Protocol Design for Congestion Management in Narrowband Integrated Networks

by

Sona S. Kapadia

B.E., University of Bombay, Bombay, 1992

Submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science.

____
Professor in Charge

____

____
Committee Members

____
Date thesis accepted

Abstract

Narrowband integrated voice/data networks require that congestion control and bandwidth management techniques be implemented in order to provide the desired quality of service. An adaptive voice/data switch has been developed to demonstrate the application of the Sinusoidal Transform Coder (STC), a low bit-rate digital speech coder, to the implementation of integrated networks.

In this work, an integrated network management and control protocol is designed and implemented. The protocol exploits the unique capabilities of the STC for congestion control in a narrowband integrated network environment. A software testbed is constructed to evaluate the speech coding technique and to verify the protocol functionality. The results of informal listening tests on several simulated conversations suggest that this coder is well-suited for bandwidth reallocation based on dynamic rate adjustment. The performance of the adaptive voice/data switch congestion control mechanism is studied using simulation.

Acknowledgements

I would like to thank Dr. Joseph Evans, my advisor, for his help and advice in the course of my graduate studies at KU. Working on this project under his guidance has been an extremely rewarding experience.

I would like to thank Dr. Glenn Prescott and Dr. Swapan Chakrabarti for serving on my committee.

I want to thank Meetul Parikh for lending an ear in times of need, and my roommate Preeti Balse for putting up with my worse side these past few months! I would also like to thank my colleague Najeeb Ansari for the valuable discussions we have had.

Finally, I would like to thank my parents for encouraging me to further my education. I would especially like to thank my mother, Uma Kapadia, without whose support none of this would have been possible.

Contents

1 Introduction **1**

2 Background **3**

2.1 Speech Coding 3

2.1.1 Applications 3

2.1.2 Overview of Coding Techniques 4

2.1.3 Limitations of Standard Coder Technology 7

2.2 Sinusoidal Transform Coder 8

2.2.1 Parameter Space Transformations 9

2.3 Adaptive Voice/Data Switch 10

2.3.1 Network Configuration 10

2.3.2 Switch Architecture 10

3 Integrated Network Management Protocol **14**

3.1 Network Services 14

3.2 Signaling Protocol 15

3.2.1 Protocol Functions 16

3.3 Protocol Description 22

3.3.1 Control Messages 22

3.3.2 Call Setup Example 25

4 Protocol Simulation **29**

4.1 Switch Modeling 29
4.2 Simulation Goals 30
4.3 Simulation Strategy 31
4.4 Test Cases 32
4.5 Results 36

5 Congestion Control **39**
5.1 Overview of Bandwidth Allocation Algorithms for Circuit-Switched Environments 39
5.2 Congestion Control in Adaptive Voice/Data Networks 40
5.3 Adaptive Switch Performance Evaluation 41
5.3.1 Modeling 41
5.3.2 Performance Measures 43
5.3.3 Simulation Results 44
5.3.4 Conclusions 50

6 Conclusion and Future Work **53**

A Control Messages **55**

B Parser Source Code **61**
B.1 C Programs 62
B.2 Header Files 94
B.3 Sample Configuration File 99

C Simulation Source Code **104**
C.1 C Programs 105
C.2 Header Files 141

List of Tables

3.1 Control Message types and functions 25

4.1 Simulation results 38

5.1 Simulation parameters 45

A.1 Voice rates 59

A.2 Data rates 59

A.3 Maximum connection rates 59

A.4 Control Message IDs and parameters 60

List of Figures

2-1 Vocoder speech production model 6

2-2 Adaptive voice/data switch network configuration 11

2-3 Integrated voice/data switch architecture 12

3-1 Switch Protocol Suite and functionality. Hardware components associated with the protocol layers are listed on the left. 16

3-2 The RS-232C interface circuits. The pin numbers are given in parenthesis 17

3-3 Asynchronous transmission format 17

3-4 Multiplexing and demultiplexing of 9.6 kb/s voice and 2.4 kb/s data on a single connection 18

3-5 Control of call congestion using rate transformation 20

3-6 Multiplexing of multirate circuits on the modem link 22

3-7 Time-slot assignment on the output link 23

3-8 Control message flow diagram for a typical call 26

3-9 Control message flow diagram for a call using rate transformation 27

3-10 Switch-to-switch handshaking for connection setup 28

4-1 Simplified model of adaptive voice/data switch 30

4-2 Block diagram of simulation strategy 33

4-3 Simulation input speech file coded at 19.2 kbps 37

4-4 Rate transition points with no silence detection 37

4-5 Rate transition points with silence detection 37

5-1 System model: voice and data sources accessing a concentrator . . 42
5-2 Blocking probabilities for voice and data calls, C=8 46
5-3 Fractional Link Utilization, C=8 46
5-4 Fairness index for voice calls, C=8 47
5-5 Comparison of Average QOS on voice calls with different link capacities 48
5-6 Comparison of Link Utilization with different link capacities . . . 48
5-7 Blocking probabilities for voice and data calls, C=9 49
5-8 Voice call blocking probabilities with thresholding 50
5-9 Data call blocking probabilities with thresholding 51
5-10 Comparison of Average QOS for different thresholds 51
5-11 Comparison of link utilization for different thresholds 52

A-1 CONNECTION REQUEST 56
A-2 CONNECTION INDICATION 56
A-3 CONNECTION RESPONSE 56
A-4 MODIFY REQUEST/INDICATION 57
A-5 MODIFY RESPONSE 57
A-6 TRANSFORM 57
A-7 OPEN CHANNEL 57
A-8 MODIFY SOURCE 58
A-9 ACKNOWLEDGEMENT 58

Chapter 1

Introduction

Traffic integration in communication networks is currently the focus of research activity in the telecommunications and computer communications communities. The last decade has witnessed the evolution of standards such as ISDN (Integrated Services Digital Network) for end-to-end digital transport of voice and data, and B-ISDN (Broadband ISDN) for support of high-bandwidth services over high-speed links. While these technologies are based on increasing bandwidth availability due to the fiber revolution [25], parallel developments in the fields of cellular and wireless communications must contend with rapidly depleting link capacities due to spectrum congestion. These developments underline the need for the implementation of integrated networks in a narrowband environment, especially for services such as digital mobile radio and personal communications. Narrowband integrated networks gain importance also as an interim or alternative low-cost solution for the integration of voice and data over existing telecommunication networks.

Limitations in bandwidth availability have also fueled interest in voice compression. Research in speech coding has demonstrated that high quality voice can be achieved at much lower bit rates than was deemed possible a few years ago [7]. Hence low bit-rate coding techniques are finding application in secure low-

speed communications, voice mail systems, multimedia services, and in ISDN. The developments in speech coder technology are closely linked to the design of integrated voice/data networks.

The work presented here is part of the design of an adaptive voice/data switch, which demonstrates the application of the Sinusoidal Transform Coder (STC) to the implementation of integrated networks. The Sinusoidal Transform Coder has been developed at MIT Lincoln Laboratory, with Air Force support, and has been shown to possess flexibility not present in other coders that can be used to enhance overall system performance [9]. The adaptive switches are connected by modem, and support up to four voice or data connections. The switches allocate bandwidth based on the number of connections and change the bit-rate of the coded speech accordingly. The protocols and control software required to exploit the unique capabilities of the coder for bandwidth reallocation and congestion control in a multirate integrated applications environment are described in this work. The performance of the switch under various bandwidth allocation schemes is predicted using the BONeS simulation tool.

Today, several vendors are offering products such as fractional-T1 multiplexers that integrate voice and data traffic over a 64 kb/s leased line; however, the adaptive switch possesses congestion control capabilities not present in these units, especially in a multi-hop network environment. An example application environment for an adaptive voice/data network based on the techniques described above is the Air Force command and control network [9]. Such a network has limited spectrum allocations, and hence limited transmission capacity, so spectrum congestion is possible, particularly during times of crisis when communication facilities are most needed. The rate transformation capabilities of the adaptive voice/data switch can be used to dynamically re-allocate spectrum resources to allow more efficient and flexible resource usage.

Chapter 2

Background

2.1 Speech Coding

A large variety of algorithms exist for digital coding of speech signals. The appropriate coding technique for a particular application depends on system requirements such as bit rate, coding delay, complexity, speech quality, robustness to errors, and cost. Recent developments in speech coding focus on medium and low bit rate speech coding techniques. The following sections outline the different types of coding schemes and their applications.

2.1.1 Applications

- *Narrowband communications* - Low bit rate speech coding can be used to provide efficient resource usage on low bandwidth links such as cellular radio or satellite links.

- *Sub-rate Multiplexers* - The use of low bit rate speech coders enables the multiplexing of several voice connections over a single channel. For example, one 64 kb/s PCM channel can be replaced by eight channels of speech coded at 8 kb/s [21]. The extra bandwidth can also be used to accommodate data

channels.

- *Voice Storage* - Voice compression can be used for memory efficient storage of voice signals. Some example applications are voice mail systems, telephone answering machines, and spoken help messages on personal computers.

- *Encryption* - With the use of speech coding, encryption of voice messages can be easily accomplished. This is useful for secure military and business communications.

2.1.2 Overview of Coding Techniques

The general classes of speech coding techniques are described below. A large number of variations of these basic techniques exist; each using a different strategy for removal of redundancy from the speech signal and allocation of bits available for coding. In general, as the bit rate is reduced, speech quality is degraded. Higher quality at a given bit rate can be achieved by using a more complex coding algorithm; the price paid for this is increased cost, and possibly, coding delay. Speech coding schemes must thus trade-off these four parameters - bit rate, quality, complexity and delay [12].

Waveform Coders

Waveform coders attempt to approximate the shape of the input speech waveform at the coder output. These coders usually operate at medium to high bit rates. Since they attempt to reproduce the input waveform without regard to its properties, these coders are not speech specific, and are able to successfully code non-speech signals, background noise and multiple speakers [21].

Pulse Code Modulation (PCM) is a widely known waveform coding technique currently in use for voice transmission over the Public Switched Telephone Network (PSTN). The analog speech waveform is sampled at a rate of 8 kHz and

quantized using 8 bits per sample to yield a 64 kb/s digital waveform at the coder output. The decoder performs the reverse operation.

The number of bits required to quantize each sample can be reduced by encoding the difference between successive samples rather than the samples themselves; this technique is called Differential Pulse Code Modulation (DPCM). Improvement in speech quality can be achieved by using nonlinear quantization to accurately represent low amplitudes in the speech signal at some cost to the large amplitudes, which can tolerate larger degrees of quantization error. This technique, called Adaptive Differential Pulse Code Modulation (ADPCM), has been standardized by CCITT for use in telecommunications applications.

Vocoders

Parametric coders or vocoders attempt to describe the speech signal in terms of parameters of a speech production model. Vocoders can operate at much lower bit rates than waveform coders, but they are computationally demanding and generally output speech of a lower quality. The speech output is intelligible, but synthetic in nature, hence it may not be possible to identify the speaker. Vocoders traditionally perform badly in the presence of background noise and with multiple speakers and non-speech signals.

The speech production model used in a typical vocoder is shown in Figure 2-1. The model is based on an excitation signal feeding a linear filter, which simulates the behavior of the human vocal tract [21]. A set of parameters is derived at the coder, encoded, and transmitted to the decoder. The decoder uses these parameters to control speech production. For voiced speech or speech produced by the vibration of the vocal chords resulting in quasi-periodic pulses of air which are acoustically filtered by the vocal tract [22], the excitation signal is modeled by a periodic impulse train at the pitch frequency. For unvoiced speech, the excitation signal is the output of a pseudorandom noise generator. The gain

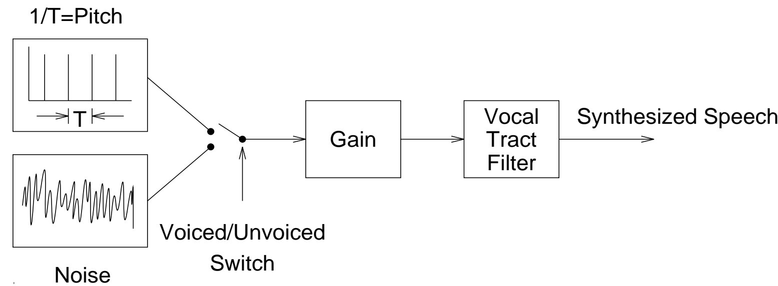

Figure 2-1: Vocoder speech production model

controls the level of excitation, and the shaping of the excitation signal by the vocal tract is modelled by a time-varying linear filter.

Linear Predictive Coding (LPC) vocoders represent the vocal tract filter as a linear combination of previous speech samples. LPC is the basis for the U.S. Government Standard LPC-10e vocoder, and is commonly used for secure military communications at 2.4 kb/s. Sinusoidal Transform Coding (STC) is another vocoding technique which forms the basis for this work. This coder is described in detail in a later section.

Hybrid Coders

Hybrid coders, or "soft" vocoders [12] combine features from both waveform coders and vocoders to provide communication quality speech at medium bit rates. Hybrid coders use a speech production model similar to the one used by vocoders, but the excitation signal is carefully optimized to yield high-quality speech.

Multipulse excited LPC (MPLPC) is a hybrid coding technique in which the excitation signal is a series of non-uniformly spaced pulses with different amplitudes [3]. The same excitation model is used for both voiced and unvoiced speech segments. A Multipulse speech codec operating at 9.6 kb/s is used in a world-wide aeronautical satellite telecommunications service known as Skyphone [3].

Code Excited Linear Prediction (CELP) uses a codebook of excitation sequences and selects the optimum excitation by minimizing the error between the

input speech signal and the synthesized speech. The address of the code is transmitted to the receiver, rather than the code itself, in order to reduce the number of bits required to quantize the excitation signal. This technique is known as vector quantization. CELP has been standardized at 4.8 kb/s for military applications and at 16 kb/s by CCITT for video conferencing support [21].

2.1.3 Limitations of Standard Coder Technology

It is evident from the above discussion that an extremely large variety of algorithms have been developed for speech coding at medium and low bit rates (approximately 2.4 kb/s to 32 kb/s). These coders provide efficient and good quality speech output, but suffer from the inherent limitation of poor interoperability with coders at different rates, even if the coders involved are based on the same model. This has been referred to as the 'Tower of Babel' problem.

Standard speech coders also do not adequately support the source- independent rate translation capability, which limits their flexibility in an integrated network environment. Source-independent rate translation involves the capability to transform the bit rate of the coded speech stream at any intermediate node in the network, without interaction with the source. One approach is to use embedded coding. In embedded coders, the parameters of the lower rate coder are embedded in the parameters of the higher rate coder [5]. To reduce the bit rate, the higher rate parameters are stripped off. However, this technique implies that the higher and lower rate coders must be co-designed, which limits the speech quality achievable at any one bit rate. Another approach is to revert to a PCM representation of the speech signal and subsequently code this signal at the desired rate. Tandeming of coders in this manner may result in unacceptable degradation in speech quality due to the cumulative effect of coding distortion and quantization noise introduced at each stage of compression.

The Sinusoidal Transform Coder (STC) allows interoperability between coders

operating at different rates and source-independent rate translation using simple parameter manipulation techniques. STC-based schemes can also be used in a narrowband conferencing environment involving multiple speakers, an area in which other vocoders traditionally perform very poorly. The speech quality obtained using STC methods is significantly higher than with any of the techniques described above.

2.2 Sinusoidal Transform Coder

The Sinusoidal Transform Coder [14, 15, 16, 18, 17] is a vocoding technique which represents the excitation signal as a sum of sine waves of arbitrary amplitudes, frequencies and phases [19]. The sine wave components are harmonically related when the excitation is voiced, and aharmonic when it is unvoiced. Feeding this sine wave excitation to the vocal tract filter results in a sinusoidal representation for the speech signal. The STC is based on a frequency domain analysis/synthesis model [8]. The spectral envelope is determined using the FFT and is used to generate a set of cepstral coefficients which are coded and transmitted along with the excitation parameters.

STC is a multirate coder - it can operate at several discrete rates from 8 kb/s to 2.4 kb/s. Optimization at higher rates, up to 19.2 kb/s, is currently being investigated. The speech quality degrades somewhat linearly with bit rate. Multirate coders are distinct from variable rate coders in that the instantaneous coder bit rate is fixed, and is based on the requirements of the transmission channel rather than on the requirements of the speech signal. This type of coder is well suited for the implementation of integrated networks with fixed capacity circuits.

2.2.1 Parameter Space Transformations

Source coders (or vocoders) such as the STC generate a parametric description of the speech signal. The STC coding technique is extremely robust to manipulation in the parameter space [4]. This characteristic greatly enhances its flexibility.

Transformations in the parameter space can be used to reduce the bit rate of the coded speech. Bit rate reduction may be achieved by 1) parameter de-scoping - coarsening the resolution allowed to the quantized parameters; and 2) frame dilation - coarsening the time resolution between speech samples [8]. Rate translation thus involves the use of signal processing algorithms to perform transformations from one parameter set to another. Generation of a waveform representation of the signal is not required, and the coder can be optimized separately at each bit rate. The transformed speech is comparable in intelligibility and quality to speech directly coded in the fundamental coder technology at the final rate. Thus, the STC can be used for multi-stage compression, without interaction with the source.

Source-independent rate transformations can be used to reduce the bit rate of the incoming speech at a network node during periods of congestion, to momentarily increase channel capacity for the purpose of transmitting signaling messages or adding error protection, or to accommodate data traffic [1]. Another application is narrowband voice conferencing. Low bit rate coders perform poorly with multiple speakers, hence conferencing in a narrowband environment has not been considered practical. This problem can be avoided by running the coders at higher rates and transforming to lower rates when necessary [8].

Since the STC parameters exhibit good time stability, techniques such as parameter freezing and parameter interpolation can be used to reconstruct lost frames in harsh data loss environments [8]. Experiments with these schemes suggest that a high channel error rate can be tolerated.

The STC parameter space transformation technique is used in this work to implement novel congestion control schemes and dynamic re-allocation of resources

in a narrowband integrated voice/data network.

2.3 Adaptive Voice/Data Switch

Narrowband integrated voice/data networks require that congestion control and bandwidth management techniques be implemented in order to provide the desired quality of service [1]. The adaptive voice/data switch exploits the capabilities of the Sinusoidal Transform Coder to address these requirements. The switch allocates bandwidth based on its load, and performs adaptive rate transformations on the coded speech to trade off quality versus channel availability. The unique features of the speech coding technique and the control protocols allow for efficient and flexible resource usage, particularly under conditions of high load.

2.3.1 Network Configuration

The integrated voice/data network configuration is shown in Figure 2-2. The switches can be connected by a point-to-point link, via modem or by a wireless link. The current implementation supports the modem configuration. Several digital terminals are hooked up to the switch. These terminals are envisioned as integrated voice/data units which incorporate the speech coder. The switch access link and the long distance link are both narrowband. The terminals compete for bandwidth on the outgoing link; the switch allocates this bandwidth dynamically depending on a number of criteria such as load and negotiated quality of service.

2.3.2 Switch Architecture

The adaptive voice/data switch is a bus-based switch with four input/output ports; a port can be connected to an integrated voice/data terminal or to a modem. The switch supports connections between terminals as well as connections to remote switches via the telephone network. The switch architecture is shown

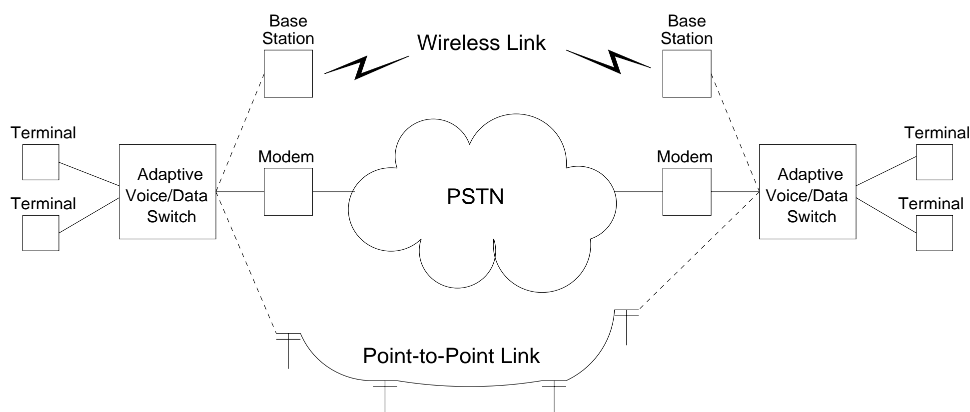

Figure 2-2: Adaptive voice/data switch network configuration

in Figure 2-3, and the functions of the various switch components are described below.

Universal Synchronous Asynchronous Receivers/Transmitters (USARTs)

The USARTs interface to external devices which include voice/data terminals and modems. The data rate for communication is selectable from a set of discrete rates in the range of 1.2 kb/s to 19.2 kb/s. The USARTs receive serial streams from external devices and output byte-wide data on the input bus in response to polling from the bus control logic. The USARTs read byte-wide data from the output bus on receiving a 'data ready' indication from the bus control logic and transmit a serial stream to the external devices. Communication in synchronous or asynchronous mode is possible.

Switch Bus Controllers (SBCs)

The Switch Bus Controllers buffer the incoming voice/data bytes and switch the data to the destination port through the output bus. The SBCs are also responsible for demultiplexing the speech, data and control information. Speech bytes are passed to the associated DSP through a serial interface, received from the DSP after processing, and held for transmission on the output bus. Control bytes are

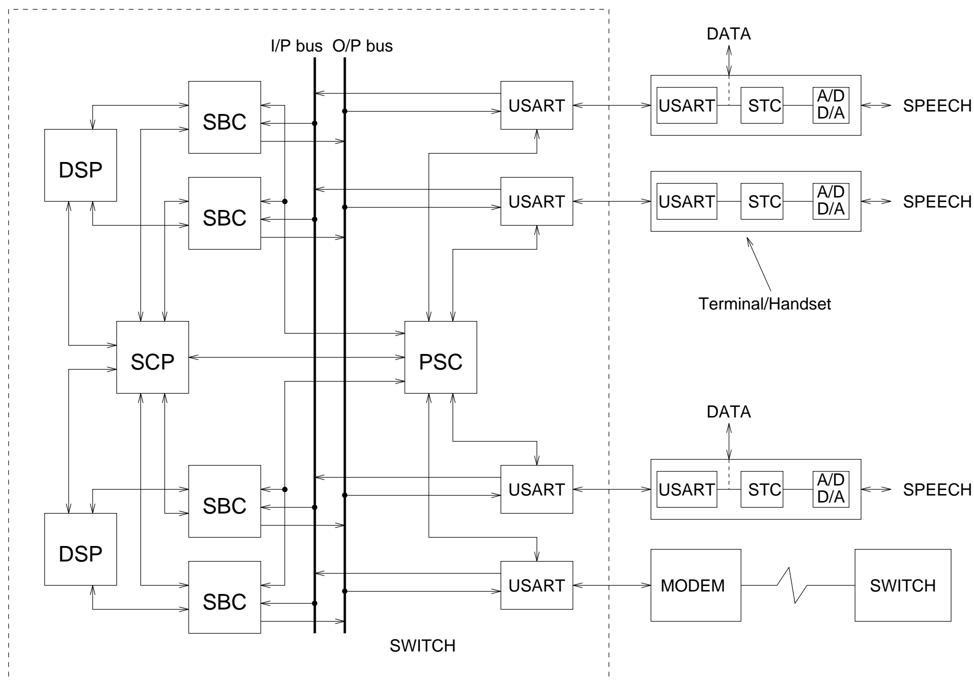

Figure 2-3: Integrated voice/data switch architecture

passed to the SCP on an interrupt basis. The SBCs insert idle bytes in the output stream if data is not available to fill the output time slot.

