

Implementation of E-Mail via Phone



Undergraduate Degree Project BESE-II

By

Ahmar Ghaffar (Project Leader)

Muhammad Usman Tahir

Project supervised by

Col. Dr. Muhammad Akbar

*Dissertation to be presented as partial requirement for the award of
B.E. Degree in Software Engineering.*

Military College of Signals

National University of Sciences and Technology, Rawalpindi.

***In The Name Of Allah,
The Most Benevolent,
The Most Merciful.***

To Our Parents.

Acknowledgments

We are grateful to Allah Almighty, for giving us the strength to visualize, undertake and complete this project.

Our humble gratitude to our teacher and project supervisor Col. Dr. Muhammad Akbar, for guiding and encouraging us through out the course of this project. We are thankful to him for providing us with innovative ideas, project management techniques and constant moral support.

We are very grateful to Dr. Masood-ul-Haque and Mr. Fakhar Hameed for providing guidance, support and in depth knowledge to understand contemporary systems and their problems.

We are thankful to all our friends for lively discussions, exchange of ideas and source of encouragement during this project.

Finally, we would like to thank our parents and family members for their perseverance and continued moral support, which helped us to get through hard times.

Abstract

This document has been prepared as a dissertation of final year degree project, to be presented to MCS/NUST in partial requirement for the award of B.E. degree in the discipline of Software Engineering.

This dissertation discusses E-Mail via Phone: An on-demand phone in service. It encompasses the vision behind the project, the way its feasibility and requirements were analyzed, the design was drawn and implementation took place. The aim of the project was to implement an automated voice messaging system providing a variety of features. Some of these features include recording voice messages for other users at the server, listening to voice messages, and download e-mail messages.

The research and subsequent development of this E-mail via Phone system was carried out by Mr. Muhammad Usman Tahir and Mr. Ahmar Ghaffar under the guidance and supervision of Col. Dr. Muhammad Akbar.

Project Specifications

Statement

E-Mail via Phone: Implementation of e-mail by phone and on demand phone-in services.

Development Environment/Tools

1. Microsoft Visual C++ 6.0.
2. Microsoft Speech SDK 4.0.
3. Microsoft TAPI.
4. Active Server Pages (ASP).

Development Languages

1. C++.
2. HTML.

Platforms Supported

1. MS Windows 95, 98.
2. MS Windows NT 4.0 with Service Pack 3.
3. MS Windows 2000.

System Modeling/Design

1. Unified Modeling Language (UML)
2. Rational ROSE.
3. OOD.

Table of Contents

ACKNOWLEDGMENTS	3
ABSTRACT	4
PROJECT SPECIFICATIONS	5
STATEMENT.....	5
DEVELOPMENT ENVIRONMENT/TOOLS.....	5
DEVELOPMENT LANGUAGES.....	5
PLATFORMS SUPPORTED	5
SYSTEM MODELING/DESIGN.....	6
1 INTRODUCTION	11
1.1 WHAT IS E-MAIL VIA PHONE, AND WHAT CAN IT DO?	12
1.2 MINIMUM REQUIREMENTS FOR E-MAIL VIA PHONE.....	13
1.3 AREAS OF APPLICATION	13
1.4 SECURITY	14
2 TEXT-TO-SPEECH	15
2.1 WHAT IS TEXT-TO-SPEECH?	16
2.2 WHY USE TEXT-TO-SPEECH?.....	17
2.3 USES OF TEXT-TO-SPEECH.....	18
2.4 GAMES AND EDUTAINMENT.....	19
2.5 HARDWARE AND SOFTWARE REQUIREMENTS	19
2.6 TEXT-TO-SPEECH CONVERSION	20
2.6.1 <i>Text Normalization</i>	20
2.6.2 <i>Homograph Disambiguation</i>	21
2.6.3 <i>Word Pronunciation</i>	22
2.6.4 <i>Prosody</i>	24
2.6.5 <i>Audio Play Back</i>	25
2.7 LIMITATIONS.....	26
2.7.1 <i>Text-to-Speech Voice Quality</i>	26
2.7.2 <i>Factors affecting Text-to-Speech Voice Quality</i>	27
2.7.3 <i>Creating and Localizing Text-to-Speech Voices</i>	28
2.7.4 <i>Where the Engine Comes From</i>	28
2.8 IMPLEMENTATION OF TTS	29
3 THE MICROSOFT SPEECH API	33
3.1 OVERVIEW	34
3.2 SPEECH API ARCHITECTURE	34
3.3 SAPI LEVELS OF ACCESS	34
3.4 TTS OPERATION WITH LOW-LEVEL SPEECH OBJECTS	38
3.5 AUDIO AND SHARING OBJECTS	39
3.6 VOICE TEXT.....	40
3.6.1 <i>Voice-Text Object</i>	41
3.6.2 <i>Voice-Text Notification Sink</i>	42
3.7 LOW-LEVEL TEXT-TO-SPEECH.....	42
3.7.1 <i>Main TTS Object</i>	43
3.7.2 <i>Main Notification Sink</i>	44
3.7.3 <i>Audio-Destination Object</i>	44
3.7.4 <i>Audio-Destination Notification Sink</i>	44
3.7.5 <i>Buffer Notification Sink</i>	45
3.8 SIGNIFICANCE IN E-MAIL VIA PHONE	45
4 E-MAIL CLIENT	46
4.1 WHY DO WE NEED AN E-MAIL CLIENT?.....	47
4.2 POST OFFICE PROTOCOL (POP3).....	47

4.2.1 Connection Establishment	48
4.2.2 Commands in POP3	48
4.2.3 Responses in POP3	48
4.2.4 State Transitions	49
5 TELEPHONY MODULE	59
5.1 TELEPHONY APPLICATIONS	60
5.1.1 Voice Mail or Answering Machine Software	60
5.1.2 Accessing Databases	60
5.1.3 Call Routing	60
5.2 HARDWARE AND SOFTWARE REQUIREMENTS	61
5.2.1 Processor speed	61
5.2.2 Memory	61
5.2.3 Telephony Card	61
5.3 APPLICATION DESIGN CONSIDERATIONS	62
5.3.1 Multi-Line Applications	62
5.4 TYPES OF TELEPHONY TODAY	62
5.4.1 Public Switched Telephony Network (PSTN)	62
5.4.2 Traditional "Computer Telephony (CT)" Technology	63
5.4.3 IP Telephony	64
5.4.4 Internet Telephony	66
5.5 MODEMS	67
5.6 THE INTELLIGENT NETWORK	69
6 DTMF DETECTION	70
6.1 WHAT IS DTMF?	71
6.2 AT&T SPECIFICATIONS FOR TONE GENERATION:	71
6.3 DTMF TONE GENERATION	72
6.3.1 Methods of Tone Generation	72
6.4 DTMF TONE DETECTION	77
6.4.1 Collecting Spectral Information Using Goertzel's Algorithm	77
6.4.2 Validity Checks	80
6.5 MODIFICATION IN GOERTZEL ALGORITHM	83
7 REGISTRATION WEB SITE	85
7.1 WHY DO WE NEED A REGISTRATION WEB SITE?	86
7.2 ACTIVE SERVER PAGES: AN INTRODUCTION	86
7.3 SOFTWARE DEVELOPMENT WITH ASP	86
7.4 ADVENT OF ASP	87
7.5 FEATURES OF ASP	87
7.6 INTERNET INFORMATION SERVER AND ASP DEVELOPMENT PLATFORM	88
7.7 IMPLEMENTATION OF THE WEB SITE	89
8 E-MAIL VIA PHONE: SYSTEM DESIGN	91
8.1 FINITE STATE MACHINES	92
8.1.1 Boot Up Services	92
8.1.2 E-mail Download using POP3 Client	93
8.1.3 User Authorization/User Menu	94
8.1.4 Message Recording	95
8.1.5 E-Mail Playback using TTS	96
8.2 CLASS DIAGRAMS	97
8.2.1 Timer Handler	97
8.2.2 Call Handler I	98
8.2.3 Call Handler II	99
8.2.4 Device Handler	100
8.2.5 POP3 Client	101
8.2.6 Data Types	102
8.3 USE CASES	103
8.3.1 Connection Establishment	103
8.3.2 User Authorization	104

8.3.3	User Menu	105
8.3.4	Message Recorder	106
8.3.5	Message Player	107
8.3.6	Delete Connection Menu	108
8.3.7	Connection Deletion	109
8.3.8	Back to Basics	110
8.3.9	Registration	111
9	E-MAIL VIA PHONE: IMPLEMENTATION	112
9.1	TFX	113
9.1.1	WindowClass	113
9.1.2	InvisibleWindowSink	114
9.1.3	InvisibleWindow	114
9.1.4	CtAddressCaps	114
9.1.5	CtVariableData	116
9.1.6	CtAppSink	117
9.1.7	CtCall	117
9.1.8	Public Methods:	118
9.1.9	CtLine	119
9.1.10	CtReplyTarget	123
9.1.11	CtCallInfo	123
9.1.12	CtCallList	126
9.1.13	CtCallSink	127
9.1.14	CtCallStatus	127
9.1.15	CtCountryList	128
9.1.16	CtDeviceID	129
9.1.17	CtDialStringSink	130
9.1.18	CtDialString	130
9.1.19	CtDtmf	131
9.1.20	Timer	131
9.1.21	CtWaveSink	132
9.1.22	TimerSink	132
9.1.23	CtWave	132
9.1.24	TapiRecover	133
9.1.25	LineTapiRecover	134
9.1.26	PhoneTapiRecover	134
9.1.27	CtRequestList	135
9.1.28	CtLineDevCaps	135
9.1.29	CtLineSink	138
9.1.30	CtPhone	138
9.1.31	CtPhoneCaps	141
9.1.32	CtPhoneNo	143
9.1.33	CtPhoneSink	144
9.1.34	CtProviderList	144
9.1.35	CtTranslateCaps	145
9.1.36	CtTranslateOutput	147
9.2	INTEGRATION	148
9.2.1	CPtrList	148
9.2.2	HCALL	148
9.2.3	CPtrArray	148
9.2.4	HLINE	148
9.2.5	HPHONE	148
9.2.6	HLINEAPP	148
9.2.7	HPHONEAPP	148
9.2.8	WAVEHDR	148
9.2.9	WAVEFORMATEX	148
9.2.10	HWAVEIN	148
9.2.11	HWAVEOUT	148
9.3	VOICEM	149
9.3.1	CMainFrame	149

9.3.2 CPop3Message	150
9.3.3 CPop3Socket.....	150
9.3.4 CPop3Connection.....	151
9.3.5 CSettings.....	152
9.3.6 CVoiceMApp.....	153
9.3.7 CVoiceMView.....	155
9.3.8 CVoiceMDoc	155
9.3.9 CAboutDlg.....	156
10 FUTURE EXPANSION POSSIBILITIES	157
10.1 NEWS UPDATE	158
10.2 WEATHER FORECAST.....	158
10.3 SPORTS NEWS	158
10.4 FLIGHT TIMINGS.....	159
10.5 ON LINE TRANSACTION PROCESSING	159
11 CONCLUSION.....	160
12 BIBLIOGRAPHY	161
13 APPENDIX.....	165
13.1 POST OFFICE PROTOCOL – VERSION 3 (POP3) RFC.....	165
14 INDEX	180

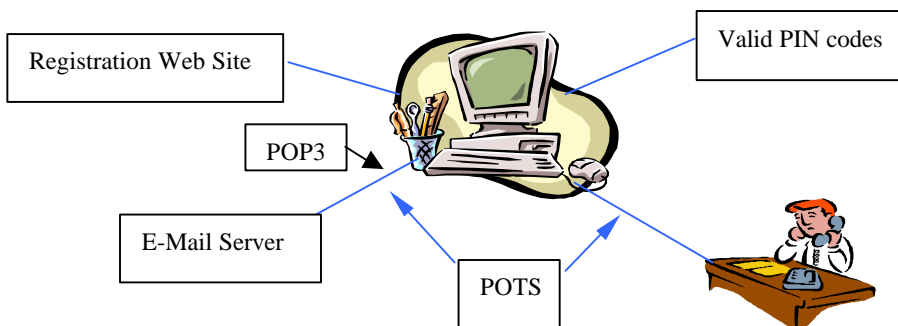
1 Introduction

The basic idea behind an E-mail via Phone system is to make a number of useful services available to the user through the normal telephone set. These services can include E-mail by phone, News update, Weather forecast, Sports news, Personalized messaging and many more.

The service that will be most useful in this context is the e-mail by phone facility that will enable the users to check their mail messages through a telephone from anywhere. This facility can also include the option of personalized messaging in which a user can record his message for another person at the server maintained by the Phone-In Service Provider (PSP). This message can later be retrieved by the person, for whom it is intended, by calling the PSP's number.

1.1 What Is E-Mail via Phone, and What Can It Do?

E-mail via Phone is a service that enables a user to access voice mail and e-mail messages using any touch-tone phone. E-mail via Phone "reads" the new e-mail messages using text-to-speech technology. E-mail via Phone also serves as a voice mail system, recording callers' messages and making them available by phone.



Block Diagram of the System

1.2 Minimum Requirements for E-Mail via Phone

- Intel Pentium Processor (100 MHz min).
- Voice Modem.
- Touch Tone Telephone.
- Microsoft Windows 9x,NTx. Operating System.
- Microsoft Speech Engine.

1.3 Areas of Application

The areas of application of such a service are as vast as the combined domain of Computer Telephony and Text-to-Speech. An existing PSTN, Internet or Internet Telephony Service Provider can add e-mail by phone facility to its existing services and provide its customers with an advanced mode of communication.

A start-up company can also initiate its business with this service. Many dot.com companies in U.S. are offering similar services. Notable among them are JFAX, ShoutMail, MailCall, eVoice, eFax, CallWave, Message Click, OneBox, uReach, Pagoo, CoolSpeak etc.

With the advent of PTCL Personal Mail Box service, and its subsequent success, it can easily be said that providing this service on a commercial basis in Pakistan is monetarily feasible plan. Two companies in Karachi have started similar services in the last few months.

1.4 Security

One of the most important issues regarding an interactive user application is Security. The syndicate has tried to assure the secure transaction of e-mail information to the user by introducing a registration procedure from its Registration Website. Each user is assigned a unique PIN code. Only a user having a valid PIN code is allowed to access the system and listen to his/her e-mails.

2 Text-to-Speech

2.1 What is Text-to-Speech?

Text-to-speech is a process through which text is rendered as digital audio and then "spoken." Most text-to-speech engines can be categorized by the method that they use to translate phonemes into audible sound. Some of the widely used TTS systems are summarized below:

- **Concatenated Word.** Although Concatenated Word systems are not really synthesizers, they are one of the most commonly used text-to-speech systems around. In a concatenated word engine, the application designer provides recordings for phrases and individual words. The engine pastes the recordings together to speak out a sentence or phrase. As in voice-mail systems the engines speaks, "[You have] [three] [new messages]." The engine has recordings for "You have", all of the digits, and "new messages".
- **Synthesis.** A text-to-speech engine that uses synthesis generates sounds similar to those created by the human vocal cords and applies various filters to simulate throat length, mouth cavity, lip shape, and tongue position. The voice produced by synthesis technology tends to sound less human than a voice produced by diphone concatenation, but it is possible to obtain different qualities of voice by changing a few parameters. IBM is working on such systems for the last twenty-five years.
- **Subword Concatenation.** A text-to-speech engine that uses subword concatenation links short digital-audio segments together and performs inter-segment smoothing to produce a continuous sound. In diphone concatenation, for example, each segment consists of two phonemes, one that leads into the sound and one that finishes the sound. Thus, the word "hello"

consists of the phonemes, h eh l æ, and the corresponding subword segments are silence-h h-eh eh-l l-æ silence.

