     (script->symTab[i].name == NULL ? "<unknown>" : script->symTab[i].name),
                     (script->symTab[i].value == NULL ? "<unknown>" : script->symTab[i].value));
            break;
          default:
            fprintf (stderr, "????? Unknown symbol type in symbol table\n");
        }
      }

} /* Cpl_SymDump() */



int Cpl_SymLookup (t_script* script, char* symName, int symType)
/*
** Find symbol named "symName" of type "symType" in symbol table and
** return index into table
*/
{
        int                     i = 0;

        DEBUG (fprintf (stderr, "\nCpl_SymLookup: searching for '%s', type=%d", symName, symType);)
        for (i = 0; i <= script->symInd; i++)
          if (symType == script->symTab[i].type)
            if (strcmp (script->symTab[i].name, symName) == 0)
              break;
        if (i <= script->symInd) {
          DEBUG (fprintf (stderr, "found\n");)
          return i;
        } else {
          DEBUG (fprintf (stderr, "not found\n");)
          return -1;
        }

} /* Cpl_SymLookup() */



int Cpl_SymSet (t_script* script, int ind, char* value, char* (*getFunc)())
/*
** Sets the value or the function name of an symbol
** table entry specified by its index passed in "i"
*/
{

        if (value != NULL) {
          if (script->symTab[ind].value != NULL)
            free (script->symTab[ind].value);
          script->symTab[ind].value = malloc (strlen (value) +1);
          if (script->symTab[ind].value == NULL) {
            fprintf (stderr, "Cannot allocate memory for value of new symbol table entry\n");
            return CPL_ERROR;
          }
          strcpy (script->symTab[ind].value, value);
        }

        if (getFunc != NULL)
          script->symTab[ind].getValueFunc = getFunc;

        return CPL_OK;

} /* Cpl_SymSet() */
```

H.3.3 CPL Script Interpreter

iSrvCplInt.h

```
/*
** iSrvCplInt.h
**
** (c) 1996 by F. Oertel
**
** 30.12.1996, FO, created
** 04.01.1997, FO, Cpl_GetCallDuration() added
** 14.01.1997, FO, Cpl_GetDay() added
** 16.01.1997, FO, Cpl_GetStatus() added
** 29.01.1997, FO, Cpl_GetHostname() added
** 07.02.1997, Fo, support for global event handling added
*/

#ifndef _I_SRVCPLINT_H
#define _I_SRVCPLINT_H

#include "iSrvStates.h"
#include "iSrvCpl.h"

extern  void                    Cpl_Dump (t_script* script);
extern  char*                   Cpl_GetCallDuration (t_callState* call);
extern  char*                   Cpl_GetDate (t_callState* call);
extern  char*                   Cpl_GetDay (t_callState* call);
extern  char*                   Cpl_GetHostname (t_callState* call);
extern  char*                   Cpl_GetStatus (t_callState* call);
extern  char*                   Cpl_GetTime (t_callState* call);
extern  char*                   Cpl_GetUsername (t_callState* call);
extern  int                     Cpl_Interprete (t_callState* callState);
extern  int                     Cpl_RunTimeErr (t_script* script, char* fmt, ...);
extern  void                    Cpl_Schedule (t_callState* state, t_events event, char* args, int globalFlag);

#endif
```

iSrvCplInt.c

```
/*
** iSrvCplInt.c -- CPL script interpreter
**
** (c) 1997 F. Oertel
**
** 01.01.1997, FO, created
** 04.01.1997, FO, enhanced
** 05.01.1997, FO, Cpl_Redirect() addded
** 08.01.1997, FO, corrections to call redirection
** 12.01.1997, FO, dtmf pattern  matching "enhanced"
** 14.01.1997, FO, handling for opCond added
** 14.01.1997, FO, Cpl_GetDay() added
** 14.01.1997, FO, date format returned by Cpl_GetDate() changed
** 16.01.1997, FO, Cpl_Convert() added
** 16.01.1997, FO, Cpl_GetStatus() added
** 27.01.1997, FO, support for incr /invite statements
** 27.01.1997, FO, command tracing added
** 29.01.1997, FO, Cpl_GetHostname() and Cpl_GetUsername() added
** 31.01.1997, FO, hangup reason added
** 02.02.1997, FO, RtpRedirect() added
** 07.02.1997, FO, support for global timers and global event handling added
** 20.02.1997, FO, corrections for DTMF pattern match check
*/

#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <malloc.h>
#include <time.h>
#include <signal.h>
#include <string.h>
#include <unistd.h>
```

```
#include <pwd.h>

#include "iSrv.h"
#include "iSrvUtils.h"
#include "iSrvStates.h"
#include "iSrvCpl.h"
#include "iSrvCplInt.h"
#include "iSrvCplComp.h"
#include "iSrvCplExec.h"

#include "IsdnLib.h"

#define DEBUG(x)

        int     testFlag = 1;

/*
** forward declarations of local functions
*/

static  char*   Convert (t_script* script, char* s, char* type);
static  void    DumpOpcode (t_script* script, int pc, t_code stmnt);
static  int     EvalCond (t_script* script, char* val1, t_operands oper, char* val2);
static  char*   EvalStr (t_callState* call, t_symTabEntry *sym);
static  void    Cpl_NewStateEntered (t_callState* call);
static  int     pop (t_script* script);
static  void    push (t_script* script, int value);
static  int     Redirect (t_callState* call, char* dest);


/*
** local functions:
*/


static char* ParseNum (char* s, int* n)
{
        char            buf[10];
        int             i = 0;

        *n = -1;
        while (i < sizeof (buf) && isdigit (*s))
          buf[i++] = *s++;
        if (i >= sizeof (buf))
          return NULL;
        buf[i] = '\0';
        *n = atoi (buf);
        return s;

} /* ParseNum() */


static char* Convert (t_script* script, char* s, char* type)
{
        int             n1, n2, n3, n4;
        char*           p;
        char*           target;

        if (strcmp (type, "DTMF2IP") == 0) {
          script->status = ERR_ILL_IP_ADDR;
          p = ParseNum (s, &n1);
          if (n1 < 0 || n1 > 255)
            return NULL;
          p++;
          p = ParseNum (p, &n2);
          if (n2 < 0 || n2 > 255)
            return NULL;
          p++;
          p = ParseNum (p, &n3);
          if (n3 < 0 || n3 > 255)
            return NULL;
          p++;
          p = ParseNum (p, &n4);
```

```
            if (n4 < 0 || n4 > 255)
              return NULL;
            target = malloc (128);
            sprintf (target, "%d.%d.%d.%d", n1, n2, n3, n4);
            script->status = 0;
            return target;
          }

          script->status = ERR_ILL_TYPE;
          return NULL;

} /* Convert() */



static void DumpOpcode (t_script* script, int pc, t_code stmnt)
{
          fprintf (stderr, "      %04d  ", pc);
          switch (stmnt.opcode) {
            case opAccept:
              fprintf (stderr, "ACCEPT\n");
              break;
            case opAudioRec:
              fprintf (stderr, "AUREC @%d\n", stmnt.op1);
              break;
            case opCall:
              fprintf (stderr, "CALL %s (PC=%s)\n",
                       (script->symTab[stmnt.op1].name == NULL ? "<unknown>" : script->symTab[stmnt.op1].name),
                       (script->symTab[stmnt.op1].value == NULL ? "<unknown>" : script->symTab[stmnt.op1].value));
              break;
            case opCond:
              fprintf (stderr, "COMP @%d %d @%d, PC(else)=%d\n", stmnt.op1, stmnt.op2, stmnt.op3, stmnt.addr1);
              break;
            case opCvt:
              fprintf (stderr, "CVT @%d := @%d @%d\n", stmnt.op1, stmnt.op2, stmnt.op3);
              break;
            case opExit:
              fprintf (stderr, "EXIT\n");
              break;
            case opExec:
              fprintf (stderr, "EXEC\n");
              break;
            case opEnter:
              fprintf (stderr, "ENTER %s\n",
                       (script->symTab[stmnt.op1].name == NULL ? "<unknown>" : script->symTab[stmnt.op1].name));
              break;
            case opGoto:
              fprintf (stderr, "GOTO %04d\n", stmnt.op1);
              break;
            case opHangup:
              fprintf (stderr, "HANGUP\n");
              break;
            case opIncr:
              fprintf (stderr, "INCR @%d %d\n", stmnt.op1, stmnt.op2);
              break;
            case opInvite:
              fprintf (stderr, "INVITE @%d @%d @%d\n", stmnt.op1, stmnt.op2, stmnt.op3);
              break;
            case opLog:
              fprintf (stderr, "LOG @%d\n", stmnt.op1);
              break;
            case opRedirect:
              fprintf (stderr, "REDIRECT @%d\n", stmnt.op1);
              break;
            case opReturn:
              fprintf (stderr, "RET\n");
              break;
            case opSet:
              fprintf (stderr, "SET @%d := @%d\n", stmnt.op1, stmnt.op2);
              break;
            case opStartTimer:
              fprintf (stderr, "TIMER @%d %s @%d\n", stmnt.op1, (stmnt.op3 == TIMER_GLOBAL ? "GLBL" : "LCL"), stmnt.op2);
              break;
```

```
          case opStop:
            fprintf (stderr, "STOP\n");
            break;
          default:
            fprintf (stderr, "???\n");
      }
} /* DumpOpcode() */



static int isInteger (char* s)
{
        while (*s)
          if (!isdigit (*s))
            return FALSE;
          else
            s++;

        return TRUE;
}

static int EvalCond (t_script* script, char* val1, t_operands oper, char* val2)
{
        if (script->trace)
          fprintf (stderr, "comparing '%s' %d '%s'\n", val1, oper, val2);
        switch (oper) {
          case opEQ:
            return (strcmp (val1, val2) == 0);
          case opLT:
            if (isInteger (val1) && isInteger (val2))
              return (atoi (val1) < atoi (val2) ? 1 : 0);
            else
              return (strcmp (val1, val2) < 0);
          case opGT:
            if (isInteger (val1) && isInteger (val2))
              return (atoi (val1) > atoi (val2) ? 1 : 0);
            else
              return (strcmp (val1, val2) > 0);
          case opLE:
            if (isInteger (val1) && isInteger (val2))
              return (atoi (val1) >= atoi (val2) ? 1 : 0);
            else
              return (strcmp (val1, val2) <= 0);
          case opGE:
            if (isInteger (val1) && isInteger (val2))
              return (atoi (val1) >= atoi (val2) ? 1 : 0);
            else
              return (strcmp (val1, val2) >= 0);
          case opNE:
            return (strcmp (val1, val2) != 0);
        }