Polling Schedule Controller (PSC)

The Polling Schedule Controller schedules and controls access to the input and output buses. Signals sent to the SBCs and USARTs indicate which devices should read and write the bus at any instant. Bus scheduling is accomplished through a time-slotted polling scheme, where each port is serviced in a round-robin fashion. The input and output buses operate at high speed relative to the access link.

Digital Signal Processors (DSPs)

The Digital Signal Processors receive speech bytes from the SBCs and perform rate transformations on the coded speech if necessary. Each DSP is capable of handling two voice streams in real-time, hence two DSP chips are required to

support four ports.

Switch Control Processor (SCP)

The Switch Control Processor controls the switch initialization process, connection setup, modification of connection parameters while a call is in progress, and connection tear-down. The SCP controls scheduling and routing via control registers in the the Polling Scheduler and System Bus Controllers, and sets the rate translation parameters on the DSPs.

Chapter 3

Integrated Network Management Protocol

A management and control protocol for an integrated voice/data network has been developed. The network architecture was described in the previous chapter; the services to be offered by the network will now be defined. The signaling protocol which defines the procedures needed to support these services is described in this chapter. In addition, a technique for multiplexing speech, data and control information on the same media is presented, and the format of the control messages is specified.

3.1 Network Services

The Adaptive Voice/Data Switch essentially functions in two modes: as an integrated switch (for local conversations and data exchanges) or as an integrated multiplexer (for transmission of compressed speech and data streams over telephone lines). In addition to local and remote communications connectivity, the protocol supports several unique features which are outlined below.

Multimedia Calls

Users can set up a call by dialing a number, and then choose to speak or send data, or both, over the same connection. Thus, applications such as voice-annotated text message exchange and transmission of graphical information to be displayed on a screen during a voice conversation, are possible.

Service Request Modification

The service request parameters can be modified at any time during the call. The user can switch between voice and data traffic and change the rate dynamically. The user specifies a *peak rate* at call setup time. Subsequent requests for call modification are granted if the aggregated voice and data rate does not exceed the peak rate, and if the necessary resources are available.

Channel availability versus speech quality

Users can trade speech quality for increased voice channel capacity or higher data throughput on the modem link. This capability may be used to increase channel availability when the line is oversubscribed.

The signaling protocol must define the necessary procedures to support these services.

3.2 Signaling Protocol

A protocol is needed to control the transmission of data between integrated voice/data terminals and switches, and for switch to switch communication. The protocol suite includes link layer, network layer and transport layer functions. This protocol suite is not meant as a replacement for sophisticated protocols such as the ISDN suite, but as a testbed for the development and evaluation of unique digital speech processing algorithms and their application to integrated network implementation.

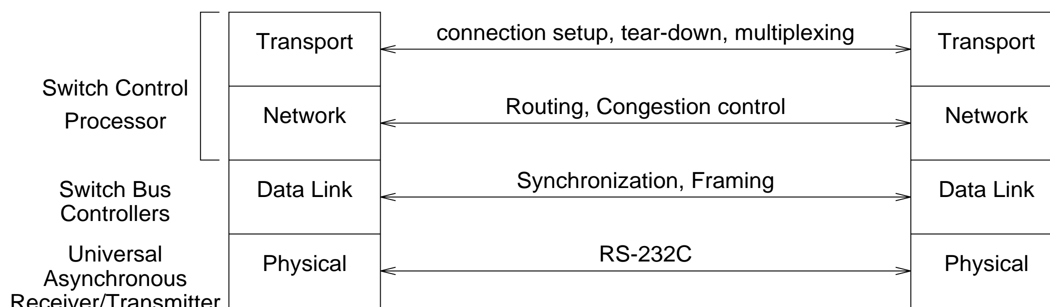

Figure 3-1: Switch Protocol Suite and functionality. Hardware components associated with the protocol layers are listed on the left.

3.2.1 Protocol Functions

The functions to be performed at each level of the protocol stack are described below. The protocol functions, and the switch components that implement the various protocol layers are summarized in Figure 3-1.

Physical Layer

A simple RS-232C interface is used between terminals and switches and between switches and modems. RS-232C is a universal standard, hence interfacing equipment (cables, connectors) is readily available. Since data rates up to 20 kb/s are permitted [26], it is ideal for narrowband applications.

Only a subset of the protocol functionality is implemented. Figure 3-2 shows the circuits that are used in the interface. An additional (non-RS232C) signal, *SYNCDET*, is provided for synchronous communication between the terminal and switch.

Data Link Layer

The link layer functions implemented by this protocol are synchronization and data framing. The lowest level of synchronization provided by the data link layer is *byte synchronization* [24]. An asynchronous or start-stop DLC (Data Link Control) protocol is used. The transmission format is shown in Figure 3-3. Simple

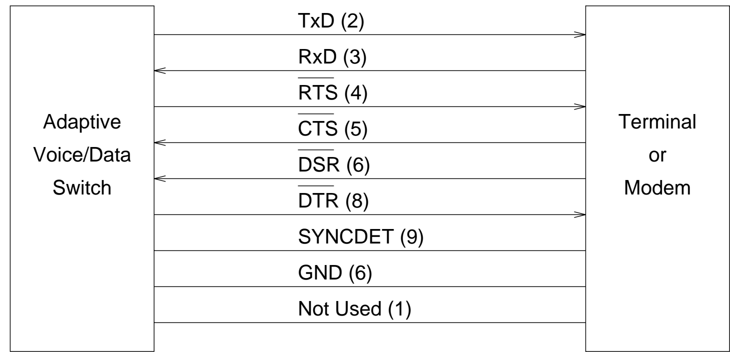

Figure 3-2: The RS-232C interface circuits. The pin numbers are given in parenthesis

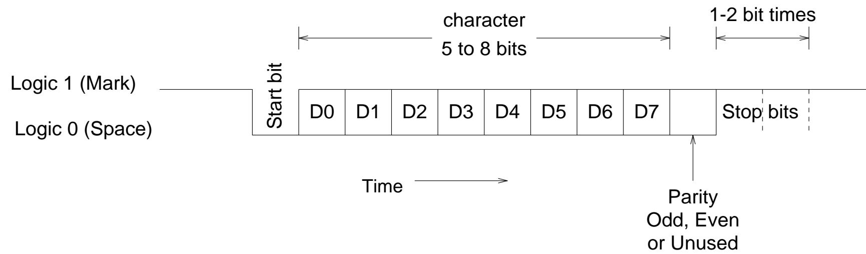

Figure 3-3: Asynchronous transmission format

error control is possible by the use of a parity bit. These functions are implemented using UARTs. The transmission parameters (character length, parity bit, STOP bit duration) can be controlled by programming the UARTs. This simple DLC protocol is adopted since the link between the terminal and switch is assumed to be slow-speed and relatively error free. If these assumptions are violated (as in a wireless system), more comprehensive functionality may be needed.

The data link layer is also responsible for *content synchronization* [24], or distinguishing control (or signaling) information from user data. This is accomplished in the following way. When a terminal is idle, all messages exchanged between the terminal and switch are control messages. When a terminal is engaged in a conversation with another terminal, control bytes are inserted at fixed intervals in the data stream. Control bytes can be identified by counting the data bytes between one control byte and the next. The distance (N) in bytes between con-

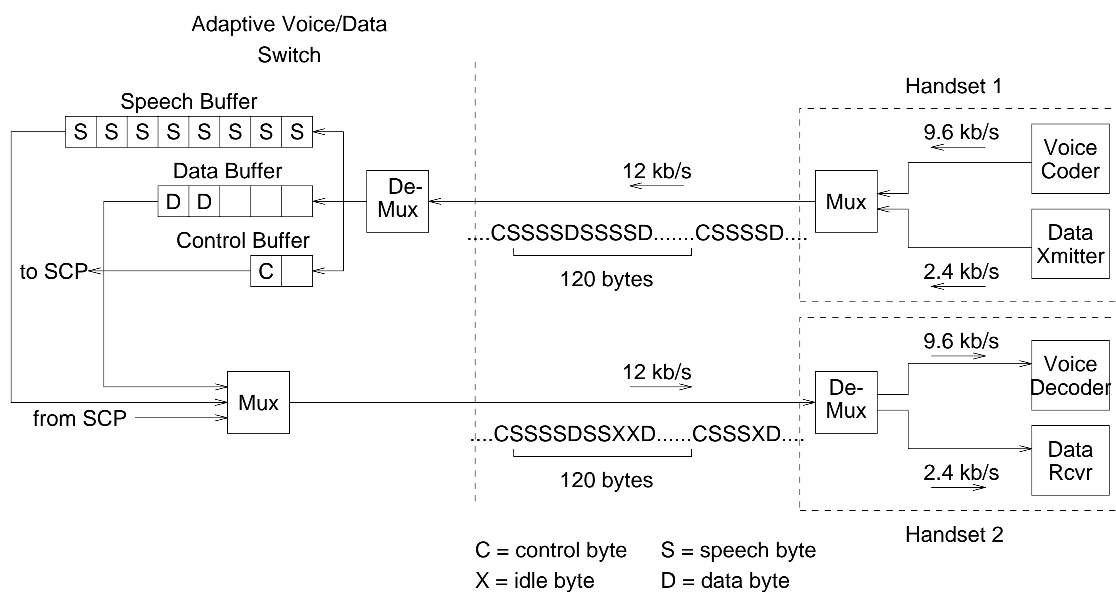

Figure 3-4: Multiplexing and demultiplexing of 9.6 kb/s voice and 2.4 kb/s data on a single connection

secutive control bytes is the ratio of the total data rate and the signaling rate, i.e. $N = R_{total}/R_{sig}$, where R_{total} is the sum of the voice and data rates on the connection, i.e. $R_{total} = R_{voice} + R_{data}$.

Content synchronization includes multiplexing and demultiplexing of speech and data on a single connection. This is performed on a link-by-link basis since the rate translation algorithm at each network node operates only on the speech stream. Speech and data is byte multiplexed in a manner dependent on the ratio R_{voice}/R_{data}. Figure 3-4 shows the multiplexing and demultiplexing of voice, data and control information for a connection supporting 9.6 kb/s voice and 2.4 kb/s data. Hence R_{voice} = 9.6 kb/s, R_{data} = 2.4 kb/s, R_{total} = 12 kb/s and R_{voice}/R_{data} = 4:1. For a signaling rate R_{sig} = 100 b/s, N = 120, i.e. a control byte will be transmitted every 120 user data bytes.

The multiplexing scheme described above depends upon the relative ordering of bytes in the transmitted byte stream. This byte order must be preserved at all times to ensure that synchronization is maintained. This requires that "filler"

bytes be transmitted in place of voice or data bytes when there is no information to be sent. The bit pattern 01111110 is used as a filler byte. This bit pattern is prevented from occurring in the voice/data stream by bit stuffing. In this technique, a "0" bit is inserted following five consecutive '1" bits appearing anywhere in the input bit stream. The bit stuffing operation is performed separately on the voice and data streams. The traffic streams are then multiplexed and transmitted, with filler bytes being inserted if necessary. At the receiver, the voice and data streams are demultiplexed, filler bytes are discarded, and then de-stuffing is performed i.e. any "0" bit following five consecutive "1" bits is removed. The filler byte insertion scheme borrows some features from the HDLC (High-Level Data Link Control) protocol [24] [11]. Figure 3-4 shows how filler bytes are inserted in a multiplexed traffic stream.

Network Layer

The network layer performs call routing and congestion control. Since the Adaptive Voice/Data Switch is a circuit switch, the network layer is static except during call setup, when switches are configured.

Call Routing

Control messages containing addresses are used to initiate connections within a local switch, and between two switches connected via the public switched telephone network (PSTN). A port address is used for local connections. For connections across the telephone network, a telephone number is required in addition to the port address.

Congestion Control

The network layer dynamically reallocates link bandwidth when the modem link experiences congestion. The bandwidth reallocation scheme is based on compressing the existing speech streams on the congested link to increase channel availability. Speech compression is performed by the switch in real time using the

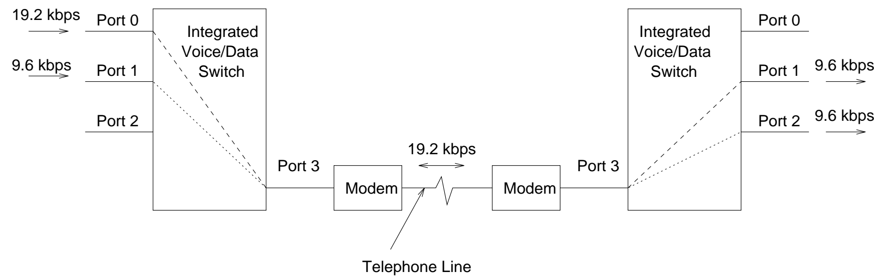

Figure 3-5: Control of call congestion using rate transformation

rate transformation capability of the STC speech coding technique.

Rate translation is illustrated in Figure 3-5. In this scenario, initially a connection exists between two terminals across the PSTN at 19.2 kb/s. Another call arrives at the originating switch, requesting a connection to a terminal on the far end switch. The switch drops the rate on the previous call to 9.6 kb/s and uses the extra bandwidth to exchange control messages with the switch at the other end. If the connection goes through (the terminal at the other end is not busy), a new transport layer connection is established on the outgoing link to accommodate the new call. If the connection fails, the old call reverts to the higher rate, and the new call request is rejected. Incoming call requests are granted or rejected on the basis of the resources available and the QOS required to be maintained on existing calls. The definition of this QOS and the fairness issues involved are discussed in a later chapter.

Figure 3-5 shows two switches connected via the PSTN. It is possible to construct a network of adaptive speech/data switches, in which case a single call may span multiple hops, and rate transformation may occur at any intermediate node in the network which is experiencing congestion.

Rate transformation is performed without interaction with the source coder. However, the decoder at the destination must be informed of the rate change. Control messages must be generated to perform the handshaking necessary to drop or increase the voice rate on a connection.

The network layer can also perform flow control by telling the source to throttle down. This "brute force" form of flow control may be used during periods of severe congestion.

Transport Layer

The transport layer is the highest layer implemented in the Adaptive Voice/Data Switch architecture. Its main functions are connection management and multiplexing. The transport layer is an end-to-end layer, i.e transport layer messages are exchanged between the endpoints of the connection, and simply forwarded by intermediate nodes in the network.

The transport layer manages the three phases of a connection: connection setup, data transfer and connection tear-down. The protocol supports modification of service parameters while a call is in progress. The transport layer also performs call multiplexing, or multiplexing of multiple calls on a single network layer connection. The multiplexing problem is complicated by the fact that neither the voice and data rates on a connection nor the total bandwidth occupied by different calls need be equal.

The multiplexing technique used is Time-Division Multiplexing (TDM). Since the switching system supports multirate circuit switching [6], every connection is built as a multiple of the basic channel rate of 2.4 kb/s. Thus, a 9.6 kb/s connection occupies 4 output time slots, and, in general, an N $\times$ 2.4 kb/s connection occupies N output time slots.

The inputs are polled in a round-robin fashion. Incoming speech/data bytes are read from the input bus and stored in an internal buffer for each port. The appropriate voice/data byte is placed on the output bus when the destination port is selected. For multiplexed connections on the modem link, the internal queues are polled in modified round-robin fashion according to the bandwidth requirements of each connection. When a new connection is established, the

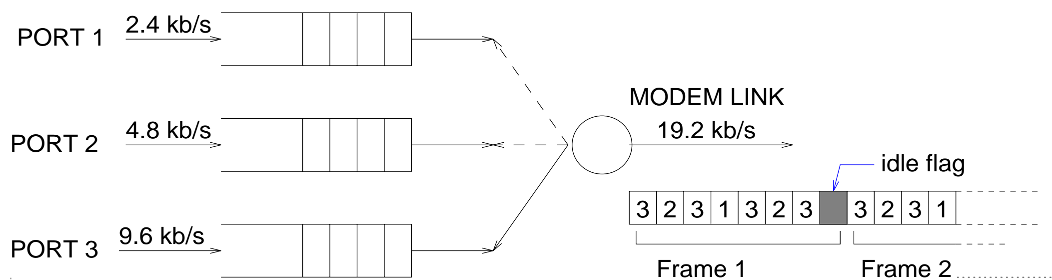

Figure 3-6: Multiplexing of multirate circuits on the modem link

switches at both ends synchronize, and establish the order in which time slots will be filled. For example, if ports 1, 2 and 3 are connected to port 4 at rates of 2.4 kb/s, 4.8 kb/s and 9.6 kb/s respectively, then the sequence in which the output time slots will be filled is as shown in Figure 3-6. Since the link capacity is 19.2 kb/s and only 16.8 kb/s is being utilized, idle bytes are transmitted in the empty slots. The algorithm for filling the slots is as follows:

1. Choose the highest rate stream and assign output slots.
2. Choose the next highest rate stream and so on, until all the streams have been assigned slots.
3. Mark the remaining slots idle.

The steps involved in assigning output time slots with the configuration in Figure 3-6 are shown in Figure 3-7.

3.3 Protocol Description

3.3.1 Control Messages

This section describes the different types of control messages and their formats. The control messages can be classified under the following categories.

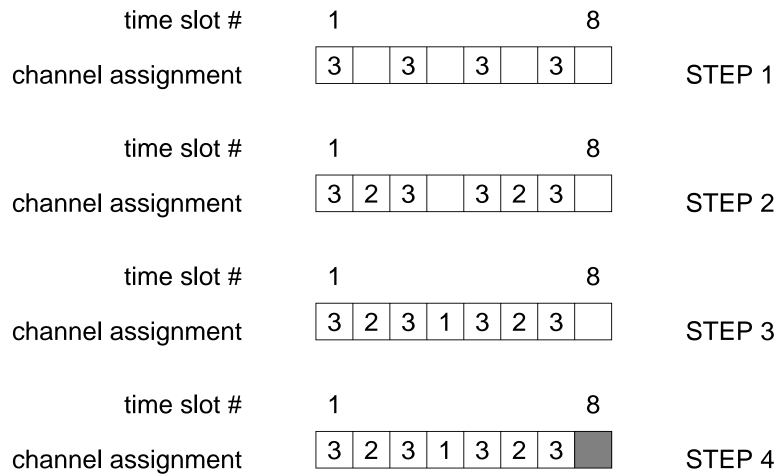

Figure 3-7: Time-slot assignment on the output link

Initialization

On power-on, or when a handset is plugged into the switch, start-up control messages need to be exchanged between the switch and the handset, before voice/data transmission can begin. The handset must communicate to the switch a "port active" signal, the data transmission rate and, in case of a modem, an indication that the port is connected to the PSTN. The switch must assign a "port id" to each active port. The data rate and handset/modem signal are set via jumpers on the switch board. A null control message from the handset to the switch acts as a port active indication, and the switch responds with the assigned port id.

Call Setup and Tear-Down

Control messages are required for setting up a connection between two ports on the switch. The maximum bandwidth required on the link is negotiated at call setup time. The voice connection rate can be specified as 2.4 kb/s, 4.8 kb/s, 9.6 kb/s or 19.2 kb/s depending on the rate supported by the coder. The allowable data rate is derived from these parameters. The data rate per call is limited to 9.6 kb/s in order to ensure fairness in the allocation of bandwidth, since bandwidth

allocated to data cannot be reallocated for the duration of the call. Call setup control messages include: Connection Request, Connection Indication, Connection Response and Connect Acknowledge. Call tear-down is needed to terminate a connection between two ports, and is accomplished using a Disconnect message.

Call Modification

The signaling protocol allows for modification of call parameters while the call is in progress, if the necessary resources are available. Call modification can be user-controlled or switch-controlled. It is possible for the user to switch between voice and data traffic and to multiplex both types of traffic on the same connection. The switch can drop the rate on the voice stream without interaction with the source during periods of congestion. Call modification is accomplished using control messages inserted into the data stream. These messages include: Modify Request, Modify Indication, Modify Response, Modify Acknowledge and Restart[Modify] (for user-controlled modification), and Transform Rate, Transform Acknowledge and Restart[Transform] (for switch-controlled modification).

Connection Management

The switch can multiplex several transport layer connections on a single network layer connection. Logical channels are assigned channel ids, and the transport layer entities at either end of the connection are resynchronized at the connection boundaries. Connection management is accomplished using Open Channel, Open Channel Acknowledge, Restart[Transport] and Close Channel control messages.

Flow Control

A Modify Source Rate control message is provided to allow the switch to throttle the source. This form of flow control is used only during periods of severe congestion.

Message Type	Direction	Function
Call Setup and Tear-down Messages		
Connection Request	u $\rightarrow$ n	Initiates call establishment
Connection Indication	n $\rightarrow$ u	Indicates incoming call
Connection Response	n $\rightarrow$ u	Indicates call progress - proceeding/granted/rejected
Connect Ack	u $\leftrightarrow$ n	Indicates call establishment successful
Disconnect	u $\leftrightarrow$ n	Requests/Indicates call termination
Call Modification Messages		
Modify Request	u $\rightarrow$ n	Initiates call parameter modification
Modify Indication	n $\rightarrow$ u	Indicates call parameter modification
Modify Response	n $\rightarrow$ u	Indicates modification request granted/rejected
Modify Ack	u $\leftrightarrow$ n	Indicates call modification successful
Call Modified	u $\leftrightarrow$ n	Indicates call restarted with modified parameters
Transform Rate	n $\rightarrow$ u	Initiates voice rate transformation
Transform Ack	u $\rightarrow$ n	Indicates rate transformation request received
Rate Transformed	u $\leftrightarrow$ n	Indicates call restarted with transformed voice rate
Connection Management Messages		
Open Channel	n $\rightarrow$ n	Initiates establishment of transport layer connection
Open Channel Ack	n $\rightarrow$ n	Indicates transport layer connection established
Start Channel	n $\rightarrow$ n	Indicates transport layer restarted
Flow Control Messages		
Modify Source Rate	n $\rightarrow$ u	Indicates source must throttle down
Modify Source Ack	u $\rightarrow$ n	Indicates source rate modified

Table 3.1: Control Message types and functions

Table 3.1 summarizes the different types of control messages and their functions. The direction of message flow is also indicated (u = user, n = network). The control message formats are illustrated in Appendix A.