Subword segments are acquired by recording many hours of a human voice and painstakingly identifying the beginning and ending of phonemes. Although this technique can produce a more realistic voice, it takes a considerable amount of work to create a new voice and the voice is not localizable because the phonemes are specific to the speaker's language.

The syndicate has decided to use Subword concatenation technique in the Text-to-Speech module of this project. All phonemes recorded in human voices are available through Microsoft's Speech Application Programming Interface (SAPI) 4.0. Text is converted to speech with the concatenation of these phonemes.

2.2 Why Use Text-to-Speech?

Text-to-speech should be used to audibly communicate information to the user, when digital audio recordings are inadequate. Generally, text-to-speech is better than audio recordings when:

1. Audio recordings are too large to store on disk or expensive to record.
2. Audio recording is impossible because the application doesn't know ahead of time what it will speak.

Text-to-speech was used in this project because every time a new mail arrives, it has to be automatically converted in speech to be accessed by the user. TTS was the only available and viable option to implement this feature.

2.3 Uses of Text-to-Speech

Text-to-speech also offers a number of benefits. In general, text-to-speech is most useful for short phrases or for situations when prerecording is not practical. Text-to-speech has the following practical uses:

- **Reading dynamic text.** Text-to-speech is useful for phrases that vary too much to record and store using all possible alternatives. For example, speaking the time is a good use for text-to-speech, because the effort and storage involved in concatenating all possible times is not manageable.
- **Proofreading.** Audible proofreading of text and numbers helps the user catch typing errors missed by visual proofreading.
- **Conserving storage space.** Text-to-speech is useful for phrases that would occupy too much storage space if they were pre-recorded in digital-audio format.
- **Notifying the user of events.** Text-to-speech works well for informational messages. For example, to inform the user that a print job is complete, an application could say "Printing complete" rather than displaying a message box and requiring the user to click OK. This should be used for non-critical notifications in case the user turns the computer's sound off or is out of hearing range.
- **Providing audible feedback.** Text-to-speech can provide audible feedback when visual feedback is inadequate or impossible. For example, the user's eyes might be busy with another task, such as transcribing data from a paper document. Users that have low vision may rely on text-to-speech as their sole means of feedback from the computer. This is the foremost reason for which Text-to-speech was incorporated in this project.

2.4 Games and Edutainment

Text-to-speech is useful in games and edutainment to allow the characters in the application to "talk" to the user instead of displaying speech balloons. Of course, it's also possible to have recordings of the speech. An application would use text-to-speech instead of recordings in the following cases:

- It's always possible to use concatenated word/phrase text-to-speech to replace recorded sentences. The application designer can easily pass the desired sentence strings to the text-to-speech engine.
- Synthesized text-to-speech inevitably sounds unnatural and weird. However, it's very good for character voices that are supposed to be robots, aliens, or maybe even foreigners.
- If the application cannot afford to have recordings of all the possible dialogs or if the dialogs cannot be recorded ahead of time, then text-to-speech is the only alternative.

2.5 Hardware and Software Requirements

A speech application requires certain hardware and software on the user's computer to run. These hardware and software requirements should be considered when designing a speech application:

- **Processor speed.** Text-to-speech engines currently on the market typically require a 486/33 (DX or SX) or faster processor.
- **Memory.** On the average, text-to-speech uses about 1 MB of RAM.
- **Sound card.** Almost any sound card will work for speech recognition and text-to-speech, including Sound Blaster™, Media Vision™, ESS Technology, cards that are compatible with the Microsoft® Windows Sound System, and the audio hardware built into multimedia computers.
- **Speakers.** The user can choose between wearing headphones and using freestanding speakers. Headphones are useful in office cubicles. Some

companies manufacture a combination headphone and microphone that can also be used for telephone conversations.

- **Operating system.** The Microsoft Speech application-programming interface (API) requires either Windows 95 or Windows NT version 4.0.
- **Text-to-speech engine.** Text-to-speech software must be installed on the user's system. Many new audio-enabled computers and sound cards are bundled with speech recognition and text-to-speech engines. As an alternative, many engine vendors offer retail packages for speech recognition or text-to-speech, and some license copies of their engines.

2.6 Text-to-Speech Conversion

Text-to-speech fundamentally functions as a pipeline that converts text into PCM digital audio. The elements of the pipeline are:

1. Text normalization
2. Homograph disambiguation
3. Word pronunciation
4. Prosody
5. Concatenate wave segments

2.6.1 Text Normalization

The "text normalization" component of text-to-speech converts any input text into a series of spoken words. Trivially, text normalization converts a string like "John rode home." to a series of words, "john", "rode", "home", along with a marker indicating that a period occurred. However, this gets more complicated when strings like "John rode home at 23.5 mph", where "23.5 mph" is converted to "twenty three point five miles per hour". Here's how text normalization works:

First, text normalization isolates words in the text. For the most part this is as trivial as looking for a sequence of alphabetic characters, allowing for an occasional apostrophe and hyphen.

Text normalization then searches for numbers, times, dates, and other symbolic representations. These are analyzed and converted to words. (Example: "Rs.54.32" is converted to "fifty four rupees and thirty two paisas.") Someone needs to code up the rules for the conversion of these symbols into words, since they differ depending upon the language and context.

Next, abbreviations are converted, such as "in." for "inches", and "St." for "street" or "saint". The normalizer will use a database of abbreviations and what they are expanded to. Some of the expansions depend upon the context of surrounding words, like "St. John" and "John St."

Once the text has been normalized and simplified into a series of words, it is passed onto the next module, homograph disambiguation.

2.6.2 Homograph Disambiguation

The next stage of text-to-speech is called "homograph disambiguation." Often it's not a stage by itself, but is combined into the text normalization or pronunciation components. It has been separated out since it doesn't fit cleanly into either.

In English and many other languages, there are hundreds of words that have the same text, but different pronunciations. A common example in English is "read," which can be pronounced "reed" or "red" depending upon it's meaning. A "homograph" is a word with the same text as another word, but with a different pronunciation. The concept extends beyond just words, and into abbreviations and numbers. "Ft." has different pronunciations in "Ft. Wayne" and "100 ft.". Likewise, the digits "1997" might be spoken as "nineteen ninety-seven" if the author is talking

about the year, or "one thousand nine hundred and ninety seven" if the author is talking about the number of people at a concert.

Text-to-speech engines use a variety of techniques to disambiguate the pronunciations. The most robust is to try to figure out what the text is talking about and decide which meaning is most appropriate given the context. Once the right meaning is known, it's usually easy to guess the right pronunciation.

Text-to-speech engines figure out the meaning of the text, and more specifically of the sentence, by parsing the sentence and figuring out the part-of-speech for the individual word. This is done by guessing the part-of-speech based on the word endings, or by looking the word up in a lexicon. Sometimes a part of speech will be ambiguous until more context is known, such as for "read". Disambiguation of the part-of-speech may require hand-written rules.

Once the homographs have been disambiguated, the words are sent to the next stage to be pronounced.

2.6.3 Word Pronunciation

The pronunciation module accepts the text, and outputs a sequence of phonemes, just like a dictionary. To get the pronunciation of a word, the text-to-speech engine first looks the word up in it's own pronunciation lexicon. If the word is not in the lexicon then the engine reverts to "letter to sound" rules.

Letter-to-sound rules guess the pronunciation of a word from the text. They're kind of the inverse of the spelling rules taught in school. There are a number of techniques for guessing the pronunciation, but the algorithm described here is one of the more easily implemented ones.

The letter-to-sound rules are "trained" on a lexicon of hand-entered pronunciations. The lexicon stores the word and its pronunciation, such as:

hello h eh l oe

An algorithm is used to segment the word and figure out which letter "produces" which sound. The "h" in "hello" produces the "h" phoneme, the "e" produces the "eh" phoneme, the first "l" produces the "l" phoneme, the second "l" nothing, and "o" produces the "oe" phoneme. Of course, in other words the individual letters produce different phonemes. The "e" in "he" will produce the "ee" phoneme.

Once the words are segmented by phoneme, another algorithm determines which letter or sequence of letters is likely to produce which phonemes. The first pass figures out the most likely phoneme generated by each letter. "H" almost always generates the "h" sound, while "o" almost always generates the "ow" sound. A secondary list is generated, showing exceptions to the previous rule given the context of the surrounding letters. Hence, an exception rule might specify that an "o" occurring at the end of the word and preceded by an "l" produces an "oe" sound. The list of exceptions can be extended to include even more surrounding characters.

When the letter-to-sound rules are asked to produce the pronunciation of a word they do the inverse of the training model. To pronounce "hello", the letter-to-sound rules first try to figure out the sound of the "h" phoneme. It looks through the exception table for an "h" beginning the word followed by "e"; Since it can't find one it uses the default sound for "h", which is "h". Next, it looks in the exceptions for how an "e" surrounded by "h" and "l" is pronounced, finding "eh". The rest of the characters are handled in the same way.

This technique can pronounce any word, even if it wasn't in the training set, and does a very reasonable guess of the pronunciation, sometimes better than humans. It doesn't work too well for names because most names are not of English origin, and use different pronunciation rules. (Example: "Usman" is pronounced as "uz-h-man" by anyone that doesn't know it is Arabic.) Some letter-to-sound rules first guess

what language the word came from, and then use different sets of rules to pronounce each different language.

Word pronunciation is further complicated by people's laziness. People will change the pronunciation of a word based upon what words precede or follow it, just to make the word easier to speak. An obvious example is the way "the" can be pronounced as "thee" or "thuh. A commonly used American phrase such as "What you doing?" sounds like "Wacha doin?"

Once the pronunciations have been generated, these are passed onto the prosody stage.

2.6.4 Prosody

Prosody is the pitch, speed, and volume that syllables, words, phrases, and sentences are spoken with. Without prosody text-to-speech sounds very robotic, and with bad prosody text-to-speech sounds like it's been drunk.

The technique that engines use to synthesize prosody varies, but there are some general techniques.

First, the engine identifies the beginning and ending of sentences. In English, the pitch will tend to fall near the end of a statement, and rise for a question. Likewise, volume and speaking speed ramp up when the text-to-speech first starts talking, and fall off on the last word when it stops. Pauses are placed between sentences.

Engines also identify phrase boundaries, such as noun phrases and verb phrases. These will have similar characteristics to sentences, but will be less pronounced. The engine can determine the phrase boundaries by using the part-of-speech information generated during the homograph disambiguation. Pauses are placed between phrases or where commas occur.

Algorithms then try to determine which words in the sentence are important to the meaning, and these are emphasized. Emphasized words are louder, longer, and will have more pitch variation. Words that are unimportant, such as those used to make the sentence grammatically correct, are de-emphasized. In a sentence such as "John and Bill walked to the store," the emphasis pattern might be "JOHN and BILL walked to the STORE." The more the text-to-speech engine "understands" what's being spoken, the better its emphasis will be.

Next, the prosody within a word is determined. Usually the pitch and volume rise on stressed syllables.

All of the pitch, timing, and volume information from the sentence level, phrase level, and word level are combined together to produce the final output. The output from the prosody module is just a list of phonemes with the pitch, duration, and volume for each phoneme.

2.6.5 Audio Play Back

The speech synthesis is almost done by this point. All the text-to-speech engine has to do is convert the list of phonemes and their duration, pitch, and volume, into digital audio.

Methods for generating the digital audio will vary, but many text-to-speech engines generate the audio by concatenating short recordings of phonemes. The recordings come from a real person. In a simplistic form, the engine receives the phoneme to speak, loads the digital audio from a database, does some pitch, time, and volume changes, and sends it out to the sound card.

It isn't quite that simple for a number of reasons. Most noticeable is that one recording of a phoneme will not have the same volume, pitch, and sound quality at

the end, as the beginning of the next phoneme. This causes a noticeable glitch in the audio. An engine can reduce the glitch by blending the edges of the two segments together so at their intersections they both have the same pitch and volume. Blending the sound quality, which is determined by the harmonics generated by the voice, is more difficult, and can be solved by the next step.

The sound that a person makes when he/she speaks a phoneme, changes depending upon the surrounding phonemes. If we record "cat" in sound recorder, and then reverse it, the reversed audio doesn't sound like "tak", which has the reversed phonemes of cat. Rather than using one recording per phoneme (about 50), the text-to-speech engine maintains thousands of recordings (usually 1000-5000). Ideally it would have all possible phoneme context combinations recorded, $50 * 50 * 50 = 125,000$, but this would be too many. Since many of these combinations sound similar, one recording is used to represent the phoneme within several different contexts.

Even a database of 1000 phoneme recordings is too large, so the digital audio is compressed into a much smaller size, usually between 8:1 and 32:1 compression. The more compressed the digital audio, the more muted the voice sounds.

Once the digital audio segments have been concatenated they're sent off to the sound card, making the computer talk to the user at the phone who is calling the E-mail via Phone system.

2.7 Limitations

2.7.1 Text-to-Speech Voice Quality

Most text-to-speech engines can render individual words successfully. However, as soon as the engine speaks a sentence, it is easy to identify the voice as synthesized

because it lacks human prosody -- i.e., the inflection, accent, and timing of speech. For this reason, most text-to-speech voices are difficult to listen to and require concentration to understand, especially for more than a few words at a time.

Some engines allow an application to define text-to-speech segments with human prosody attached, making the synthesized voice much clearer. The engine provides this capability by prerecording a human voice and allowing the application developer to transfer its intonation and speed to the text being spoken.

In effect, this acts as a highly effective voice compression algorithm. Although text with prosody attached requires more storage than ASCII text (1K per minute compared to a few hundred bytes per minute), it requires considerably less storage than pre-recorded speech, which requires at least 30K per minute.

2.7.2 Factors affecting Text-to-Speech Voice Quality

These factors also influence the quality of a synthesized voice:

Emotion. Although many text-to-speech engines can parse and interpret punctuation, such as periods, commas, exclamation points, and questions, none of the engines that are currently available can render the sound of human emotion.

Mispronunciation. Text-to-speech engines use a set of pronunciation rules to translate text into phonemes. This is fairly easy for languages with phonetic alphabets, but it is very difficult for the English language, especially if last names are to be pronounced correctly. (Pronunciation rules seldom fail on common words, but they almost always fail on names that are unusual or of non-English origin.)

If an engine mispronounces a word, the only way that the user can change the pronunciation is by entering either the phonemes, which is not an easy task, or by

choosing a series of "sound-alike" words that combine to make the correct pronunciation.

2.7.3 Creating and Localizing Text-to-Speech Voices

Creating a new voice for an engine that uses synthesis can be done relatively quickly by altering a few parameters of an existing voice. Although the pitch and timbre of the new voice are different, it uses the same speaking style and prosody rules as the existing voice.

Creating a new voice for a text-to-speech engine that uses diphone concatenation can take a considerable amount of work, because the diphones must be acquired by recording a human voice and identifying the beginning and ending of phonemes, which are specific to the speaker's language.