} /* EvalCond() */



static char* EvalStr (t_callState* call, t_symTabEntry* sym)
/*
** Performs some trivial string substitution: Replaces all variables
** (prefixed by $) with its current value.
*/
{
        int                     i, j, ti;
        char*                   s;
        char*                   target;
        char                    varName[1000];
        int                     x;

        target = malloc (1000);
        i = 0;
        ti = 0;
        if (sym->getValueFunc != NULL) {
          DEBUG (fprintf (stderr, "EvalSTR: evaluating %s()", sym->name);)
```

```
          return sym->getValueFunc (call); }
        else {
          if (sym->value == NULL) {
            Cpl_RunTimeErr (call->script, "Variable '%s' not declared (ERR201)", sym->name);
            return NULL;
          }
          DEBUG (fprintf (stderr, "EvalSTR: evaluating '%s'", sym->value);)
        }
        while (sym->value[i]) {
          if (sym->value[i] == '$') {
            x = 0;
            i++;
            while (isdigit (sym->value[i]) || sym->value[i] == '_' || isalpha (sym->value[i]))
              varName[x++] = sym->value[i++];
            varName[x] = '\0';
            DEBUG (fprintf (stderr, "EvalSTR: varName=(%s)\n", varName);)
            if (( j = Cpl_SymLookup (call->script, varName, symVar)) > -1)
              if (call->script->symTab[j].getValueFunc != NULL) {
                s = call->script->symTab[j].getValueFunc (call);
                for (x = 0; x < strlen (s); x++)
                  target[ti++] = s[x];
                free (s);
              } else
                if (call->script->symTab[j].value)
                  for (x = 0; x < strlen (call->script->symTab[j].value); x++)
                    target[ti++] = call->script->symTab[j].value[x];
          } else
            target[ti++] = sym->value[i++];
        }

        target[ti] = '\0';
        DEBUG (fprintf (stderr, "->'%s'\n", target);)
        return target;

} /* EvalStr() */


static int FindCodeEntry (t_script* script, t_events event, char* evArgs, int globalFlag)
/*
** Returns the program counter (PC) for the event handling code
*/
{
        int                     flag = FALSE;
        int                     i;
        t_eventDef*             p;
        char*                   s;
        char                    tmpDtmf[MAX_DTMF_LEN];

        DEBUG (fprintf (stderr, "looking for entrypoint for event %d\n", event);)
        if (event == evDtmf) {
          if (strlen (script->dtmfSeq) == MAX_DTMF_LEN)
            strcpy (script->dtmfSeq, "");
          strcat (script->dtmfSeq, evArgs);
        }
        if (globalFlag && event == evTimeout)
          p = script->eventDefs;                        /* search in global event list */
        else
          p = script->states[script->state].eventDefs;  /* search in local event list */
        while (p) {
          if (p->type == event)
            if (event == evTimeout) {
              if (strcmp (evArgs, script->symTab[p->argInd].value) == 0) {
                script->pc = p->codeStart;
                return p->codeStart;
              }
            } else
              if (event == evDtmf) {
                s = malloc (strlen (script->symTab[p->argInd].value)+1);
                strcpy (s, script->symTab[p->argInd].value);
                if (Cpl_CompileMatchPattern (script, s) == CPL_ERROR) {
                  free (s);
                  return -1;
                }
```

```
                if (Cpl_Match (s, script->dtmfSeq)) {
                  if ((i = Cpl_SymLookup (script, "sysDtmf", symVar)) == -1) {
                    fprintf (stderr, "INTERNAL ERROR: Can't find $sysDtmf in symbol table\n");
                    exit (1);
                  }
                  Cpl_SymSet (script, i, script->dtmfSeq, NULL);
                  strcpy (script->dtmfSeq, "");
                  script->pc = p->codeStart;
                  free (s);
                  return p->codeStart;
                }
                free (s);
              } else {
                script->pc = p->codeStart;
                return p->codeStart;
              }
          p = p->next;
        }

        return -1;

} /* FindCodeEntry() */



static void Cpl_NewStateEntered (t_callState* call)
/*
** This function is called whenever a new state is entered
*/
{
        Cpl_QuitLocalTimers (call);

} /* Cpl_NewStateEntered() */



static int pop (t_script* script)
{
        if (script->stackPtr > -1)
          return script->stack[script->stackPtr--];

} /* pop() */


static void push (t_script* script, int value)
{
        if (script->stackPtr == STACKSIZE) {
          fprintf (stderr, "STACK OVERFLOW!\n");
          exit (1);
        }
        script->stack[++(script->stackPtr)] = value;

} /* push() */



static void QuitAudioLog (t_callState* call)
/*
** Terminates audio recording if it is enabled
*/
{
        if (call->audioLog > -1) {
          DEBUG (fprintf (stderr, "Audio logging stopped\n");)
          close (call->audioLog);
          call->audioLog = -1;
        }

} /* QuitAudioLog() */



static int Redirect (t_callState* call, char* dest)
```

```
{
        t_parsedUrl*            parsedUrl;

        if (call->script->trace)
          fprintf (stderr, "redirect '%s'\n", dest);

        if ((parsedUrl = Cpl_ParseUrl (dest)) == NULL)
          return Cpl_RunTimeErr (call->script, "Syntax error in URL '%s' (ERR207)", dest);

        switch (parsedUrl->type) {
          case phone_url:
            Cpl_IsdnRedirect (call, parsedUrl->phoneNo);
            break;
          case rtp_url:
            Cpl_RtpRedirect (call, parsedUrl->host, parsedUrl->port);
            break;
          default:
            return Cpl_RunTimeErr (call->script, "Unexpected URL type in destination specification (ERR206)");
        }

        free (parsedUrl);
        return CPL_OK;

} /* Redirect() */



/*
** exported functions
*/

void Cpl_Dump (t_script* script)
/*
** Dumps the misc. interpreter tables for debugging purposes
*/
{
        int                     i;
        t_eventDef*             p;
        char*                   eventNames[] = {"enter     ", "disconnect", "timer     ", "dtmf      ", "speech    ", "reschedule"};

        Cpl_SymDump (script);

        fprintf (stderr, "\nSTATE TABLE:\n");
        for (i = 0; i < script->stateCount; i++) {
          fprintf (stderr, "  %s\n", script->states[i].name);
          p = script->states[i].eventDefs;
          while (p) {
            fprintf (stderr, "    ON %s", eventNames[p->type]);
            if (p->type == evDtmf || p->type == evSpeech)
              fprintf (stderr, " '%s'", script->symTab[p->argInd].value);
            fprintf (stderr, " PC=%04d\n", p->codeStart);
            p = p->next;
          }
        }

        fprintf (stderr, "\nCODE TABLE:\n");
        for (i = 0; i < script->codeSP; i++)
          DumpOpcode (script, i, script->codeStack[i]);


} /* Cpl_Dump() */



char* Cpl_GetCallDuration (t_callState* call)
{
        time_t                  now;
        time_t                  td;
        char*                   s;

        time (&now);
        td = now - call->startTime;
        s = malloc (10);
```

```
        sprintf (s, "%02d:%02d:%02d\n", td / 3600, (td % 3600) / 60, (td % 60));
        return s;

} /* Cpl_GetCallDuration() */



char* Cpl_GetDate (t_callState* call)
{
        char*                   s;
        struct tm               *tm;
        time_t                  t;

        s = malloc (20);
        time (&t);
        tm = localtime (&t);
        sprintf (s, "%04d/%02d/%02d", 1900+tm->tm_year, 1+tm->tm_mon, tm->tm_mday);
        DEBUG (fprintf (stderr, "CPL_GETDATE: %s\n", s);)
        return s;

} /* Cpl_GetDate() */



char* Cpl_GetDay (t_callState* call)
/*
** Returns a char* to a buffer containing the current day of week (1=monday,..7)
** This function is invoked via accessing the $sysDay CPL variable.
*/
{
        char*                   s;
        struct tm               *tm;
        time_t                  t;

        s = malloc (20);
        time (&t);
        tm = localtime (&t);
        sprintf (s, "%d", (tm->tm_wday == 0 ? 7 : tm->tm_wday));
        DEBUG (fprintf (stderr, "CPL_TIME: %s\n", s);)
        return s;

} /* Cpl_GetDay() */



char* Cpl_GetHostname (t_callState* call)
{
        char*                   s;

        s = malloc (80);
        gethostname (s, 80);
        return s;

} /* Cpl_GetHostname() */



char* Cpl_GetStatus (t_callState* call)
/*
** Return the current script status
*/
{
        char*                   s;

        s = malloc (20);
        sprintf (s, "%d", call->script->status);
        return s;

} /* Cpl_GetStatus() */



char* Cpl_GetTime (t_callState* call)
```

```
/*
** Returns a char* to a buffer containing the system time (format HH:MM:SS).
** This function is invoked by accessing the $sysTime CPL variable.
*/
{
        char*                   s;
        struct tm               *tm;
        time_t                  t;

        s = malloc (20);
        time (&t);
        tm = localtime (&t);
        sprintf (s, "%02d:%02d:%02d", tm->tm_hour, tm->tm_min, tm->tm_sec);
        DEBUG (fprintf (stderr, "CPL_TIME: %s\n", s);)
        return s;