3.3.2 Call Setup Example

An example of the use of the protocol to setup a call across the PSTN is shown in Figure 3-8. When the caller lifts the handset and dials a number, a CONNECTION REQUEST is generated and sent to the switch. The CONNECTION REQUEST specifies the destination type (local or remote), destination address, traffic type and data rates. Some means must be provided in the handset for the caller to select these parameters. For instance, the destination type may be

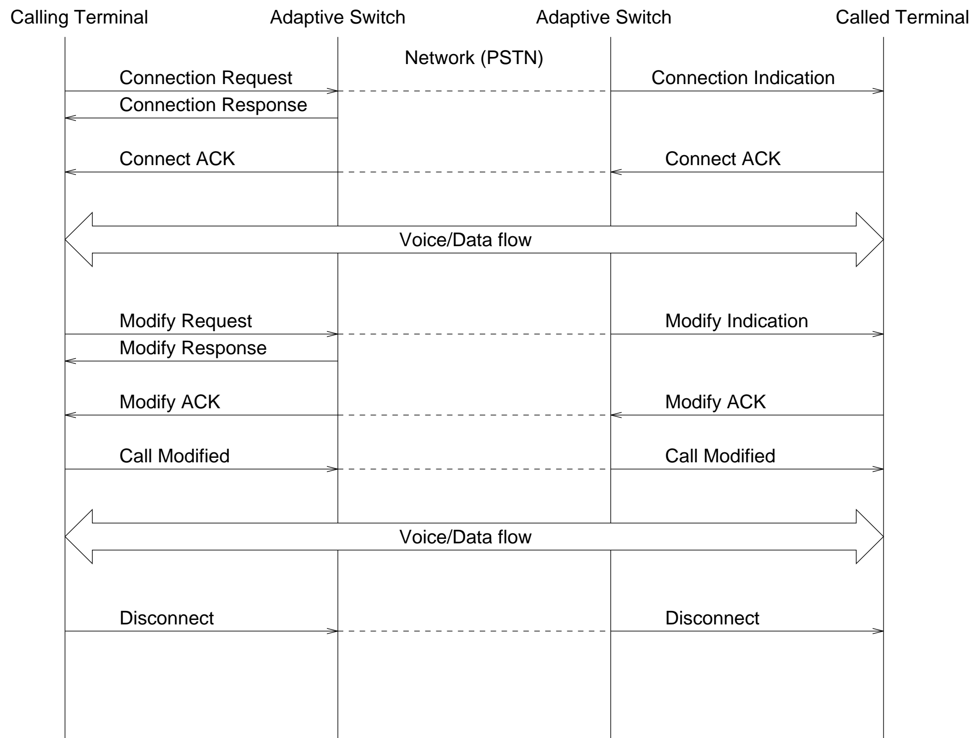

Figure 3-8: Control message flow diagram for a typical call

indicated in a manner similar to that used in PBX systems - all calls are assumed to be local, and a special digit must be dialed before dialing a remote destination.

On receiving a CONNECTION REQUEST, the local switch forwards this request to the remote switch, and sends a CONNECTION RESPONSE to the calling terminal indicating that the call is proceeding. If the resources required for the call were not available, i.e. if the modem line were connected to a different destination or could not accommodate a new call, the CONNECTION RESPONSE would indicate that the call was rejected. When the CONNECTION REQUEST reaches the remote switch, it sends a CONNECTION INDICATION to the called terminal, which generates a ringing tone. When the called party lifts the handset, a CONNECT ACK is sent back to the calling terminal. A connection now exists between the two handsets, and voice/data transmission can begin.

If the user at either end desires to modify the call parameters, he/she may do

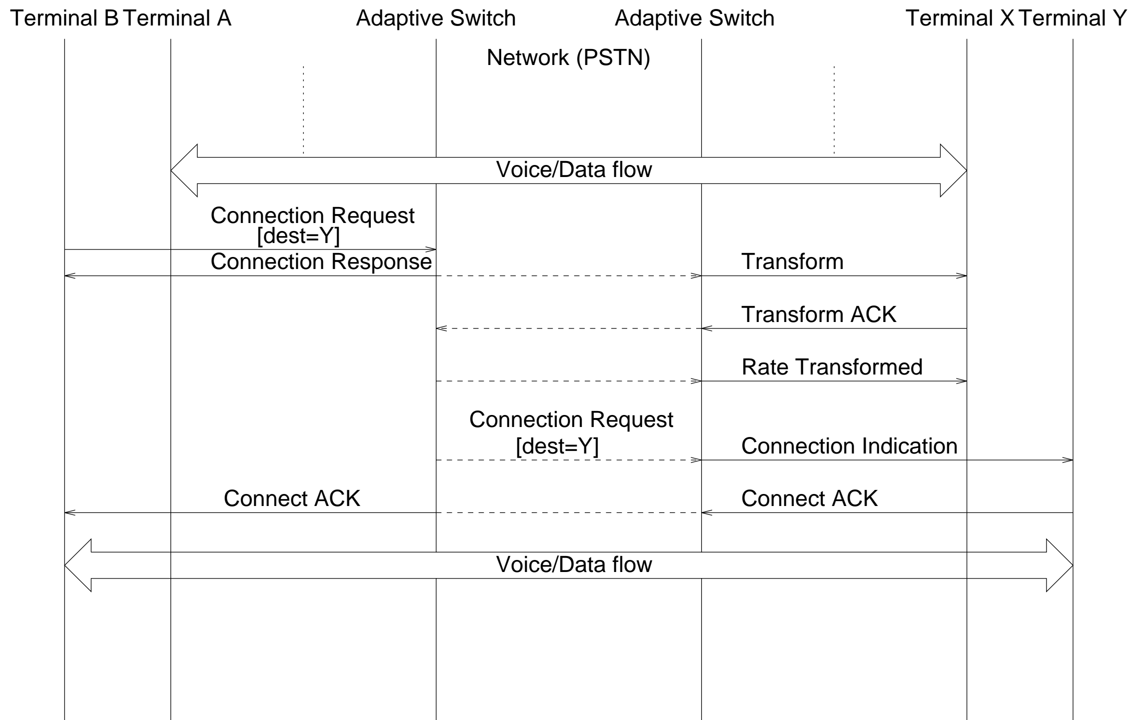

Figure 3-9: Control message flow diagram for a call using rate transformation

so by sending a MODIFY REQUEST to the local switch. This request is handled in a manner similar to the CONNECTION REQUEST, except that a three-way handshake is used to ensure synchronization between the two endpoints. When one of the users hangs up, a DISCONNECT message is sent to the local switch, forwarded to the remote switch and from there to the handset at the other end, and the connection is terminated.

The control message flow diagram for call setup over a link using rate transformation is shown in Figure 3-9. In this scenario, the modem link is being utilized for data transmission between Terminal A and Terminal X. When Terminal B requests a connection to Terminal Y, the necessary resources are freed by sending a TRANSFORM message to Terminal X. The freed-up resources are then used to setup a connection between Terminal B and Terminal Y, and subsequently, for data transfer between them.

The above flow diagrams do not explicitly indicate the switch-to-switch hand-

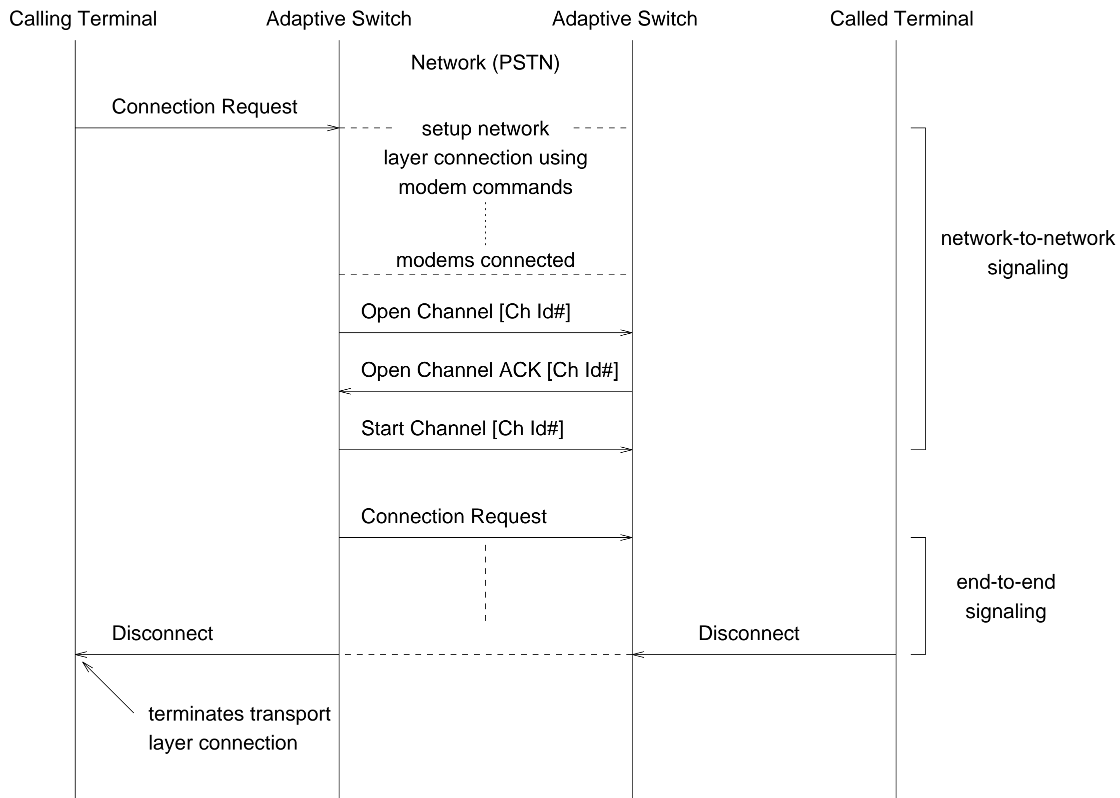

Figure 3-10: Switch-to-switch handshaking for connection setup

shaking required to setup and release connections. The standard modem commands (the AT commands) are used to establish a connection between the modems, i.e. a network layer connection. The control messages used to setup a transport layer connection between the two ends are shown in Figure 3-10.

Chapter 4

Protocol Simulation

This chapter describes the development of a software testbed for the evaluation of the speech coding technique and network management protocols. The management and control software is integrated with the speech coder software to form a system for emulating an integrated network environment. Configuration files are used to select particular inputs and test scenarios. Several simulated conversations are used to test the control protocols under various network load conditions, and to evaluate the transformation algorithm.

4.1 Switch Modeling

The switch architecture described in Chapter 2 shows the distribution of hardware tasks among the switch components. For simulation purposes, these tasks are lumped together into three main components as shown in Figure 4-1, in order to simplify the switch model. The control software developed emulates the function of the switch control processor (SCP). The rate translation and speech file processing functions are performed off-line (using shell scripts).

The switch is modeled as having four input-output ports, each of which is connected either to a handset or to the PSTN via a modem. The switch supports

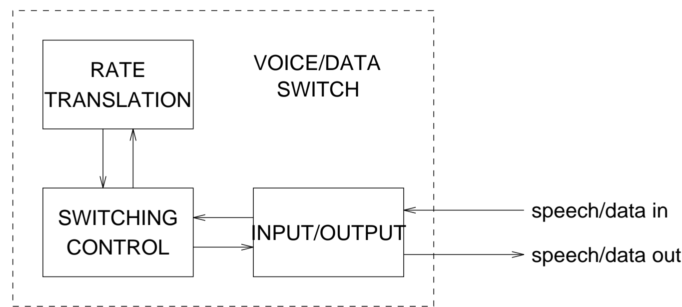

Figure 4-1: Simplified model of adaptive voice/data switch

connections between its ports as well as connections to remote switches via the telephone network. Each port sends and receives streams of bytes, which contain signaling messages multiplexed with voice and data. Binary files are used to represent the input and output byte streams for each port. The switch polling mechanism is simulated by reading a byte at a time from the input files in a round-robin fashion.

4.2 Simulation Goals

The main objectives of the simulation are to verify the control protocols and to observe the effects of call modification, in particular rate translation, on perceived speech quality.

Protocol verification includes the following tasks:

- Delineation of control messages and user data in the incoming byte stream.
- Verification of the capability of the protocol to handle the following types of requests for service:
  - requests for connection (*call setup*),
  - requests for modifying call parameters within negotiated limits (*call modification*),
  - congestion control and bandwidth management (*rate translation*),

– requests for disconnection (*call teardown*).

Tests on speech quality involve observing the speech output when the voice stream is subjected to multiple rate transformations at the switch.

4.3 Simulation Strategy

The simulation process is illustrated in Figure 4-2. The simulation essentially consists of the five phases described below.

1. **Phase I: Speech Analysis**

   In this phase, the speech file for each input port is analyzed using *stc_ana*, the STC analysis program.

2. **Phase II: Control Message Insertion**

   The coded speech for each input port is merged with the corresponding control message stream to produce a composite input file for that port. The formats and types of control messages inserted depends on the test scenario.

3. **Phase III: Switching**

   The switching module reads the composite speech file for each input port, extracts signaling information from the byte stream, and takes appropriate action on detecting a control message. Call control and supervision is accomplished by maintaining "state" for each active port. The switching module then routes the incoming data stream to the appropriate destination, and inserts signaling messages in the outgoing stream as necessary.

4. **Phase IV: Control Message Removal**

   The output file for each port is decomposed into control and voice streams. The control stream can be examined to verify the functioning of the protocol, and the voice stream is submitted to the next phase for processing.

5. **Phase V: Rate Transformation and Speech Synthesis**

   Rate Transformation is performed on the voice (if necessary) using *stc_xform*, the STC transform program, followed by synthesis of the speech using *stc_syn*, the STC synthesis program. The speech output can now be played back to observe the change in quality.

The simulation programs were developed in 'C' on a Unix platform. Workstation audio devices and associated audio tools were used to record and play back speech. Appendix B contains the code listing for the program used to parse the configuration files and insert control messages into coded speech files. A sample configuration file is also included. Appendix C contains the listing for the simulation program.

4.4 Test Cases

The switch protocols were tested and verified using various test scenarios including local and remote call setup between two terminals with different sets of call parameters, handling of blocked calls due to contention for a destination port or resource unavailability and call modification. Of particular interest is the rate transformation algorithm involving the speech coder. A two-state rate translation algorithm was developed and subsequently enhanced to accomplish multi-stage rate translations. In addition to the speech and control message files generated as output, the simulation program can be made to output text messages to the screen when operating in "verbose" mode. This provides a convenient method for examining the switching actions performed. The screen output for a two-stage rate translation scenario is reproduced below. In this scenario, port 1 is connected to a modem, and ports 0, 2 and 3 are connected to terminals.

```
1   time slot #0     cntrl byte = 0x1a
2   Received CONN_REQ on port 0
3   Connection Request parameters:
```

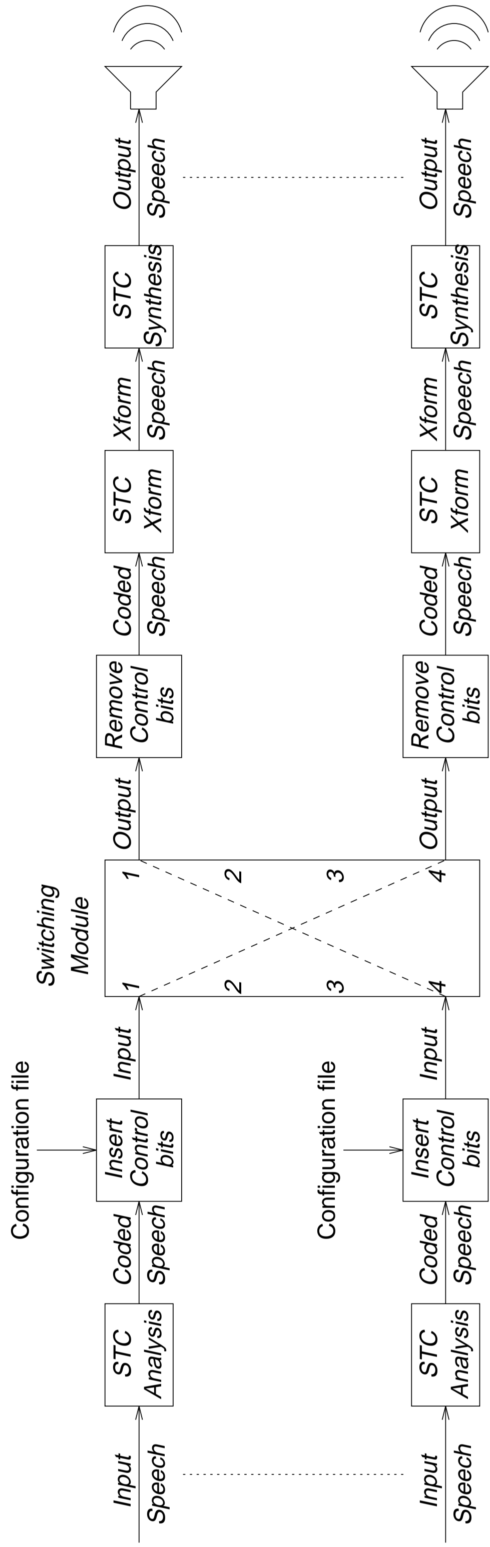

Figure 4-2: Block diagram of simulation strategy

```
4          destination type = 1
5          destination address = 2
6   time slot #1     cntrl byte = 0xbe
7          voice on = 1
8          data on = 0
9          maximum rate = 7
10         voice rate = 3
11  time slot #2     cntrl byte = 0x50
12         coder init = 5
13  time slot #3     cntrl byte = 0x39
14  time slot #4     cntrl byte = 0x31
15  time slot #5     cntrl byte = 0x33
16  time slot #6     cntrl byte = 0x38
17  time slot #7     cntrl byte = 0x36
18  time slot #8     cntrl byte = 0x34
19  time slot #9     cntrl byte = 0x37
20  time slot #10    cntrl byte = 0x37
21  time slot #11    cntrl byte = 0x33
22  time slot #12    cntrl byte = 0x38
23  Request parameters received from port 0
24  Sent WAIT...CALL PROCEEDING to port 0
25  Dialing telephone number on port 1(modem)
26  The dialing sequence is: ATDT 9138647738
27  time slot #13    cntrl byte = 0x31
28  Port 1: Modem result code = CONNECT
29  Sent OPEN_CHANNEL to port 1(modem)
30  time slot #14    cntrl byte = 0xf0
31  Received OPEN_CHANNEL_ACK from port 1
32  Sent CONN_REQ on external line (port 1) ch 1
33  time slot #16    cntrl byte = 0xf9
34  Recd CONN_ACK from port 1(dest) ch 1
35         coder init = 9
36  Sent CONN_ACK to port 0(src)
37  time slot #2400         cntrl byte = 0x1b
38  Received CONN_REQ on port 2
39  Connection Request parameters:
40         destination type = 1
41         destination address = 3
42  time slot #2401         cntrl byte = 0xbe
43         voice on = 1
44         data on = 0
45         maximum rate = 7
```

```
46          voice rate = 3
47   time slot #2402          cntrl byte = 0x50
48          coder init = 5
49   time slot #2403          cntrl byte = 0x39
50   time slot #2404          cntrl byte = 0x31
51   time slot #2405          cntrl byte = 0x33
52   time slot #2406          cntrl byte = 0x38
53   time slot #2407          cntrl byte = 0x36
54   time slot #2408          cntrl byte = 0x34
55   time slot #2409          cntrl byte = 0x37
56   time slot #2410          cntrl byte = 0x37
57   time slot #2411          cntrl byte = 0x33
58   time slot #2412          cntrl byte = 0x38
59   Request parameters received from port 2
60   Outgoing line busy...trying to multiplex connection
61   Physical connection to dest exists, checking bandwidth available
62   Bandwidth currently not available...trying to transform rate
63   Sent CONN_RESP to port 2
64   Sent TRANSFORM to port 1(modem) ch 1
65   Pending request from port 2(src) to port 1(dest)
66   time slot #2413          cntrl byte = 0xf0
67   Received TRANSFORM_ACK from port 4
68   Enable DSP
69   Wait for Started Xform signal from DSP
70   Started Xform signal received
71   Sent START_XFORM to port 4
72   Sent OPEN_CHANNEL to port 1(modem)
73   time slot #2414          cntrl byte = 0xf0
74   Received OPEN_CHANNEL_ACK from port 1
75   Sent CONN_REQ on external line (port 1) ch 2
76   time slot #2416          cntrl byte = 0xf9
77   Recd CONN_ACK from port 1(dest) ch 2
78          coder init = 9
79   Sent CONN_ACK to port 2(src)
80   Reached EOF on port 0
81   Reached EOF on port 4
82   Reached EOF on port 1
83   Reached EOF on port 2
84   Reached EOF on port 3
85   Reached EOF on port 5
```

Whenever a non-null control byte is received from any port, it is displayed in

hexadecimal format. The time slot counter is incremented each time one complete round-robin cycle is traversed. Since the input bus is time-division multiplexed, with a slot assigned to each port, one round-robin cycle can be considered as an input bus "frame", and hence the time slot counter value actually represents the frame count. The control message types are displayed in capital letters.

On receiving a connection request from port 0 (lines 1-23), a switch-to-switch connection is first established (lines 25-28) using the modem. A virtual circuit is now established and assigned a channel id (lines 29-31), and call setup is completed (lines 32-36). Call setup for the subsequent connection request from port 2 proceeds similarly except that the voice rate on channel 1 must first be dropped (lines 64-71) in order to accommodate an additional channel. A three-stage rate dropping scenario can similarly be envisioned.

4.5 Results

Informal listening tests were performed to evaluate the speech quality in the presence of rate transformation. The following discussion is based on a three-stage rate dropping scenario; similar observations hold for the general case of n-stage rate transformation. Initially, the rate was dropped at arbitrary points in the speech stream. The tests performed showed a perceivable degradation in speech quality at the receive end when transitioning from a higher rate to a lower rate. The effects observed included audible 'clicks', fading and distortion of the spoken syllable. It is interesting to note that the steady-state rate change was not as easily perceivable as the transition point. These observations suggested the use of silence detection to improve speech quality.