Whether a text-to-speech engine uses synthesis or diphone concatenation, the work of localizing an engine for a new language requires a skilled linguist to design pronunciation and prosody rules and reprogram the engine to simulate the sound of the language's phonemes. In addition, diphone-concatenation systems require a new voice to be constructed for the new language. As a consequence, most engines support only five to ten major languages.

2.7.4 Where the Engine Comes From

Of course, for text-to-speech to work on an end user's PC the system must have a text-to-speech engine installed on it. The application has two choices:

1. The application can come bundled with a text-to-speech engine and install it itself. This guarantees that text-to-speech will be installed and also guarantees a certain level of quality from the text-to-speech. However, if an application does this, royalties will need to be paid to the engine vendor.

- Alternatively, an application can assume that the text-to-speech engine is already on the PC or that the user will purchase one if they wish to use text-to-speech. The user may already have text-to-speech because many PCs and sound cards will come bundled with an engine, or, the user may have purchased another application that included an engine. If the user has no text-to-speech engine installed then the application can tell the user that they need to purchase a text-to-speech engine and install it. Several engine vendors offer retail versions of their engines.

The syndicate has decided to prefer option 2 and assume that a text-to-speech engine is already on the PC running the application developed during this project. It is recommended to install Microsoft SAPI Suite 4.0 before installing e-mail via phone application.

2.8 Implementation of TTS

There are two basic technologies: speech recognition (SR) and speech synthesis, depending on who is doing the talking, person or the computer. Speech synthesis is commonly called "text-to-speech" or TTS, since the speech is usually synthesized from text data. **Figure 1** shows the architecture of a typical text-to-speech engine.

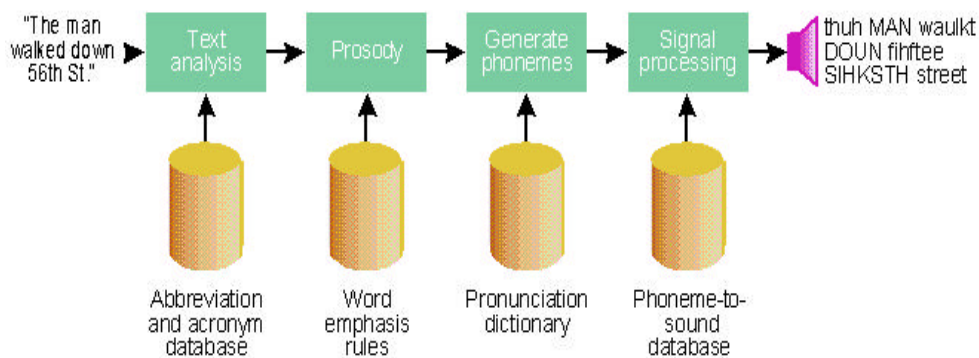


Figure 1 Text-to-Speech Engine

The process begins when the application hands the engine a string of text such as, "The man walked down 56th St." The text analysis module converts numbers into words, identifies punctuation such as commas, periods, and semicolons, converts

abbreviations to words, and even figures out how to pronounce acronyms. Some acronyms are spelled out (MSJ) whereas others are pronounced as a word (NUST).

Text analysis is quite complex because written language can be so ambiguous. A human has no trouble pronouncing "St. John St." as "Saint John Street," but a computer, in typically mechanical fashion, might come up with "Street John Street" unless a clever programmer gives it some help.

Once the text is converted to words, the engine figures out what words should be emphasized by making them louder or longer, or giving them a higher pitch. Other words may be de-emphasised. Without word emphasis, or "prosody," the result is a monotone voice that sounds robotic.

Next, the text-to-speech engine determines how the words are pronounced, either by looking them up in a pronunciation dictionary, or by running an algorithm that guesses the pronunciation. Some text strings have ambiguous pronunciations, such as "read." The engine must use context to disambiguate the pronunciations. The result of this analysis is the original sentence expressed as phonemes. "Th-uh M-A-Nw-au-l-k-tD-OU-Nf-ih-f-t-eeS-IH-K-S-TH s-t-r-ee-t".

Next, the phonemes are parsed and their pronunciations retrieved from a phoneme-to-sound database that numerically describes what the individual phonemes sound like. If speech were simple, this table would have only forty-four entries, one for each of the forty-four English phonemes. In practice, each phoneme is modified slightly by its neighbours, so the table often has as many as 1600 or more entries. Depending on the implementation, the table might store either a short wave recording or parameters that describe the mouth and tongue shape. Either way the sound database values are finally smoothed together using signal processing techniques, and the digital audio signal is sent to an output device such as a PC sound card and out the speakers to human ears.

Both text-to-speech and speech recognition involve quite a bit of processing, but speech recognition is harder because it usually requires more processing for equivalent user satisfaction. A few years ago, one needed a high-end workstation to

do speech recognition. Today, just about every new PC and even many older PCs can handle speech. But Speech Recognition is not used in this project because it is a vast research field in itself.

While the exact requirements vary from one speech engine to another, **Figure 2** gives a rough idea of the hardware needed to run various kinds of speech applications under Windows. The faster the CPU and the more memory available, the higher the accuracy for speech recognition and the better the text-to-speech sounds.

Figure 2 Speech Hardware Minimum Requirements

Technology	CPU	RAM
Discrete command and control		
User speaks simple commands like "mail," "change time," "minimize."	386/33	500KB
Continuous command and control		
User speaks complex commands, like "Send mail to Fred," "Change the time to ten o'clock," and "Minimize the window."	486/33	1MB
Discrete dictation		
Transcribes whatever the user says into a word processor. The user must pause between words.	486/66	8MB
Continuous dictation		
Transcribes natural speech into a word processor	P6	16MB
Text-to-speech		
Convert ASCII or Unicode strings to natural speech.	486/33	1MB

A sound card, microphone, and speakers are also needed. Most speech engines will work with any sound card. Some systems offload processing onto a DSP (digital

signal processor) chip that comes on some high-end sound cards, which cuts the CPU speed requirement in half. Better microphones and speakers will also improve things.

As speech has become more feasible on average PCs, vendors have been busy developing and promoting their speech engines. Many multimedia PCs and sound cards come bundled with speech software. Others vendors sell their engines as standalone products. Some applications even come bundled with speech engines.

Unfortunately, as with any budding technology, the situation is a bit chaotic. Even though they all support similar functionality, each speech engine has its own specific features and proprietary API. If one wants to use speech in one's application, one first got to pick which engine to use, and write program for that engine. If a better engine comes along, one's out of luck. One will probably have to rewrite a program substantially to use the other API. The syndicate has decided to limit themselves to MS SAPI Suite 4.0 and the engine that comes with it.

3 The Microsoft Speech API

3.1 Overview

The Microsoft Speech API is specified as a collection of OLE Component Object Model (COM) objects. Using OLE makes speech readily available to developers writing in Visual Basic®, C/C++, or any other programming language that can access OLE objects directly or through automation. The Speech API requires Windows 95 or Windows NT 3.51.

3.2 Speech API Architecture

As with other Windows Open Services Architecture (WOSA) services, the Speech API is intended as a standard interface that application developers and engine vendors alike can code to. Programmers can write applications without worrying about which engine to use, engine vendors can get instant compatibility with all speech apps, and users gain the freedom to choose whichever speech engine meets their budget and performance requirements. The situation is analogous to GDI, which lets programs draw graphics without worrying about what kind of display card or monitor the user has. Just like GDI, the Speech API provides escape hooks to access proprietary engine features when we need to do something special.

3.3 SAPI levels of access

The Speech API offers two levels of access: high-level objects designed to make implementation easy, and low-level objects that offer total control but make us do a little more work. If all a program does is listen for a few voice commands and utter some simple phrases, one can use the high-level objects. To do more sophisticated stuff, one needs the low-level.

The high-level objects, provided by Microsoft, don't do any SR or TTS themselves; they just call the low-level objects to do the work. The low-level objects are provided by the speech engine vendor, just like the video and sound card drivers that come with display or sound card. In our case, the engine is provided by Microsoft as well.

Figure-3 describes the relationship between MS SAPI and third party engines, which too is provided by Microsoft in this project:

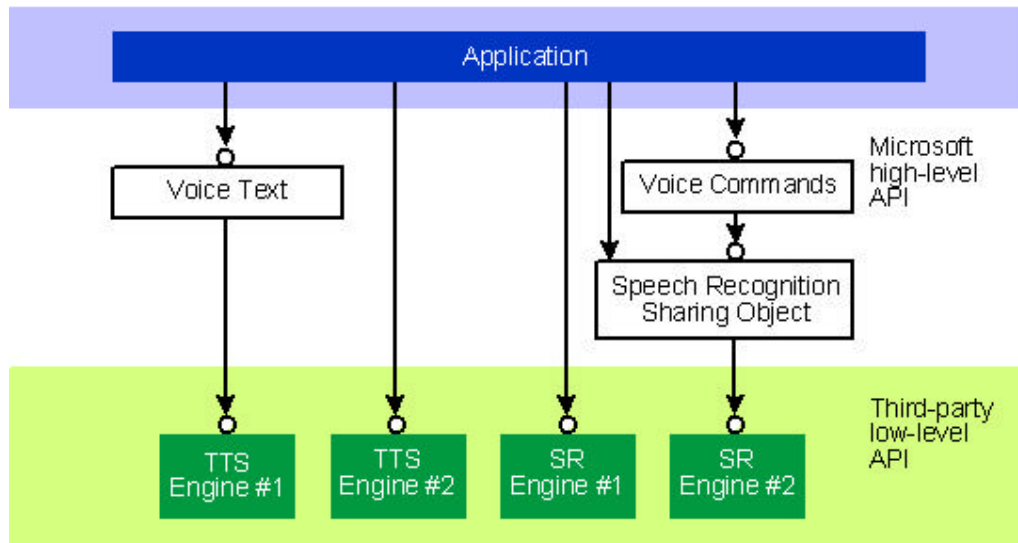


Figure 3 Using the Low-level Speech API

Figures 4 and 5 show the main OLE objects and interfaces that constitute the Speech API. The objects used by the syndicate voice text for text-to-speech. Microsoft also provides a speech recognition-sharing object that lets several applications share engines.

Figure 4 High-level Speech Objects

Voice Commands Object

- IUnknown Provide access to other interfaces in the object.
- IVoiceCmd Simple command and control speech recognition. Member functions let the app create Voice Menu objects.
- IVCmdAttributes Controls the attributes of the speech recognition engine such as the automatic gain, speaker name, and recognition threshold.
- IVCmdDialogs Displays Windows dialog boxes that let the user configure the speech recognition engine, such as training.
- IVCmdNotifySink (Supplied by the app.) Used to notify the app when a command is recognized, the

user is speaking too loudly or softly, or something else happens.

Voice Menu Object

IUnknown Provide access to other interfaces in the object.

IVCmdMenu Methods to add/remove/modify voice commands, and to start listening for them.

Voice Text Object

IUnknown Provide access to other interfaces in the object.

IVoiceText Main interface for generating speech; contains the Speak function.

IVTxtAttributes ControlstheattributesoftheTTSenginesuchasthevoice'spitchandgender.

IVTxtDialogs Displays dialog boxes that let the user configure the TTS engine.

IVTxtNotifySink Supplied by the app. Used to notify the app when talking has begun or ended, or when a bookmark is reached or something else happens.

Figure 5 Low-level Speech Objects

Speech Recognition Engine Object

IUnknown Provides access to other interfaces in the object.

ISRAAttributes Controls the attributes of the speech recognition engine such as the automatic gain, processor usage, speaker name, and recognition threshold.

ISRCentral Controls the engine object. Methods let the app create grammar objects and register notification sinks.

ISRDialogs Displays Windows dialog boxes that let the user configure the speech recognition engine, such as training.

ISRNotifySink Supplied by the app. Used to pass information asynchronously from the engine to the application.

ISRSpeaker Optional. Manages speaker profile information, such as for "training" the SR engine to recognize phrases.

ILexPronounce Optional. Lets apps query and control the pronunciation of words.

Speech Recognition Grammar Object

IUnknown	Provides access to other interfaces in the object.
ISRGramCommon	Provides methods to activate and deactivate the grammar object, or archive it to disk.
ISRGramCFG	Provides interfaces specific to context-free grammars and methods to manage lists of words and link grammars together.
ISRGramDictation	Used for dictation grammars. Apps can supply hints about what the user might be dictating next.
ISRGramNotifySink	Supplied by the app. Used to pass grammar notifications from the engine to the app.

Speech Recognition Results Object

(All interfaces are optional except IUnknown)

IUnknown	Provides access to other interfaces in the object.
ISRResAudio	Gets an audio recording of what was spoken.
ISRResBasic	Provides general information about what was spoken, such as the phrase that was recognized and when it was spoken.
ISRResCorrection	Lets the app confirm that the phrase was correctly or incorrectly recognized, so the engine can learn from its mistakes.
ISRResEval	Tells the engine to re-evaluate a recognition decision based on what it now knows about the context.
ISRResGraph	Provides a graph of alternate recognition hypotheses, either for words or phonemes.
ISRResMemory	Since storing results objects consumes memory, this interface is provided to let apps control how results objects are stored.
ISRResMerge	To merge or split two results objects.
ISRResModifyGUI	Tells the engine to display a graphical user interface so the user can correct a recognition result.
ISRResSpeaker	If an engine supports this, the application can use it to identify that spoke.

Text-to-Speech Engine Object

IUnknown	Provides access to other interfaces in the object.
ITTSAttributes	Controls the attributes of the text-to-speech engine such as the volume, processor usage, speaking speed, and pitch.
ITTSCentral	Controls the engine object. Member functions allow an application to add buffers, and start and stop speech.
ITTSDialogs	Displays windows dialog boxes that allow the end-user to configure the text-to-speech engine, such as correcting word pronunciations.
ITTSBufNotifySink	Supplied by the app. Used to notify the app of changes to text buffer, such as when bookmarks are reached.
ITTSNotifySink	Supplied by the app. Used to notify the app when audio starts or stops, or when attributes are changed.
ILexPronounce	Optional. Lets app query and control the pronunciation of words.