} /* Cpl_GetTime() */



char* Cpl_GetUsername (t_callState* call)
{
        char*                   s;
        struct passwd* p;

        s = malloc (80);
        p = getpwuid (getuid ());
        if (p)
          strcpy (s, p->pw_name);
        else
          strcpy (s, "<unknown>");

        return s;

} /* Cpl_GetUsername() */




static int Cpl_AbortScript (t_callState* callState)
{
        fprintf (stderr, "\nScript aborted!\n");
        Cpl_Cleanup (callState);
        return CPL_ERROR;

} /* Cpl_AbortScript() */



int Cpl_Interprete (t_callState* callState)
/*
** Interprete the pseudo code pointed to by the program counter (PC).
** Note, that the current state may only changed by the enter statement.
*/
{
        int                     incr;
        char                    newVal[80];
        t_script*               script = callState->script;
        char*                   s;
        char*                   s1;
        char*                   s2;
        char*                   s3;
        t_code                  stmnt;

        if (callState->script == NULL)
          return CPL_ERROR;

        while (1) {
          if (callState->script->trace)
            fprintf (stderr, " PC=%04d  ", script->pc);
          stmnt = script->codeStack[script->pc];
          script->pc++;
          switch (stmnt.opcode) {
```

```
case opAccept:
  if (callState->state != SRV_S_CONNECTED) {
    Cpl_Accept (callState);
    callState->script->status = 0;
    return CPL_OK;
  } else {
    callState->script->status = ERR_ALREADY_CONNECTED;
    break;
  }
case opAudioRec:
  s = EvalStr (callState, &script->symTab[stmnt.op1]);
  Cpl_AudioRec (callState, s);
  free (s);
  break;
case opCall:
  push (script, script->pc);
  if (callState->script->trace)
    fprintf (stderr, "call '%s': ", script->symTab[stmnt.op1].name);
  script->pc = atoi (script->symTab[stmnt.op1].value);
  if (callState->script->trace)
    fprintf (stderr, "branching %04d\n", script->pc);
  callState->script->status = 0;
  continue;
case opCond:
  s1 = EvalStr (callState, &script->symTab[stmnt.op1]);
  s2 = EvalStr (callState, &script->symTab[stmnt.op3]);
  if (!EvalCond (callState->script, s1, stmnt.op2, s2))
    script->pc = stmnt.addr1;
  free (s1);
  free (s2);
  break;
case opCvt:
  s1 = EvalStr (callState, &script->symTab[stmnt.op2]);
  s2 = EvalStr (callState, &script->symTab[stmnt.op3]);
  if (callState->script->trace)
    fprintf (stderr, "cvt %s '%s' '%s'\n", script->symTab[stmnt.op1].name, s1, s2);
  s = Convert (script, s1, s2);
  if (s) {
    Cpl_SymSet (script, stmnt.op1, s, NULL);
    free (s);
  }
  free (s1);
  free (s2);
  break;                                /* resume script execution! */
case opEnter:
  if (callState->script->trace)
    fprintf (stderr, "enter '%s': ", script->symTab[stmnt.op1].name);
  script->state = atoi (script->symTab[stmnt.op1].value);
  script->dtmfDetection =               /* DTMF detection needed for */
    script->states[script->state].dtmfDetection;    /* this state? */
  DEBUG (fprintf (stderr, "\nDTMF detection %s\n", (script->dtmfDetection ? "on" : "off"));)
  if (script->states[script->state].enterEvent == NULL) {
    fprintf (stderr, "No enter event definition for state %d\n", script->state);
    callState->script->status = 0;
    return CPL_OK;                      /* deschedule */
  } else {
    script->pc = script->states[script->state].enterEvent->codeStart;
    Cpl_NewStateEntered (callState);
    fprintf (stderr, "branching to state %d, PC=%04d\n", script->state, script->pc);
    callState->script->status = 0;
    continue;                           /* go and process enter event statements! */
  }
case opExec:
  s = EvalStr (callState, &script->symTab[stmnt.op1]);
  if (Cpl_Exec (callState, s) != CPL_OK) {
    free (s);
    break;
  } else {
    free (s);
    return CPL_OK;                      /* deschedule! */
  }
case opExit:
  if (callState->state == SRV_S_CONNECTED)
```

```
        Cpl_Hangup (callState, NULL);
      Cpl_Cleanup (callState);
      return CPL_OK;                          /* never come back! */
    case opGoto:
      if (callState->script->trace)
        fprintf (stderr, "goto %04d\n", stmnt.op1);
      script->pc = stmnt.op1;
      continue;
    case opHangup:
      if (callState->state == SRV_S_CONNECTED) {
        if (stmnt.op1 > -1)
          s = EvalStr (callState, &script->symTab[stmnt.op1]);
        else
          s = NULL;
        Cpl_Hangup (callState, s);
        callState->script->status = 0;
        if (stmnt.op1 > -1)
          free (s);
        return CPL_OK;                          /* deschedule! */
      }
      callState->script->status = ERR_NOT_CONNECTED;
      break;
    case opIncr:
      if (stmnt.op2 > -1) {
        if ((s = EvalStr (callState, &script->symTab[stmnt.op2])) == NULL)
          return Cpl_AbortScript (callState);
        incr = atoi (s);
        free (s);
      } else
        incr = 1;
      if (callState->script->trace)
        fprintf (stderr, "incr %s %d\n", script->symTab[stmnt.op1].name, incr);
      sprintf (newVal, "%d", atoi (script->symTab[stmnt.op1].value) + incr);
      Cpl_SymSet (script, stmnt.op1, newVal, NULL);
      callState->script->status = 0;
      break;
    case opInvite:

      callState->script->status = 0;
      break;
    case opLog:
      s = EvalStr (callState, &script->symTab[stmnt.op1]);
      if (write (script->logFile, s, strlen (s)) == -1)
        perror ("write (logFile)");
      if (callState->script->trace)
        if (strlen (s) > 30) {
          s[30] = '\0';
          fprintf (stderr, "log '%s...'\n", s);
        } else
          fprintf (stderr, "log '%s'\n", s);
      callState->script->status = 0;
      free (s);
      break;                                  /* resume script execution */
    case opRedirect:
      s = EvalStr (callState, &script->symTab[stmnt.op1]);
      if (Redirect (callState, s) != CPL_OK) {
        free (s);
        return Cpl_AbortScript (callState);
      } else {
        free (s);
        return CPL_OK;                          /* deschedule! */
      }
    case opReturn:
      if (callState->script->trace)
        fprintf (stderr, "return\n");
      script->pc = pop (script);
      continue;
    case opSet:
      s = EvalStr (callState, &script->symTab[stmnt.op2]);
      if (callState->script->trace)
        fprintf (stderr, "set %s '%s'\n", script->symTab[stmnt.op1].name, s);
      Cpl_SymSet (script, stmnt.op1, s, NULL);
      free (s);
```

```
        if (strcmp (script->symTab[stmnt.op1].name, "sysTrace") == 0)
          callState->script->trace = atoi (s);
        break;                                    /* resume script execution! */
      case opStartTimer:
        s = EvalStr (callState, &script->symTab[stmnt.op1]);
        s1 = EvalStr (callState, &script->symTab[stmnt.op2]);
        if (callState->script->trace)
          fprintf (stderr, "timer %s %s %s ms\n", s, (stmnt.op3 == TIMER_LOCAL ? "local" : "global"), s1);
        Cpl_StartTimer (callState, s, atoi (s1), stmnt.op3);
        free (s);
        break;                                    /* resume script execution! */
      case opStop:
        if (callState->script->trace)
          fprintf (stderr, "stopped... ");
        script->pc--;
        return CPL_OK;                            /* end of procbody, deschedule! */
      case opStopTimer:
        s = EvalStr (callState, &script->symTab[stmnt.op1]);
        if (callState->script->trace)
          fprintf (stderr, "stop timer %s\n", 1);
        Cpl_StopTimer (callState, s);
        free (s);
        break;                                    /* resume script execution!*/
      default:
        Cpl_RunTimeErr (script, "Illegal CPL instruction, PC=%04d (ERR202)", script->pc);
        return Cpl_AbortScript (callState);
    }
    if (Cpl_Reschedule (callState))
      return CPL_OK;

  } /* while */

} /* Cpl_Interprete() */



int Cpl_RunTimeErr (t_script* script, char* fmt, ...)
{
        char                    buf[512];
        va_list                 args;

        va_start (args, fmt);
        fprintf (stderr, "\n%s: ", script->srcFilename);
        sprintf (buf, "\n%s: ", script->srcFilename);
        write (script->logFile, buf, strlen (buf));
        vfprintf (stderr, fmt, args);
        vsprintf (buf, fmt, args);
        write (script->logFile, buf, strlen (buf));
        fprintf (stderr, "\n");
        write (script->logFile, "\n", 1);
        va_end (args);
        return CPL_ERROR;

} /* Cpl_RunTimeErr() */



void Cpl_Schedule (t_callState* state, t_events event, char* parms, int globalFlag)
{

        if (state->script == NULL)              /* script already terminted? */
          return;                               /* leave! */

        if (event == evReschedule) {
          QuitAudioLog (state);
          Cpl_QuitExecProc (state);             /* kill pending exec process */
          Cpl_QuitRedirect (state);             /* terminate pending call redirection */
          if (state->script->trace)
            fprintf (stderr, " resuming...\n");
          Cpl_Interprete (state);
        } else
          if (FindCodeEntry (state->script, event, parms, FALSE) > -1) {
            QuitAudioLog (state);
```

```
          Cpl_QuitExecProc (state);           /* kill pending exec process */
          Cpl_QuitRedirect (state);           /* terminate pending call redirection */
          if (state->script->trace)
            fprintf (stderr, " resuming...\n");
          Cpl_Interprete (state);
        } else
          if (event == evDisconnect) {
            QuitAudioLog (state);
            Cpl_QuitRedirect (state);         /* terminate pending call redirection */
            Cpl_Cleanup (state);
          } else
            if (event == evTimeout)
              if (FindCodeEntry (state->script, event, parms, TRUE) > -1) {
                QuitAudioLog (state);
                Cpl_QuitExecProc (state);     /* kill pending exec process */
                Cpl_QuitRedirect (state);     /* terminate pending call redirection */
                if (state->script->trace)
                  fprintf (stderr, " resuming...\n");
                Cpl_Interprete (state);
              }