Figure 4-3 is a representation of the input speech file coded at 19.2 kb/s using the STC. The sound segments are displayed as boxes and the silence regions are displayed as lines connecting these boxes. These pauses usually indicate breaks

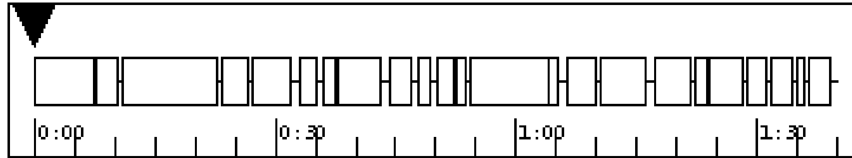

Figure 4-3: Simulation input speech file coded at 19.2 kbps

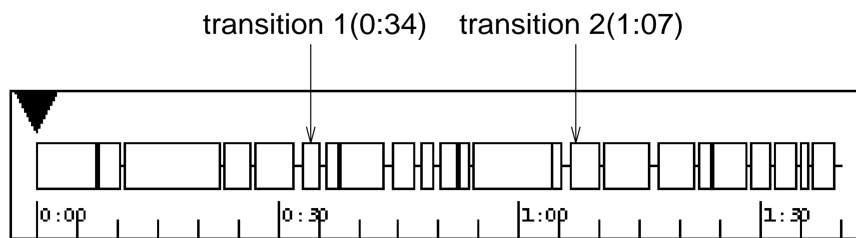

Figure 4-4: Rate transition points with no silence detection

Figure 4-5: Rate transition points with silence detection

Output Speech Quality	
no silence detection	with silence detection
audible clicks, fading at transition point	transition point inaudible

Table 4.1: Simulation results

between sentences or phrases. Time is measured along the horizontal axis. Figure 4-4 shows the points at which rate transitions occur with no silence detection. Point A represents a transition from 19.2 kb/s to 9.6 kb/s and Point B represents a further down translation to 4.8 kb/s. In Figure 4-5 the transition points are shifted so as to coincide with silence intervals. The transitions are no longer noticeable in the audio output. The results are tabulated in Table 4.1.

These results suggest the use of silence detection at the switch accompanied by synchronization of rate translation instances with silence periods to achieve a smoother transition between discrete speech rates. Silence detection can be accomplished by stealing a bit from each speech frame for speech activity indication. A simple thresholding algorithm can be incorporated in the STC analyzer to mark the activity bit. The signal processing algorithm for performing rate translation at the switch can be modified so that a rate transition occurs only after N successive inactive frames. The parameter N depends on the stationary properties of the speech waveform, and must be determined by studying the speech production model used by the coder. A time-out mechanism may be needed to force a transition in case of anomalous speech conditions.

Chapter 5

Congestion Control

5.1 Overview of Bandwidth Allocation Algorithms for Circuit-Switched Environments

Traditionally, in a connection-oriented network (such as the telephone network), bandwidth is dedicated to the user for the duration of the call, and new connections are refused if insufficient bandwidth is available. Thus, performance is predictable once a connection is set up. Connectionless networks, on the other hand, have the ability to allocate bandwidth dynamically, but respond to overloads by degrading the performance seen by all users. This is due, in large part, to their limited ability to restrict access to the network. The term *congestion control* is mostly used with reference to packet-switched networks to mean "the collection of methods used to ensure each user acceptable performance under a variety of load conditions" [27]. Quality of service (QOS) negotiations have also been traditionally associated with packet-switched systems.

Most of the previous work on bandwidth allocation in circuit-switched communication networks with integrated traffic focuses on the *access control* problem. A user is granted or refused access to the network based on a policy designed to meet some performance criterion (such as minimum blocking or maximum utiliza-

tion). The simplest bandwidth allocation policy is *complete sharing*, in which an incoming call is always accepted if sufficient resources are available. However, this policy is unfair in situations where a heavily loaded traffic stream monopolizes the available bandwidth. A *complete partitioning* policy overcomes this drawback by allocating a fixed amount of the total bandwidth to each traffic stream. However, this approach could lead to wasted capacity if the load offered by a traffic stream drops below its allocated capacity. Several other bandwidth allocation schemes have been developed which lie between these two extremes - for example, the thresholding policy advocated in [10]).

The policies described above fall under the category of static bandwidth allocation schemes. Bandwidth allocation schemes which dynamically track network load variations have also been proposed. These include moveable boundary schemes [2, 20], in which slots in a TDM frame (or channels on a link) are partitioned among different traffic streams, and the partitioning varies dynamically with instantaneous traffic levels. Only very recently, mention of dynamic rate control mechanisms appears in the literature. These schemes use a variable rate source coder for dynamic rate adjustment based on load [13, 28].

5.2 Congestion Control in Adaptive Voice/Data Networks

The previous section discusses bandwidth allocation schemes employed for existing network architectures. The dynamic bandwidth reallocation and congestion control capabilities of adaptive voice/data networks are now examined, and contrasted with the features described above.

The adaptive voice/data switch operates essentially as a circuit switch. Resources are allocated when the connection is established, and held for the duration of the connection. However, due to the unique capabilities of the speech coder

and the control protocols, it is possible to retrieve allocated resources from existing connections and reallocate these resources dynamically. This is accomplished using the rate transformation capability of the STC coding technique. In particular, this capability can be used to free up bandwidth to accommodate new connections during periods of overload. Thus, a mechanism for connection-level overload control is available, based on degrading the quality of service provided to all the users in the system. The overload control is only effective, however, until the maximum limit on the number of users is reached. All subsequent connection requests are blocked.

It is evident from the above discussion that the adaptive switch possesses dynamic bandwidth allocation capabilities similar to those of connectionless networks, while retaining the access control capabilities of connection-oriented networks. A congestion control scheme is needed that exploits these unique capabilities, and addresses the issues of fairness and QOS guarantees.

5.3 Adaptive Switch Performance Evaluation

A comparative simulation study of switch performance under a variety of channel assignment schemes is conducted. A model is developed for the adaptive voice/data switch that incorporates the rate transformation algorithm. A set of performance measures that can be used as figures of merit for different resource allocation schemes is identified. The modeling and simulation tool used is the Block Oriented Network Simulator (BONeS).

5.3.1 Modeling

Figure 5-1 shows the top-level block diagram of the switch model. For communication over the PSTN, the switch functions as a traffic concentrator. Several voice/data terminals compete for bandwidth on the outgoing link. The termi-

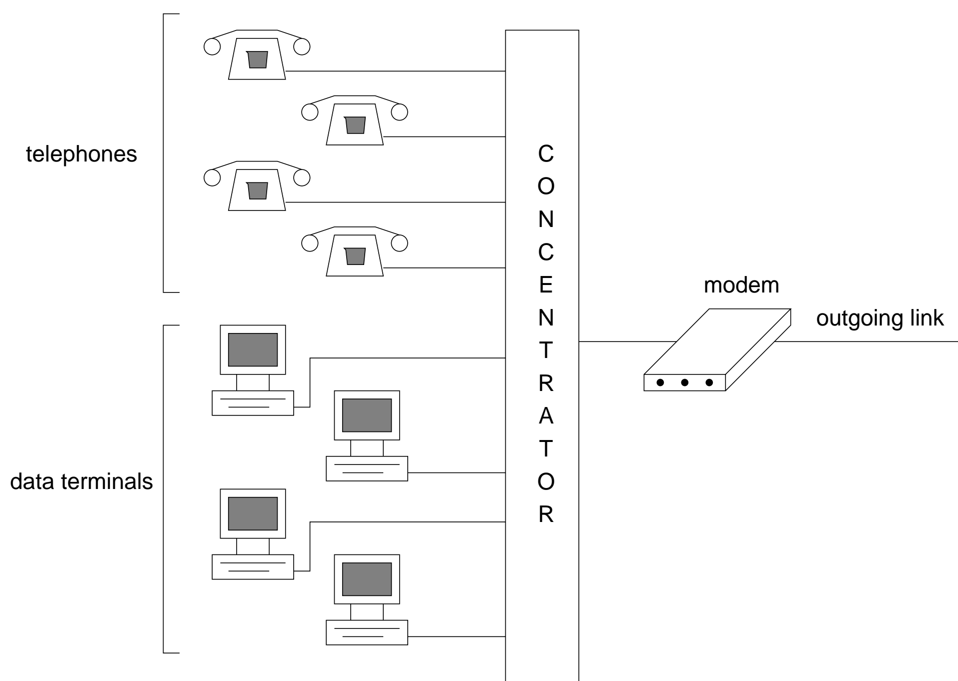

Figure 5-1: System model: voice and data sources accessing a concentrator

nals are modeled as independent voice and data sources rather than as integrated voice/data sources in order to simplify the source model. This simplification has the effect that the arrival processes for voice and data calls are not correlated. Voice and data call arrivals are assumed to occur according to a Poisson process, and the service time is assumed to be exponentially distributed. The bandwidth requested by each type of call (voice or data), the mean call holding time, the offered load per source and the capacity of the outgoing link are parameters that can be set at simulation time.

The main component of the simulation model is the concentrator that multiplexes voice and data calls onto the outgoing link. The algorithm used to modify voice calls is as follows. A voice call requires a bandwidth of V slots. If V slots are available, the call is accepted. If V slots are not available, an attempt is made to grant V/2 slots. If V/2 unused slots cannot be found, a V-slot call currently in service is transformed to a V/2-slot call to free up V/2 slots. If no V-slot call is currently in progress, an attempt is made to grant V/4 slots to the incoming

request, either directly (if available), or by transforming a V/2-slot call currently in progress to a V/4-slot call. In the model, data calls are accepted or rejected on the basis of unused resource units available at the instant the call arrives, i.e. voice quality is not degraded in order to accommodate data calls.

The higher rate call to be victimized is chosen in a random manner. This is partly due to the modeling difficulties faced in assuming otherwise (BONeS 'allocate resource' blocks do not have provision for preemption). A solution to this problem which would allow the call to be transformed to be selected in a deterministic manner was not pursued since the aim was to keep the model simple. Such an assumption is considered to be reasonable, as long as the results obtained are interpreted in the light of the modeling assumptions made. Actually, it turns out (due to the implementation method) that the call selected for rate-transformation is most likely (but not necessarily) the one that has been in service the longest. This is in fact a good candidate for rate reduction.

5.3.2 Performance Measures

The figures of merit used to compare the performance of different resource allocation schemes are listed below.

1. **Fractional Utilization**: This is the ratio of the average number of resource units allocated to the total capacity.

2. **Blocking Probability**: The voice call blocking probability is conditional on the amount of modification allowed. Hence, three different definitions of blocking exist for voice calls; the fraction of voice calls rejected with (a) no modification allowed, (b) modification to half-rate, and (c) modification to quarter-rate. A voice call is said to be *ultimately blocked* if it cannot be accepted even after modification to quarter rate [23]. The data call blocking probability is the ratio of the number of data calls rejected to the total

number of data calls.

3. **Average quality of service (QOS)**: This performance measure is defined to indicate the average grade of service on voice calls. Assuming that voice quality degrades linearly with rate (this is a feature of the coding scheme, which is approximately true), the Average QOS for voice calls is the mean of the per call QOS, which is defined as:

$$\text{QOS/call} = \frac{1 \times T_V + 0.5 \times T_{V/2} + 0.25 \times T_{V/4}}{\sum N} \tag{5.1}$$

   where T_i is the duration of time spent in the i-slot state. It is assumed that a V slot voice call corresponds to a QOS of 1 i.e. the QOS is normalized with respect to the maximum voice call bandwidth. It must be noted that the average QOS is defined only for those calls that ultimately receive service, i.e. this measure does not account for blocking.

4. **Fairness Index**: From the user's viewpoint, an indication of the uniformity of quality of service provided by the network can be obtained by measuring the fluctuation of the average QOS about the mean value. A fairness index is thus defined as:

$$\text{Fairness Index} = \sigma^2(\text{QOS}) \text{ or } var(\text{QOS}) \tag{5.2}$$

5.3.3 Simulation Results

The parameters used for the simulation studies conducted are listed in Table 5.1. The link capacity (C), and the bandwidth required per call are measured in "slots". One slot refers to the smallest bandwidth unit available on the outgoing link. Data traffic is modeled as being constant and low-load. The performance of the system is measured under overload conditions by allowing the offered voice

Parameter	Value
Number of voice terminals	4
Number of data terminals	4
Voice call bandwidth	8 slots
Data call bandwidth	2 slots
Average voice call holding time	6 minutes
Average data call holding time	12 minutes
Average offered load per data terminal	0.25

Table 5.1: Simulation parameters

load to vary over a range of values. This approach allows us to examine the effect of a highly loaded traffic stream on itself, and on non-interfering traffic of a different type.

Complete Sharing System with C=8

Figures 5-2 and 5-3 show the blocking probabilities and average link utilization for a link capacity of 8 slots. The improvement in blocking performance that can be obtained at the cost of voice quality can be estimated from Figure 5-2. It is observed that data calls experience higher blocking than voice calls with modification allowed to 2 slots/call even though the data bandwidth is also 2 slots/call. This is reasonable, because voice calls can "borrow" bandwidth from higher rate calls when resources are not available, whereas data calls are cleared. Figure 5-3 shows that the link utilization does not saturate even when the link is overloaded by a factor of 4. In Figure 5-4, the average QOS is plotted as a function of the load. The length of the vertical bars represents the QOS variance. It can be observed that the complete sharing scheme treats the users with increasing uniformity as the load increases.

Figure 5-3: Fractional Link Utilization, C=8

Figure 5-4: Fairness index for voice calls, C=8

Comparison of Performance with Different Link Capacities

Figures 5-5 and 5-6 compare the average QOS and utilization for different link capacities. A 20% improvement in average voice QOS can be obtained by increasing the link capacity from C=8 slots to C=10 slots (i.e. by 25%) with only a marginal reduction in utilization. An interesting observation is that with C=9, the QOS on voice calls actually degrades drastically. The reason for this can be inferred from Figure 5-7. It appears that data calls grab that one “extra” slot (since the data blocking probability improves considerably) which is then never released, so that voice calls can never be allocated 8 slots. This factor must be considered when choosing link capacities that are non-integer multiples of the minimum required bandwidth for voice calls.

Figure 5-6: Comparison of Link Utilization with different link capacities

Figure 5-7: Blocking probabilities for voice and data calls, C=9

Thresholding Policy for Data Calls

Foschini et al [10] suggest that the optimum channel allocation policy for a system with two traffic classes is a thresholding policy, i.e. a policy that restricts the maximum number of channels allocated to one traffic class. Figures 5-8 through 5-11 show the results obtained by setting the data threshold to 1, 2 and 3 calls. An exact analytical proof to determine the optimal policy is not attempted. However, the following inferences can be made from the graphs.

1. The optimal policy for maximum utilization is one which limits the maximum number of data calls to 3.

2. If minimum overall blocking is desired, the objective function to be minimized can be expressed as

$$J_{\text{min blocking}} = PB_v \times \rho_v + PB_d \times \rho_d \tag{5.3}$$

Figure 5-8: Voice call blocking probabilities with thresholding

where PB_v=voice call blocking probability, PB_d=data call blocking probability, ρ_v=offered voice load and ρ_d =offered data load.

3. The thresholding policy is more effective in improving the QOS delivered on voice calls than it is in reducing the voice call blocking probability.

5.3.4 Conclusions

A model is developed for the adaptive voice/data switch that can be used to predict performance. The advantages of using a simulation tool are: the ability to model the unique call modification capabilities of the adaptive switch, flexibility in choosing resource request parameters (such as traffic characteristics) and the ability to model a wide range of resource allocation policies. A set of performance metrics is identified that can adequately describe the behavior of the system. The simulation results verify that the improvement in overload control resulting

Figure 5-10: Comparison of Average QOS for different thresholds

Figure 5-11: Comparison of link utilization for different thresholds

from the use of rate transformation is indeed substantial. These results can be used to determine the required link capacity given some performance constraints. The effect of a simple access control scheme in enhancing the overload control capabilities of the switch is demonstrated.

Chapter 6

Conclusion and Future Work

A protocol has been developed for the management and control of a narrowband integrated voice/data network. The protocol architecture, the services offered and the procedures needed to support these services have been defined. The control protocol has several unique features, including in-call service re-negotiation and voice rate modification. The protocol exploits the novel capabilities of the Sinusoidal Transform Coder to implement dynamic bandwidth reallocation in a narrowband communication environment.

A software testbed was constructed to verify the control functionality and to demonstrate the interaction of the protocols with the speech coder. The software developed can be ported to a real application environment with minor modifications, hence the use of the code is two-fold.

Scope for further work exists in several areas. The protocol does not incorporate any error correction mechanisms (except for parity detection at the link layer). This is irrelevant for voice traffic, but has significance for data transmission, especially in harsh data loss environments (such as cellular or wireless transmissions). Enhancements to the protocol that make it more robust in the presence of channel errors will greatly aid its application in wireless environments.

The performance evaluation conducted served to illustrate the interrelation

between the different performance metrics. It would be instructive to extend these studies to a multi-hop network environment. Another interesting subject for further study would be to compare the performance of a source-coder based dynamic rate control scheme to the STC-based rate transformation scheme in a multi-hop network environment.

The quality of service (QOS) definition in this work is 'open-loop', i.e. there is no mechanism for the user to demand a certain QOS level. This is mainly due to the difficulty in anticipating the users' demands. For an application in which users require QOS guarantees, it may be necessary to include a field in the connection establishment control messages for QOS negotiation.

Another area for future investigation is the design of integrated voice/data terminals which will support the protocol developed. One approach is to use a general-purpose PC with audio input/output capabilities and an add-on DSP card for speech processing.

Appendix A

Control Messages

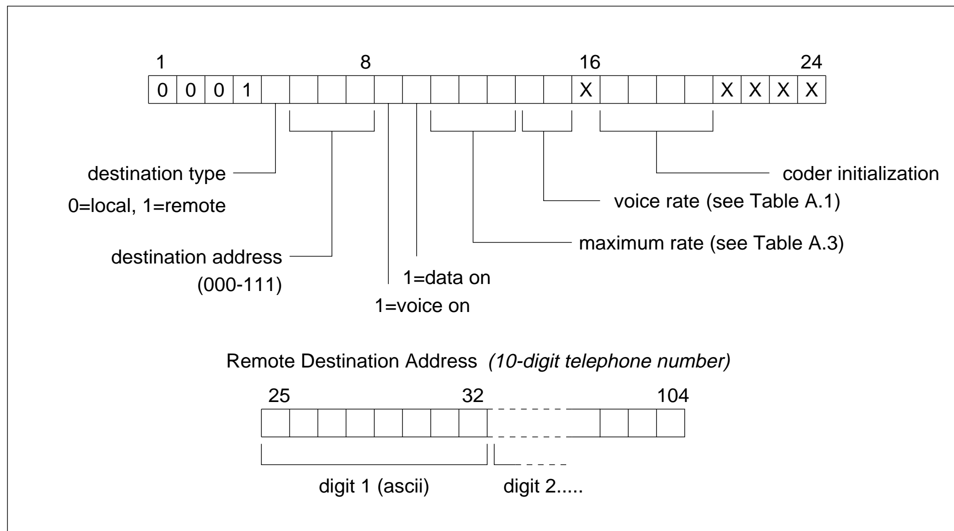

Figure A-1: CONNECTION REQUEST

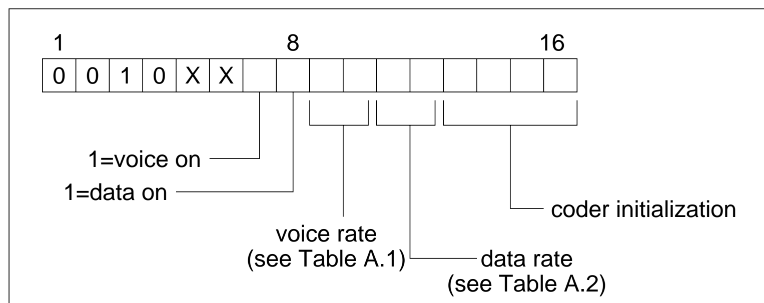

Figure A-2: CONNECTION INDICATION

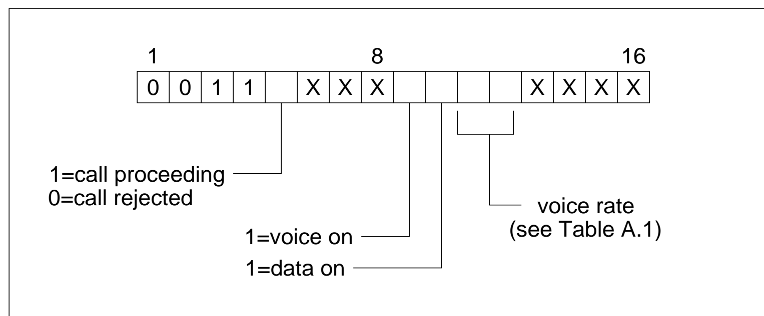

Figure A-3: CONNECTION RESPONSE

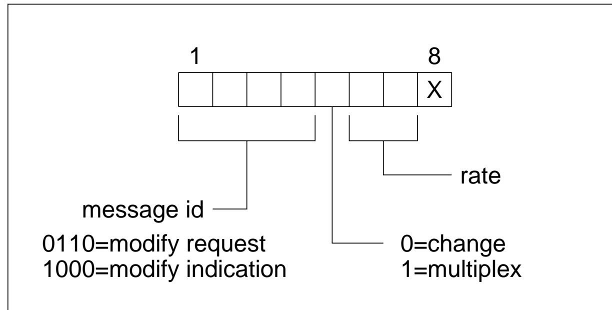

Figure A-4: MODIFY REQUEST/INDICATION

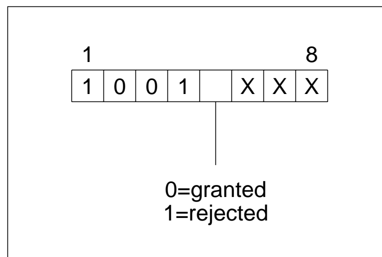

Figure A-5: MODIFY RESPONSE

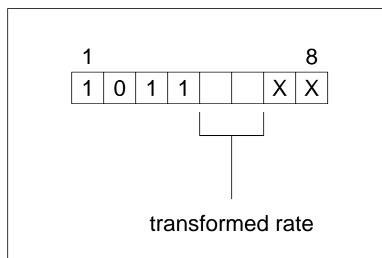

Figure A-6: TRANSFORM

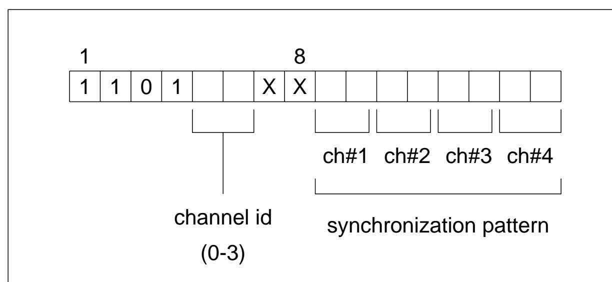

Figure A-7: OPEN CHANNEL

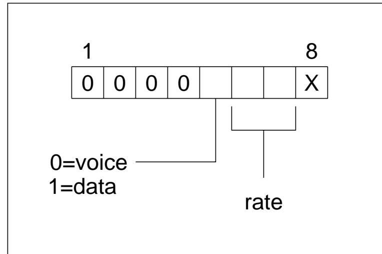

Figure A-8: MODIFY SOURCE

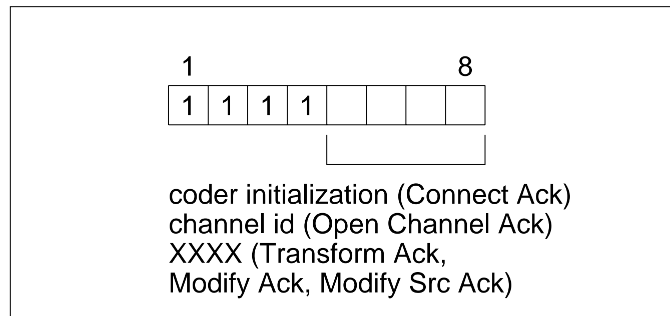

Figure A-9: ACKNOWLEDGEMENT

BIT PATTERN	VOICE RATE
00	2400 bps
01	4800 bps
10	9600 bps
11	19200 bps

Table A.1: Voice rates

BIT PATTERN	DATA RATE
00	2400 bps
01	4800 bps
10	9600 bps
11	illegal

Table A.2: Data rates

BIT PATTERN	MAXIMUM CONNECTION RATE
000	2400 bps
001	4800 bps
010	7200 bps
011	9600 bps
100	12000 bps
101	14400 bps
110	16800 bps
111	19200 bps