3.4 TTS operation with low-level speech objects

The operation of a TTS engine with a custom destination, as explained in the last chapter is shown in **Figure 6** below:

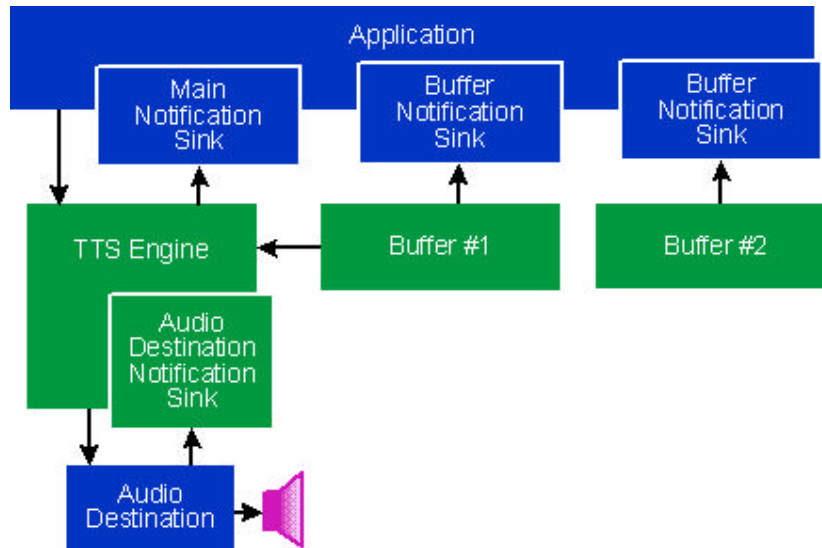
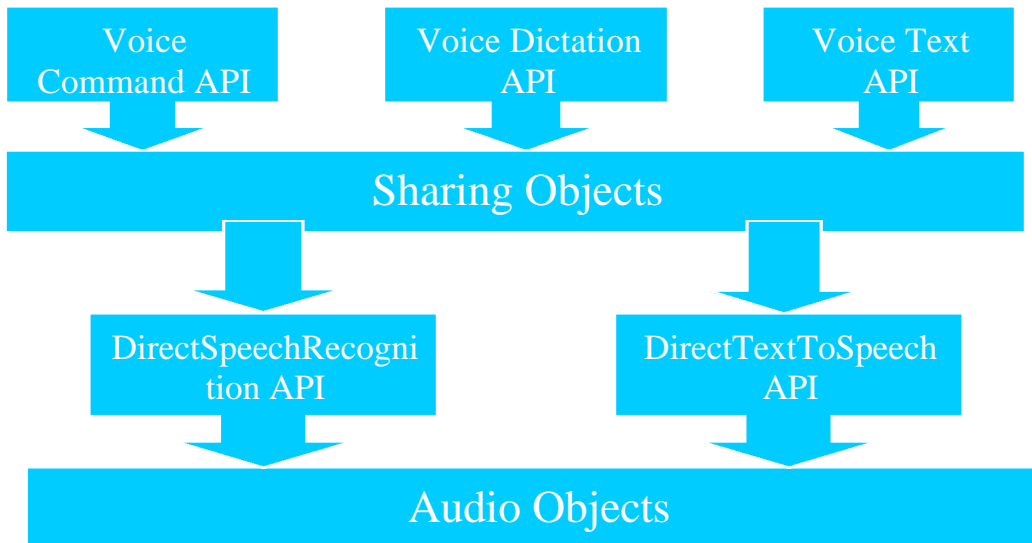


Figure 6 Low-level TTS Objects with Custom Audio Destination

3.5 Audio and Sharing Objects

The speech objects are implemented on several levels. The Voice Command, Voice Dictation, and Voice Text objects occupy the highest level. A Sharing object (which allows the high-levels objects to share speech engines) occupies the next level. The DirectSpeechRecognition and DirectTextToSpeech objects occupy the next level. The audio objects occupy the lowest level. These levels are shown in the following figure:



The DirectSpeechRecognition and DirectTextToSpeech objects provide full access to speech engines. They interface to speech engines at the lowest possible level, giving good speed and maximum control. They load the engines in process and take control of the speakers and microphone. The Voice Command, Voice Dictation, and Voice Text objects provide higher-level access to speech engines.

The Voice Telephony objects allow speech synthesis, speech recognition, wave synthesis, and DTMF on single or multi-line voice telephone devices.

The Speech Tools objects provide commonly used functionality to speed application development.

3.6 Voice Text

Voice Text is the high-level interface for text-to-speech. Adding voice text is fairly simple. An application has to make only the following modifications:

```
CoCreateInstance (CLSID_VTtxt, NULL, CLSCTX_LOCAL_SERVER,
IID_IVoiceText, &pIVoiceText);
```

Calling **CoCreateInstance** creates an instance of the voice text object.


```
pIVoiceText->Register ("", "Demo Application", NULL,  
IID_IVTxtNotifySink, NULL, NULL);
```

Applications have to call **Register** so the voice text knows the name of the application and to what audio device the speech will be played, because some applications will be telephone-aware. Also, an application can provide a notification sink so that it's alerted when speaking starts or stops; but this isn't necessary.

The next step is to send out text to be spoken.

```
pIVoiceText->Speak ("Hello world.", 0, NULL);
```

Finally, when the application is finished using voice text it releases the object.

```
pIVoiceText->Release();
```

Two objects are involved in text-to-speech:

1. **Voice-Text Object** is the object that handles speech and the only one that the application deals with.
2. **Notification Sink** code is optional and is supplied by the application. The voice-text object calls methods in this object when audio starts or stops playing, or with mouth-animation cues.

3.6.1 Voice-Text Object

The Voice-Text Object supports four interfaces:

1. **IDispatch** allows an application to use voice text through OLE Automation.
2. **IVTxtAttributes** controls attributes of a voice-text site such as the audio device, the speaking speed, the text-to-speech mode, and whether text-to-speech is enabled.
3. **IVTxtDialogs** displays Windows dialog boxes that allow an end user to configure the text-to-speech engine.

4. **IVoiceText** registers an application to use voice text on a site and controls playback of text.

3.6.2 Voice-Text Notification Sink

The Voice-Text Notification Sink supports one interface:

1. **IVTxtNotifySink**. is used by the voice-text module to notify an application that speaking has started or stopped, or that an attribute has been changed for the site.

3.7 Low-Level Text-to-Speech

When an application uses the low-level text-to-speech interfaces, it is talking directly to the engine. This provides the application with much more control, but requires more work of it.

The low-level API consists of many more objects than the high-level API (voice text). The process works as follows:

- The application determines where the text-to-speech audio should be sent and creates an audio-destination object through which the engine sends the data. Microsoft supplies an audio-destination object that sends its audio to the multimedia wave-out device, but the application may use customized audio destinations, such as an audio destination that writes to a .wav file.
- The application, through a text-to-speech enumerator object (not shown here, but provided by Microsoft), locates a text-to-speech engine and voice that it wants to use. It then creates an instance of the engine object and passes it the audio-destination object.

- The engine object has a dialog with the audio-destination object to find a common data format for the digital audio. Once an acceptable format is established, the engine creates an Audio-Destination Notification Sink that it passes to the audio-destination object. From then on, the audio-destination object submits status information to the engine through the notification sink.
- The application can then register a Main Notification Sink that receives buffer-independent notifications, such as whether the synthesized voice is speaking and mouth positions for animation.
- When it is ready, the application passes one or more text buffers down to the engine. These will be queued up and then spoken (to the audio destination) by the engine.
- To find out what words are currently being spoken, the application can create a Buffer Notification Sink for every buffer object. When the engine speaks a word, reaches a bookmark, or some other event occurs, it calls functions in the Buffer Notification Sinks. The notification sink is released when the buffer is finished being read.

3.7.1 Main TTS Object

The Main TTS Object supports four interfaces:

1. **ILexPronounce** allows an application to query and control the pronunciation lexicon for a speech-recognition or text-to-speech engine.
2. **ITTSAttributes** controls the attributes of a text-to-speech engine. Member functions allow an application to adjust the pitch, speed, and volume of the voice and the engine's share of the processor.
3. **ITTSCentral** controls an engine object. Member functions allow an application to send text to the engine; inject speech-inflection tags into text as it is spoken; convert Unicode text to a phonemic representation; pause,

resume, and reset the audio output; get information about the text-to-speech mode; register or release a notification interface; get the time that a byte in the audio stream was played, and convert the time to a Win32 FILETIME value.

4. **ITTSDialogs** displays Windows dialog boxes that allow the end-user to configure the text-to-speech engine, such as controlling how symbols and currencies are pronounced.

3.7.2 Main Notification Sink

The Main Notification Sink supports one interface:

1. **ITTSNotifySink** notifies an application of engine-specific events related to processing text into speech, such as a change of attributes, the time that audio starts or stops playing, and hints for synchronizing animation with the phoneme that is being spoken.

3.7.3 Audio-Destination Object

The Audio-Destination Object supports three interfaces:

1. **IAudio** allows an audio-destination or audio-source object to manage its internal buffer and control attributes of the audio device it represents.
2. **IAudioMultiMediaDevice** allows an audio-destination or audio-source object to access features specific to multimedia devices.
3. **IAudioDest** sends information and data to an audio-destination object.

3.7.4 Audio-Destination Notification Sink

The Audio-Destination Notification Sink supports one interface:

1. **IAudioDestNotifySink** notifies a text-to-speech engine of changes to the internal buffer of an audio-destination object.

3.7.5 Buffer Notification Sink

The Buffer Notification Sink supports one interface:

1. **ITTSBufNotifySink** notifies an application of changes to the buffer that contains the text being spoken.

3.8 Significance in E-Mail via Phone

The Microsoft Speech API provides complete application control text-to-speech, which is the backbone of e-mail via phone system. The syndicate selected Microsoft Speech API because it exposes two levels for development, one that is easy for application writing, and the other that provides more flexibility. Its universal acceptability is an added advantage and was one of the main reasons for which it was selected.

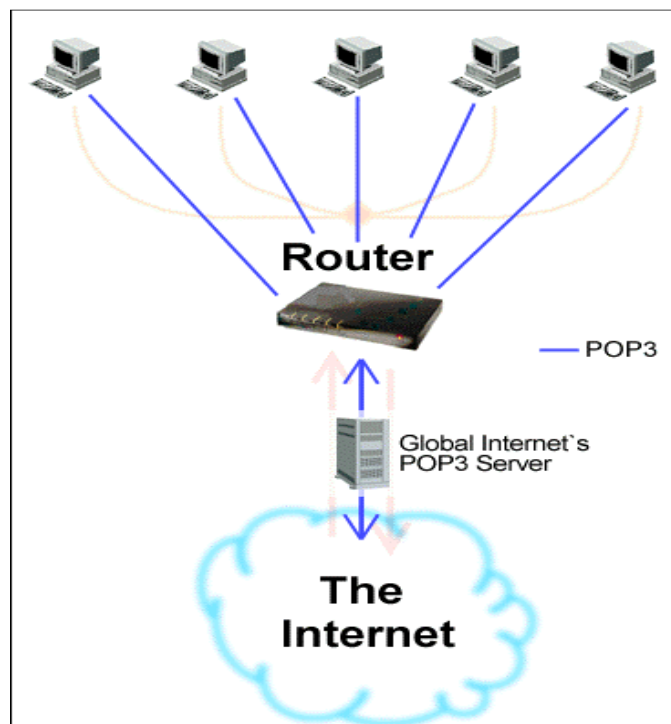
4 E-Mail Client

4.1 Why do we need an E-Mail Client?

The aim of this project was to facilitate a user to check his/her e-mail from any location using a normal touch-tone telephone. For that purpose we needed to develop an e-mail client to download users' e-mail to the system to be converted into speech. Post Office Protocol Version 3 (POP3) was used to develop such a client. Features provided by this e-mail client include list, download and delete the e-mails on a particular user client. POP3 protocol and its implementation are explained in the following.

4.2 Post Office Protocol (POP3)

POP3 is intended to permit a workstation to dynamically access a mail drop on a server host in a useful fashion. Usually, this means that the POP3 protocol is used to allow a workstation or another server to retrieve mail that the server is holding for it, as shown in the following figure:



POP3 is not intended to provide extensive manipulation operations of mail on the server; normally, mail is downloaded and then deleted.

4.2.1 Connection Establishment

Initially, the server host starts the POP3 service by listening on TCP port 110. When a client host wishes to make use of the service, it establishes a TCP connection with the server host. When the connection is established, the POP3 server sends a greeting. The client and POP3 server then exchange commands and responses (respectively) until the connection is closed or aborted.

4.2.2 Commands in POP3

Commands in the POP3 consist of a case-insensitive keyword, possibly followed by one or more arguments. All commands are terminated by a CRLF pair. Keywords and arguments consist of printable ASCII characters. Keywords and arguments are each separated by a single SPACE character. Keywords are three or four characters long. Each argument may be up to 40 characters long.

4.2.3 Responses in POP3

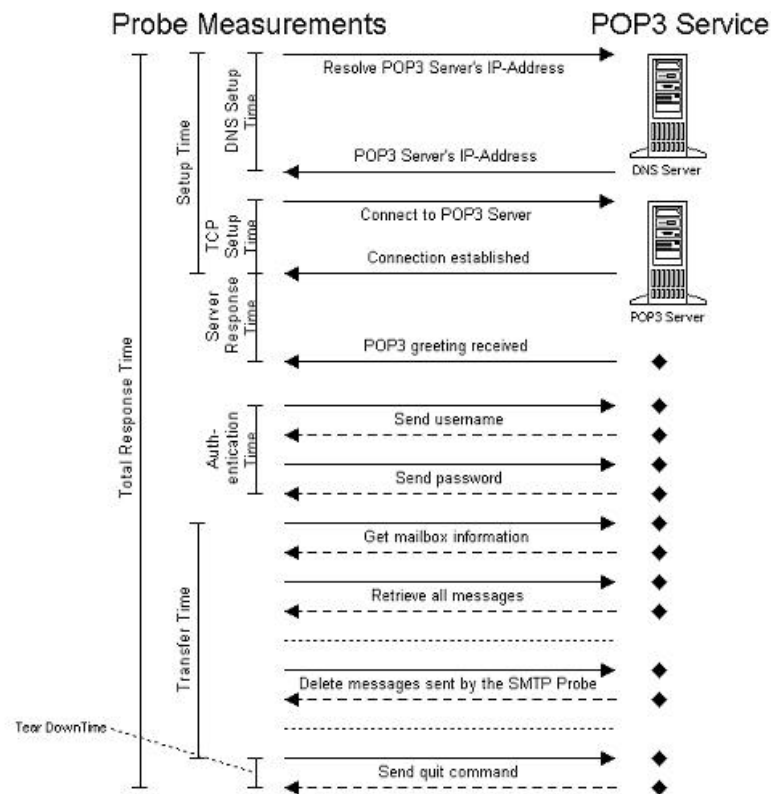
Responses in the POP3 consist of a status indicator and a keyword possibly followed by additional information. All responses are terminated by a CRLF pair. Responses may be up to 512 characters long, including the terminating CRLF. There are currently two status indicators: positive ("+OK") and negative ("-ERR"). Servers **MUST** send the "+OK" and "-ERR" in upper case.

Responses to certain commands are multi-line. In these cases, which are clearly indicated below, after sending the first line of the response and a CRLF, any additional lines are sent, each terminated by a CRLF pair. When all lines of the response have been sent, a final line is sent, consisting of a termination octet (decimal code 046, ".") and a CRLF pair. If any line of the multi-line response begins with the termination octet, the line is "byte-stuffed" by pre-pending the termination octet to that line of the response. A server must respond to an unrecognised, unimplemented, or syntactically invalid command by responding with

a negative status indicator. A server must respond to a command issued when the session is in an incorrect state by responding with a negative status indicator. There is no general method for a client to distinguish between a server that does not implement an optional command and a server that is unwilling or unable to process the command.

4.2.4 State Transitions

The functionality of POP3 Client consists of three states. These states are briefly explained below and are depicted in the following figure:



4.2.4.1 Authorization State

A POP3 session progresses through a number of states during its lifetime. Once the TCP connection has been opened and the POP3 server has sent the greeting, the

session enters the Authorization state. In this state, the client must identify itself to the POP3 server.

Once the POP3 server has determined through the use of an authentication command that the client should be given access to the appropriate mail drop, the POP3 server then acquires an exclusive-access lock on the mail drop, as necessary to prevent messages from being modified or removed before the session enters the UPDATE state.