} /* Cpl_Schedule() */
```

H.3.4 CPL Run-Time-Functions

`iSrvCplExec.h`

```
/*
** iSrvCplExec.h
*/

#ifndef _ISRVCPLEXEC_H
#define _ISRVCPLEXEC_H

extern  void    ExecStdOutCb (void *ClientData, int mask);
extern  void    Cpl_Accept (t_callState* call);
extern  void    Cpl_AudioRec (t_callState* call, char* fn);
extern  void    Cpl_Cleanup (t_callState* call);
extern  int     Cpl_Exec (t_callState* call, char* cmd);
extern  void    Cpl_Hangup (t_callState* call, char* reason);
extern  int     Cpl_IsdnRedirect (t_callState* call, char* toNumber);
extern  void    Cpl_QuitAllTimers (t_callState* call);
extern  void    Cpl_QuitExecProc (t_callState* call);
extern  void    Cpl_QuitLocalTimers (t_callState* call);
extern  void    Cpl_QuitRedirect (t_callState* call);
extern  int     Cpl_Reschedule (t_callState* call);
extern  int     Cpl_RtpRedirect (t_callState* call, char* host, int port);
extern  int     Cpl_StartTimer (t_callState* call, char* timerName, int milliSecs, int scopeFlag);
extern  int     Cpl_StopTimer (t_callState* call, char* timerName);
#endif
```

`iSrvCplExec.c`

```
/*
** iSrvCplExec.c -- Execution of CPL commands
**
** (c) 1997 F. Oertel
**
** 04.01.1997, FO, Cpl_Exec() (and others) implemented
** 05.01.1997, FO, Cpl_Audiorec() finished
** 05.01.1997, FO, Cpl_Exec() finish
** 05.01.1997, FO, Cpl_Redirect(), Cpl_Mbone(), Cpl_Scip() added
** 06.01.1997, FO, Cpl_Exec reenginered due to timing problems
** 07.01.1997, FO, Cpl_Audioplay() removed
** 08.01.1997, FO, Cpl_Start/StopTimer() added
** 19.01.1997, FO, bug in returning of system() exit status fixed
** 06.02.1997, FO, Cpl_Reschedule() added
** 07.02.1997, FO, support for global timers
*/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/wait.h>
#include <malloc.h>
#include <signal.h>

#define DEBUG(x)

#include "iSrv.h"
#include "iSrvStates.h"
#include "iSrvAudio.h"
#include "iSrvCplExec.h"
#include "iSrvCplInt.h"
#include "iSrvUtils.h"

#include "IsdnLib.h"

#define PIPE_BUF_SIZE           512

static  void                    HandleTimeOut (void* data);
static  void                    iFwdBchannelCb (void *ClientData, int mask);
static  void                    iFwdHandleChannelInd (t_callState *state);
static  void                    iFwdHandleConnectInd (t_callState *state);
static  void                    iFwdHandleDisconnectInd (t_callState *state);
static  void                    iFwdHandleReleaseInd (t_callState *state);

/*
** internals:
*/

static void HandleTimeOut (void* data)
{
        t_timers*               timerId = (t_timers*) data;
        t_callState*            call = ((t_callState*)((t_timers*)data)->state);

        DEBUG (fprintf (stderr, "TIMEOUT FOR '%s'!\n", timerId->name);)

        strcpy (call->script->dtmfSeq, "");
        Cpl_Schedule (call, evTimeout, timerId->name, timerId->scope);

        Cpl_StopTimer (call, timerId->name);

} /* HandleTimeOut() */



static void iFwdBchannelCb (void *ClientData, int mask)
/*
** read data from second B channel
*/
{
        t_callState*            state = (t_callState*)ClientData;
        char                    buf[320];
        int                     n;

        n = read (state->bChannelFd, buf, sizeof (buf));
        if (n < 0) {
          perror ("read (bChannel)");
          return;
        }
        if (((t_callState*)state->fwdState)->bChannelFd > -1)
          if (write (((t_callState*)state->fwdState)->bChannelFd, buf, n) == -1) {
            perror ("write() to first B channel");
            return;
          }

} /* iFwdBchannelCb() */
```

```
static void iFwdHandleChannelInd (t_callState *state)
{
        IsdnCreateFileHandler (state->bChannelFd, ISDN_READABLE, iFwdBchannelCb, state);
        return;

} /* iFwdHandleChannelInd() */



static void iFwdHandleConnectInd (t_callState *state)
{
        return;

} /* iFwdHandleConnectInd() */



static void iFwdHandleDisconnectInd (t_callState *state)
{
        t_IsdnReleaseReq        iReq;

        ZERO (iReq)
        iReq.callId = state->callId;
        if (IsdnReleaseReq (&iReq) < 0)
          IsdnError ("*** Error while RELEASE:");
/*
        if (state->fwdState != NULL) {
          ZERO (iReq)
          iReq.callId = ((t_callState*)state->fwdState)->callId;
          if (IsdnReleaseReq (&iReq) < 0)
            IsdnError ("*** Error while RELEASE:");
        }
*/
        return;

} /* iFwdHandleDisconnectInd() */



void iFwdHandleReleaseInd (t_callState *state)
{

        ((t_callState*)state->fwdState)->fwdState = NULL;
        return;

} /* iFwdHandleReleaseInd() */



void Reschedule (void* data)
{
        t_callState*            call = (t_callState*) data;

        DEBUG (fprintf (stderr, "reschedule\n");)
        Cpl_Schedule (call, evReschedule, NULL, FALSE);

} /* Reschedule() */



/*
** externals:
*/

void Cpl_Accept (t_callState* call)
{
        t_IsdnConnectReq        iReq;

        call->state = SRV_S_CONNECTING;

        fprintf (stderr, "accept, ");
        ZERO (iReq);
        iReq.callId = call->callId;
```

```
        if (IsdnConnectReq (&iReq) < 0) {
          IsdnError ("*** Error while CONNECT:");
          return;
        }

} /* Cpl_Accept() */



void Cpl_AudioRec (t_callState* call, char* fn)
{
        t_auHeader              header;

        fprintf (stderr, "audiorec '%s'\n", fn);
        call->audioLog = open (fn, O_CREAT | O_WRONLY | O_TRUNC, 0777);
        if (call->audioLog == -1) {
          Cpl_RunTimeErr (call->script, "Can't open file '%s' for recording audio (ERR203)", fn);
          call->audioLog = -1;
          call->script->status = ERR_OPENING_FILE;
        }
        /*
        ** write au file header
        */
        strncpy (header.magic, ".snd", 4);
        header.off = sizeof (t_auHeader);
        header.len = -1;
        header.fmt = 27;                        /* A law */
        header.rate = 8000;
        header.channels = 1;
        write (call->audioLog, &header, sizeof (header));
        call->script->status = 0;

} /* Cpl_AudioRec() */


void Cpl_Cleanup (t_callState* call)
{

        if (call->script->trace)
          fprintf (stderr, "exit '%s'\n", call->script->srcFilename);

        IsdnDeleteFileHandler (call->bChannelFd);
        call->bChannelFd = -1;

        Cpl_QuitExecProc (call);
        Cpl_QuitAllTimers (call);

        call->script->state = -1;       /* ? */
        close (call->script->logFile);
        Cpl_FreeScript (call->script);
        call->script = NULL;

} /* Cpl_Cleanup() */



int Cpl_Exec (t_callState* call, char* cmd)
/*
** Execute system command via system()
*/
{
        int                     i;
        int                     null;
        int                     p[2];
        int                     rc;

        if (pipe (p) == -1) {
          perror ("pipe()");
          return CPL_ERROR;
        }
        call->script->execStdOut        = -1;
        call->script->execRptr          = 0;
        call->script->execWptr          = 1;
```

```
        ZERO (call->script->execRbuf);
        switch (call->script->execPid = fork ()) {
          case -1:
            Cpl_RunTimeErr (call->script, "Error forking in server process (ERR204)");
            return CPL_ERROR;
          case 0:
            close (p[0]);
            if (call->script->trace)
              fprintf (stderr, "exec '%s' ", cmd);

            null = open ("/dev/null", O_RDWR);
            dup2 (null, 0);                              /* stdin is /dev/null */
            dup2 (call->script->logFile, 2);             /* stderr is /dev/null */
            dup2 (p[1], 1);                              /* redirect stdout to pipe */
            for (i = 3; i < 40; i++)                     /* MAX FD ?!?!?!?!? */
              close (i);
            signal (SIGCHLD, SIG_DFL);
            rc = system (cmd);                           /* execute command */
            if (WEXITSTATUS (rc) != 0)                   /* error occured? */
              fprintf (stderr, "system('%s') exits with %d\n", cmd, WEXITSTATUS (rc));
            exit (WEXITSTATUS (rc));

          default:
            DEBUG (fprintf (stderr, "exec PID=%d\n", call->script->execPid);)
            call->script->execStdOut = p[0];
            close (p[1]);
            call->script->execBytes = 0;
            call->bChannelBytes = 0;
            IsdnCreateFileHandler (p[0], ISDN_READABLE, ExecStdOutCb, call);
        }

        call->script->status = 0;
        return CPL_OK;