Table A.3: Maximum connection rates

MESSAGE TYPE	ID	PARAMETERS
Connection Request	0001	Destination (local/remote), Destination Address, Traffic Type (Voice/Data/Multiplexed), Data Rate, Coder Initialization
Connection Indication	0010	Traffic Type (Voice/Data/Multiplexed), Voice Rate, Data Rate, Coder Initialization
Connection Response	0011	Proceeding/Refused, Traffic Type, Voice Rate
Alerting	0100	–
Disconnect	0101	–
Modify Request	0110	Change/Multiplex, Rate
Modify Indication	1000	Change/Multiplex, Rate
Modify Response	1001	Granted/Rejected
Call Modified	1010	–
Transform Rate	1011	New Rate
Rate Transformed	1100	–
Open Channel	1101	Channel Id, Synchronization pattern
Start Channel	1110	–
Modify Source	0000	Voice/Data, Rate
Null Message	0111	–
ACK	1111	–

Table A.4: Control Message IDs and parameters

Appendix B

Parser Source Code

B.1 C Programs

```
/*
 ***************
 * Name: main.c
 ***************
 *
 * Purpose: parse the configuration file and
 * insert control messages in coded speech files
 */


#include<stdio.h>
#include<string.h>
#include "common.h"
#include "event.h"
#include "queue.h"

#define NULL_MSG 0x7e

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

  signaling_event *event;
  queue_item *q_head, *q_tail;
  unsigned char msgs_array[NUM_PORTS][MAX_MSGS];
  char *cmd_line, line[200], cmd_comp[3][20];
  int i, j, ret_val, line_num=0;
  FILE *fp_config;


  if (argc != 2) {
    printf("usage: %s <configuration file>\n", argv[0]);
    exit(-1);
  }

  fp_config = fopen(argv[1], "r");

  /*
   * Fill message array with NULL messages
   */

  for(i=0; i<NUM_PORTS; i++) {
```

```
    for(j=0; j<MAX_MSGS; j++) {
      msgs_array[i][j] = NULL_MSG;
    }
  }

  event = (signaling_event *)malloc(sizeof(signaling_event));
  init_event(event);

  q_head = (queue_item *)malloc(sizeof(queue_item));
  q_tail = q_head;

  while ((cmd_line = fgets(line, 200, fp_config)) != (char *)0) {

    line_num++;

    /*
     * Parse the command line
     */
    parse_cmd_line(cmd_line, cmd_comp);

    ret_val = store_event(cmd_comp, event);

    if (ret_val < 0) {

      /*
       * remove the end-of-line character
       */
      cmd_line[strlen(cmd_line)-1] = '\0';

      fprintf(stderr, "error in line %d: '%s'\n", line_num, cmd_line);
      exit(-1);
    }
    else if (ret_val > 0) {
      /*
       * end of current event
       */
      q_tail = (queue_item *)insert_msgs(event, msgs_array, q_tail);
    }
  }

#ifdef DEBUG
  printf("startup file parsed\n");
```

```
  printf("inserting control messages in speech files\n");
#endif

  insert_speech(msgs_array, q_head);
}
```

```
/*
 ***************
 * Name: init.c
 ***************
 *
 * Purpose: routines to initialize state and
 * event structs
 */


#include<stdio.h>
#include "common.h"


#include "state.h"

void init_state(port_state state[], int num_ports) {

  int port;

  /*
   * Initialize state struct for each port
   */

  for (port = 0; port < num_ports; port++) {
    state[port].active = FALSE;
    state[port].talk = FALSE;
    state[port].voice_frame = 0;
    state[port].fp_speech = (FILE *)0;
    state[port].fp_out = (FILE *)0;
  }
}


#include "event.h"

void init_event(signaling_event *event) {

  /*
   * Initialize the event struct to avoid processing
   * parameters from previous events
   */
```

```
  event→event_time = (int *)0;
  event→event_type = (char *)0;
  event→port = (int *)0;
  event→dest = (int *)0;
  event→dest_type = (int *)0;
  event→traffic_type = (int *)0;
  event→max_rate = (int *)0;
  event→voice_conn_rate = (int *)0;
  event→stc_init = (char *)0;
  event→tel_no = (char *)0;
  event→grant_reject = (int *)0;
  event→modem_code = (char *)0;
  event→change_mux = (int *)0;
  event→modify_rate = (int *)0;
  event→active = (int *)0;
  event→talk = (int *)0;
  event→voice_frame = (int *)0;
  event→speechfile = (char *)0;
  event→outputfile = (char *)0;
}


#include "queue.h"

void init_queue(queue_item *item) {

  /*
   * Initialize the next item in the event countdown queue
   */

  item→next = (queue_item *)0;
  item→time = (int *)0;
  item→port = (int *)0;
  item→active = (int *)0;
  item→talk = (int *)0;
  item→voice_frame = (int *)0;
  item→speechfile = (char *)0;
  item→outputfile = (char *)0;
}
```

```
/*
 ****************
 * Name: parse.c
 ****************
 *
 * Purpose: parse a command line and find the components
 * (X,operator,Y) of the command.  A command is of the
 * form: "X<space>operator<space>Y<newline>"
 */


#include<stdio.h>
#include<string.h>

#define SEPARATOR  " "


void parse_cmd_line(char *cmd_line, char *cmd_comp[3]) {

  char *cmd, *newline_ptr;
  int next = 0;


  cmd = (char *)malloc(strlen(cmd_line)+1);
  strcpy(cmd, cmd_line);

#ifdef DEBUG
  printf("----->%s<-----\n", cmd);
#endif

  /*
   * Find the components of the command
   */

  cmd_comp[next] = (char *)malloc(strlen(cmd_line)+1);
  cmd_comp[next] = strtok(cmd, SEPARATOR);

  do {
#ifdef DEBUG
    printf ("cmd_comp = --->%s<---\n", cmd_comp[next]);
#endif
    next++;
```

```
  } while ((cmd_comp[next] = strtok((char *)0,  SEPARATOR)) ≠ (char *)0);

  /*
   * Remove the "\n" from the last command component
   */

  if ((newline_ptr = strchr(cmd_comp[2], '\n')) ≠ NULL) {
    *newline_ptr = '\0';
  }
}
```

```
/*
 ****************
 * Name: event.c
 ****************
 *
 * Purpose: store the command parameters in the event
 * struct
 *
 * Return values:
 *   -1 => failure
 *    0 => success
 *    1 => end of event
 */


#include<stdio.h>
#include<string.h>
#include "event.h"


int store_event(char *cmd_comp[3], signaling_event *event) {

  int ret_val;


  /*
   * Check for NULL command
   */

  if (cmd_comp[0] == (char *)0) {
    /*
     * No command, so do nothing
     */
#ifdef DEBUG
    printf("did nothing\n");
#endif
    return(0);
  }


  /*
   * Check for end of event
```

```
   */

  if (strcmp(cmd_comp[0], "\n") == 0) {
    if (event->event_type != NULL) {
      /*
       * End of current event
       */
#ifdef DEBUG
      printf("event done\n");
#endif
      return(1);
    }
    return(0);
  }


  /*
   * Store event parameter in event struct
   */

  if (strcmp(cmd_comp[0], "time") == 0) {
    event->event_time = (int *)malloc(sizeof(int));
    *event->event_time = atoi(cmd_comp[2]);
  }

  else if (strcmp(cmd_comp[0], "event") == 0) {
    event->event_type = (char *)malloc(strlen(cmd_comp[2]+1));
    strcpy(event->event_type, cmd_comp[2]);
  }

  else {
    if ((ret_val = get_params(event, cmd_comp)) < 0) {
      return(-1);
    }
  }
  return(0);
}
```

```
/*
 ****************
 * Name: params.c
 ****************
 *
 * Purpose: parse the event parameters and store the
 * parameter values in the event struct
 */


#include <stdio.h>
#include<string.h>
#include "common.h"
#include "event.h"
#include "params.h"


int get_params(signaling_event *event, char *param_comp[]) {

  /*
   * Macro to print error message for invalid parameter
   * value
   */

#define printerr  fprintf(stderr,"invalid parameter value:'%s'\n", param_value); \
              return(-1)


  int param_index = 0, rate_index = 0;
  char param_name[20], param_value[20];
  char *workstring, *separators, *str_pointer, *tmp1, *tmp2;


  strcpy(param_name, param_comp[0]);
  strcpy(param_value, param_comp[2]);

  while (strcmp(param_name, params_array[param_index]) != 0) {
    param_index++;
    if (param_index == MAX_PARAMS) {
      fprintf(stderr, "invalid param type: '%s'\n", param_name);
      return(-1);
    }
```

```
  }

#ifdef DEBUG
  printf ("param_name = %s\n", param_name);
  printf ("param_value = %s\n", param_value);
#endif


  switch(param_index) {

  case PORT:
    event->port = (int *)malloc(sizeof(int));
    *event->port = atoi(param_value);

#ifdef DEBUG
    printf("port = 0x%x\n", *event->port);
#endif

    break;

  case DEST:
    event->dest = (int *)malloc(sizeof(int));
    *event->dest = atoi(param_value);

#ifdef DEBUG
    printf("destination = 0x%x\n", *event->dest);
#endif

    break;

  case DEST_TYPE:
    event->dest_type = (int *)malloc(sizeof(int));
    if (strcmp(param_value, "INTERNAL") == 0) {
      *event->dest_type = 0;
    }
    else if (strcmp(param_value, "EXTERNAL") == 0) {
      *event->dest_type = 1;
    }
    else {
      printerr;
    }
    break;
```

```
case TRAFFIC_TYPE:
  event→traffic_type = (int *)malloc(sizeof(int));
  if (strcmp(param_value, "VOICE") == 0) {
    *event→traffic_type = 2;
  }
  else if (strcmp(param_value, "DATA") == 0) {
    *event→traffic_type = 1;
  }
  else if (strcmp(param_value, "MULTIPLEXED") == 0) {
    *event→traffic_type = 3;
  }
  else {
    printerr;
  }
  break;

case MAX_RATE:
  event→max_rate = (int *)malloc(sizeof(int));
  while (strcmp(param_value, total_rates[rate_index]) ≠ 0) {
    rate_index++;
    if (rate_index > MAX_TOTAL_RATES) {
      printerr;
    }
  }
  *event→max_rate = rate_index;
  break;

case VOICE_CONN_RATE:
  event→voice_conn_rate = (int *)malloc(sizeof(int));
  while (strcmp(param_value, voice_data_rates[rate_index]) ≠ 0) {
    rate_index++;
    if (rate_index > MAX_VOICE_RATES) {
      printerr;
    }
  }
  *event→voice_conn_rate = rate_index;
  break;

case STC_INIT:
  event→stc_init = (char *)malloc(sizeof(char));
  *event→stc_init = (char)atoi(param_value);
```

```
  break;

case TEL_NO:
  workstring = param_value;
  separators = "()-";

  tmp1 = (char *)malloc(strlen(workstring)+1);
  tmp2 = (char *)malloc(strlen(workstring)+1);

  while (*separators ≠ '\0') {
    str_pointer = strchr(workstring, *separators);
    strcpy(tmp1, (str_pointer+1));
    *str_pointer++ = '\0';
    strcpy(tmp2, workstring);
    workstring = strcat(tmp2, tmp1);
    separators++;
  }

  event→tel_no = (char *)malloc(strlen(workstring)+1);
  str_pointer = event→tel_no;

  while (*workstring ≠ '\0') {
    *str_pointer++ = *workstring++;
  }
  *str_pointer = '\0';
  break;

case GRANT_REJECT:
  event→grant_reject = (int *)malloc(sizeof(int));
  if (strcmp(param_value, "GRANTED") == 0) {
    *event→grant_reject = 1;
  }
  else if (strcmp(param_value, "REJECTED") == 0) {
    *event→grant_reject = 0;
  }
  else {
    printerr;
  }
  break;

case MODEM_CODE:
  event→modem_code = (char *)malloc(sizeof(char));
```

```
    *event→modem_code = *param_value;
    break;

  case CHANGE_MUX:
    event→change_mux = (int *)malloc(sizeof(int));
    if (strcmp(param_value, "CHANGE") == 0)
      *event→change_mux = 0;
    else if (strcmp(param_value, "MULTIPLEX") == 0)
      *event→change_mux = 1;
    else {
      printerr;
    }
    break;

  case MODIFY_RATE:
    event→modify_rate = (int *)malloc(sizeof(int));
    while (strcmp(param_value, voice_data_rates[rate_index]) ≠ 0) {
      rate_index++;
      if (rate_index == MAX_VOICE_RATES) {
        printerr;
      }
    }
    *event→modify_rate = rate_index;
    break;

  case STATE:
    event→active = (int *)malloc(sizeof(int));
    if (strcmp(param_value, "ACTIVE") == 0) {
      *event→active = TRUE;
    }
    else if (strcmp(param_value, "INACTIVE") == 0) {
      *event→active = FALSE;
    }
    else {
      printerr;
    }
    break;

  case ACTIVITY:
    event→talk = (int *)malloc(sizeof(int));
    if (strcmp(param_value, "TALK") == 0) {
      *event→talk = TRUE;
```

```
    }
    else if (strcmp(param_value, "IDLE") == 0) {
      *event->talk = FALSE;
    }
    else {
      printerr;
    }
    break;

  case VOICE_FRAME:
    event->voice_frame = (int *)malloc(sizeof(int));
    *event->voice_frame = atoi(param_value);
    break;

  case SPEECHFILE:
    event->speechfile = (char *)malloc(strlen(param_value)+1);
    strcpy(event->speechfile, param_value);
    break;

  case OUTPUTFILE:
    event->outputfile = (char *)malloc(strlen(param_value)+1);
    strcpy(event->outputfile, param_value);
    break;
  }

  return(1);
}
```

```
/*
 **********************
 * Name: insert_msgs.c
 **********************
 *
 * Purpose: read the event struct and perform the
 * necessary action depending on the event type
 *   event type = control message
 *    => construct control message and insert into
 *       control message array
 *   event type = change port state
 *    => store new port state in event queue
 *
 * Return value: pointer to the current position
 * in the event queue
 */


#include <stdio.h>
#include "common.h"
#include "event.h"
#include "queue.h"
#include "msgs.h"


queue_item *insert_msgs(signaling_event *event, unsigned char msgs[][MAX_MSGS], queue_item
*q_ptr) {

  /*
   * Debugging printf macro
   */

#define dprintf(a) printf("time=%d port=%d %s=0x%x\n",tnow,port,a,msgs[port][tnow])


  unsigned char ctrl_byte, *ptr;
  int tnow, port;
  int bits_left, zeros = 0x00;


  tnow = *event->event_time;
  port = *event->port;
```

```
#ifdef DEBUG
  printf("event time = %d\n", *event->event_time);
  printf("event type = %s\n", event->event_type);
#endif


/*
 *
 *  CONNECTION REQUEST
 *
 */

  if (strcmp(event->event_type, "CONN_REQ") == 0) {

    /*
     * BYTE 1
     *   message id       = 4 bits
     *   destination type = 1 bit
     *   internal address = 3 bits
     */

    bits_left = pack_bits(CONN_REQ, nbits.msg_id, &msgs[port][tnow]);
    bits_left = pack_bits(*event->dest_type, nbits.dest_type, &msgs[port][tnow]);
    bits_left = pack_bits(*event->dest, nbits.dest_add, &msgs[port][tnow]);

#ifdef DEBUG
    dprintf("CONN_REQ_1");
#endif

    /*
     * BYTE 2
     *   traffic type = 2 bits
     *   maximum rate = 3 bits
     *   voice rate   = 2 bits
     *   XXXX         = 1 bit
     */

    tnow++;
    bits_left = pack_bits(*event->traffic_type, nbits.traffic_type, &msgs[port][tnow]);
    bits_left = pack_bits(*event->max_rate, nbits.max_rate, &msgs[port][tnow]);
    bits_left = pack_bits(*event->voice_conn_rate, nbits.voice_rate, &msgs[port][tnow]);
```

```
    bits_left = pack_bits(zeros, bits_left, &msgs[port][tnow]);

#ifdef DEBUG
    dprintf("CONN_REQ_2");
#endif

    /*
     * BYTE 3
     *   stc initialization = 4 bits
     *   XXXX               = 4 bits
     */

    tnow++;
    bits_left = pack_bits(*event→stc_init, nbits.coder_init, &msgs[port][tnow]);
    bits_left = pack_bits(zeros, bits_left, &msgs[port][tnow]);

#ifdef DEBUG
    dprintf("STC_INIT");
#endif

    /*
     * external address
     */

    tnow++;
    if (*event→dest_type == EXTERNAL) {
      ptr = event→tel_no;
      while (*ptr ≠ '\0') {
        msgs[port][tnow] = *ptr;
        ptr++;
        tnow++;
      }

#ifdef DEBUG
      printf("time=%d port=%d TEL NUMBER=%s\n", tnow, port, event→tel_no);
#endif
    }
  }


/*
 *
```

```
 * CONNECTION RESPONSE
 *
 */

 if (strcmp(event->event_type, "CONN_RESP") == 0) {

   /*
    * BYTE 1:
    *   message id                = 4 bits
    *   call proceeding/rejected = 1 bit
    *   XXXX                      = 3 bits
    */

   bits_left = pack_bits(CONN_RESP, nbits.msg_id, &msgs[port][tnow]);
   bits_left = pack_bits(*event->grant_reject, nbits.flag, &msgs[port][tnow]);
   bits_left = pack_bits(zeros, bits_left, &msgs[port][tnow]);

#ifdef DEBUG
   dprintf("CONN_RESP_1");
#endif

   if (*event->grant_reject == GRANTED) {

     /*
      * BYTE 2
      *   traffic type = 2 bits
      *   voice rate   = 2 bits
      *   XXXX         = 4 bits
      */

     tnow++;
     bits_left = pack_bits(*event->traffic_type, nbits.traffic_type, &msgs[port][tnow]);
     bits_left = pack_bits(*event->voice_conn_rate, nbits.voice_rate, &msgs[port][tnow]);
     bits_left = pack_bits(zeros, nbits.flag, &msgs[port][tnow]);

#ifdef DEBUG
     dprintf("CONN_RESP_2");
#endif
   }
 }
```

```
/*
 *
 *  CONNECT ACK
 *
 */

  if (strcmp(event->event_type, "CONN_ACK") == 0) {

    bits_left = pack_bits(CONN_ACK, nbits.msg_id, &msgs[port][tnow]);
    bits_left = pack_bits(*event->stc_init, nbits.coder_init, &msgs[port][tnow]);

#ifdef DEBUG
    dprintf("CONN_ACK");
#endif
  }


/*
 *
 *  ALERTING
 *
 */

  if (strcmp(event->event_type, "ALERTING") == 0) {

    bits_left = pack_bits(ALERTING, nbits.msg_id, &msgs[port][tnow]);
    bits_left = pack_bits(zeros, bits_left, &msgs[port][tnow]);

#ifdef DEBUG
    dprintf("ALERTING");
#endif
  }


/*
 *
 *  MODEM RESULT CODE
 *
 */

  if (strcmp(event->event_type, "MODEM_RESPONSE") == 0) {
```

```
   msgs[port][tnow] = *event->modem_code;

#ifdef DEBUG
   dprintf("MODEM_CODE");
#endif
 }


/*
 *
 *  OPEN CHANNEL ACK
 *
 */

 if (strcmp(event->event_type, "OPEN_CH_ACK") == 0) {

   bits_left = pack_bits(OPEN_CH_ACK, nbits.msg_id, &msgs[port][tnow]);
   bits_left = pack_bits(zeros, bits_left, &msgs[port][tnow]);

#ifdef DEBUG
   dprintf("OPEN_CH_ACK");
#endif
 }


/*
 *
 *  MODIFY
 *
 */

 if (strcmp(event->event_type, "MODIFY_REQ") == 0) {

   /*
    * Modify Request
    *   message id = 4 bits
    *   change/mux = 1 bit
    *   rate       = 2 bits
    *   XXXX       = 1 bit
    */

   bits_left = pack_bits(MODIFY_REQ, nbits.msg_id, &msgs[port][tnow]);
```

```
    bits_left = pack_bits(*event→change_mux, nbits.change_mux, &msgs[port][tnow]);
    bits_left = pack_bits(*event→modify_rate, nbits.modify_rate, &msgs[port][tnow]);
    bits_left = pack_bits(zeros, bits_left, &msgs[port][tnow]);

#ifdef DEBUG
    dprintf("MODIFY_REQ");
#endif
  }

/*
 *
 *  START_MODIFY
 *
 */

  if (strcmp(event→event_type, "START_MODIFY") == 0) {

    bits_left = pack_bits(START_MODIFY, nbits.msg_id, &msgs[port][tnow]);
    bits_left = pack_bits(zeros, bits_left, &msgs[port][tnow]);

#ifdef DEBUG
    dprintf("START_MODIFY");
#endif
  }


/*
 *
 *  MODIFY ACK
 *
 */

  if (strcmp(event→event_type, "MODIFY_ACK") == 0) {

    bits_left = pack_bits(MODIFY_ACK, nbits.msg_id, &msgs[port][tnow]);
    bits_left = pack_bits(zeros, bits_left, &msgs[port][tnow]);

#ifdef DEBUG
    dprintf("MODIFY_ACK");
#endif
  }
```

```
/*
 *
 *  TRANSFORM ACK
 *
 */

  if (strcmp(event->event_type, "TRANSFORM_ACK") == 0) {

    bits_left = pack_bits(TRANSFORM_ACK, nbits.msg_id, &msgs[port][tnow]);
    bits_left = pack_bits(zeros, bits_left, &msgs[port][tnow]);

#ifdef DEBUG
    dprintf("TRANSFORM_ACK");
#endif
  }


/*
 *
 *  DISCONNECT
 *
 */

  if (strcmp(event->event_type, "DISCONNECT") == 0) {

    bits_left = pack_bits(DISCONNECT, nbits.msg_id, &msgs[port][tnow]);
    bits_left = pack_bits(zeros, bits_left, &msgs[port][tnow]);

#ifdef DEBUG
    dprintf("DISCONNECT");
#endif
  }


/*
 *
 *  CHANGE STATE
 *
 */

#define intalloc  (int *)malloc(sizeof(int))
```

```
if (strcmp(event→event_type, "CHANGE_STATE") == 0) {

  /*
   * Schedule the state change by adding an event to the
   * event countdown queue
   */

  if (q_ptr→time ≠ NULL) {
    /*
     * Allocate memory for next queue item
     */
    q_ptr→next = (queue_item *)malloc(sizeof(queue_item));
    q_ptr = q_ptr→next;
  }

  q_ptr→time = intalloc;
  *q_ptr→time = tnow;

  q_ptr→port = intalloc;
  *q_ptr→port = port;

  if (event→active ≠ NULL) {
    q_ptr→active = intalloc;
    *q_ptr→active = *event→active;
  }
  else {
    q_ptr→active = NULL;
  }
  if (event→talk ≠ NULL) {
    q_ptr→talk = intalloc;
    *q_ptr→talk = *event→talk;
  }
  else {
    q_ptr→talk = NULL;
  }
  if (event→voice_frame ≠ NULL) {
    q_ptr→voice_frame = intalloc;
    *q_ptr→voice_frame = *event→voice_frame;
  }
  else {
    q_ptr→voice_frame = NULL;
```

```
    }
    if (event→speechfile ≠ NULL) {
      q_ptr→speechfile = (char *)malloc(strlen(event→speechfile)+1);
      strcpy(q_ptr→speechfile, event→speechfile);
    }
    else {
      q_ptr→speechfile = NULL;
    }
    if (event→outputfile ≠ NULL) {
      q_ptr→outputfile = (char *)malloc(strlen(event→outputfile)+1);
      strcpy(q_ptr→outputfile, event→outputfile);
    }
    else {
      q_ptr→outputfile = NULL;
    }
    q_ptr→next = NULL;
  }


  /*
   * initialize event struct to avoid processing previous
   * events
   */

  init_event(event);

  return(q_ptr);
}
```

```
/*
 ****************
 * Name: state.c
 ****************
 *
 * Purpose: store the configuration for each port in the
 * state struct
 */


#include<stdio.h>
#include<string.h>
#include "common.h"
#include "state.h"


void store_state(char *cmd_comp[3], port_state state[]) {

  char *port, filename[20];
  static int port_num;


  if (strcmp(cmd_comp[0], "PORT") == 0) {
    port_num = atoi(cmd_comp[2]);
    state[port_num].active = TRUE;
  }

  else if (strcmp(cmd_comp[0], "STATE") == 0) {
    if (strcmp(cmd_comp[2], "TALK") == 0) {
      state[port_num].talk = TRUE;
    }
  }

  else if (strcmp(cmd_comp[0], "VOICE_FRAME") == 0) {
    state[port_num].voice_frame = atoi(cmd_comp[2]);
  }

  else if (strcmp(cmd_comp[0], "speechfile") == 0) {
    strcpy(filename, cmd_comp[2]);
    state[port_num].fp_speech = fopen(filename, "r");
    printf("speech file = %s \n", filename);
  }
```

```
  else if (strcmp(cmd_comp[0], "outputfile") == 0) {
    strcpy(filename, cmd_comp[2]);
    state[port_num].fp_out = fopen(filename, "w");
    printf("output file = %s\n", filename);
  }
}
```

```
/*
 ***********************
 * Name: insert_speech.c
 ***********************
 *
 * Purpose: This routine reads the control message array
 * and inserts control messages in the input speech
 * files. It also examines the event countdown queue and
 * modifies the port states if required.
 */


#include<stdio.h>
#include "common.h"
#include "queue.h"
#include "state.h"


/*
 * Maximum length speech frame
 *   = 20ms speech frame at 19.2 kb/s
 *   = 384 bits
 *   = 48 bytes
 */

#define BUFFERLENGTH  48


void insert_speech(unsigned char msgs[][MAX_MSGS], queue_item *q_head) {

 int port, tnow, len;
 port_state state[NUM_PORTS];
 unsigned char buffer[BUFFERLENGTH];
 FILE *fileptr;


 init_state(state, NUM_PORTS);

 for (tnow=0; tnow < MAX_MSGS; tnow++) {

   /*
    * Check the event queue and update port states if
```

```
 * necessary
 */

while((q_head ≠ NULL) && (*q_head→time == tnow)) {

  if (q_head→active ≠ NULL) {
    state[*q_head→port].active = *q_head→active;
  }

  if (q_head→talk ≠ NULL) {
    state[*q_head→port].talk = *q_head→talk;
  }

  if (q_head→voice_frame ≠ NULL) {
    state[*q_head→port].voice_frame = *q_head→voice_frame;
  }

  if (q_head→speechfile ≠ NULL) {
    if ((fileptr = fopen(q_head→speechfile, "r")) == NULL) {
      printf("file '%s' could not be opened for reading\n", q_head→speechfile);
      exit(−1);
    }
    else {
      state[*q_head→port].fp_speech = fileptr;
    }
  }

  if (q_head→outputfile ≠ NULL) {
    if ((fileptr = fopen(q_head→outputfile, "w")) == NULL) {
      printf("file '%s' could not be opened for writing\n", q_head→outputfile);
      exit(−1);
    }
    else {
      state[*q_head→port].fp_out = fileptr;
    }
  }

  q_head = q_head→next;
}


for (port=0; port < NUM_PORTS; port++) {
```

```
      if (state[port].active) {
        if (state[port].talk) {
          /*
           * If port is in the "talk" state, check for EOF on speech
           * file
           */
          if (len = fread(&buffer,1,state[port].voice_frame,state[port].fp_speech) > 0) {
            /*
             * Read control byte from array and write to output file
             */
            fwrite(&msgs[port][tnow],1,1,state[port].fp_out);
            /*
             * Read speech frame from the input file and write to
             * output file
             */
            fwrite(&buffer,1,state[port].voice_frame,state[port].fp_out);
          }
          else {
            /*
             * If EOF is reached on the speech file, terminate control
             * message file
             */
            printf("EOF reached on port %d\n", port);
            state[port].active = FALSE;
            fclose(state[port].fp_speech);
            fclose(state[port].fp_out);
          }
        }
        else {
          /*
           * If port is not in the "talk" state, write only the
           * control byte
           */
          fwrite(&msgs[port][tnow],1,1,state[port].fp_out);
        }
      }
    }
  }
}
```

```
/*
 *******************
 * Name: pack_bits.c
 *******************
 *
 * Purpose: this routine takes parameters (param) of given
 * length (index bits) and packs them into control
 * bytes. The resulting control byte is written to chptr.
 *
 * Return value: the number of bits left in the current byte
 */


#include<stdio.h>
#include<math.h>

#define WORD 8

static int count = 0;
static int bits_left_index, bits_left_count;
static char bitpack;
static short masks[] = { 0, 0x01, 0x03, 0x07, 0x0f,
                         0x01f, 0x03f, 0x07f, 0x0ff,
                         0x01ff, 0x03ff, 0x07ff, 0x0fff,
                         0x01fff, 0x03fff, 0x07fff, 0x0ffff };


int pack_bits(short param, int index, char *chptr){

  short part1, part2;


  bits_left_count = WORD - count;

  if ( index < bits_left_count ) {
    bitpack = bitpack<<index;
    bitpack += param;
    count += index;

  } else if ( bits_left_count == index) {

    bitpack = bitpack<<index;
```

```
    bitpack += param;

    *chptr = bitpack;
    count = 0;
    bitpack = 0;

  } else {
    bits_left_index = index − bits_left_count;

    part1 = (param≫bits_left_index) & masks[bits_left_count];

    bitpack = bitpack≪bits_left_count;
    bitpack += part1;

    *chptr = bitpack;
    bitpack=0;

    if (bits_left_index ≥ WORD) {

      bits_left_count = WORD;

      bits_left_index = bits_left_index − bits_left_count;

      part1 = (param≫bits_left_index) & masks[bits_left_count];

      bitpack = bitpack≪bits_left_count;
      bitpack += part1;

      *chptr = bitpack;
    }

    bitpack = param & masks[bits_left_index];
    count = bits_left_index;

  }
  bits_left_count = WORD − count;
  return(bits_left_count);
}
```