If the lock is successfully acquired, the POP3 server responds with a positive status indicator. The POP3 session now enters the TRANSACTION state, with no messages marked as deleted. If the mail drop cannot be opened for some reason (for example, a lock can not be acquired, the client is denied access to the appropriate mail drop, or the mail drop cannot be parsed), the POP3 server responds with a negative status indicator. (If a lock was acquired but the

POP3 server intends to respond with a negative status indicator, the POP3 server must release the lock prior to rejecting the command.) After returning a negative status indicator, the server may close the connection. If the server does not close the connection, the client may either issue a new authentication command and start again, or the client may issue the QUIT command.

After the POP3 server has opened the mail drop, it assigns a message-number to each message, and notes the size of each message in octets. The first message in the mail drop is assigned a message-number of "1", the second is assigned "2", and so on, so that the nth message in a mail drop is assigned a message-number of "n". In POP3 commands and responses, all message-numbers and message sizes are expressed in base-10 (i.e., decimal).

Here is the summary for the QUIT command when used in the AUTHORIZATION state:

4.2.4.1.1 QUIT

Arguments:

None

Restrictions:

None

Possible Responses:

+OK

Examples:

C: QUIT

S: +OK dewey POP3 server signing off

4.2.4.2 Transaction State

Once the client has successfully done this, the server acquires resources associated with the client's mail drop, and the session enters the Transaction state. In this state, the client requests actions on the part of the POP3 server.

The POP3 commands valid in the TRANSACTION state are:

4.2.4.2.1 STAT

Arguments:

None

Restrictions:

May only be given in the TRANSACTION state

The POP3 server issues a positive response with a line containing information for the mail drop. This line is called a "drop listing" for that mail drop.

In order to simplify parsing, all POP3 servers are required to use a certain format for drop listings. The positive response consists of "+OK" followed by a single space, the number of messages in the mail drop, a single space, and the size of the mail drop in octets. This memo makes no requirement on what follows the mail drop size.

Minimal implementations should just end that line of the response with a CRLF pair. More advanced implementations may include other information.

Possible Responses:

+OK nn mm

Examples:

C: STAT

S: +OK 2 320

4.2.4.2.2 LIST [msg]

Arguments:

A message-number (optional), which, if present, may NOT refer to a message marked as deleted

Restrictions:

May only be given in the TRANSACTION state

If an argument was given and the POP3 server issues a positive response with a line containing information for that message. This line is called a "scan listing" for that message.

If no argument was given and the POP3 server issues a positive response, then the response given is multi-line. After the initial +OK, for each message in the mail drop, the POP3 server responds with a line containing information for that message. This line is also called a "scan listing" for that message. If there are no messages in the mail drop, then the POP3 server responds with no scan listings--it issues a positive response followed by a line containing a termination octet and a CRLF pair.

In order to simplify parsing, all POP3 servers are required to use a certain format for scan listings. A scan listing consists of the message-number of the message, followed by a single space and the exact size of the message in octets. Methods for calculating the exact size of the message are described in the "Message Format" section below. This memo makes no requirement on what follows the message size in the scan listing. Minimal implementations should just end that line of the response with a CRLF pair. More advanced implementations may include other information, as parsed from the message.

Possible Responses:

+OK scan listing follows

-ERR no such message

Examples:

C: LIST

S: +OK 2 messages (320 octets)

S: 1 120

S: 2 200

S: .

...

C: LIST 2

S: +OK 2 200

...

C: LIST 3

S: -ERR no such message, only 2 messages in mail drop

4.2.4.2.3 RETR [msg]

Arguments:

A message-number (required) which may NOT refer to a message marked as deleted

Restrictions:

May only be given in the TRANSACTION state

If the POP3 server issues a positive response, then the response given is multi-line. After the initial +OK, the POP3 server sends the message corresponding to the given message-number, being careful to byte-stuff the termination character (as with all multi-line responses).

Possible Responses:

+OK message follows

-ERR no such message

Examples:

C: RETR 1

S: +OK 120 octets

S: <the POP3 server sends the entire message here>

S: .

4.2.4.2.4 DELE [msg]

Arguments:

A message-number (required) that may NOT refer to a message marked as deleted

Restrictions:

May only be given in the TRANSACTION state

The POP3 server marks the message as deleted. Any future reference to the message-number associated with the message in a POP3 command generates an error. The POP3 server does not actually delete the message until the POP3 session enters the UPDATE state.

Possible Responses:

+OK message deleted

-ERR no such message

Examples:

C: DELE 1

S: +OK message 1 deleted

...

C: DELE 2

S: -ERR message 2 already deleted

4.2.4.2.5 NOOP

Arguments:

None

Restrictions:

May only be given in the TRANSACTION state

The POP3 server does nothing, it merely replies with a positive response.

Possible Responses:

+OK

Examples:

C: NOOP

S: +OK

4.2.4.2.6 RSET

Arguments:

None

Restrictions:

May only be given in the TRANSACTION state

If any messages have been marked as deleted by the POP3 server, they are unmarked. The POP3 server then replies with a positive response.

Possible Responses:

+OK

Examples:

C: RSET

S: +OK mail drop has 2 messages (320 octets)

4.2.4.3 Update State

When the client has issued the QUIT command, the session enters the Update state. In this state, the POP3 server releases any resources acquired during the Transaction state and says goodbye. The TCP connection is then closed. A POP3 server may have an inactivity auto logout timer. Such a timer must be of at least 10 minutes' duration. The receipt of any command from the client during that interval should suffice to reset the auto logout timer. When the timer expires, the session does not enter the Update state--the server should close the TCP connection without removing any messages or sending any response to the client.

If a session terminates for some reason other than a client-issued QUIT command, the POP3 session does NOT enter the UPDATE state and MUST not remove any messages from the mail drop.

4.2.4.3.1 QUIT

Arguments:

None

Restrictions:

None

The POP3 server removes all messages marked as deleted from the mail drop and replies as to the status of this operation. If there is an error, such as a resource shortage, encountered while removing messages, the mail drop may result in having some or none of the messages marked as deleted be removed. In no case may the server remove any messages not marked as deleted.

Whether the removal was successful or not, the server then releases any exclusive-access lock on the mail drop and closes the TCP connection.

Possible Responses:

+OK

-ERR some deleted messages not removed

Examples:

C: QUIT

S: +OK dewey POP3 server signing off (mail drop empty)

...

C: QUIT

S: +OK dewey POP3 server signing off (2 messages left)

5 Telephony Module

Telephony applications are applications that are accessed via the telephone rather than locally over the PC. A GUI application may also support telephony features, although the user interface design for the two interaction mechanisms are significantly different. Many GUI applications support telephony because of the flexibility that a long-distance connection to the PC provides. The telephony module in E-Mail via phone allows the user to dial in to the server from where the service is provided.

5.1 Telephony Applications

Some typical telephony applications include:

5.1.1 Voice Mail or Answering Machine Software.

Most users are familiar with "Voice mail" or computerized answering machine software. These pieces of software allow users to call into a computer and access audio messages that have been left for them. Voice-mail and answering machine software programs are often extended to E-mail, address books, and other types of data. E-Mail via phone can be included in this kind of telephony applications.

5.1.2 Accessing Databases.

Large numbers of telephony applications allow users to access databases such as movie listings, stock quotes, or news.

5.1.3 Call Routing.

Many of the same telephony applications that provide voice-mail or database access also allows incoming calls to be routed to other phone lines. Because most

contemporary call routing systems rely on DTMF (touch-tone) to rout the call they ask for an extension number, but with speech recognition this could just as easily be a name.

5.2 Hardware and Software Requirements

Telephony applications use the same speech recognition engines used for Command and Control speech recognition, and the same text-to-speech engines used on the PC. These hardware and software requirements should be considered when designing a speech application:

5.2.1 Processor speed

The speech recognition and text-to-speech engines currently on the market typically require a 486/66 or faster processor.

5.2.2 Memory

On the average, the combination of speech recognition and text-to-speech will use 2 megabytes (MB) of random-access memory (RAM) in addition to that required by the running application.

5.2.3 Telephony Card

A number of telephony cards are on the market today. On the low end are cards that use FAX/MODEM chips that have been augmented to handle speech. These are included in almost every new home PC. Higher end cards include DSPs or support for multiple phone lines.

5.3 Application Design Considerations

5.3.1 Multi-Line Applications

Most telephony applications are designed to handle several phone lines coming into the same PC. Multi-line telephony applications need to be designed to handle the multiple input channels in such a way that one channel doesn't slow down or harm another channel.

The easiest multi-line application has one process running at least one thread per phone line. Because each line has its own thread, the lines are independent and (generally) one line will not cause another line to slow down. Multi-threaded lines also allow for improved performance on multi-processor CPUs.

The most stable multi-line telephony design is to have one process per phone line. This insures that one phone line cannot crash and pull down the other lines. It also parallelizes well. It is more difficult to code. E-Mail via Phone is initially implemented to cater for one user at a time. Special devices are available in the market that allow up to 16 incoming lines. This feature is proposed in the future expansion possibilities.

5.4 Types of Telephony Today

Different types of telephony services provided these days are briefly mentioned in the following. Since E-Mail via phone is a cross roads of PSTN, IP, Internet and Computer Telephony; it is but obvious that they should be defined.

5.4.1 Public Switched Telephony Network (PSTN)

- Wired or wireless circuit switched phone service delivered in the form of cell phone service, analog lines, or digital lines (BRI ISDN, PRI ISDN, T1/E1).

- Signaling (Call Setup/Teardown/Billing/Number Lookup/Etc) done by a separate network SS7 network – closed controlled secure network.
- Digital signaling for call progress, etc.

5.4.2 Traditional "Computer Telephony (CT)" Technology

5.4.2.1 Basics

- Interfaces a HOST to the PSTN
- Interface cards provide telephony services at the edge of the PSTN network
- Traditionally has been stand-alone systems – purely a host-based solution

5.4.2.2 Common uses

- PBX or IVR type applications (MoviePhone, bank balance, auto-attendant, etc.)

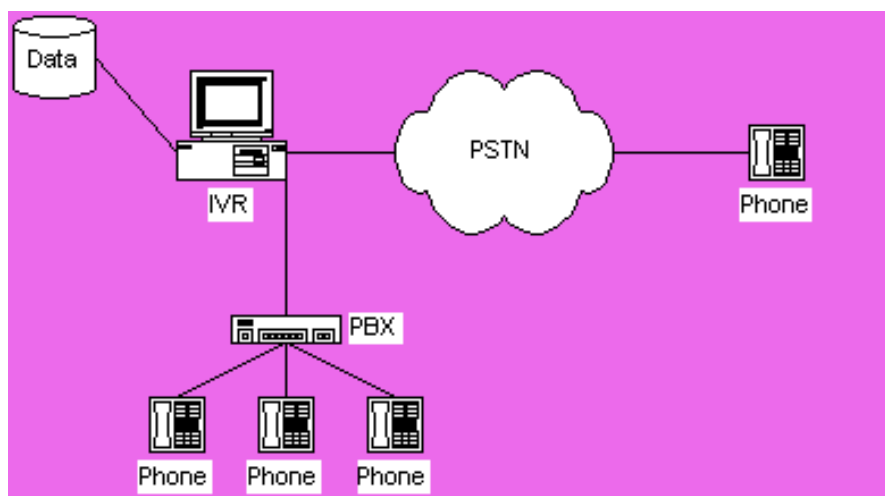


Figure 8: Traditional Computer Telephony

5.4.3 IP Telephony

5.4.3.1 Basics

- Interfaces a LAN to the PSTN.
- Minimum use is traditional CT applications with a network twist.
- Typical use is a basic IP phone system.

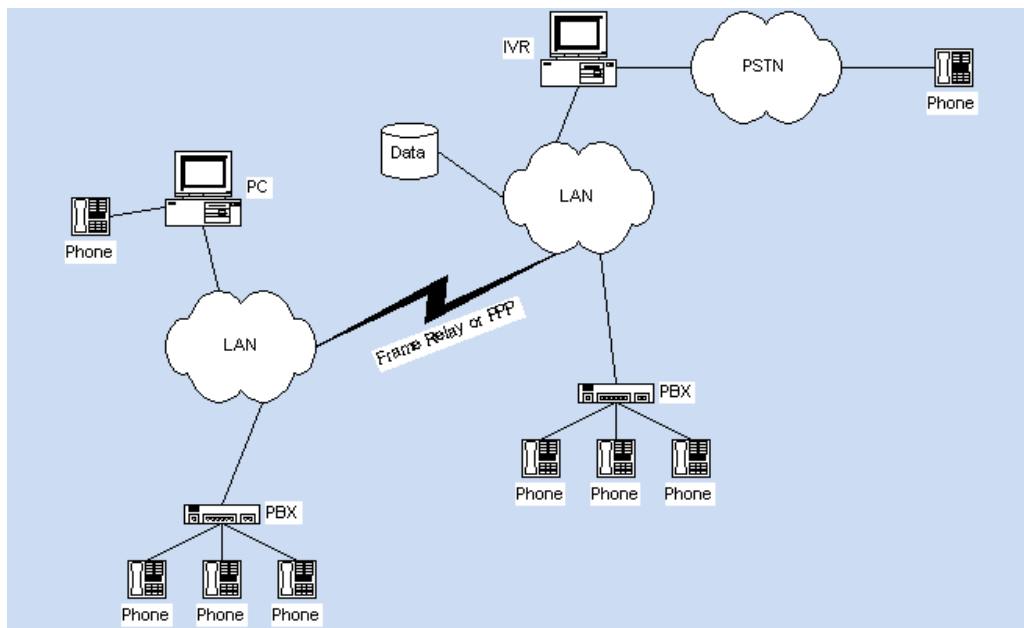


Figure 9: IP Telephony

5.4.3.2 Common uses

- Next generation phone systems with email-vmail bridges, unified messaging, etc.
- Uses either low or high-density hardware, often uses Voice over Frame Relay, dedicated networks.

- Strong deployment of this technology for dedicated phone links over data networks.
- Inter-Office PBX links, especially for international offices.
- Audio requires no significant needs beyond that needed for traditional CT.
- Bandwidth use and latency are not issues, since it's a dedicated LAN application, often on a managed network.
- Signaling is done over the network, not on a separate (SS7) private network.
- Needed a totally new signaling protocol that works over IP networks.
- There are issues with this – security, privacy, etc.

5.4.3.3 Solution

- H.323.
- SIP.
- MGCP.

5.4.4 Internet Telephony

5.4.4.1 Basics

- Telephony applications across a WAN, perhaps with interfaces to the PSTN.
- Core differences: bandwidth and latency are major issues.

5.4.4.2 Requirements

- Audio compression to reduce bandwidth.
- Sensitivity to latency on the audio path – 200 milliseconds is considered acceptable.
- Realization that some packet loss is inevitable and network conditions unpredictable.
- Use Jitter Buffers and codecs that minimize the network impacts.