} /* Cpl_Exec() */



void ExecStdOutCb (void *ClientData, int mask)
/*
** Read process' stdout from pipe and write into ring buffer
*/
{
        char                    buf[PIPE_BUF_SIZE];
        int                     bFree;
        int                     i;
        t_callState*            state = (t_callState*)ClientData;
        int                     n;
        int                     rPtr, wPtr;

        n = read (state->script->execStdOut, buf, sizeof (buf));
        DEBUG (fprintf (stderr, "got %d bytes from pipe\n", n);)
        if (n <= 0) {
          IsdnDeleteFileHandler (state->script->execStdOut);
          close (state->script->execStdOut);           /* close pipe */
          state->script->execStdOut = -1;
          return;
        }
        DEBUG (fprintf (stderr, "f=%d ", n);)
        wPtr = state->script->execWptr;
        rPtr = state->script->execRptr;
        bFree = (wPtr >= rPtr ? R_BUF_SIZE - wPtr + rPtr : rPtr - wPtr);
        if (bFree < n) {
          fprintf (stderr, "INTERNAL ERROR ExecStdOutCb(): Ringbuffer overflow!\n");
          IsdnDeleteFileHandler (state->script->execStdOut);
          return;
        }
        for (i = 0; i < n; i++) {
          state->script->execRbuf[wPtr++] = buf[i];
          if (wPtr >= R_BUF_SIZE)
            wPtr = 0;
        }
        bFree = (wPtr >= rPtr ? R_BUF_SIZE - wPtr + rPtr : rPtr - wPtr);
```

```
        if (sizeof (buf) * 2 > bFree) {
          DEBUG (fprintf (stderr, "ringbuf full, r=%d w=%d free=%d\n", state->script->execRptr, state->script->execWptr, bFree);)
          IsdnDeleteFileHandler (state->script->execStdOut);
        }
        state->script->execWptr = wPtr;
        state->script->execBytes += n;

        return;

} /* ExecStdOutCb() */



void Cpl_Hangup (t_callState* call, char* reason)
/*
** Disconnect from ISDN line
*/
{
        t_IsdnReleaseReq        iReq;

        fprintf (stderr, "hangup '%s'\n", (reason == NULL ? "" : reason));
        ZERO (iReq);
        iReq.callId = call->callId;
        if (IsdnReleaseReq (&iReq) < 0) {
          IsdnError ("*** Error while RELEASE:");
          return;
        }

} /* Cpl_Hangup() */



int Cpl_IsdnRedirect (t_callState* call, char* toNumber)
{
        int                     i;
        t_IsdnCallReq           iReq;
        t_callState*            fwdCall;

        ZERO (iReq);
        strcpy (iReq.calledNo.number, toNumber);
        if (IsdnCallReq (&iReq) < 0) {
          IsdnError ("*** Error while CALL-REQ:");
          call->script->status = ERR_CALL_FAILED;
          return CPL_OK;
        }
        if ((fwdCall = CallStateNew ()) == NULL) {
          fprintf (stderr, "*** Too many ISDN connections!\n");
          return CPL_ERROR;
        }
        call->fwdState = fwdCall;
        fwdCall->fwdState = call;
        fwdCall->state = SRV_S_CALLING;
        fwdCall->callId = iReq.callId;
        fwdCall->bChannelFd = -1;
        fwdCall->clientAudioFd = -1;
        fwdCall->clientAudioPort = -1;
        strcpy (fwdCall->calledParty, toNumber);

        fwdCall->ConnectIndHandler = iFwdHandleConnectInd;
        fwdCall->ChannelIndHandler = iFwdHandleChannelInd;
        fwdCall->DisconnectIndHandler = iFwdHandleDisconnectInd;
        fwdCall->ReleaseIndHandler = iFwdHandleReleaseInd;

        call->script->status = 0;
        return CPL_OK;

} /* Cpl_IsdnRedirect() */


void Cpl_QuitAllTimers (t_callState* call)
{
        int                     i;
```

```
        for (i = 0; i < MAX_TIMERS; i++)
          if (call->script->timers[i]) {
            fprintf (stderr, "disabling timer '%s'\n", call->script->timers[i]->name);
            IsdnDeleteTimerHandler (call->script->timers[i]->id);
            free (call->script->timers[i]);
            call->script->timers[i] = NULL;
          }

} /* Cpl_QuitAllTimers() */



void Cpl_QuitExecProc (t_callState* call)
{
        int                     pid;

        if (call->script->execPid < 0)
          return;

        IsdnDeleteFileHandler (call->script->execStdOut);
        close (call->script->execStdOut);
        call->script->execStdOut = -1;
        pid = call->script->execPid;
        call->script->execPid = -1;
        DEBUG (fprintf (stderr, "killing %d\n", pid);)
        kill (pid, SIGKILL);

} /* Cpl_QuitExecProc() */



void Cpl_QuitLocalTimers (t_callState* call)
{
        int                     i;

        fprintf (stderr, "stopping local timers\n");
        for (i = 0; i < MAX_TIMERS; i++)
          if (call->script->timers[i])
            if (call->script->timers[i]->scope == TIMER_LOCAL) {
              fprintf (stderr, "disabling timer '%s'\n", call->script->timers[i]->name);
              IsdnDeleteTimerHandler (call->script->timers[i]->id);
              free (call->script->timers[i]);
              call->script->timers[i] = NULL;
            }

} /* Cpl_QuitLocalTimers() */



void Cpl_QuitRedirect (t_callState* call)
/*
** Terminate pending ISDN call redirection by releasing
** the second B channel to the redirection destination.
*/
{
        t_IsdnReleaseReq        iReq;

        if (call->fwdState == NULL)
          return;
        DEBUG (fprintf (stderr, "canceling call redirection\n");)
        IsdnDeleteFileHandler (call->bChannelFd);
        ZERO (iReq)
        iReq.callId = ((t_callState*)call->fwdState)->callId;
        if (IsdnReleaseReq (&iReq) < 0)
          IsdnError ("*** Error while RELEASE:");
        ((t_callState*)call->fwdState)->fwdState = NULL;

} /* Cpl_QuitRedirect() */



int Cpl_Reschedule (t_callState* call)
/*
```

```
** Install timer, to resume code execution
*/
{

        IsdnCreateTimerHandler (1, Reschedule, call);
        return 1;

} /* Cpl_Reschedule() */



int Cpl_RtpRedirect (t_callState* call, char* host, int port)
{
        return CPL_OK;

} /* Cpl_RtpRedirect() */



int Cpl_StartTimer (t_callState* call, char* timerName, int milliSecs, int scopeFlag)
{
        int                     i;

        for (i = 0; i < MAX_TIMERS; i++)
          if (call->script->timers[i] == NULL)
            break;
        if (i == MAX_TIMERS) {
          Cpl_RunTimeErr (call->script, "Too many timers active at the same time (ERR205)");
          call->script->status = ERR_TOO_MANY_TIMERS;
          return CPL_ERROR;
        }

        call->script->timers[i] = malloc (sizeof (t_timers));
        if (call->script->timers[i] == NULL) {
          Cpl_RunTimeErr (call->script, "Can't allocate space for timer");
          call->script->status = ERR_MALLOC_FAILED;
          return CPL_ERROR;
        }

        strcpy (call->script->timers[i]->name, timerName);
        call->script->timers[i]->scope = scopeFlag;
        (t_callState*)call->script->timers[i]->state = call;
        call->script->timers[i]->id = IsdnCreateTimerHandler (milliSecs, HandleTimeOut, call->script->timers[i]);
        if (call->script->timers[i]->id == -1) {
          free (call->script->timers[i]);
          call->script->timers[i] = NULL;
          Cpl_RunTimeErr (call->script, "Error creating timer '%s'", timerName);
          call->script->status = ERR_CREATE_TIMER;
          return CPL_ERROR;
        }

        call->script->status = 0;
        return CPL_OK;

} /* Cpl_StartTimer() */



int Cpl_StopTimer (t_callState* call, char* timerName)
{
        int                     i;

        for (i = 0; i < MAX_TIMERS; i++)
          if (call->script->timers[i] != NULL)
            if (strcmp (call->script->timers[i]->name, timerName) == 0) {
              IsdnDeleteTimerHandler (call->script->timers[i]->id);
              free (call->script->timers[i]);
              call->script->timers[i] = NULL;
              call->script->status = 0;
              return CPL_OK;
            }

        call->script->status = ERR_NO_SUCH_TIMER;
```

```
        return CPL_ERROR;

} /* Cpl_StopTimer() */
```

H.3.5 Utility Functions

`iSrvUtils.h`

```
/*
** iSrvUtils.h
**
** 31.01.1997, FO, Cpl_ParseUrl() added
*/

#ifndef _ISRVUTILS_H
#define _ISRVUTILS_H

#include "iSrvCpl.h"

extern  int                     Cpl_CompileMatchPattern (t_script* script, char *pattern);
extern  int                     Cpl_Match (char* p, char* s);
extern  char*                   GetSym (char* src, char* target);
extern  void                    Msg (char* fmt, ...);
extern  int                     MsgOpen (char* fn);
extern  t_parsedUrl*            Cpl_ParseUrl (char* url);
#endif
```

`iSrvUtils.c`

```
/*
** iSrvUtils.c  --  some helpful functions used by the isdn server
**
** (c) 1996 F. Oertel
**
** 13.01.1997, FO, new functions for pattern mathcing
** 31.01.1997, FO, URL parsing function added
** 20.02.1997, FO, changes to Cpl_CompileMatchPattern()
**
*/

#include <stdio.h>
#include <string.h>
#include <stdarg.h>
#include <time.h>
#include <malloc.h>

#define DEBUG(x)
#include "iSrv.h"
#include "iSrvUtils.h"

static  FILE*                   logf = NULL;


#define VALID_PATTERN_CHARS     "0123456789ABCD*#'ad?"


static int CharMatch (char pc, char c)
{
        switch (pc) {
          case '_':     return 1;
          case '%':     return 1;
          case 'i':     return (isdigit (c) ? 1 : 0);
          case 'I':     return (isdigit (c) ? 1 : 0);
          case 'z':     return (strchr ("ABCD", c) ? 1 : 0);
          case 'Z':     return (strchr ("ABCD", c) ? 1 : 0);
          default:      return (pc == c ? 1 : 0);
        }
} /* CharMatch() */
```