B.2 Header Files

```
/****************
 * Name: common.h
 ****************/


#define NUM_PORTS  8
#define MAX_MSGS   6000

#define TRUE   1
#define FALSE  0



/*
 ***************
 * Name: event.h
 ***************
 *
 * Defines the data structure used to store
 * signaling events and their parameters
 * (event struct)
 */


typedef struct {
  int *event_time;
  char *event_type;
  int *port;
  /*
   * parameters for control messages
   */
  int *dest;
  int *dest_type;
  int *traffic_type;
  int *max_rate;
  int *voice_conn_rate;
  char *stc_init;
  char *tel_no;
  int *grant_reject;
  char *modem_code;
```

```
  int *change_mux;
  int *modify_rate;
  /*
   * parameters for port state
   */
  int *active;
  int *talk;
  int *voice_frame;
  char *speechfile;
  char *outputfile;
} signaling_event;


/*
 ***************
 * Name: queue.h
 ***************
 *
 * Defines the structure describing an element
 * in the event queue
 */


struct Queue_Item {
  struct Queue_Item *next;
  int *time;
  int *port;
  int *active;
  int *talk;
  int *voice_frame;
  char *speechfile;
  char *outputfile;
};

typedef struct Queue_Item queue_item;


/*
 ***************
 * Name: state.h
 ***************
```

```
 *
 * Structure for storing the current state of each port
 */

typedef struct {
  int active;
  int talk;
  int voice_frame;
  FILE *fp_speech;
  FILE *fp_out;
} port_state;


/*
 ****************
 * Name: params.h
 ****************
 *
 * Header file defining control message
 * parameters
 */

#define PORT            0
#define DEST            1
#define DEST_TYPE       2
#define TRAFFIC_TYPE    3
#define MAX_RATE        4
#define VOICE_CONN_RATE 5
#define STC_INIT        6
#define TEL_NO          7
#define GRANT_REJECT    8
#define MODEM_CODE      9
#define CHANGE_MUX      10
#define MODIFY_RATE     11
#define STATE           12
#define ACTIVITY        13
#define VOICE_FRAME     14
#define SPEECHFILE      15
#define OUTPUTFILE      16
```

```
#define TRUE   1
#define FALSE  0

#define MAX_PARAMS        17
#define MAX_TOTAL_RATES    7
#define MAX_VOICE_RATES    3


static char *params_array[] = { "PORT", "DEST", "DEST_TYPE", "TRAFFIC_TYPE",
                        "MAX_RATE", "VOICE_CONN_RATE", "STC_INIT",
                        "TEL_NO", "GRANT_REJECT", "MODEM_CODE",
                        "CHANGE_MUX", "MODIFY_RATE", "STATE",
                        "ACTIVITY", "VOICE_FRAME", "SPEECHFILE",
                        "OUTPUTFILE" };

static char *total_rates[] = { "2400", "4800", "7200", "9600",
                       "12000", "14400", "16800", "19200" };

static char *voice_data_rates[] = { "2400", "4800", "9600", "19200" };


/*
 ***************
 * Name: msgs.h
 ***************
 *
 * Header file defining Message IDs and
 * bit allocations for control messages
 */


#define EXTERNAL  1
#define INTERNAL  0

#define GRANTED   1
#define REJECTED  0


/*
 * Structure defining number of bits for message
 * components
```

```
 */

struct {
  int msg_id;
  int dest_type;
  int dest_add;
  int traffic_type;
  int max_rate;
  int voice_rate;
  int coder_init;
  int change_mux;
  int modify_rate;
  int flag;
} nbits = { 4, 1, 3, 2, 3, 2, 4, 1, 2, 1 };


/*
 * Message IDs
 */

#define NIL            0x07
#define CONN_REQ       0x01
#define CONN_IND       0x02
#define CONN_RESP      0x03
#define ALERTING       0x04
#define CONN_ACK       0x0f
#define DISCONNECT     0x05
#define MODIFY_REQ     0x06
#define MODIFY_IND     0x08
#define MODIFY_RESP    0x09
#define MODIFY_ACK     0x0f
#define START_MODIFY   0x0a
#define TRANSFORM      0x0b
#define TRANSFORM_ACK  0x0f
#define START_XFORM    0x0c
#define OPEN_CH        0x0d
#define OPEN_CH_ACK    0x0f
#define START_CH       0x0e
#define MODIFY_SRC     0x00
#define MODIFY_SRC_ACK 0x0f
```

B.3 Sample Configuration File

The following is a sample configuration file for a 3-stage rate translation scenario.

```
time = 0
event = CHANGE_STATE
PORT = 0
STATE = ACTIVE
ACTIVITY = IDLE
OUTPUTFILE = port0.out
SPEECHFILE = sona.19200.bits

time = 0
event = CHANGE_STATE
PORT = 1
STATE = ACTIVE
ACTIVITY = IDLE
OUTPUTFILE = port1.out
SPEECHFILE = sona.19200.bits

time = 0
event = CHANGE_STATE
PORT = 2
STATE = ACTIVE
ACTIVITY = IDLE
OUTPUTFILE = port2.out
SPEECHFILE = sona.19200.bits

time = 0
event = CHANGE_STATE
PORT = 3
STATE = ACTIVE
ACTIVITY = IDLE
OUTPUTFILE = port3.out
SPEECHFILE = sona.19200.bits

time = 0
event = CONN_REQ
PORT = 0
DEST = 2
DEST_TYPE = EXTERNAL
TRAFFIC_TYPE = VOICE
```

```
MAX_RATE = 19200
VOICE_CONN_RATE = 19200
STC_INIT = 5
TEL_NO = (913)864-7738

time = 13
event = MODEM_RESPONSE
PORT = 1
MODEM_CODE = 1

time = 14
event = OPEN_CH_ACK
PORT = 1

time = 14
event = CHANGE_STATE
PORT = 4
STATE = ACTIVE
ACTIVITY = IDLE
OUTPUTFILE = port4.out
SPEECHFILE = sona.19200.bits

time = 14
event = CONN_RESP
PORT = 4
GRANT_REJECT = GRANTED
TRAFFIC_TYPE = VOICE
VOICE_CONN_RATE = 19200

time = 16
event = CONN_ACK
PORT = 4
STC_INIT = 9

time = 17
event = CHANGE_STATE
PORT = 0
ACTIVITY = TALK
VOICE_FRAME = 48

time = 17
event = CHANGE_STATE
```

```
PORT = 4
ACTIVITY = TALK
VOICE_FRAME = 48

time = 1683
event = CONN_REQ
PORT = 2
DEST = 3
DEST_TYPE = EXTERNAL
TRAFFIC_TYPE = VOICE
MAX_RATE = 19200
VOICE_CONN_RATE = 19200
STC_INIT = 5
TEL_NO = (913)864-7738

time = 1696
event = TRANSFORM_ACK
PORT = 4

time = 1697
event = OPEN_CH_ACK
PORT = 1

time = 1697
event = CHANGE_STATE
PORT = 5
STATE = ACTIVE
ACTIVITY = IDLE
OUTPUTFILE = port5.out
SPEECHFILE = sona.19200.bits

time = 1697
event = CONN_RESP
PORT = 5
GRANT_REJECT = GRANTED
TRAFFIC_TYPE = VOICE
VOICE_CONN_RATE = 19200

time = 1699
event = CONN_ACK
PORT = 5
STC_INIT = 9
```

```
time = 1700
event = CHANGE_STATE
PORT = 2
ACTIVITY = TALK
VOICE_FRAME = 48

time = 1700
event = CHANGE_STATE
PORT = 5
ACTIVITY = TALK
VOICE_FRAME = 48

time = 3333
event = CONN_REQ
PORT = 3
DEST = 1
DEST_TYPE = EXTERNAL
TRAFFIC_TYPE = VOICE
MAX_RATE = 19200
VOICE_CONN_RATE = 19200
STC_INIT = 7
TEL_NO = (913)864-7738

time = 3346
event = TRANSFORM_ACK
PORT = 4

time = 3347
event = OPEN_CH_ACK
PORT = 1

time = 3347
event = CHANGE_STATE
PORT = 6
STATE = ACTIVE
ACTIVITY = IDLE
OUTPUTFILE = port6.out
SPEECHFILE = sona.19200.bits

time = 3347
event = CONN_RESP
```

```
PORT = 6
GRANT_REJECT = GRANTED
TRAFFIC_TYPE = VOICE
VOICE_CONN_RATE = 19200

time = 3349
event = CONN_ACK
PORT = 6
STC_INIT = 9

time = 3350
event = CHANGE_STATE
PORT = 3
ACTIVITY = TALK
VOICE_FRAME = 48

time = 3350
event = CHANGE_STATE
PORT = 6
ACTIVITY = TALK
VOICE_FRAME = 48
```