5.4.4.3 Applications

- Direct Point-to-Point Internet Telephony
- "Hop-on" and/or "Hop-off" applications for Toll-Bypass

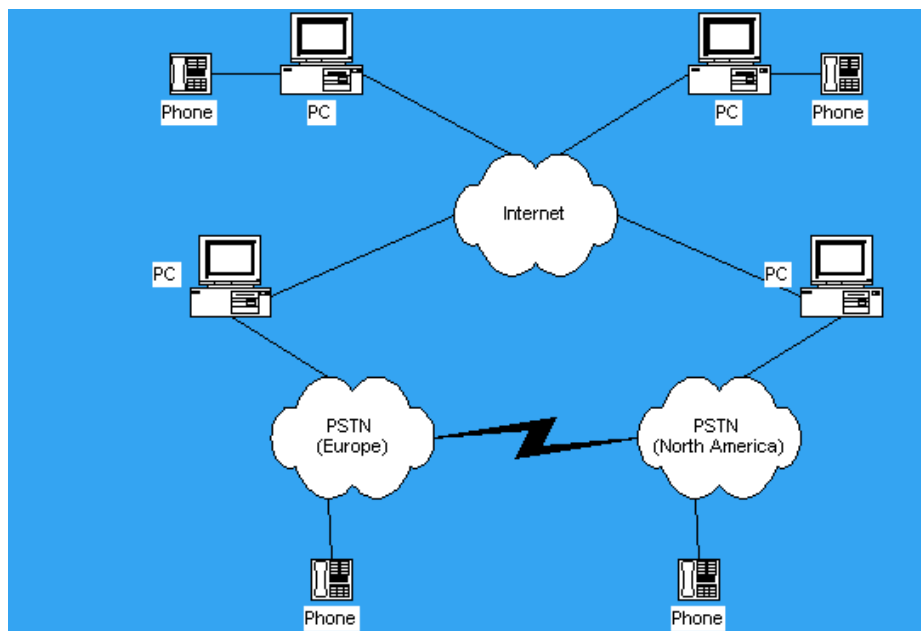


Figure 10: Internet Telephony

5.5 Modems

The word **modem** is a contraction of the words **modulator-demodulator**. A modem is typically used to send digital data over a phone line. The sending modem **modulates** the data into a signal that is compatible with the phone line, and the receiving modem **demodulates** the signal back into digital data. Wireless modems are also frequently seen converting data into radio signals and back.

Modems came into existence in the 1960s as a way to allow terminals to connect to computers over the phone lines. A typical arrangement is shown below:

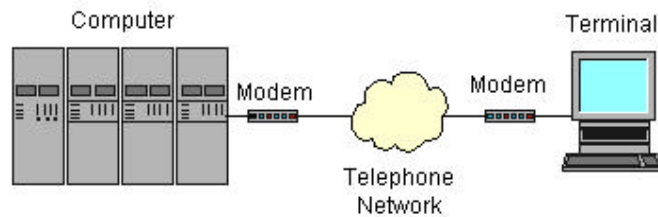


Figure 11: Modem Operation

In a configuration like this, a **dumb terminal** at an off-site office or store could "dial in" to a large, central computer. The 1960s were the age of **time-shared** computers, so a business would often buy computer time from a time-share facility and connect to it via a 300 bit-per-second (BPS) modem.

A dumb terminal is simply a keyboard and a screen. A very common dumb terminal at the time was called the DEC VT-100. The VT-100 could display 25 lines of 80 characters each. When the user typed a character on the terminal, the modem sent the ASCII code the character to the computer. The computer would then send the character back to the computer so it would appear on the screen.

When personal computers started appearing in the late 1970s, **bulletin board systems** became the rage. A person would set up a computer with a modem or two and some BBS software, and other people would dial in to connect to the bulletin board. The users would run **terminal emulators** on their computers to emulate a dumb terminal.

People got along at 300 BPS for quite awhile. The reason this speed was tolerable was because 300 BPS represents about 30 characters per second, and that is much faster than a person can type characters or read. Once people started transferring large programs and images to and from bulletin.

For E-Mail via Phone system, we need a voice modem with a high speed (56 kbps is recommended) at the server end. This modem is the service point between server's telephony module and the user through a normal touch-tone phone. It is assumed that the whole system is functioning in an ideal Intelligent Network.

5.6 The Intelligent Network

The PSTN is an intelligent network throughout much of the world. In practical terms, this means that the network has the capacity to utilize real-time database interactions to control the routing of telephone calls. Many of the services that modern telephone users expect rely upon this capability. As mentioned earlier the environment in which E-Mail via Phone system will be working is supposed to be an Intelligent Network. This supposition is not far from the fact as well because almost 90% of data lines in Pakistan are on fibre optics and controlled by digital switches of PTCL.

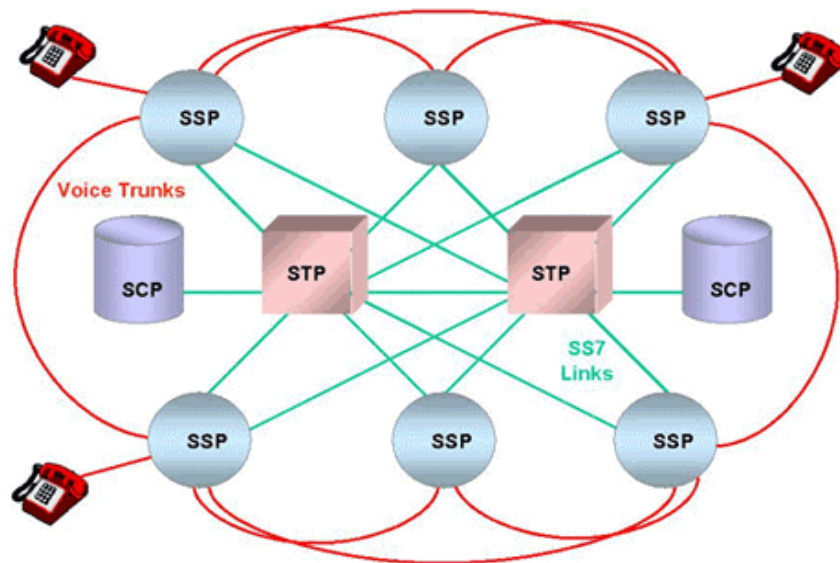


Figure 12: An Intelligent Network

6 DTMF Detection

6.1 What is DTMF?

The word DTMF is the acronym for “Dual Tone Multiple Frequency”. DTMF tones are the sounds emitted when one dials a number on a touch-tone phone.

A DTMF codec incorporates an encoder that translates key strokes or digit information into dual-tone signals, as well as a decoder that detects the presence and the information content of incoming DTMF tone signals. Each key on the keypad is identified uniquely by its row frequency and its column frequency.

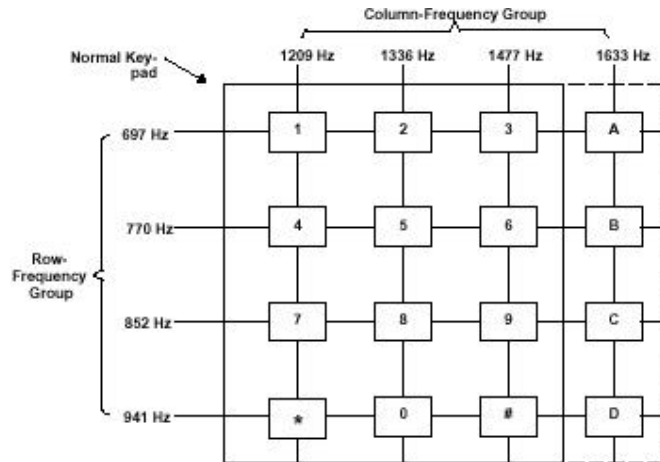


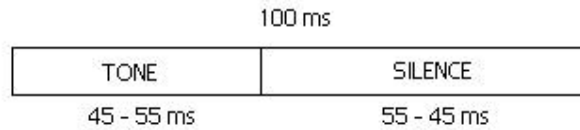
Figure 1. Touch-Tone Telephone Keypad
(A row tone and column tone are associated with each digit.)

Touch Tone Keypad

6.2 AT&T Specifications for Tone Generation:

Tone duration specifications by AT&T state the following: 10 digits/sec is the maximum data rate for touch-tone signals. For a 100-msec time slot, the duration for

the actual tone is at least 45 msec and not longer than 55 msec. The tone generator must be quiet during the remainder of the 100-msec time slot.



6.3 DTMF Tone Generation

Though the domain of this project is limited to DTMF tone detection, the syndicate conducted a thorough study to understand the mechanism of DTMF tone generation. These techniques, in theory, directly affect the algorithm used to detect DTMF tones.

Modems have traditionally been the device used to generate these tones from a computer. But the more sophisticated modems on the market today are nothing more than a DSP (digital signal processor) with accompanying built-in software to generate and interpret analog sounds into digital data. The computers sitting on desk have more CPU power, a more complex OS, and very often a just as sophisticated DSP. There is no reason one cannot duplicate the functionality of a modem from right inside of software, providing with a lot easier to understand and modify code.

6.3.1 Methods of Tone Generation

There are three methods for generating DTMF tones by summing two sine waves. These are:

- Table Look-up
- Taylor Series
- Harmonic Resonator

6.3.1.1 Table Look-up Method

The table look-up method retrieves previously computed sine wave values from memory. The sine function is periodic and only one period must be computed. Since this is sampled data, an accurate sine wave generator must confirm that the sample's starting and ending point are the same. The easiest way to determine this is to find the smallest value of I (an integer) that when multiplied by the ratio below will result in an integer.

$$(F_s / F_o) * I = \text{integer \# of samples}$$

where

F_s = sampling frequency

F_o = frequency of tone to be generated

The period of the frequency to be generated must be evenly divisible by a multiple of the sampling rate. This method can require large amounts of memory if the frequency is not an easy divisor of the sampling rate. If there are numerous frequencies to generate, or the frequency is unknown beforehand, then the table look-up method may not be the best solution.

6.3.1.2 Taylor Series Expansion

The Taylor series expansion method reduces the memory required to compute an approximation of the sine value. The accuracy can be selected. The Taylor series

expansion method expresses a function by polynomial approximation. The expansion for a sine function order 5 is:

$$\sin(x) = 3.140625 * x + 0.02026367 * x^2 - 5.325196 * x^3 + 0.544678 * x^4 + 1.800293 * x^5$$

where $0 < x < \pi/2$.

Note that x is in radians and that the other three quadrants must be accounted for by manipulating the sign and the input value, x . The Taylor series expansion method requires more computations but less memory than the table.

6.3.1.3 Harmonic Resonator

This method is based on two programmable, second-order digital sinusoidal oscillators, one for the row tone and one for the column tone.

Two oscillators, instead of eight, facilitate the code and reduce the code size. Of course, for each digit that is to be encoded, each of the two oscillators needs to be loaded with the appropriate coefficient and initial conditions before oscillation can be initiated.

Since typical DTMF frequencies range from approximately 700 Hz to 1700 Hz, a sampling rate of 8 kHz for this implementation is within a safe area of the Nyquist criteria. The following figure displays the block diagram of the digital oscillator pair. Note that $1/z$ corresponds to a delay of one sampling period.

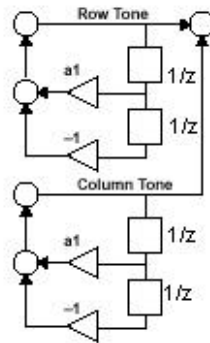


Figure 1 Two Second-Order Digital Sinusoidal Oscillators
(Program-flow description of the DTMF generator)

Figure 14: Two Second Order Digital Sinusoidal Oscillators

6.3.1.3.1 Working of Harmonic Resonator

It is a direct implementation of the Z-transform of a discrete sine function, $\sin(n\omega T)$. Where T is the sampling time, ω is the frequency to be generated in radians per sec.

The transfer function of the oscillator is:

$$H(z) = \frac{b_0}{1 + a_1 z^{-1} + a_2 z^{-2}}$$

$$b_0 = A \sin(\omega_0)$$

$$a_1 = -2 \cos(\omega_0)$$

$$a_2 = 1$$

The complex conjugate poles of the system lie on the unit circle:

$$P_{1,2} = e^{\pm j\omega}$$

The discrete time impulse response

$$\mathbf{H(n)} = \mathbf{A} \sin((\mathbf{n}+1)\omega_0) * \mathbf{u(n)}$$

corresponding to the above second-order system clearly indicates a clean sinusoidal output due to a given impulse input. Therefore, this system can be termed a digital sinusoidal oscillator or digital sinusoidal generator. For the actual implementation of a digital sinusoidal oscillator, the corresponding difference equation is the essential system descriptor, given by

$$\mathbf{y(n)} = -\mathbf{a_1} * \mathbf{y(n-1)} - \mathbf{a_2} * \mathbf{y(n-2)} + \mathbf{b_0} * \delta(\mathbf{n})$$

Where initial conditions $y(-1)$ and $y(-2)$ are zero. Note that the impulse applied at the system input serves the purpose of beginning the sinusoidal oscillation. Thereafter, the oscillation is self-sustaining, as the system has no damping and is exactly marginally stable. Instead of applying a delta impulse at the input, let the initial condition $y(-2)$ be the systems oscillation initiator and remove the input. With this in mind, the final difference equation is given by:

$$\mathbf{y(n)} = 2 * \cos(\omega_0) * \mathbf{y(n-1)} - \mathbf{y(n-2)}$$

where

$$y(-1) = 0$$

$$y(-2) = -A \sin(\omega_0)$$

$$\omega_0 = 2 * \pi * f_0 / f_s$$

With f_s being the sampling frequency, f_0 being the frequency of tone and A being the amplitude of the sinusoid to be generated. Note that the initial condition $y(-2)$ solely determines the actual amplitude of the sine wave.

Coefficients and Initial Conditions for Sinusoidal Oscillators

f(Hz)	a1	y[-1]	y[-2]/A
697	0.85382	0	-0.52047
770	0.82283	0	-0.56857
852	0.78433	0	-0.62033
941	0.73911	0	-0.67358
1209	0.58206	0	-0.81314
1336	0.49820	0	-0.86706
1477	0.39932	0	-0.91680
1633	0.28424	0	-0.95874

6.4 DTMF TONE DETECTION

DTMF tone detection was one of the major tasks involved in this project. Different functions are performed by the system with different keystrokes of the user. Theory behind this whole process is explained in the following.

The task to detect DTMF tones in an incoming signal and to convert them into actual digits is certainly more complex than the encoding process. The decoding process is by its nature a continuous process, meaning it needs to continually search an incoming data stream for the presence of DTMF tones.

6.4.1 Collecting Spectral Information Using Goertzel's Algorithm

The **Goertzel algorithm** is the basis of the DTMF detector. This method is a very effective and fast way to extract spectral information from an input signal. This algorithm essentially utilizes two-pole IIR type filters to compute DFT values effectively. It is, thereby, a recursive structure (always operating on one incoming sample at a time), as compared to the DFT (or FFT) that needs a block of data before being able to start processing.

Another major advantage of Goertzel's algorithm is that it gives only the magnitude of the frequency in demand.

For the actual tone detection, the magnitude (here, squared magnitude) information of the DFT is sufficient. After a certain number of samples N (equivalent to a DFT block size), the Goertzel filter output converges towards a pseudo DFT value $v_k(n)$, which can then be used to determine the squared magnitude.

The Goertzel algorithm is much faster than a true FFT, as only few of the set of spectral line values are needed and only for those values are filters provided.

Squared magnitudes are needed for eight row/column frequencies and for their eight-second harmonics. The second harmonics information later enables discrimination of DTMF tones from speech or music.

The choice of N is mainly driven by the frequency resolution needed, which sets a lower boundary. N also is chosen so that $(k/N)f_s$ most accurately coincides with the actual DTMF frequencies (see Table 1) assuming k_s are integer values and f_s is a sampling frequency of 8 ksp/s.