```
int Cpl_CompileMatchPattern (t_script* script, char *p)
{
        int                     flag = 0;
        int                     i;
        char*                   pattern;
        int                     plen = strlen (p);
        int                     x = 0;

        for (i = 0; i < plen; i++)
          if (!strchr (VALID_PATTERN_CHARS, p[i]))
            return Cpl_RunTimeErr (script, "Invalid character '%c' in match string", p[i]);

        pattern = malloc (plen +1);
        for (i = 0; i < plen; i++) {
          if (p[i] == '\'') {
            if (flag)
              flag = 0;
            else
              flag = 1;
            continue;
          }
          if (flag) {
            pattern[x++] = p[i];
            continue;
          }
          switch (p[i]) {
            case 'a': pattern[x++] = 'z'; break;
            case 'A': pattern[x++] = 'Z'; break;
            case 'd': pattern[x++] = 'i'; break;
            case 'D': pattern[x++] = 'I'; break;
            case '?': pattern[x++] = '_'; break;
            case '*': pattern[x++] = '%'; break;
            default:
              pattern[x++] = p[i];
          }
        } /* for */
        pattern[x] = '\0';
        if (flag) {
          free (pattern);
          return Cpl_RunTimeErr (script, "Missing closing ' in match pattern '%s'\n", p);
        }

        strcpy (p, pattern);
        free (pattern);
        return CPL_OK;

} /* Cpl_CheckMatchPattern() */


int Cpl_Match (char* p, char* s)
{
        char                    c;
        int                     flag = 0;
        int                     i, j;
        int                     plen = strlen (p);
        int                     slen = strlen (s);
        int                     x = 0;

        DEBUG (fprintf (stderr, "pattern='%s', string='%s'\n", p, s);)

        if (slen < plen)
          return 0;

        j = 0;
        i = 0;
        while (1) {
          DEBUG (printf ("pattern[%d]=%c==s[%d]=%c\n", j, p[j], i, s[i]);)
          if (j >= plen) {                      /* end of pattern string? */
            return (i < slen ? 0 : 1);
          }
          if (i >= slen) {
            return (j < plen ? 0 : 1);
```

```
        }
        switch (p[j]) {
          case '_':                           /* one char ? */
            i++;
            j++;
            break;
          case 'i':                           /* one digit ? */
            if (isdigit (s[i])) {
              i++;
              j++;
              break;
            }
            return 0;
          case 'I':                           /* one or more digits ? */
            if (!isdigit (s[i])) {            /* at least one is required! */
              return 0;
            }
            do {
              i++;
            } while (i < slen && isdigit (s[i]));
            j++;
            break;
          case 'z':                           /* one alpha ? */
            if (strchr ("ABCD", s[i])) {
              i++;
              j++;
              break;
            }
            return 0;
          case 'Z':                           /* one or more alphas ? */
            if (!strchr ("ABCD", s[i])) {     /* at least one is required! */
              return 0;
            }
            do {
              i++;
            } while (i < slen && strchr ("ABCD", s[i]));
            j++;
            break;
          case '%':                           /* one or more characters ? */
            c = (j + 1 < plen ? p[j +1] : '\0' );
            do {
              i++;
            } while (i < slen && (c == '\0' || !CharMatch (c, s[i])));
            j++;
            break;
          default:
            if (p[j] == s[i]) {
              i++;
              j++;
              break;
            }
            return 0;
        } /* switch */
      } /* while */

} /* Cpl_Match() */



char* GetSym (char* src, char* target)
{
        while (*src && isspace (*src))          /* skip leading white space chars */
          src++;
        while (*src && !isspace (*src)) {       /* copy source chars to target */
          if (!isspace (*src))
            *target++ = *src;
          src++;
        }
        *target = '\0';
        while (*src && isspace (*src))          /* skip trailing white spaces */
          src++;

        return src;
```

```
} /* GetSym */


void Msg (char* fmt, ...)
/*
** Generate server log entry, format: YYYY/MM/DD HH:MM:SS msg
*/
{
        va_list                 args;
        char                    s[128];
        struct tm               *tm;
        time_t                  t;

        if (logf == NULL) {
          fprintf (stderr, "INTERNAL ERROR: Server log file not opend!\n");
          exit (1);
        }
        time (&t);
        tm = localtime (&t);
        sprintf (s, "%04d/%02d/%02d %02d:%02d:%02d ", 1900+tm->tm_year,
                 1+tm->tm_mon, tm->tm_mday, tm->tm_hour, tm->tm_min, tm->tm_sec);
        fprintf (stderr, "%s", s);
        fprintf (logf, "%s", s);
        va_start (args, fmt);
        vfprintf (stderr, fmt, args);
        vfprintf (logf, fmt, args);
        va_end (args);
        fprintf (stderr, "\n");
        fflush (stderr);
        fprintf (logf, "\n");
        fflush (logf);

} /* Msg() */


void MsgClose ()
{
        fclose (logf);

} /* MsgClose() */


int MsgOpen (char* fn)
{
        if ((logf = fopen (fn, "a")) == NULL) {
          logf = NULL;
          return -1;
        }

} /* MsgOpen() */


t_parsedUrl* Cpl_ParseUrl (char* url)
{
#define BUF_SIZE                200

        char                    s[BUF_SIZE];
        int                     i = 0;
        t_parsedUrl*            p;

        DEBUG (fprintf (stderr, "parsing URL '%s'\n", url);)

        while (*url && isspace (*url))
          url++;                        /* skip leading spaces */
        while (*url && *url != ':' && i < BUF_SIZE)
          s[i++] = tolower(*url++);     /* scan url scheme */
        if (!*url)
          return NULL;
        if (*(++url) != '/')            /* skip : */
```

```
  return NULL;
if (*(++url) != '/')            /* skip / */
  return NULL;
s[i] = '\0';
DEBUG (fprintf (stderr, "URL type: %s\n", s);)
p = malloc (sizeof (t_parsedUrl));
memset (p, '\0', sizeof (t_parsedUrl));

url++;                          /* skip / */
if (!strcmp (s, "phone")) {
  /*
  ** phone URL syntax:
  **
  **    "phone://" [ "+" countrycode "-" ] phoneno
  */
  p->type = phone_url;
  if (*url == '+') {            /* country code prefix? */
    i = 0;
    url++;                      /* skip + */
    while (*url && *url != '-' && i < 256)
      if (isdigit (*url))
        p->countryCode[i++] = *url++;
      else
        url++;
    if (*url != '-') {
      free (p);
      return NULL;
    }
    p->countryCode[i] = '\0';
    url++;                      /* - */
  }
  i = 0;
  while (*url && i < 256)
    if (isdigit (*url))
      p->phoneNo[i++] = *url++;
    else
      url++;
  if (i == 0) {
    free (p);
    return NULL;
  }
  p->phoneNo[i] = '\0';
  return p;
}

if (!strcmp (s, "rtp")) {
  /*
  ** rtp URL syntax: "rtp://" host ":" portnumber
  */
  p->type = rtp_url;
  i = 0;
  while (*url && *url != ':' && i < 256)
    p->host[i++] = *url++;              /* extract host spec */
  if (i == 0) {                         /* host spec is required! */
    free (p);
    return NULL;
  }
  p->host[i] = '\0';
  DEBUG (fprintf (stderr, "URL host: %s\n", p->host);)
  if (!*url) {
    p->port = 0;                        /* no port spec supplied */
    return p;
  }
  i = 0;
  while (*url && i < BUF_SIZE)
    if (isdigit (*url))
      s[i++] = *url++;                  /* extract port number */
    else
      url++;
  if (i == 0) {
    free (p);
    return NULL;
  }
```

```
          s[i] = '\0';
          p->port = atoi (s);
          if (p->port < 0 || p->port > 65535) {
            free (p);
            return NULL;
          }
          return p;
        }

        free (p);
        return NULL;

#undef BUF_SIZE
} /* Cpl_ParseUrl() */
```

H.3.6 DTMF Tone Detection

iSrvDTMF.h

```
/*
** iSrvDTMF.h
*/

#ifndef _ISRVDTMF_H
#define _ISRVDTMF_H

#include "iSrvStates.h"

extern  int                     DtmfDetect (t_callState* state, char* buf, int len, int fmt);
extern  int                     DtmfInit (t_callState* state);

#endif
```

iSrvDTMF.c

```
/*
** iSrvDTMF -- DTMF tone detection/generation subroutines
**
*/

#include <stdio.h>
#include <stdlib.h>

#include "debug.h"
#include "iSrvDTMF.h"
#include "tabxlaw.h"
#include "/u/czahl/ibm.paul/audio.h.paul"

#define NCOEFF           16     /* number of frequencies to be analyzed       */
#define DTMF_TRESH    50000     /* above this is dtmf                         */
#define SILENCE_TRESH   100     /* below this is silence                      */
#define H2_TRESH      10000     /* 2nd harmonic                               */
#define AMP_BITS          9     /* bits per sample, reduced to avoid overflow */
#define LOGRP             0
#define HIGRP             1

#define MIN(a,b)                ((a)<(b)?(a):(b))
/*
        dtmf_state*             state = (dtmf_state*) NULL;
*/
typedef struct {
        int grp;        /* low/high group     */
        int k;          /* k                  */
        int k2;         /* k fuer 2. harmonic */
} dtmf_t;

/* For DTMF recognition:
 * 2 * cos(2 * PI * k / N) precalculated for all k
 */
```

```
static int cos2pik[NCOEFF] = {
        55812,  29528, 53603,  24032, 51193,  14443, 48590,   6517,
        38113, -21204, 33057, -32186, 25889, -45081, 18332, -55279
};

static dtmf_t dtmf_tones[8] = {
        { LOGRP,  0,  1 }, /*  697 Hz */
        { LOGRP,  2,  3 }, /*  770 Hz */
        { LOGRP,  4,  5 }, /*  852 Hz */
        { LOGRP,  6,  7 }, /*  941 Hz */
        { HIGRP,  8,  9 }, /* 1209 Hz */
        { HIGRP, 10, 11 }, /* 1336 Hz */
        { HIGRP, 12, 13 }, /* 1477 Hz */
        { HIGRP, 14, 15 }  /* 1633 Hz */
};

static char dtmf_matrix[4][4] = {
        {'1', '2', '3', 'A'},
        {'4', '5', '6', 'B'},
        {'7', '8', '9', 'C'},
        {'*', '0', '#', 'D'}
};


dtmf_state* isdn_audio_dtmf_init ()
{
        dtmf_state              *s;

        s = (dtmf_state*) malloc (sizeof (dtmf_state));
        if (s) {
          s->idx   = 0;
          s->last  = ' ';
          s->keySeqLen = 0;
        }
        return s;