Appendix C

Simulation Source Code

C.1 C Programs

```
/*
 * NAME: main.c
 *
 * PURPOSE:
 *     Simulate the function of the control processor
 *     in the Adaptive Voice/Data Switch
 */


#include <stdio.h>
#include "scp.h"


int more_messages()
{
  int i, flag;

  flag = 0;
  for (i = 0; i <= LOGICAL_PORTS; i++)
    flag = flag || !feof(state[i].fp_in);
  return(flag);
}



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

  int time, i, j, ch;
  unsigned char byte_in;
  char indir[64], outdir[64];
  char *in[8], *out[8], *cntrl[8], *speech[8], *xform[8];


  if(argc < 2) {
    printf("usage: switch.out <inputdir> <outputdir>\n");
    exit(1);
  }

  strcpy(indir, argv[1]);
  strcpy(outdir, argv[2]);
```

```
/*
 * Initialization
 */

in[0] = "/sp_port_0";
in[1] = "/sp_port_1";
in[2] = "/sp_port_2";
in[3] = "/sp_port_3";
in[4] = "/sp_port_4";
in[5] = "/sp_port_5";
in[6] = "/sp_port_6";
in[7] = "/sp_port_7";

out[0] = "/outfile0";
out[1] = "/outfile1";
out[2] = "/outfile2";
out[3] = "/outfile3";
out[4] = "/outfile4";
out[5] = "/outfile5";
out[6] = "/outfile6";
out[7] = "/outfile7";

cntrl[0] = "/control0";
cntrl[1] = "/control1";
cntrl[2] = "/control2";
cntrl[3] = "/control3";
cntrl[4] = "/control4";
cntrl[5] = "/control5";
cntrl[6] = "/control6";
cntrl[7] = "/control7";

speech[0] = "/speech0";
speech[1] = "/speech1";
speech[2] = "/speech2";
speech[3] = "/speech3";
speech[4] = "/speech4";
speech[5] = "/speech5";
speech[6] = "/speech6";
speech[7] = "/speech7";
```

```
xform[0] = "/drop1";
xform[1] = "/drop2";
xform[2] = "/drop3";
xform[3] = "/drop4";


for (i = 0; i ≤ MAX_PORTS; i++) {
  strcpy(state[i].in_file, indir);
  strcat(state[i].in_file, in[i]);
  strcpy(state[i].out_file, outdir);
  strcat(state[i].out_file, out[i]);
  strcpy(state[i].cntrl, outdir);
  strcat(state[i].cntrl, cntrl[i]);
  strcpy(state[i].speech, outdir);
  strcat(state[i].speech, speech[i]);
  for(j=0; j < MAX_CH; j++) {
    strcpy(state[i].xform_file[j], outdir);
    strcat(state[i].xform_file[j], xform[j]);
  }
}

for (i = 0; i ≤ MAX_PORTS; i++) {
  state[i].alive = ON;
  state[i].modem = OFF;
  state[i].idle = ON;
  state[i].connect = OFF;
  state[i].busy = OFF;
  state[i].xform = OFF;
  state[i].data_on = OFF;
  state[i].voice_on = OFF;
  state[i].req_flag = OFF;
  state[i].master = i;
  state[i].ch_id = 0;
  state[i].frame_count = 0;
  state[i].eof_flag = 0;
  state[i].idle_to_busy = OFF;
  state[i].xform_count = 0;
}

state[MODEM].modem = ON;

for (i = 0; i ≤ PHYSICAL_PORTS; i++) {
```

```
    state[i].fp_in = fopen(state[i].in_file, "r");
    state[i].fp_out = fopen(state[i].out_file, "w");
    state[i].fp_speech = fopen(state[i].speech, "w");
    state[i].fp_cntrl = fopen(state[i].cntrl, "w");
  }

#ifdef VERBOSE
  printf("finished init\n");
#endif


  /*
   * Main loop to read control messages from ports
   * 0x7e => NULL message
   */

  time = 0;
  LOGICAL_PORTS = 3;

  while(more_messages()) {

    for(i = 0; i ≤ LOGICAL_PORTS; i++) {

      if((ch = fgetc(state[i].fp_in)) ≠ EOF) {
        byte_in = (char)ch;

        if(state[i].frame_count == 0) {

          if((byte_in & 0xff) ≠ 0x7e) {

#ifdef VERBOSE
            printf("time slot #%d \t cntrl byte = 0x%x \n", time, (int)byte_in);
#endif

            configure(i, byte_in);
            fputc(byte_in, state[i].fp_cntrl);
          }

          if(state[i].busy) {
            if(state[i].idle_to_busy == OFF) {
              state[i].frame_count = state[i].voice_frame;
              for(j = 0; j < state[i].voice_frame; j++) {
```

```
          ch = (char)fgetc(state[i].fp_in);
          fputc(ch, state[state[i].talk_to].fp_speech);
          state[i].frame_count--;
         }
        }
        else
         state[i].idle_to_busy = OFF;
       }
      }
     }

     else {
      if(!state[i].eof_flag) {

#ifdef VERBOSE
       printf("Reached EOF on port %d\n", i);
#endif

       state[i].eof_flag = 1;
      }
     }
    }
    time++;
  }
}
```

```
/*
 * NAME: configure.c
 *
 * PURPOSE:
 *      Perform the appropriate action depending on the control byte
 *
 * INPUTS:
 *      port: port number
 *      cntrl_in: control byte received from port
 *
 * OUTPUT:
 *      The switch state (the "state" struct) is updated
 */


#include <stdio.h>
#include "scp.h"
#include "msgs.h"


void configure(int port, unsigned char cntrl_in) {

  char cntrl_out, new_traffic;
  int line_count, no_line_free, no_ckt_exists;
  int modify, other_end;
  int msg_id, bits_left;


  msg_id = unpack_bits(NBITS_MSG_ID, &cntrl_in);


  /*
   * IDLE state
   */

  if(state[port].idle) {
    if(msg_id == CONN_REQ) {
#ifdef VERBOSE
      printf("Received CONN_REQ on port %d\n", port);
#endif
      state[port].idle = OFF;
      state[port].connect = ON;
```

```
      state[port].req_flag = ON;
      state[port].req_count = 0;
    }
  }


  /*
   * CONNECT state
   * (establishing a connection)
   */

  if(state[port].connect) {
    if(state[port].req_flag == ON) {
      /*
       * Receiving request parameters
       */
      state[port].req_count++;
      switch(state[port].req_count)
        {
        case 1 :
          state[port].dest_type = unpack_bits(NBITS_DEST_TYPE, (char *)0);
          state[port].talk_to = unpack_bits(NBITS_DEST_ADD, (char *)0);
          state[port].conn_req_1 = cntrl_in;
#ifdef VERBOSE
          printf("Connection Request parameters:\n");
          printf("\tdestination type = %d\n", state[port].dest_type);
          printf("\tdestination address = %d\n", state[port].talk_to);
#endif
          if(state[port].dest_type == EXTERNAL)
            state[port].max_req = REMOTE_REQ_LEN;
          else
            state[port].max_req = LOCAL_REQ_LEN;
          state[state[port].talk_to].talk_to = port;
          break;

        case 2 :
          state[port].voice_on = unpack_bits(NBITS_FLAG, &cntrl_in);
          state[port].data_on = unpack_bits(NBITS_FLAG, (char *)0);
          state[port].max_rate = unpack_bits(NBITS_MAX_RATE, (char *)0);
          state[port].voice_rate = unpack_bits(NBITS_VOICE_RATE, (char *)0);
          state[port].xform_voice = state[port].voice_rate;
          state[port].conn_req_2 = cntrl_in;
```

```
#ifdef VERBOSE
        printf("\tvoice on = %d\n", state[port].voice_on);
        printf("\tdata on = %d\n", state[port].data_on);
        printf("\tmaximum rate = %d\n", state[port].max_rate);
        printf("\tvoice rate = %d\n", state[port].voice_rate);
#endif
        break;

      case 3 :
        state[port].stc_init = unpack_bits(NBITS_CODER_INIT, &cntrl_in);
        state[port].conn_req_3 = cntrl_in;
#ifdef VERBOSE
        printf("\tcoder init = %d\n", state[port].stc_init);
#endif
        break;

      case 4 :
      case 5 :
      case 6 :
      case 7 :
      case 8 :
      case 9 :
      case 10:
      case 11:
      case 12:
      case 13:
        state[port].phone_no[state[port].req_count-4] = cntrl_in;
        break;
      }
  } /* end of "if(state[port].req_flag == ON)" */


  if(state[port].req_flag == OFF) {
    if(msg_id == CONN_ACK) {
      state[port].stc_init = unpack_bits(NBITS_CODER_INIT, (char *)0);
#ifdef VERBOSE
      printf("Recd CONN_ACK from port %d(dest) ch %d\n", state[port].master, state[port].ch_id);
      printf("\tcoder init = %d\n", state[port].stc_init);
#endif
      cntrl_out = cntrl_in;
      fputc(cntrl_out, state[state[port].talk_to].fp_out);
#ifdef VERBOSE
```

```
        printf("Sent CONN_ACK to port %d(src)\n", state[port].talk_to);
#endif
        state[state[port].talk_to].idle = OFF;
        state[state[port].talk_to].connect = OFF;
        state[state[port].talk_to].busy = ON;
        state[port].idle = OFF;
        state[port].connect = OFF;
        state[port].busy = ON;
        state[port].idle_to_busy = ON;
        state[port].voice_on = state[state[port].talk_to].voice_on;
        state[port].data_on = state[state[port].talk_to].data_on;
        state[port].max_rate = state[state[port].talk_to].max_rate;
        state[port].voice_rate = state[state[port].talk_to].voice_rate;
        state[port].xform_voice = state[state[port].talk_to].xform_voice;
        state[port].data_rate = state[state[port].talk_to].data_rate;
        state[port].voice_frame = frame_length(state[port].voice_rate, 'v');
        state[state[port].talk_to].voice_frame = state[port].voice_frame;
        state[port].data_frame = frame_length(state[port].data_rate, 'd');
        state[state[port].talk_to].data_frame = state[port].data_frame;
      }

      else if(msg_id == CONN_RESP) {
        cntrl_out = cntrl_in;
        fputc(cntrl_out, state[state[port].talk_to].fp_out);
#ifdef VERBOSE
        printf("Recd CONN_RESP from port %d(modem) ch %d, fwd to port %d(src)\n",
               state[port].master, state[port].ch_id, state[port].talk_to);
#endif
        if(unpack_bits(NBITS_FLAG, (char *)0) == NO) {
          /*
           * connection refused
           */
          state[state[port].master].num_ch--;
          LOGICAL_PORTS--;
          state[state[port].talk_to].connect = OFF;
          state[state[port].talk_to].idle = ON;
#ifdef VERBOSE
          printf("Destination is busy\n");
#endif
        }
      }
    } /* end of "if(state[port].req_flag == OFF)" */
```

```
   if((state[port].req_count == state[port].max_req) && (state[port].req_flag == ON)) {
     state[port].req_flag = OFF;
#ifdef VERBOSE
     printf("Request parameters received from port %d\n", port);
#endif
     if(state[port].dest_type == EXTERNAL) {
       line_count = 0;
       no_line_free = TRUE;
       while((no_line_free) && (line_count < PHYSICAL_PORTS)) {
         if(line_count ≠ port) {
           if(state[line_count].modem == ON) {
             if((state[line_count].busy == OFF) && (state[line_count].connect == OFF)
&& (state[line_count].modem_resp == OFF))
               no_line_free = FALSE;
             else
               line_count++;
           }
           else
             line_count++;
         }
         else
           line_count++;
       }

       if(no_line_free) {
#ifdef VERBOSE
         printf("Outgoing line busy...trying to multiplex connection\n");
#endif
         line_count = 0;
         no_ckt_exists = TRUE;
         while((no_ckt_exists) && (line_count < PHYSICAL_PORTS)) {
           if(line_count ≠ port) {
             if(state[line_count].modem == ON) {
               if(cmp_phone(state[line_count].talk_to, port) == 1) {
#ifdef VERBOSE
                 printf("Physical connection to dest exists, checking bandwidth available\n");
#endif
                 no_ckt_exists = FALSE;
               }
               else
```

```
            line_count++;
          }
          else
            line_count++;
        }
        else
          line_count++;
      }
    }

    if(!no_line_free) {
      state[port].ext = state[port].talk_to;
      state[port].talk_to = line_count;
      state[line_count].pending_req = port;
      bits_left = pack_bits(CONN_RESP, NBITS_MSG_ID, &cntrl_out);
      bits_left = pack_bits(YES, NBITS_FLAG, &cntrl_out);
      bits_left = pack_bits(0, bits_left, &cntrl_out);
      fputc(cntrl_out, state[port].fp_out);
      bits_left = pack_bits(state[port].voice_on, NBITS_FLAG, &cntrl_out);
      bits_left = pack_bits(state[port].data_on, NBITS_FLAG, &cntrl_out);
      bits_left = pack_bits(state[port].voice_rate, NBITS_VOICE_RATE, &cntrl_out);
      bits_left = pack_bits(0, bits_left, &cntrl_out);
      fputc(cntrl_out, state[port].fp_out);
#ifdef VERBOSE
      printf("Sent WAIT...CALL PROCEEDING to port %d\n", port);
      printf("Dialing telephone number on port %d(modem)\n", state[port].talk_to);
#endif
      dial(port, state[port].talk_to);
      state[state[port].talk_to].idle = OFF;
      state[state[port].talk_to].modem_resp = ON;
    }

    if(no_line_free && !no_ckt_exists) {
      state[port].ext = state[port].talk_to;
      state[line_count].pending_req = port;
      if(check_bandwidth(port, line_count) == 0) {
#ifdef VERBOSE
        printf("Bandwidth currently not available...trying to transform rate\n");
#endif
        if(transform (port, line_count) == 0) {
#ifdef VERBOSE
          printf("Sorry, can't transform\n");
```

```
#endif
      }
      else {
       if(state[port].xform_flag == ON) {
        state[port].xform_flag = OFF;
        bits_left = pack_bits(CONN_RESP, NBITS_MSG_ID, &cntrl_out);
        bits_left = pack_bits(YES, NBITS_FLAG, &cntrl_out);
        bits_left = pack_bits(0, bits_left, &cntrl_out);
        fputc(cntrl_out, state[port].fp_out);
        bits_left = pack_bits(state[port].voice_on, NBITS_FLAG, &cntrl_out);
        bits_left = pack_bits(state[port].data_on, NBITS_FLAG, &cntrl_out);
        bits_left = pack_bits(state[port].voice_rate, NBITS_VOICE_RATE, &cntrl_out);
        bits_left = pack_bits(0, bits_left, &cntrl_out);
        fputc(cntrl_out, state[port].fp_out);
#ifdef VERBOSE
        printf("Sent CONN_RESP to port %d\n", port);
#endif
       }
       if(state[line_count].xform_flag == ON) {
        state[line_count].xform_flag = OFF;
        state[state[line_count].port_map[state[line_count].xform_ch]].xform_flag = OFF;
        bits_left = pack_bits(TRANSFORM, NBITS_MSG_ID, &cntrl_out);
        bits_left = pack_bits(state[state[line_count].port_map[state[line_count].xform_ch]].xform_voice,
NBITS_VOICE_RATE, &cntrl_out);
        fputc(cntrl_out, state[state[line_count].port_map[state[line_count].xform_ch]].fp_out);
#ifdef VERBOSE
        printf("Sent TRANSFORM to port %d(modem) ch %d\n", line_count, state[line_count].xform_ch)
#endif
        state[state[line_count].port_map[state[line_count].xform_ch]].xform = ON;
#ifdef VERBOSE
        printf("Pending request from port %d(src) to port %d(dest)\n", port, line_count);
#endif
       }
      }
     }
     else {
      bits_left = pack_bits(CONN_RESP, NBITS_MSG_ID, &cntrl_out);
      bits_left = pack_bits(YES, NBITS_FLAG, &cntrl_out);
      bits_left = pack_bits(0, bits_left, &cntrl_out);
      fputc(cntrl_out, state[port].fp_out);
      bits_left = pack_bits(state[port].voice_on, NBITS_FLAG, &cntrl_out);
      bits_left = pack_bits(state[port].data_on, NBITS_FLAG, &cntrl_out);
```

```
          bits_left = pack_bits(state[port].voice_rate, NBITS_VOICE_RATE, &cntrl_out);
          bits_left = pack_bits(0, bits_left, &cntrl_out);
          fputc(cntrl_out, state[port].fp_out);
#ifdef VERBOSE
          printf("Sent WAIT...CALL PROCEEDING to port %d\n", port);
#endif
          bits_left = pack_bits(OPEN_CH, NBITS_MSG_ID, &cntrl_out);
          bits_left = pack_bits(0, bits_left, &cntrl_out);
          fputc(cntrl_out, state[line_count].fp_out);
#ifdef VERBOSE
          printf("Sent OPEN_CHANNEL to port %d(modem)\n", line_count);
#endif
          state[line_count].open_ch = ON;
        }
      }
    }


    if(state[port].dest_type == INTERNAL) {
      if(state[state[port].talk_to].busy) {
        bits_left = pack_bits(CONN_RESP, NBITS_MSG_ID, &cntrl_out);
        bits_left = pack_bits(NO, NBITS_FLAG, &cntrl_out);
        bits_left = pack_bits(0, bits_left, &cntrl_out);
        fputc(cntrl_out, state[port].fp_out);
#ifdef VERBOSE
        printf("port %d(dest) busy, sent CONN_RESP (NO) to port %d(src)\n", state[port].talk_to,
port);
#endif
        state[port].connect = OFF;
      }
      else {
#ifdef VERBOSE
        printf("Setting up connection to port %d(dest)\n", state[port].talk_to);
#endif
        setup_conn(port);
      }
    }
  }
 }/* end of "if(state[port].connect)" */


 /*
```

```
 * MODEM RESULT state
 * (waiting for modem result code)
 */

else if(state[port].modem_resp) {
  state[port].modem_resp = OFF;
  if(cntrl_in == '1') {
#ifdef VERBOSE
    printf("Port %d: Modem result code = CONNECT\n", port);
#endif
    state[port].busy = ON;
    cntrl_out = OPEN_CH;
    fputc(cntrl_out, state[port].fp_out);
#ifdef VERBOSE
    printf("Sent OPEN_CHANNEL to port %d(modem)\n", port);
#endif
    state[port].open_ch = ON;
  }
  else {
#ifdef VERBOSE
    if(cntrl_in == '6') {
      printf("Port %d: Modem result code = NO DIALTONE\n", port);
    }
    if(cntrl_in == '7') {
      printf("Port %d: Modem result code = BUSY\n", port);
    }
    if(cntrl_in == '8') {
      printf("Port %d: Modem result code = NO ANSWER\n", port);
    }
#endif
    bits_left = pack_bits(CONN_RESP, NBITS_MSG_ID, &cntrl_out);
    bits_left = pack_bits(NO, NBITS_FLAG, &cntrl_out);
    bits_left = pack_bits(0, bits_left, &cntrl_out);
    fputc(cntrl_out, state[state[port].talk_to].fp_out);
#ifdef VERBOSE
    printf("Sent CONN_RESP (NO) to port %d \n", state[port].talk_to);
#endif
    state[state[port].pending_req].connect = OFF;
    state[state[port].pending_req].idle = ON;
  }
}
```

```
/*
 * BUSY state
 * (talking to another port)
 */

else if(state[port].busy) {
  if(msg_id == MODIFY_REQ) {
#ifdef VERBOSE
    printf("Received MODIFY_REQ from port %d \t",port);
#endif
    state[port].mod_type = unpack_bits(NBITS_CHANGE_MUX, (char *)0);
    state[port].mod_rate = unpack_bits(NBITS_MODIFY_RATE, (char *)0);
    modify = OFF;

    if(state[port].mod_type == CHANGE) {
      if(state[port].voice_on)
        new_traffic = 'd';
      if(state[port].data_on)
        new_traffic = 'v';
      if(decode(state[port].mod_rate, new_traffic) <= decode(state[port].max_rate, 't'))
        modify = ON;
    }

    if(state[port].mod_type == MULTIPLEX) {
      if(state[port].voice_on) {
        if(decode(state[port].voice_rate,'v') + decode(state[port].mod_rate,'d') <=
           decode(state[port].max_rate,'t'))
          modify = ON;
      }

      if(state[port].data_on) {
        if(decode(state[port].data_rate,'d') + decode(state[port].mod_rate,'v') <=
           decode(state[port].max_rate,'t'))
          modify = ON;
      }
    }

    if(modify) {
      bits_left = pack_bits(MODIFY_RESP, NBITS_MSG_ID, &cntrl_out);
      bits_left = pack_bits(YES, NBITS_FLAG, &cntrl_out);
      fputc(cntrl_out, state[port].fp_out);
```

```
#ifdef VERBOSE
      printf("Sent a Modify Response (YES) to port %d (source) \n", port);
#endif
      bits_left = pack_bits(MODIFY_IND, NBITS_MSG_ID, &cntrl_out);
      bits_left = pack_bits(state[port].mod_type, NBITS_CHANGE_MUX, &cntrl_out);
      bits_left = pack_bits(state[port].mod_rate, NBITS_MODIFY_RATE, &cntrl_out);
      fputc(cntrl_out, state[state[port].talk_to].fp_out);
#ifdef VERBOSE
      printf("Sent MODIFY_IND: port %d (dest) \n", state[port].talk_to);
#endif
      state[port].busy = OFF;
      state[port].modify = ON;
      state[port].mod_src = ON;
      state[state[port].talk_to].busy = OFF;
      state[state[port].talk_to].modify = ON;
      state[state[port].talk_to].mod_src = OFF;
    }
    else {
      bits_left = pack_bits(MODIFY_RESP, NBITS_MSG_ID, &cntrl_out);
      bits_left = pack_bits(NO, NBITS_FLAG, &cntrl_out);
      fputc(cntrl_out, state[port].fp_out);
#ifdef VERBOSE
      printf("Sent MODIFY_RESP (NO): port %d (src) \n", port);
#endif
    }
  }

  if(msg_id == DISCONNECT) {
#ifdef VERBOSE
    printf("Received DISCONNECT from port %d\n", port);
#endif
    other_end = state[state[port].talk_to].master;
    if(state[port].modem || state[other_end].modem) {
      printf("....reconfiguring modem \n");
    }
    if(state[port].master != port) {
      state[state[port].master].num_ch--;
      LOGICAL_PORTS--;
    }
    state[port].idle = ON;
    state[port].busy = OFF;
    cntrl_out = DISCONNECT;
```

```
    fputc(cntrl_out, state[state[port].talk_to].fp_out);
#ifdef VERBOSE
    printf("Sent DISCONNECT to port %d ch %d\n", state[state[port].talk_to].master,
          state[state[port].talk_to].ch_id);
#endif
    if(state[state[port].talk_to].master ≠ state[port].talk_to) {
      state[state[state[port].talk_to].master].num_ch--;
      LOGICAL_PORTS--;
    }
    state[state[port].talk_to].idle = ON;
    state[state[port].talk_to].busy = OFF;
  }

  if(state[port].open_ch == ON) {
    if(msg_id == OPEN_CH_ACK) {
      state[port].open_ch = OFF;
#ifdef VERBOSE
      printf("Received OPEN_CHANNEL_ACK from port %d\n", port);
#endif
      add_channel(port);
      cntrl_out = state[state[port].pending_req].conn_req_1 & 0xdf; /* flip dest type */
      fputc(cntrl_out, state[LOGICAL_PORTS].fp_out);
      cntrl_out = state[state[port].pending_req].conn_req_2;
      fputc(cntrl_out, state[LOGICAL_PORTS].fp_out);
      cntrl_out = state[state[port].pending_req].conn_req_3;
      fputc(cntrl_out, state[LOGICAL_PORTS].fp_out);
#ifdef VERBOSE
      printf("Sent CONN_REQ on external line (port %d) ch %d\n", port,
            state[LOGICAL_PORTS].ch_id);
#endif
    }
  }


  if(state[port].xform == ON) {
    if(msg_id == TRANSFORM_ACK) {
      state[port].xform_count++;
      state[port].xform = OFF;
#ifdef VERBOSE
      printf("Received TRANSFORM_ACK from port %d \n", port);
      printf("Enable DSP\n");
      printf("Wait for Started Xform signal from DSP\n");
```

```
      printf("Started Xform signal received\n");
#endif
      cntrl_out = START_XFORM;
      fputc(cntrl_out, state[port].fp_out);
#ifdef VERBOSE
      printf("Sent START_XFORM to port %d\n", port);
#endif
      strcpy(state[port].speech, state[port].xform_file[state[port].xform_count-1]);
      state[port].fp_speech = fopen(state[port].speech, "w");
      cntrl_out = OPEN_CH;
      fputc(cntrl_out, state[state[port].master].fp_out);
#ifdef VERBOSE
      printf("Sent OPEN_CHANNEL to port %d(modem)\n", state[port].master);
#endif
      state[state[port].master].open_ch = ON;
     }
   }
 }


 else if(state[port].modify) {
   if(state[port].mod_src == ON) {
     if(msg_id == START_MODIFY) {
#ifdef VERBOSE
      printf("Received CALL MODIFIED: port %d \n", port);
#endif
      if(state[port].voice_on) {
        if(state[port].mod_type == CHANGE)
          state[port].voice_on = OFF;
        state[port].data_on = ON;
        state[port].data_rate = state[port].mod_rate;
      }
      if(state[port].data_on) {
        if(state[port].mod_type == CHANGE)
          state[port].data_on = OFF;
        state[port].voice_on = ON;
        state[port].voice_rate = state[port].mod_rate;
      }
      state[port].modify = OFF;
      state[port].busy = ON;
      state[state[port].talk_to].modify = OFF;
      state[state[port].talk_to].busy = ON;
```

```
      }
    }
  }
} /* end of configure */
```

```
/*
 * NAME: code_decode_rate.c
 *
 * PURPOSE:
 *     These routines perform code conversion
 *
 *     encoded rate | decoded rate
 *        binary    | voice  data
 *          00      |  1      1
 *          01      |  2      2
 *          10      |  4      4
 *          11      |  8      4
 */


#include<stdio.h>

int decode(int code_in, char type) {

  int rate;
  static int voice_rate[]={1, 2, 4, 8};
  static int data_rate[]={1, 2, 4, 4};


  if(type == 't')
    rate = code_in + 1;

  else if(type == 'v')
    rate = voice_rate[code_in];

  else if(type == 'd')
    rate = data_rate[code_in];
  return(rate);
}


int code(int rate, char type) {

  int code_out;
  static int voice_code[]={0, 0, 1, 0, 2, 0, 0, 0, 3};
  static int data_code[]={0, 0, 1, 1, 2, 2, 2, 2, 2};
```

```
  if(type == 'v')
    code_out = voice_code[rate];

  else if(type == 'd')
    code_out = data_code[rate];
  return(code_out);
}
```

```
/*
 * NAME: frame_length.c
 *
 * PURPOSE:
 *     This routine returns the frame length in bytes
 *     for a given rate and traffic type
 *
 * INPUTS:
 *        rate: voice/data rate (00,01,10,11)
 *        frame_type: voice/data
 *
 * RETURN VALUE:
 *        frame length in bytes
 */


#include <stdio.h>

int frame_length(int rate, char frame_type) {

  int length;
  static int data_frame_len[] = { 64, 64, 64, 0 };
  static int voice_frame_len[] = { 6, 12, 24, 48 };


  switch(frame_type) {
  case 'd': length = data_frame_len[rate];
    break;

  case 'v': length = voice_frame_len[rate];
    break;
  }
  return(length);
}
```

```
/*
 * NAME: cmp_phone.c
 *
 * PURPOSE:
 *     This routine compares the remote destination addresses
 *     (telephone numbers) for two calls to determine if call
 *     multiplexing is possible
 *
 * INPUTS:
 *       port1, port2 : calling ports
 *
 * RETURN VALUE:
 *       1 => phone numbers not matched
 *       0 => phone numbers are identical
 */


#include<stdio.h>
#include "scp.h"

int cmp_phone(int port1, int port2) {

  int cmp_flag=1, i=0;


  while (cmp_flag && (i<=PHYSICAL_PORTS))
    {
      if (state[port1].phone_no[i] == state[port2].phone_no[i])
        cmp_flag = 1;
      else
        cmp_flag = 0;
      i++;
    }
  return(cmp_flag);
}
```

```
/*
 * NAME: dial.c
 *
 * PURPOSE:
 *      This routine uses the AT command set to dial
 *      a phone number on the modem
 *
 * INPUTS:
 *      src: calling port
 *      modem: modem port
 */


#include <stdio.h>
#include "scp.h"

void dial(int src, int modem) {

  int i;
  char cntrl_out;

  cntrl_out = 'A';
  fputc (cntrl_out, state[modem].fp_out);
  cntrl_out = 'T';
  fputc (cntrl_out, state[modem].fp_out);
  cntrl_out = 'D';
  fputc (cntrl_out, state[modem].fp_out);
  cntrl_out = 'T';
  fputc (cntrl_out, state[modem].fp_out);

#ifdef VERBOSE
  printf ("The dialing sequence is: ATDT ");

  for (i=0; i < PHONE_NO; i++) {
    cntrl_out = state[src].phone_no[i];
    fputc (cntrl_out, state[modem].fp_out);
    printf ("%c", state[src].phone_no[i]);
  }
  printf ("\n");
#endif
}
```

```
/*
 * NAME: setup_conn.c
 *
 * PURPOSE:
 *    This routine sets up a connection between
 *    src and state[src].talk_to
 */


#include <stdio.h>
#include "scp.h"
#include "msgs.h"


void setup_conn(src)
    int src;
{
  int total, voice, data, bits_left;
  char temp, cntrl_out;

  total = decode(state[src].max_rate, 't');
  voice = decode(state[src].voice_rate, 'v');
  data = total - voice;
  state[src].data_rate = code(data, 'd');
  if (state[src].voice_on)
    state[src].voice_frame = frame_length(state[src].voice_rate, 'v');
  else
    state[src].voice_frame = 0;
  if (state[src].data_on)
    state[src].data_frame = frame_length(state[src].data_rate, 'd');
  else
    state[src].data_frame = 0;
  bits_left = pack_bits(CONN_RESP, NBITS_MSG_ID, &cntrl_out);
  bits_left = pack_bits(YES, NBITS_FLAG, &cntrl_out);
  bits_left = pack_bits(0, bits_left, &cntrl_out);
  fputc (cntrl_out, state[src].fp_out);
  bits_left = pack_bits(state[src].voice_on, NBITS_FLAG, &cntrl_out);
  bits_left = pack_bits(state[src].data_on, NBITS_FLAG, &cntrl_out);
  bits_left = pack_bits(state[src].voice_rate, NBITS_VOICE_RATE, &cntrl_out);
  bits_left = pack_bits(0, bits_left, &cntrl_out);
  fputc (cntrl_out, state[src].fp_out);
#ifdef VERBOSE
```

```
  printf ("Sent CONN_RESP to port %d \t");
#endif
  bits_left = pack_bits(CONN_IND, NBITS_MSG_ID, &cntrl_out);
  bits_left = pack_bits(0, 2, &cntrl_out);
  bits_left = pack_bits(state[src].voice_on, NBITS_FLAG, &cntrl_out);
  bits_left = pack_bits(state[src].data_on, NBITS_FLAG, &cntrl_out);
  fputc (cntrl_out, state[state[src].talk_to].fp_out);
  bits_left = pack_bits(state[src].voice_rate, NBITS_VOICE_RATE, &cntrl_out);
  bits_left = pack_bits(state[src].data_rate, NBITS_DATA_RATE, &cntrl_out);
  bits_left = pack_bits(state[src].stc_init, NBITS_CODER_INIT, &cntrl_out);
  fputc (cntrl_out, state[state[src].talk_to].fp_out);
#ifdef VERBOSE
  printf ("Sent CONN_IND to port %d (dest) \n", state[src].talk_to);
#endif
  state[state[src].talk_to].connect = ON;
  state[state[src].talk_to].pending_req = src;
}
```

```
/*
 * NAME: add_channel.c
 *
 * PURPOSE:
 *      This routine opens a transport layer connection
 *      on an existing modem link. This is simulated by
 *      activating a virtual "port", setting up the
 *      necessary file pointers for that port, and then
 *      redirecting all i/o to or from the modem on that
 *      channel to the new port.
 */


#include <stdio.h>
#include "scp.h"

void add_channel(modem)
    int modem;
{
  state[modem].num_ch++;
  LOGICAL_PORTS++;
  state[modem].port_map[state[modem].num_ch] = LOGICAL_PORTS;
  state[LOGICAL_PORTS].active = ON;
  state[LOGICAL_PORTS].master = modem;
  state[LOGICAL_PORTS].ch_id = state[modem].num_ch;
  state[LOGICAL_PORTS].idle = OFF;
  state[LOGICAL_PORTS].connect = ON;
  state[LOGICAL_PORTS].talk_to = state[modem].pending_req;
  state[LOGICAL_PORTS].fp_in = fopen (state[LOGICAL_PORTS].in_file, "r");
  state[LOGICAL_PORTS].fp_out = fopen (state[LOGICAL_PORTS].out_file, "w");
  state[LOGICAL_PORTS].fp_speech = fopen (state[LOGICAL_PORTS].speech, "w");
  state[LOGICAL_PORTS].fp_cntrl = fopen (state[LOGICAL_PORTS].cntrl, "w");
  state[state[modem].pending_req].talk_to = LOGICAL_PORTS;
}
```

```
/*
 * NAME: check_bandwidth.c
 *
 * PURPOSE:
 *     This routine checks if bandwidth is available on
 *     the modem link to accomodate an additional caller
 *
 * INPUTS:
 *        caller = calling port
 *        modem  = modem port
 *
 * RETURN VALUE:
 *        1 => bandwidth available
 *        0 => not available
 */


#include<stdio.h>
#include "scp.h"

#define MAX_LINK_RATE  8

int check_bandwidth(caller, modem)
    int caller, modem;
{
  int i, port;
  int old_total, old_voice=0, old_data=0;
  int new_total, new_voice=0, new_data=0;


  for (i=1; i<=state[modem].num_ch; i++)
    {
      port = state[modem].port_map[i];
      if(state[port].active)
        {
          if(state[port].voice_on)
            {
              old_voice += decode(state[port].xform_voice, 'v');
            }
          if(state[port].data_on)
            old_data += decode(state[port].data_rate,'d');
        }
```

```
    }
  old_total = old_voice + old_data;

  if(state[caller].voice_on)
    {
      new_voice = decode(state[caller].xform_voice, 'v');
    }
  if(state[caller].data_on)
    new_data = decode(state[caller].data_rate,'d');
  new_total = new_voice + new_data;

  if((old_total + new_total) ≤ MAX_LINK_RATE)
    return(1);
  else
    return(0);
}
```

```
/*
 * NAME: transform.c
 *
 * PURPOSE:
 *   Tries to accomodate new call by xforming rate on
 *   current and/or previous call(s). Priority for
 *   dropping rate:  (i) Higher rate calls
 *                   (ii) Older calls
 *
 * INPUTS:
 *   caller = calling port
 *   modem  = modem port
 *
 * RETURN VALUE:
 *   0 => transform not possible
 *   1 => transform possible
 */


#include<stdio.h>
#include "scp.h"

int transform(caller, modem)
    int caller, modem;
{
  int caller_voice, ch_number, voice, port, high_voice, high_port, done_flag;
  int save_caller_voice, save_high_voice;
  static int drop_rate[] = { 0, 0, 1, 0, 2, 0, 0, 0, 4 };


  caller_voice = decode(state[caller].voice_rate, 'v');

  ch_number = 1;
  high_port = state[modem].port_map[ch_number];
  high_voice = decode(state[high_port].xform_voice, 'v');
  while (ch_number <= state[modem].num_ch)
    {
      ch_number++;
      port = state[modem].port_map[ch_number];
      if (state[port].active)
        voice = decode(state[port].xform_voice, 'v');
      else
```

```
      voice = 0;
    if (high_voice < voice)
      {
        high_voice = voice;
        high_port = port;
      }
  }

done_flag = OFF;
while ((caller_voice ≥ high_voice) && (done_flag == OFF))
  {
    caller_voice = drop_rate[caller_voice];
    save_caller_voice = state[caller].xform_voice;
    state[caller].xform_voice = code(caller_voice, 'v');
    state[caller].xform_flag = ON;
    if (check_bandwidth(caller, modem) == 1)
      done_flag = ON;
  }
if (done_flag == OFF)
  {
    save_high_voice = state[high_port].xform_voice;
    state[high_port].xform_voice = code(drop_rate[high_voice], 'v');
    state[high_port].xform_flag = ON;
    state[modem].xform_flag = ON;
    state[modem].xform_ch = state[high_port].ch_id;
    if (check_bandwidth(caller, modem) == 1)
      done_flag = ON;
  }

if (done_flag == OFF)
  {
    state[caller].xform_voice = save_caller_voice;
    state[high_port].xform_voice = save_high_voice;
    state[caller].xform_flag = OFF;
    state[high_port].xform_flag = OFF;
    state[modem].xform_ch = 0;
  }

return(done_flag);
}
```

```
/*
 * NAME: pack_bits.c
 *
 * PURPOSE:
 *   This routine takes parameters of given
 *   length and packs them into control bytes
 *
 * INPUTS:
 *   param: parameter to be packed
 *   index: #bits in param
 *   chptr: the packed control byte
 *
 * RETURN VALUE:
 *   the number of bits left in the current byte
 */


#include<stdio.h>
#include<math.h>

#define WORD 8

static int count = 0;
static int bits_left_index, bits_left_count;
static char bitpack;
static short masks[] = { 0, 0x01, 0x03, 0x07, 0x0f,
                         0x01f, 0x03f, 0x07f, 0x0ff,
                         0x01ff, 0x03ff, 0x07ff, 0x0fff,
                         0x01fff, 0x03fff, 0x07fff, 0x0ffff };


int pack_bits(short param, int index, char *chptr){

  short part1, part2;


  bits_left_count = WORD - count;

  if ( index < bits_left_count ) {
    bitpack = bitpack<<index;
    bitpack += param;
    count += index;
```

```
} else if ( bits_left_count == index) {

  bitpack = bitpack<<index;
  bitpack += param;

  *chptr = bitpack;
  count = 0;
  bitpack = 0;

} else {
  bits_left_index = index - bits_left_count;

  part1 = (param>>bits_left_index) & masks[bits_left_count];

  bitpack = bitpack<<bits_left_count;
  bitpack += part1;

  *chptr = bitpack;
  bitpack=0;

  if (bits_left_index >= WORD) {

    bits_left_count = WORD;

    bits_left_index = bits_left_index - bits_left_count;

    part1 = (param>>bits_left_index) & masks[bits_left_count];

    bitpack = bitpack<<bits_left_count;
    bitpack += part1;

    *chptr = bitpack;
  }

  bitpack = param & masks[bits_left_index];
  count = bits_left_index;

}
bits_left_count = WORD - count;
return(bits_left_count);
}
```

```
/*
 * NAME: unpack_bits.c
 *
 * PURPOSE:
 *      This subroutine unpacks parameters from control bytes
 *
 * INPUTS:
 *      index: #bits the variable param is to have
 *      chptr: control byte to be unpacked
 *
 * RETURN VALUE:
 *      param: the unpacked version
 */


#include<math.h>
#include<stdio.h>

#define INT_WORD  8

static int count = 0;
static int bits_left_index, bits_left_count;
static char bitpack;
int param;
static short masks[] = { 0, 0x01, 0x03, 0x07, 0x0f,
                          0x01f, 0x03f, 0x07f, 0x0ff,
                          0x01ff, 0x03ff, 0x07ff, 0x0fff,
                          0x01fff, 0x03fff, 0x07fff, 0x0ffff };

int unpack_bits(int index, char *chptr){

  short part1, part2, middle_part, left;
  int nmbr;

  if (chptr != (char *)0) {
    bitpack = *chptr;
    count = 0;
  }

  bits_left_count = INT_WORD - count;

  if (count == 0) {
```

```
  bitpack = *chptr;
}

if ( index <= bits_left_count ) {

  left = bits_left_count - index;
  param = (bitpack>>left) & masks[index];
  count+=index;
}

else {          /* parameter longer than current word */

  bits_left_index = index - bits_left_count;

  part1 = bitpack & masks[bits_left_count];
  part1 = part1<<bits_left_index;

  if (bits_left_index > INT_WORD) {

    bits_left_count = INT_WORD;
    bits_left_index = bits_left_index - bits_left_count;

    bitpack = *chptr;

    middle_part = bitpack & masks[bits_left_count];
    middle_part = middle_part<<bits_left_index;
    part1 = part1 + middle_part;
  }

  bitpack = *chptr;

  left = INT_WORD - bits_left_index;

  part2 = bitpack>>left;
  part2 = part2 & masks[bits_left_index];

  param = part1 + part2;
  count = bits_left_index;

}

if (count == 8) {    /* need a new word, but wait until next time */
```

```
    count = 0;
  }

  return(param);
}
```