As the first stage in the tone-detection process, the Goertzel algorithm is one of the standard schemes used to extract the necessary spectral information from an input signal. Essentially, the Goertzel algorithm is a very fast way to compute DFT values under certain conditions. It takes advantage of two facts:

- The periodicity of phase factors w_N^k allows the expression of the computation of the DFT as a linear filter operation utilizing recursive difference equations.
- Only a few of the spectral values of an actual DFT are needed (in this application, there are eight row/column tones plus an additional eight tones or corresponding 2nd harmonics).

Keeping in mind that a DFT of size N is defined as

$$X(k) = \sum_{m=0}^{N-1} x(m) e^{-j 2 \pi k m / N}$$

It is possible to find the sequence of a one-pole resonator

$$Y_k(n) = \sum_{m=0}^{N-1} x(m) e^{j 2 \pi k (n-m) / N}$$

which has a sample value at $n = N$ coinciding exactly with the actual DFT value. In other words, each DFT value $X(k)$ can be expressed in terms of the sample value at $n = N$ resulting from a linear filter process (one-pole filter).

It can be verified that

$$X(k) = Y_k(N) = \sum_{m=0}^{N-1} X(m) e^{-j 2 \pi k m / N}$$

The difference equation corresponding to the above one-pole resonator which is essential for the actual implementation, is given by

$$y_k(n) = e^{j 2 \pi k / N} y_k(n-1) + x(n)$$

with $y(-1) = 0$ and pole location .

Being a one-pole filter, this recursive filter description yet contains complex multiplications, not very convenient for a DSP implementation. Instead, by using a two-pole filter with complex conjugate poles and only real multiplications in its difference equation,

$$v_k(n) = 2 \cos(2\pi k / N) * v_k(n-1) - v_k(n-2) + x(n)$$

where $v_k(-1)$ and $v_k(-2)$ are zero.

In the N th iteration, only a complex multiplication is needed to compute the DFT value, which is

$$X_k = y_k(N) = v_k(N) - e^{-j2\pi k / N} v_k(N-1)$$

However, the DTMF tone-detection process does not need the phase information of the DFT; squared magnitudes of the computed DFT values, in general, suffices.

$$|X(k)|^2 = y_k(N) y_k^*(N)$$

After some arithmetical manipulation, it is found that

$$|X(k)|^2 = v_k^2(N) + v_k^2(N-1) - 2 \cos(2\pi k / N) v_k(N) v_k(N-1)$$

Which gives the energy of the tone.

6.4.2 Validity Checks

Once the spectral information (in the form of squared magnitude at each of the row and column frequencies and their second harmonics) is collected, a series of tests need to be executed to determine the validity of tone and digit results.

6.4.2.1 Signal Strength Check

A first check makes sure the signal strength of the possible DTMF tone pair is sufficient. The sum of the squared magnitudes of the peak spectral row component and the peak spectral column component needs to be above a certain threshold. Since already small twists (row and column tone strength are not equal) result in significant row and column peak differences, the sum of row and column peak provides a better parameter for signal strength than separate row and column checks.

6.4.2.2 Twist Check

Tone twists (the ratio of column to row or row to column signal strength) are investigated in a separate check to make sure the twist ratio specifications are met.

The spectral information can reflect the types of twists.

The more likely one, called “**reverse twist**”, assumes the row peak to be larger than the column peak. Row frequencies (lower frequency band) are typically less attenuated as compared to column frequencies (higher frequency band), assuming a low-pass filter type telephone line. The decoder, therefore, computes a reverse twist ratio and sets a threshold of 8 dB acceptable reverse twist.

The other twist, called “**standard twist**”, occurs when the row peak is smaller than the column peak. Similarly, a “standard twist ratio” is computed and its threshold is set to 4 dB acceptable standard twist.

6.4.2.3 Relative Peak Check

The program makes a comparison of spectral components within the row group as well as within the column group. The strongest component must stand out (in terms of squared amplitude) from its proximity tones within its group by more than a certain threshold ratio.

6.4.2.4 Second Harmonic Strength Check

Finally, the program checks on the strength of the second harmonics in order to be able to discriminate DTMF tones from possible speech or music. It is assumed that the DTMF generator generates tones only on the fundamental frequency; however, speech will always have significant even-order harmonics added to its fundamental frequency component. This second harmonics check, therefore, makes sure that the ratio of the second harmonics component and the fundamental frequency component is below a certain threshold. If the DTMF signal pair passes all these checks, we say a valid DTMF tone pair, which corresponds to a digit, is present.

6.4.2.5 Check For Validity Of Tone

We now need to determine if the valid DTMF tone information contains stable digit information. This is done by mapping the tone-pair to its corresponding digit and comparing it with the previously detected digit. We call the digit information stable if it has been detected twice successively.

6.4.2.6 Check Whether New Digit Pressed

Finally, we compare the detected digit with the previous-to-last digit. Only if the last digit was preceded by a pause do we accept the current digit as a valid digit. The detector is then forced into a state where it waits for a pause before being able to

accept a new digit. This last step is necessary to ensure the discrimination of identical keystrokes succeeding one another.

6.5 Modification in Goertzel Algorithm

Since we only require the magnitude information associated with Goertzel we modify it further to output only the energies.

The block diagram for the further modified version of Goertzel is shown below.

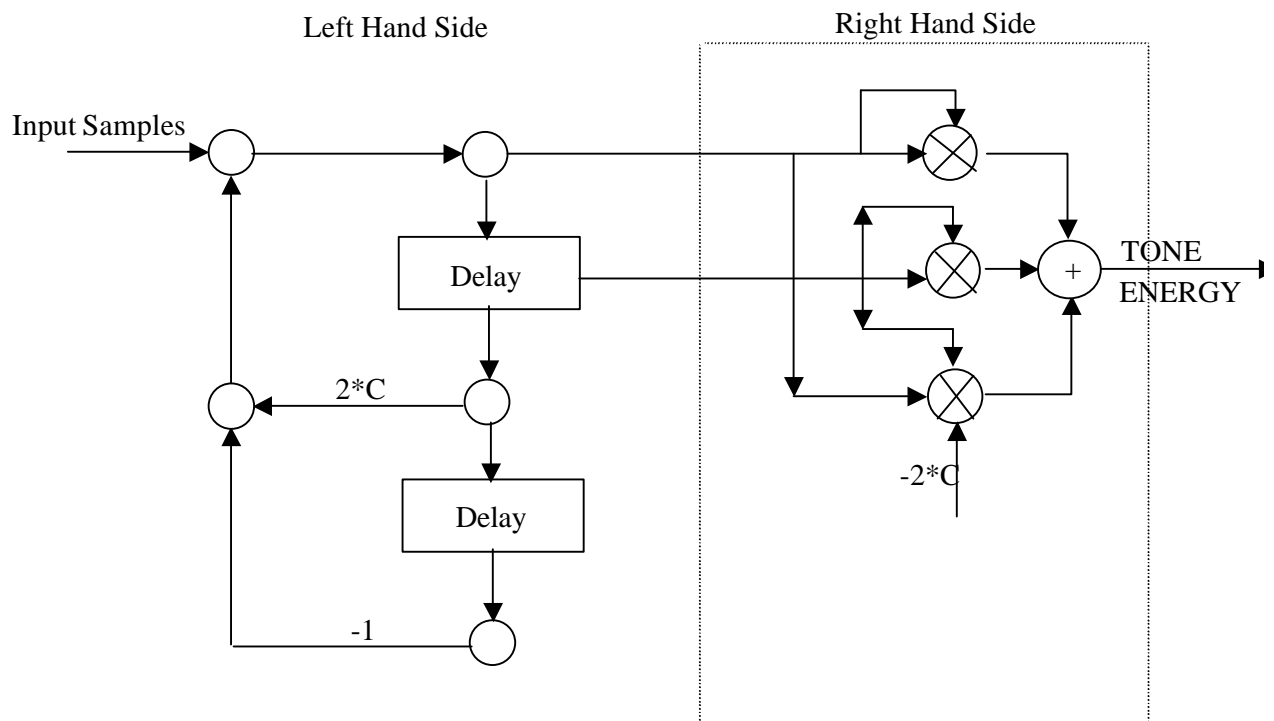
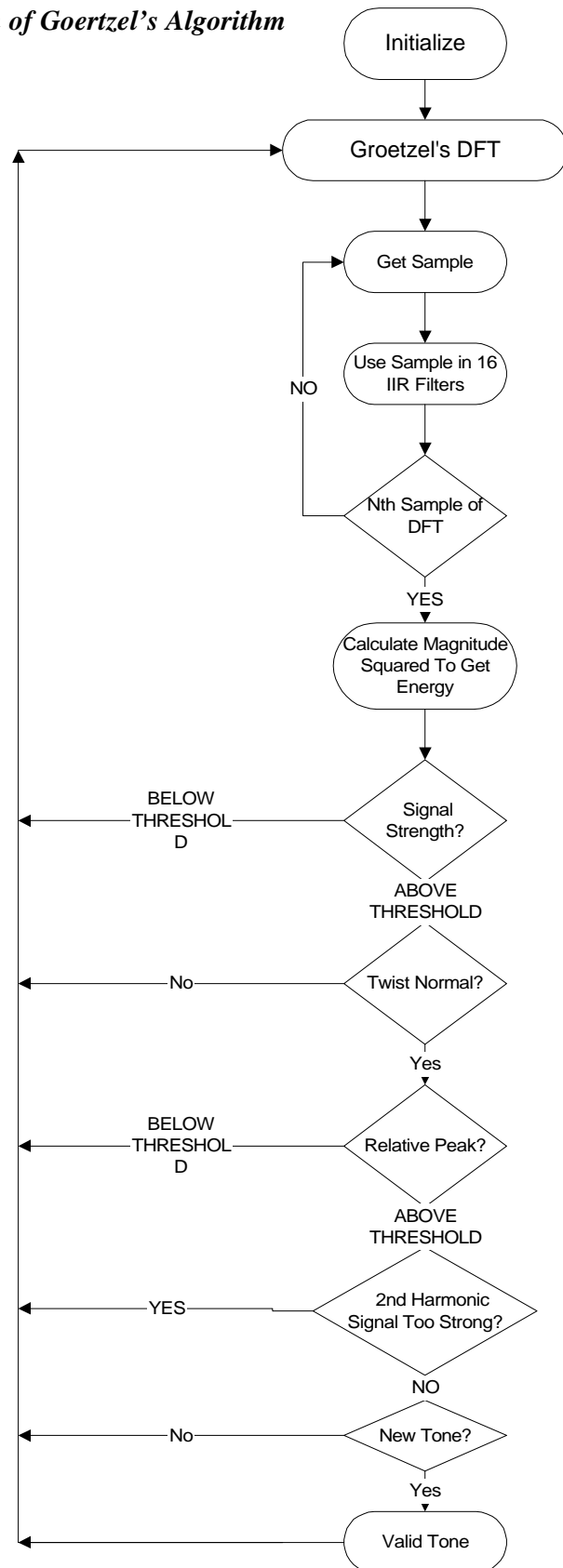


Figure 16: A Simplified Form of Goertzel's Algorithm To Calculate Tone Energy

Figure 17: Flow Chart for the implementation of Goertzel's Algorithm



7 Registration Web Site

7.1 Why do we need a Registration Web Site?

In order to ensure the security of the system and to make it foolproof, some sort of authorization was necessary. For this purpose it was decided by the syndicate that a registration web site will be developed, where the users can register themselves with their e-mail addresses. A unique PIN code is allotted to each user. This PIN is later used as ID cum Password of that user. This web site is developed using Active Server Pages (ASP).

7.2 Active Server Pages: An Introduction

Active Server components actually constitute what we traditionally think of as the 'middle' tier, or business rule layer of an enterprise application. These components are responsible for encapsulating the majority of an application's transaction and database logic. For example, one might decide to place the logic for a complex calculation that requires access to several database tables inside an Active Server component. This component would then be invoked by other Active Server components during execution of one or more of our enterprise applications.

The server-side execution environment that makes much of this possible is named Active Server Pages (formerly code-named "Denali"), an environment in Microsoft Internet Information Server that executes ActiveX Scripts and ActiveX Components on a server.

7.3 Software Development with ASP

Active Server Pages are a key component of Microsoft's dynamic web content strategy. With Active Server Pages, a software developer can create interactive and personalized web pages for their World Wide Web site or corporate intranet without having to understand the internals of a web server or complex application programming interfaces. In addition, Active Server Pages is extensible via software components written using Microsoft's Component Object Model. This last feature

was the main reason because of which the authors of this document decide to develop the web site in ASP, as it allows to take advantage of code we have already written using languages such as Visual Basic, C++ etc.

7.4 Advent of ASP

Active Server Pages were introduced with release 3 of Microsoft's web server, Internet Information Server or IIS. Active Server Pages are actually a series of dynamic link libraries or DLLs that are installed on a web server by either a standalone installation program or as part of the Visual Studio 97 setup for Visual InterDev. These DLLs give IIS the ability to interpret and process information via the use of a script file (called an ASP script) that is resident in a web application directory.

7.5 Features of ASP

Some of the several features of ASP include:

- Active Server Pages is an environment that hosts one of several scripting languages that can be used to produce output, in the form of HTML. This interactivity is the key method used to 'activate' a web site.
- To create an Active Server Page script, we simply write a combination of script and HTML and place it in a file with the extension .ASP.
- Once installed, the Active Server Pages environment will process the script and interact with the server environment to produce HTML that will be sent to the requesting browser.

7.6 Internet Information Server and ASP Development Platform

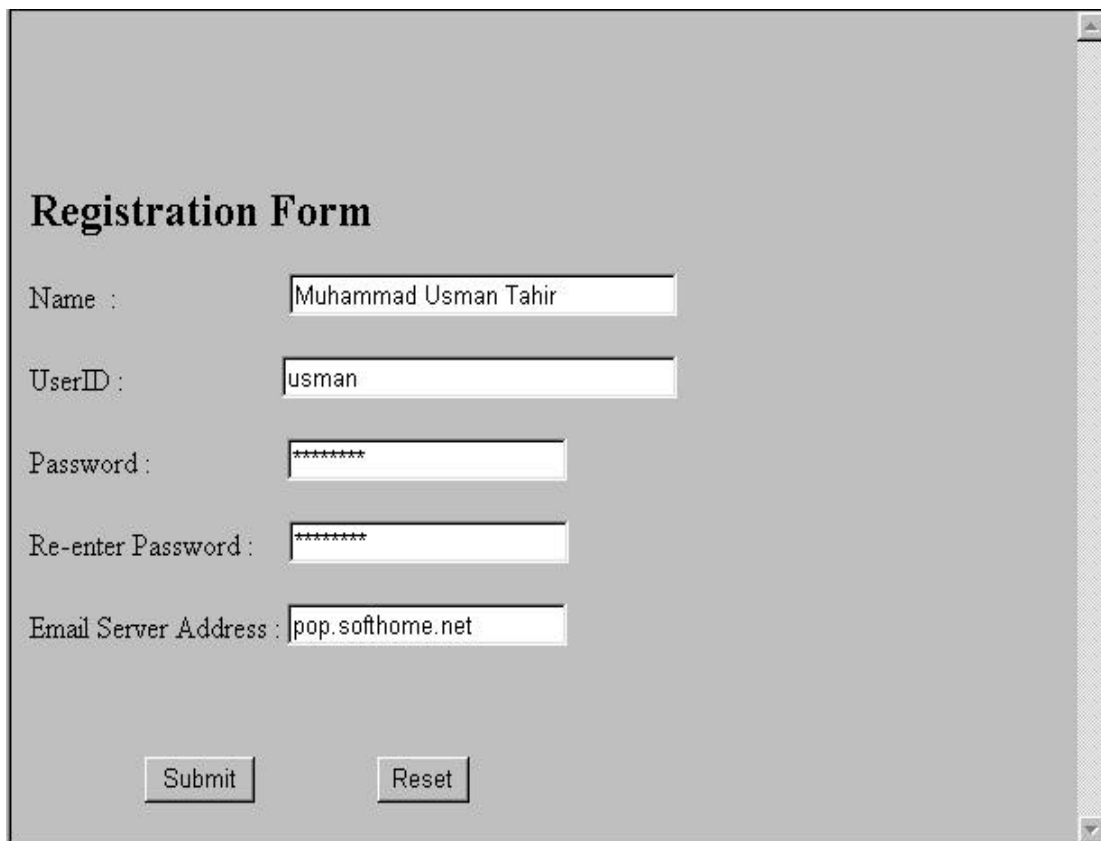
Microsoft's Internet Information Server (IIS) is an open platform designed specifically for creating powerful, scaleable Internet and intranet applications. However, IIS is not just a web server, it's much more. IIS is a complete set of software components that provide the capability to develop robust data-driven web applications.