} /* isdn_audio_dtmf_init() */



void isdn_audio_eval_dtmf (dtmf_state* s, int *result)
{
        int                     silence;
        int                     i;
        int                     di;
        int                     ch;
        unsigned long           flags;
        int                     grp[2];
        char                    what;
        char                    *p;

        grp[LOGRP] = grp[HIGRP] = -2;
        silence = 0;
        for(i = 0; i < 8; i++) {
          if ((result[dtmf_tones[i].k] > DTMF_TRESH) &&
              (result[dtmf_tones[i].k2] < H2_TRESH)    )
            grp[dtmf_tones[i].grp] = (grp[dtmf_tones[i].grp] == -2)?i:-1;
          else
            if ((result[dtmf_tones[i].k] < SILENCE_TRESH) &&
                (result[dtmf_tones[i].k2] < SILENCE_TRESH)  )
              silence++;
        }
        if(silence == 8)
          what = ' ';
        else {
          if((grp[LOGRP] >= 0) && (grp[HIGRP] >= 0)) {
            what = dtmf_matrix[grp[LOGRP]][grp[HIGRP] - 4];
            if(s->last != ' ' && s->last != '.')
              s->last = what;  /* min. 1 non-DTMF between DTMF */
          } else
              what = '.';
        }
        if ((what != s->last) && (what != ' ') && (what != '.')) {
```

```
          DEBUG (fprintf (stderr, "DTMF=%c\n", what);)
          if (s->keySeqLen +1 < sizeof (s->keySeq))
            s->keySeq[s->keySeqLen++] = what;
        }
        s->last = what;

} /* isdn_audio_eval_dtmf() */


/*
 * Goertzel algorithm.
 * See http://ptolemy.eecs.berkeley.edu/ pino/Ptolemy/papers/96/dtmf_ict/
 * for more info.
 * Result is stored into an sk_buff and queued up for later
 * evaluation.
 */
void isdn_audio_goertzel (dtmf_state *dtmf_state)
{
        int                     sk, sk1, sk2;
        int                     k, n;
        struct sk_buff          *skb;
        int                     result[NCOEFF];
        int                     *sample = dtmf_state->buf;

        for (k = 0; k < NCOEFF; k++) {
          sk = sk1 = sk2 = 0;
          for (n = 0; n < DTMF_NPOINTS; n++) {
            sk = sample[n] + ((cos2pik[k] * sk1) >> 15) - sk2;
            sk2 = sk1;
            sk1 = sk;
          }
          result[k] =
            ((sk * sk) >> AMP_BITS) -
            ((((cos2pik[k] * sk) >> 15) * sk2) >> AMP_BITS) +
            ((sk2 * sk2) >> AMP_BITS);
        }
        isdn_audio_eval_dtmf (dtmf_state, result);

}



/*
 * Decode DTMF tones, queue result in separate sk_buf for
 * later examination.
 * Parameters:
 *   s    = pointer to state-struct.
 *   buf  = input audio data
 *   len  = size of audio data.
 *   fmt  = audio data format
 */
void isdn_audio_calc_dtmf (dtmf_state *s, unsigned char *buf, int len, int fmt)
{
        int i;
        int c;

        while (len) {
          c = MIN(len, (DTMF_NPOINTS - s->idx));
          if (c <= 0)
            break;
          for (i = 0; i < c; i++) {
            if (fmt)
              s->buf[s->idx++] = ALAW_TO_LIN(*buf++) >> (15 - AMP_BITS);
            else
              s->buf[s->idx++] = ULAW_TO_LIN(*buf++) >> (15 - AMP_BITS);
          }
          if (s->idx == DTMF_NPOINTS) {
            isdn_audio_goertzel (s);
            s->idx = 0;
          }
          len -= c;
        }
```

```
} /* isdn_audio_calc_dtmf() */

/*
** Exported stuff:
*/

int DtmfDetect (t_callState *state, char* buf, int len, int fmt)
/*
** Check audio buffer for DTMF tones, return number of detected tones
*/
{

        state->dtmf_state->keySeqLen = 0;
        isdn_audio_calc_dtmf (state->dtmf_state, buf, len, fmt);
        state->dtmf_state->keySeq[state->dtmf_state->keySeqLen] = '\0';

        return state->dtmf_state->keySeqLen;

} /* DtmfDetect() */



int DtmfInit (t_callState *state)
{
        state->dtmf_state = isdn_audio_dtmf_init ();
        return 0;

} /* DtmfInit() */
```

H.3.7 Call State Handling

iSrvStates.h

```
/*
** iSrvStates.h
*/

#ifndef _STATES_H
#define _STATES_H

#include <time.h>

#include "iSrvCpl.h"

#define MAX_CALL_STATES         1000
        int                     execPid;        /* process id for executing system command via CPL exec command */
        int                     execStdOut;     /* stdout from exec process */
        int                     execBytes;      /* bytes read from exec pipe */

#define SRV_S_CALLED            0
#define SRV_S_CONNECTING        1
#define SRV_S_CONNECTED         2
#define SRV_S_DISCONNECTED      3
#define SRV_S_RELEASING         4
#define SRV_S_RELEASED          5
#define SRV_S_CALLING           6

#define AUBUF_SIZE                10000

typedef struct {
        char                    magic[4];
        int                     off;
        int                     len;
        int                     fmt;
        int                     rate;
        int                     channels;
} t_auHeader;

typedef struct {
        int                     Play;
```

```
        unsigned char           buf[AUBUF_SIZE];
        t_auHeader              header;
        int                     channels;
        int                     BPS;
        int                     rate;
        int                     format;
        int                     Vol;
        int                     loop;
        int                     flags;
        int                     f;
} t_auplay;


#define ms              *8              /* samples/ms */
#define DTMF_NPOINTS    40 ms           /* Number of samples for DTMF recognition */

typedef struct dtmf_state {
          char                  keySeq[100];
          int                   keySeqLen;
          char                  last;
          int                   idx;
          int                   buf[DTMF_NPOINTS];
} dtmf_state;

typedef struct {
          int                   used;           /* 0=slot free, 1=slot in use */
          int                   state;          /* current call state */
          int                   callId;         /* call id supplied by the Q.931 library */
          int                   bChannelFd;     /* filedesc. for B channel */
          int                   bChannelBytes;  /* bytes read from B channel */
          time_t                startTime;      /* arrival time of ISDN connect indication */
          char                  callingParty[80];
          char                  calledParty[80];
          /*
          ** structures for the iPhone service
          */
          int                   cConnection;    /* index into cConnections table */
          int                   clientAudioFd;  /* audio stream connect to client */
          unsigned short        clientAudioPort;
          /*
          ** structures for the iPlay service
          */
          char                  OGM_filename[80];
          int                   OGM_loop;
          /*
          ** structures for the iScript service
          */
          dtmf_state*           dtmf_state;     /* context structure for recognizing DTMF tones */
          t_auplay*             auplay;         /* context structure for playing audio files */
          int                   audioLog;       /* fd for audio logging (CPL audiorec command) */
          t_script*             script;
          void*                 fwdState;       /* ptr to companion state for call forwarding */
          /*
          ** Handler for misc. ISDN indications
          */
          void                  (*ChannelIndHandler) ();
          void                  (*ConnectIndHandler) ();
          void                  (*DisconnectIndHandler) ();
          void                  (*ReleaseIndHandler) ();
} t_callState;


extern  void                    CallStateDelete (t_callState *state);
extern  t_callState*            CallStateFind (int CallId);
extern  t_callState*            CallStateFindByConInd (int fd);
extern  t_callState*            CallStateFindByPid (int pid);
extern  int                     CallStateInit ();
extern  t_callState*            CallStateNew ();

#endif
```

iSrvStates.c

```
/*
** iSrvStates.c -- server state handling routines
**
** (c) 1996 F. Oertel
**
** 04.01.1997, FO; CallStateFindByPid() added
*/

#include <stdio.h>

#include "iSrv.h"
#include "iSrvStates.h"


static  t_callState             callStates[MAX_CALL_STATES];


void CallStateCleanUp (int conInd, void (*CleanUpFunc) ())
{
        int                     i;

        for (i=0; i < MAX_CALL_STATES; i ++)
          if (callStates[i].used)
            if (callStates[i].cConnection == conInd) {
              callStates[i].used = 0;
              CleanUpFunc (&callStates[i]);
            }

} /* CallStateCleanUp() */



void CallStateDelete (t_callState *state)
{
        state->used = 0;
} /* CallStateDelete() */



t_callState* CallStateFind (int callId)
{
        int                     i;

        for (i = 0; i < MAX_CALL_STATES; i++)
          if (callStates[i].used)
            if (callStates[i].callId == callId)
              break;
        if (callStates[i].callId != callId)
          return NULL;
        else
          return &callStates[i];

} /* CallStateFind() */



t_callState* CallStateFindByConInd (int ind)
/*
** Find a state using the clients connection index
*/
{
        int                     i;

        for (i = 0; i < MAX_CALL_STATES; i++)
          if (callStates[i].used)
            if (callStates[i].cConnection == ind)
              break;
        if (callStates[i].cConnection != ind)
          return NULL;
        else
          return &callStates[i];
```

```
} /* CallStateFindByConInd() */


t_callState* CallStateFindByPid (int pid)
{
        int                     i;

        for (i = 0; i < MAX_CALL_STATES; i++)
          if (callStates[i].used && callStates[i].script != NULL)
            if (callStates[i].script->execPid == pid)
              break;
        if (i < MAX_CALL_STATES)
          return &callStates[i];
        else
          return NULL;