C.2 Header Files

```
/*
 * NAME: scp.h
 *
 * PURPOSE:
 *   Header file defining the structure used to
 *   maintain port state
 *
 * Description of fields in port struct:
 *
 *   modem = modem port
 *   alive = port active
 *   idle = port not busy
 *   connect = call setup in progress
 *   modem_resp = waiting for modem response
 *   busy = talking to another port
 *   xform = rate modification in progress
 *   modify = call modification in progress
 *   conn_req_1 |
 *   conn_req_2 | connection request
 *   conn_req_3 |
 *   stc_init = coder initialization
 *   dest_type: 0=>local, 1=>remote
 *   talk_to = port connected to
 *   ext = extension address for remote call
 *   phone_no = 10-digit tel# (remote switch address)
 *   data_on: 0=>no data, 1=>data
 *   voice_on: 0=>no voice, 1=>voice
 *   max_rate = maximum link rate
 *   voice_rate = voice connection rate
 *   data_rate = allowable data rate
 *   voice_frame = voice frame length in bytes
 *   data_frame = data frame length in bytes
 *   max_req = number of bytes in connection request
 *           = 3  if destination is local
 *           = 8  if destination is remote
 *   req_count = #connection request bytes received
 *   req_flag = indicates end of request
 *   pending_req: port with conn req pending
 *   open_ch: waiting for open_ch ack
 *   master = self => physical port
```

```
*         = other => logical port
*   ch_id = channel number (if logical port)
*   num_ch = #channels active (if physical port)
*   port_map[x] = channel #x is mapped to logical
*                port #port_map[x]
*   mod_type: 0=>change, 1=>multiplex
*   mod_rate: rate for new traffic
*   mod_src = waiting for modify source ack
*   xform_ch    |
*   xform_voice | used to communicate with transform
*   xform_flag  | routine
*   xform_count |
*   in_file = input file (coded speech+control)
*   out_file = output file (echoes control from in_file)
*   cntrl = output control message file
*   speech = output speech file (no rate translation)
*   xform_file[n] = output speech file after
*                 nth stage of rate translation
*   frame_count = #bytes left in current frame
*   idle_to_busy = signifies state transition
*   eof_flag: 1=>EOF reached on port
*/


#define PHONE_NO   10
#define ON         1
#define OFF        0
#define TRUE       1
#define FALSE      0
#define YES        1
#define NO         0
#define CHANGE     0
#define MULTIPLEX  1
#define INTERNAL   0
#define EXTERNAL   1
#define MAX_CH     4
#define MAX_PORTS  7
#define PHYSICAL_PORTS 3

#define MAX_PATH_NAME 80
```

```
int LOGICAL_PORTS;


struct PortState {
  unsigned int  modem        : 1;
  unsigned int  alive        : 1;
  unsigned int  idle         : 1;
  unsigned int  connect      : 1;
  unsigned int  modem_resp   : 1;
  unsigned int  busy         : 1;
  unsigned int  xform        : 1;
  unsigned int  modify       : 1;
          char  conn_req_1;
          char  conn_req_2;
          char  conn_req_3;
  unsigned int  dest_type    : 1;
  unsigned int  talk_to      : 3;
  unsigned int  data_on      : 1;
  unsigned int  voice_on     : 1;
  unsigned int  max_rate     : 3;
  unsigned int  voice_rate   : 2;
  unsigned int  data_rate    : 2;
          char  stc_init;
          char  phone_no[PHONE_NO];
  unsigned int  ext          : 3;
           int  voice_frame;
           int  data_frame;
           int  max_req;
           int  req_count;
  unsigned int  req_flag     : 1;
  unsigned int  pending_req  : 3;
  unsigned int  open_ch      : 1;
  unsigned int  master       : 3;
           int  ch_id;
           int  num_ch;
  unsigned int  port_map[MAX_CH];
  unsigned int  mod_type     : 1;
  unsigned int  mod_rate     : 2;
  unsigned int  mod_src      : 1;
  unsigned int  xform_ch;
  unsigned int  xform_voice  : 2;
  unsigned int  xform_flag   : 1;
```

```
        int  xform_count;
        char  in_file[MAX_PATH_NAME], out_file[MAX_PATH_NAME];
        char  cntrl[MAX_PATH_NAME], speech[MAX_PATH_NAME];
        char  xform_file[MAX_CH][MAX_PATH_NAME];
        FILE  *fp_in, *fp_out, *fp_cntrl, *fp_speech;
        int  frame_count;
  unsigned int  idle_to_busy  : 1;
  unsigned int  eof_flag      : 1;
} state[8];
```

```
/*
 * NAME: msgs.h
 *
 * Header file describing control
 * message IDs and bit allocations
 */


/*
 * Definitions of number of bits
 * for message components
 */

#define NBITS_MSG_ID        4
#define NBITS_DEST_TYPE     1
#define NBITS_DEST_ADD      3
#define NBITS_TRAFFIC_TYPE  2
#define NBITS_MAX_RATE      3
#define NBITS_VOICE_RATE    2
#define NBITS_DATA_RATE     2
#define NBITS_CODER_INIT    4
#define NBITS_CHANGE_MUX    1
#define NBITS_MODIFY_RATE   2
#define NBITS_FLAG          1


/*
 * Misc control message defs
 */

#define LOCAL_REQ_LEN 3
#define REMOTE_REQ_LEN 13

/*
 * Message Id's
 */

#define NIL           0x07
#define CONN_REQ      0x01
#define CONN_IND      0x02
#define CONN_RESP     0x03
#define ALERTING      0x04
```

```
#define CONN_ACK        0x0f
#define DISCONNECT      0x05
#define MODIFY_REQ      0x06
#define MODIFY_IND      0x08
#define MODIFY_RESP     0x09
#define MODIFY_ACK      0x0f
#define START_MODIFY    0x0a
#define TRANSFORM       0x0b
#define TRANSFORM_ACK   0x0f
#define START_XFORM     0x0c
#define OPEN_CH         0x0d
#define OPEN_CH_ACK     0x0f
#define START_CH        0x0e
#define MODIFY_SRC      0x00
#define MODIFY_SRC_ACK 0x0f
```

Bibliography

[1] Najeeb I. Ansari, Sona S. Kapadia, and Joseph B. Evans. A narrowband adaptive voice/data switch. In *IEEE Wichita Conference on Communications, Networking and Signal Processing*, April 1994.

[2] O. A. Avellaneda, J. F. Hayes, and M. M. Nassehi. A capacity allocation problem in voice-data networks. *IEEE Trans. Comm.*, COM-30(7), July 1982.

[3] I. Boyd. Speech coding for telecommunications. In F. A. Westall and S. F. A. Ip, editors, *Digital Signal Processing in Telecommunications*. Chapman & Hall, 1993.

[4] T. G. Champion. Theory of parameter space transformation techniques. Tech. rep., Rome Laboratories, to be published.

[5] T. G. Champion and J. B. Evans. A flexible multirate speech coder. In *Proc. Int. Conf. Signal Proc. Appl. & Tech.*, pages 1440–1443, Sept 1993.

[6] Martin de Prycker. *Asynchronous Transfer Mode Solution for Broadband ISDN*. Ellis Horwood Limited, 1992.

[7] Spiros Dimolitsas. Standardizing speech-coding technology for network applications. *IEEE Communications Magazine*, Nov 1993.

[8] J. B. Evans and T. G. Champion. Robust speech coding and reconstruction techniques. In *Proc. Int. Conf. Signal Proc. Appl. & Tech.*, pages 928–931, Nov 1992.

[9] Joseph B. Evans and Timouthy Johnson. Adaptive voice/data networks. Proposal to Rome Laboratories, University of Kansas, March 1993.

[10] G. J. Foschini, B. Gopinath, and J. F. Hayes. Optimum allocation of servers to two types of competing customers. *IEEE Trans. Comm.*, COM-29(7), July 1981.

[11] Dale Gulick. The basics of high-level data link control. In Gary R. McClain, editor, *The Handbook of International Connectivity Standards*. Van Nostrand Reinhold, 1992.

[12] N. S. Jayant. Coding speech at low bit rates. *IEEE Spectrum*, Aug 1986.

[13] Whay C. Lee and Michael G. Hluchyj. Dynamic connection management for call-level QOS guarantee in integrated communication networks. In *Proc. IEEE Infocom*, 1994.

[14] R. J. McAulay and T. G. Champion. Improved interoperable 2.4 kb/s LPC using sinusoidal transform coder techniques. In *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, pages 641–643, 1990.

[15] R. J. McAulay and T. F. Quatieri. Magnitude-only reconstruction using a sinusoidal speech model. In *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, pages 27.6.1–27.6.4, 1984.

[16] R. J. McAulay and T. F. Quatieri. Mid-rate speech coding based on a sinusoidal representation of speech. In *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, pages 945–948, 1985.

[17] R. J. McAulay and T. F. Quatieri. Multirate sinusoidal transform coding at rates from 2.4 kbps to 8 kbps. In *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, pages 1645–1648, 1986.

[18] R. J. McAulay and T. F. Quatieri. Speech analysis-synthesis based on a sinusoidal representation. *IEEE Trans. Acoust., Speech, Signal Processing*, ASSP-34(4):744–754, Aug 1986.

[19] Robert J. McAulay and Thomas F. Quatieri. Low-rate speech coding based on the sinusoidal model. In Sadaoki Furui and M. Mohan Sondhi, editors, *Advances in Speech Signal Processing*. M. Dekker, 1991.

[20] Gopalkrishnan Meempat and Malur Sundareshan. Optimum channel allocation policies for access control of circuit-switched traffic in isdn environments. *IEEE Trans. Comm.*, 41(2), Feb 1993.

[21] Amanda Moody, Cliff Parris, and Danny Wong. Selecting a speech coding algorithm. *DSP & Multimedia Technology*, May 1994.

[22] D. Petr, L. DaSilva, and V. Frost. Priority discarding of speech in integrated packet networks. *IEEE Journ. Select. Areas Commun.*, SAC-7(5):644–656, June 1989.

[23] David W. Petr, K. M. S. Murthy, Victor S. Frost, and Lyn A. Neir. Modeling and simulation of the resource allocation process in a bandwidth-on-demand satellite communications network. *IEEE Journ. Select. Areas Commun.*, 10(2), Feb 1992.

[24] John D. Spragins, Joseph L. Hammond, and Krzysztof Pawlikowski. *Telecommunications Protocols and Design*. Addison-Wesley Publishing Company, 1991.

[25] William Stallings. *ISDN and Broadband ISDN.* Macmillan Publishing Company, 1992.

[26] Andrew S. Tannenbaum. *Computer Networks.* Prentice Hall, Inc., 1988.

[27] Jonathan S. Turner. New directions in communications (or which way to the information age. *IEEE Commun. Mag.*, 24(10):8–15, Oct 1986.

[28] Nanying Yin and Michael G. Hluchyj. A dynamic rate control mechanism for source coded traffic in a fast packet network. *IEEE Journ. Select. Areas Commun.*, 9(7):1003–1012, Sept 1991.