} /* CallStateFindByPid() */


int CallStateInit ()
{
        int                     i;

        for (i = 0; i < MAX_CALL_STATES; i++)
          callStates[i].used = 0;
        return 0;

} /* CallStateInit() */


t_callState* CallStateNew ()
{
        int                     i;

        for (i = 0; i < MAX_CALL_STATES; i ++)
          if (!callStates[i].used)
            break;
        if (i == MAX_CALL_STATES)
          return NULL;

        ZERO (callStates[i]);
        callStates[i].used = 1;
        callStates[i].bChannelFd = -1;
        callStates[i].clientAudioFd = -1;
        callStates[i].script = NULL;

        return &callStates[i];

} /* CallStateNew() */
```

H.3.8 Configuration File Handling

`iSrvCfg.h`

```
/*
** iSrvCfg.h
*/
#ifndef _ISRVCFG_H
#define _ISRVCFG_H

typedef struct {
          char                  callingParty[20];
          char                  calledParty[20];
          char                  action[80];
        } t_serviceDef;

extern  char*                   FindServiceDefinition (char* callingParty, char* calledParty);
```

```
extern  int                     ReadCfgFile (char* fn);


#endif
```

iSrvCfg.c

```
/*
** iSrvCfg.c -- server configuration file handling routines
**
** (c) 1996 F. Oertel
**
** 31.12.1996, FO, config file syntax changed (slashes replaced by spaces)
*/


#include <stdio.h>
#include <stdlib.h>

#include "iSrvCfg.h"

static  t_serviceDef            callTable[100];
static  int                     callTabCount = 0;


/*
** internal funtions:
*/

char* getField (char* src, char* target)
{
        char*                   p = target;

        while (*src && isspace (*src))          /* skip trailing white spaces */
          src++;
        while (*src && !isspace (*src))         /* scan field */
          *target++ = *src++;
        *target = '\0';

        return (strlen (p) == 0 ? NULL : src);

} /* getField() */


int matches (char* value, char* pattern)
{

        while (*pattern && *value && *pattern != '*' && *pattern == *value) {
          value++;
          pattern++;
        }
        if (*pattern == '*')
          return 1;
        if (*pattern == *value)
          return 1;

        return 0;

} /* matches() */


/*
** exported functions:
*/


char* FindServiceDefinition (char* callingParty, char* calledParty)
{
        int                     i;


        for (i = 0; i < callTabCount; i++)
```

```
          if (matches (callingParty, callTable[i].callingParty))
            if (matches (calledParty, callTable[i].calledParty))
              return callTable[i].action;

        return NULL;

} /* FindServiceDefinition() */


int ReadCfgFile (char* fn)
/*
** read the isdn servers configuration file into memory
*/
{
        FILE                    *f;
        int                     i;
        char                    line[512];
        int                     lineCount = 0;
        int                     n = 0;
        char*                   p;
        int                     x;

        if ((f = fopen (fn, "r")) == NULL)
          return -1;

        while (fgets (line, sizeof(line), f)) {
          line[strlen (line) -1] = '\0';
          lineCount++;
          i = 0;
          while (isspace (line[i]) && line[i])
            i++;
          if (!line[i])                         /* skip blank line */
            continue;
          if (line[i] == '#')
            continue;                           /* skip comment line */
          p = line;
          if (!(p = getField (p, callTable[n].callingParty))) {
            fprintf (stderr, "Line %03d: Calling party number is blank or missing!\n", lineCount);
            return -1;
          }
          if (!(p = getField (p, callTable[n].calledParty))) {
            fprintf (stderr, "Line %03d: Called party number is blank or missing!\n", lineCount);
            return -1;
          }
          while (*p && isspace (*p))            /* skip trailing spaces */
            p++;
          x = 0;
          while (*p && *p != '#')
            callTable[n].action[x++] = *p++;    /* scan until end of line or comment introducer */
          callTable[n].action[x] = '\0';
          if (strlen (callTable[n].action) == 0) {
            fprintf (stderr, "Line %03d: Action for incoming calls is blank or unspecified!\n", lineCount);
            return -1;
          }
          n++;
        } /* while */

        if (n == 0) {                           /* no entries in cfg file! */
          fprintf (stderr, "%s contains no call handling rules!\n", fn);
          return -1;
        }

        callTabCount = n;
        return 0;

} /* ReadCfgFile() */
```

Literaturverzeichnis

[AESP96] ARSLAN, Guener ; EVANS, Brian L. ; SAKARYA, Ayhan F. ; PINO, Jose L.: Performance Evaluation and Real-Time Implementation of Subspace Adaptive, and DFT Algorithm for Multi-Tone Detection / Yildiz Technical University, Instanbul, Turkey, University of California, Berkeley. 1996. – Forschungsbericht

[ASU88] AHO, Alfred v. ; SETHI, Ravi ; ULLMANN, Jeffrey D.: *Compilerbau*. Reading, Massachusetts : Addison-Wesley, 1988. – ISBN 3–89319–150–X

[BLMM94] BERNERS-LEE, T. ; MASINTER, L. ; MCCAHILL, M.: Uniform Resource Locators (URL) / Internet Engineering Task Force. 1994 (RFC 1738). – Request for Comments (Proposed Standard) 25 Seiten

[CAP94] CAPI Working-Group: *Common ISDN Application Programming Interface (CAPI) Version 2.0*. 2 1994

[Cro82] CROCKER, David H.: STANDARD FOR THE FORMAT OF ARPA INTERNET TEXT MESSAGES / Internet Engineering Task Force. 1982 (RFC 822). – Request for Comments (Standards Track) 47 Seiten

[Cro96] CRONIN, Paul: An Introduction to TSAPI and Network Telephony / Novell Inc. IEEE Communications Magazine, 4 1996. – Forschungsbericht

[Dee89] DEERING, S.: Host Extensions for IP Multicasting / Internet Engineering Task Force. 1989 (RFC 1112). – Request for Comments 17 Seiten

[Han96] HANDLEY, M.: Session Announcement Protocol SAP / Internet Engineering Task Force. 1996. – Internet-Draft work-in-progress 14 Seiten

[HJ96] HANDLEY, M. ; JACOBSON, V.: Session Description Protocol SDP / Internet Engineering Task Force. 1996. – Internet-Draft work-in-progress 28 Seiten

[HSS96] HANDLEY, M. ; SCHULZRINNE, H. ; SCHOOLER, E.: Session Invitation Protocol SIP / Internet Engineering Task Force. 1996. – Internet-Draft work-in-progress 30 Seiten

[ITU89a] ITU: Multifrequency Push-Button Signal Reception / INTERNATIONAL TELECOMMUNICATION UNION. 1989 (Q.24). – ITU-T Recommendation

[ITU89b] ITU: Technical Features of Push-Button Telephone Sets / INTERNATIONAL TELECOMMUNICATION UNION. 1989 (Q.23). – ITU-T Recommendation

[ITU90] ITU: 40, 32, 24, 16 kbit/s ADAPTIVE DIFFERENTIAL PULSE CODE MODULATION / INTERNATIONAL TELECOMMUNICATION UNION. 1990 (G.726). – ITU-T Recommendation

[ITU92] ITU: CODING OF SPEECH AT 16 kbit/s USING LOW-DELAY CODE EXCITED LINEAR PREDICTION / INTERNATIONAL TELECOMMUNICATION UNION. 1992 (G.728). – ITU-T Recommendation

[ITU93a] ITU: Arrangement of Digits, Letters and Symbols on Telephones and Other Devices that can be used to gain access to a Telephone Network / INTERNATIONAL TELECOMMUNICATION UNION. 1993 (E.161). – ITU-T Recommendation

[ITU93b] ITU: DIGITAL SBSCRIBER SIGNALING SYSTEM No. 1 (DSS 1) ISDN USER-NETWORK INTERFACE LAYER 3 SPECIFICATION FOR BASIC CALL CONTROL / INTERNATIONAL TELECOMMUNICATION UNION. 1993 (Q.931). – ITU-T Recommendation

[ITU93c] ITU: Progamming Languages – CCITT specification and description language SDL / INTERNATIONAL TELECOMMUNICATION UNION. 1993 (Z.100). – ITU-T Recommendation 256 Seiten

[Mes96] MESSERSCHMITT, David G.: The Future of Computer Telecommunications Integration. IEEE Communications Magazine, 4 1996. – Forschungsbericht

[Mil92] MILLS, David L.: Network Time Protocol (Version 3) Specification, Implementation and Analysis / Internet Engineering Task Force. 1992 (RFC 1305). – Request for Comments

[Moc87] MOCKAPETRIS, P.: DOMAIN NAMES - CONCEPTS AND FACILITIES / Internet Engineering Task Force. 1987 (RFC 1034). – Request for Comments (Standards Track) 55 Seiten

[Ous94] OUSTERHOUT, John K.: *Tcl and the Tk Toolkit*. Reading, Massachusetts : Addison-Wesley, 1994. – ISBN 0–201–63337–X

[SCFJ96] SCHULZRINNE, H. ; CASNER, S. ; FREDERICK, R. ; JACOBSON, V.: RTP: A Transport Protocol for Real-Time Applications / Internet Engineering Task Force. 1996 (RFC 1889). – Request for Comments (Standards Track) 75 Seiten

[Str96] STRATHMEYER, Carl R.: An Introduction To Computer Telephony. IEEE Communications Magazine, 5 1996. – Forschungsbericht

[Uhl93] UHLER, Stephen A.: PhoneStation, Moving the Telephone into the Virtual Desktop. **In:** *1993 Winter USENIX - January 25-29, 1993 San Diego, CA*. San Diego, California, Januar 1993, S. 131–140

[Zah97] ZAHL, Christian: *Aufbau und Steuerung von Internet-Multimedia-Konferenzen ueber Verbindungen niedriger Bandbreite*, TU-Berlin, GMD Fokus, Diplomarbeit, Februar 1997

[Zig96] ZIGMOND, D.: Uniform Resource Locators for Television and Telephony / Internet Engineering Task Force. 1996. – Internet-Draft work-in-progress 3 Seiten