ASP applications run within the context of IIS. In simple terms, ASP provides the 'glue' that lets developers, like the authors, integrate these components and take advantage of their functionality in applications.

7.7 Implementation of the Web Site

The registration web site was developed, as discussed earlier, using ASP in Microsoft InterDev. This site has an MS Access database at its backend. All the information provided by the user is stored in the database and can be later accessed by the application. Every new user is assigned a unique randomly generated PIN that can later be used for authorization purposes.

A sample screen shot of the main registration form is shown in the below:

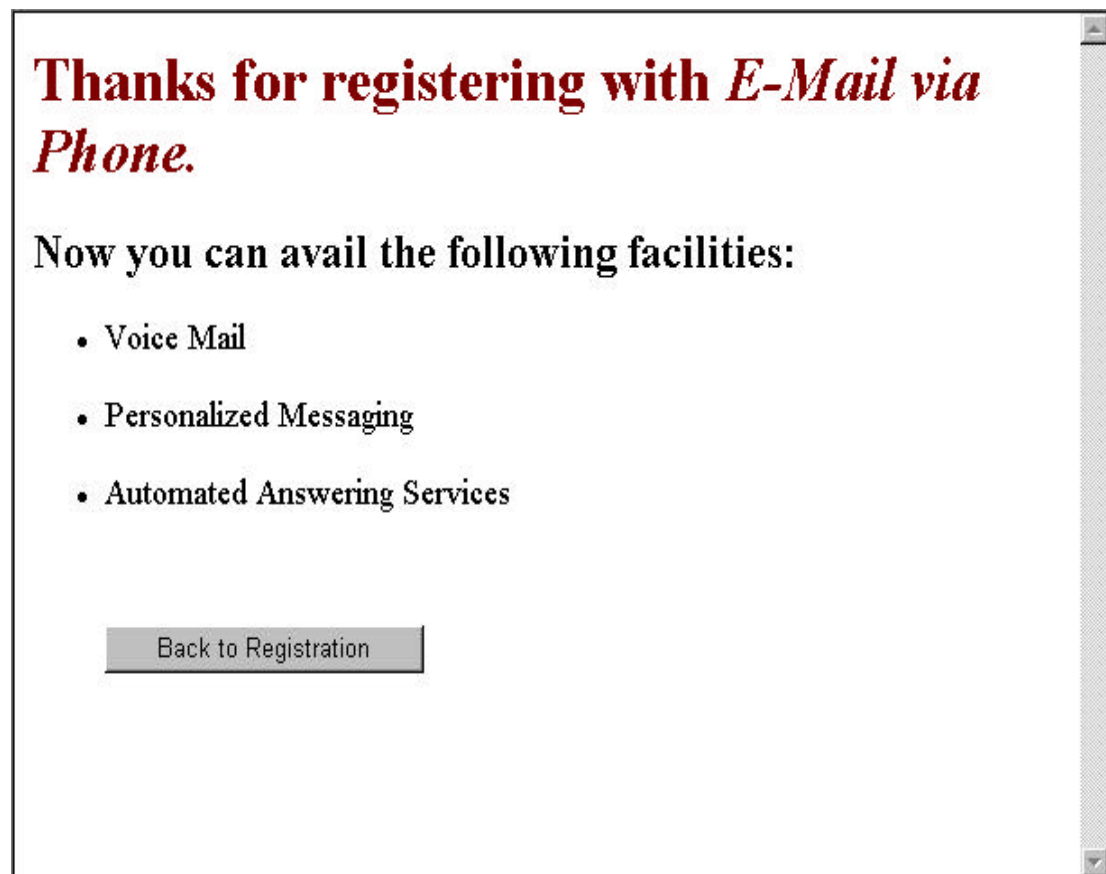


The screenshot shows a web browser window displaying a registration form. The form is titled "Registration Form" and contains several input fields and two buttons. The fields are labeled as follows:

- Name : Muhammad Usman Tahir
- UserID : usman
- Password : *****
- Re-enter Password : *****
- Email Server Address : pop.softhome.net

At the bottom of the form, there are two buttons: "Submit" and "Reset".

When the user is finished with filling in the information and submits his/her request for registration a unique PIN code is assigned and a page similar to the one below appears:



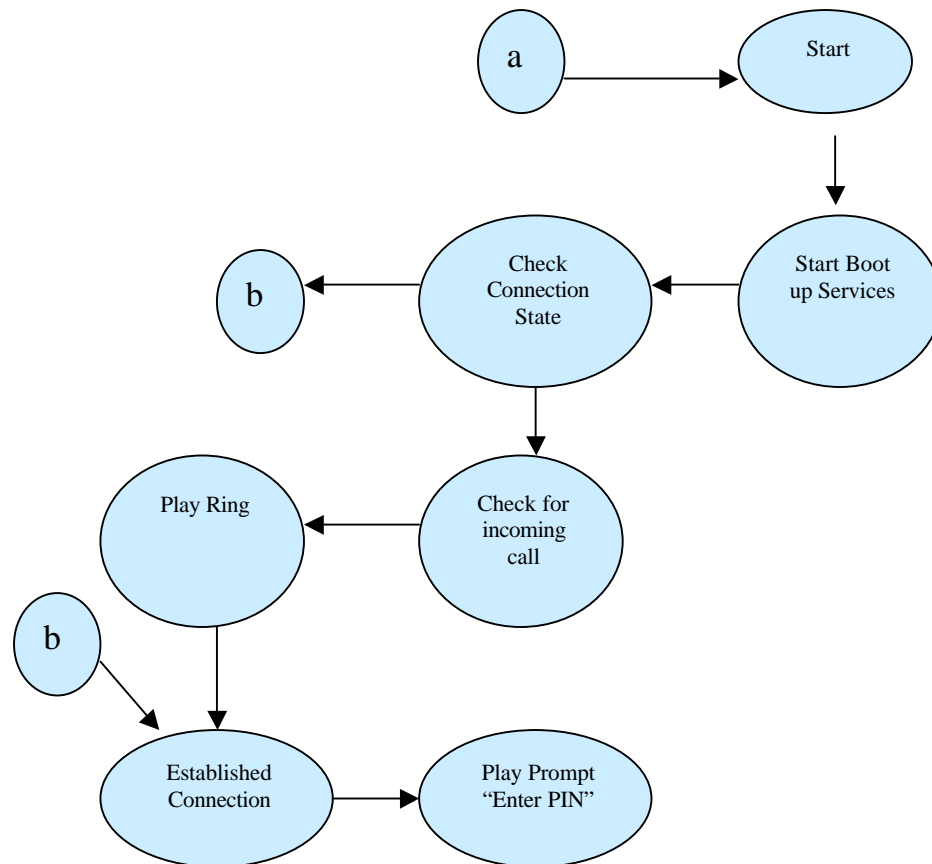
8 E-mail via Phone: System Design

This chapter describes the low level design of E-Mail via Phone system, including the finite state machines, UML class diagrams and use cases developed for implementation of the whole system.

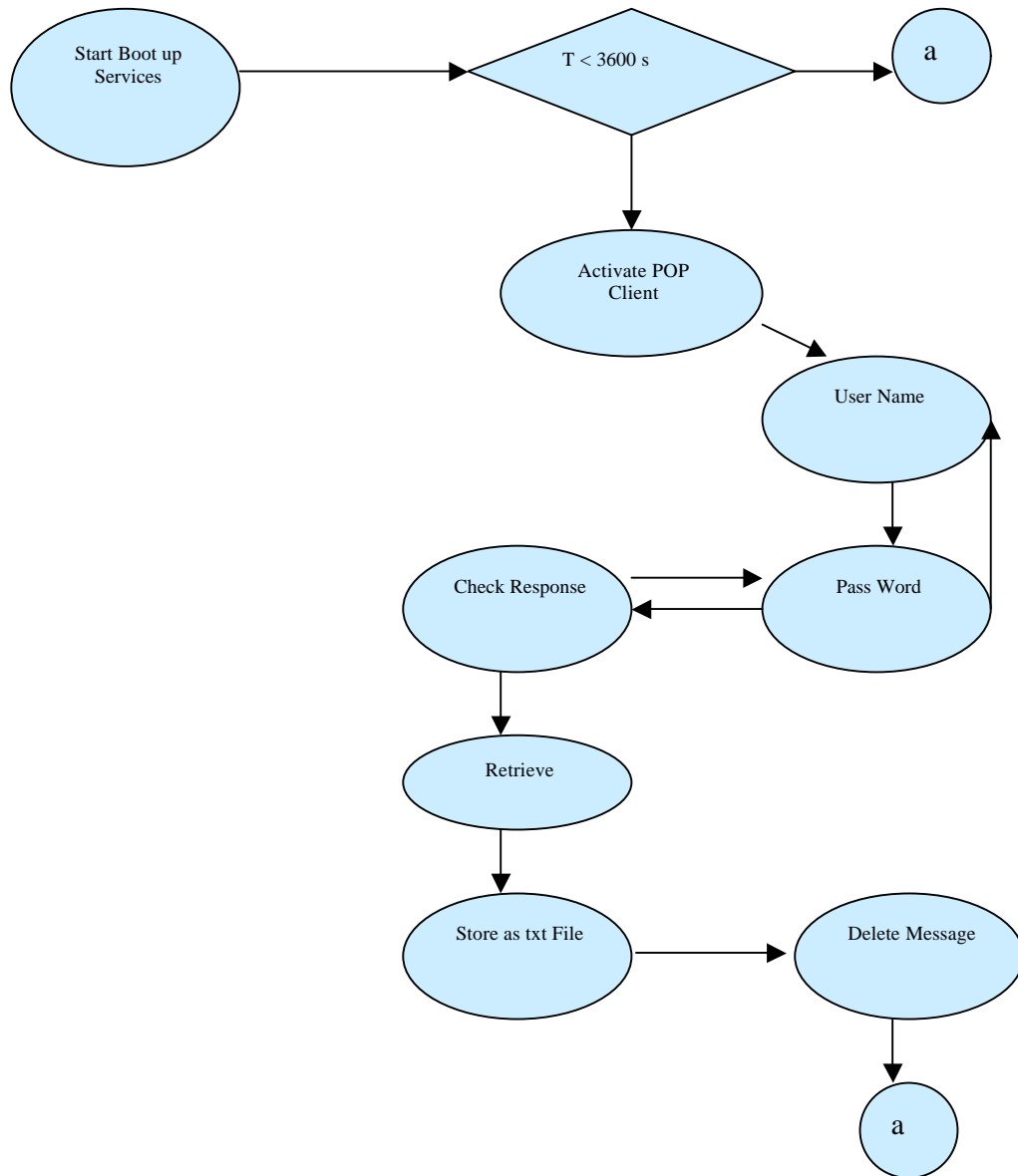
8.1 Finite State Machines

Finite State Machines (FSM) graphically represent all the states E-Mail via Phone system goes through during its lifetime. Different FSM's shown below explain various states of operation, which the system may be in during its entire operation cycle.

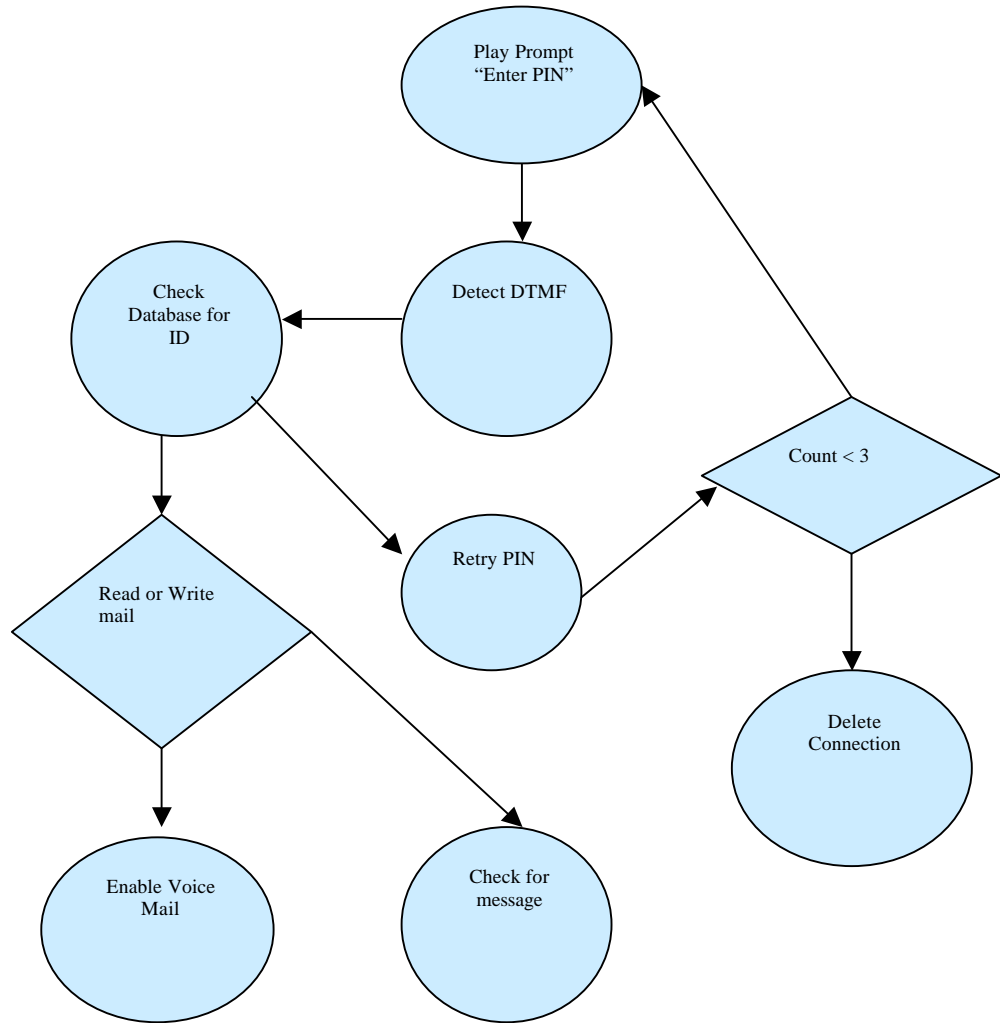
8.1.1 Boot Up Services



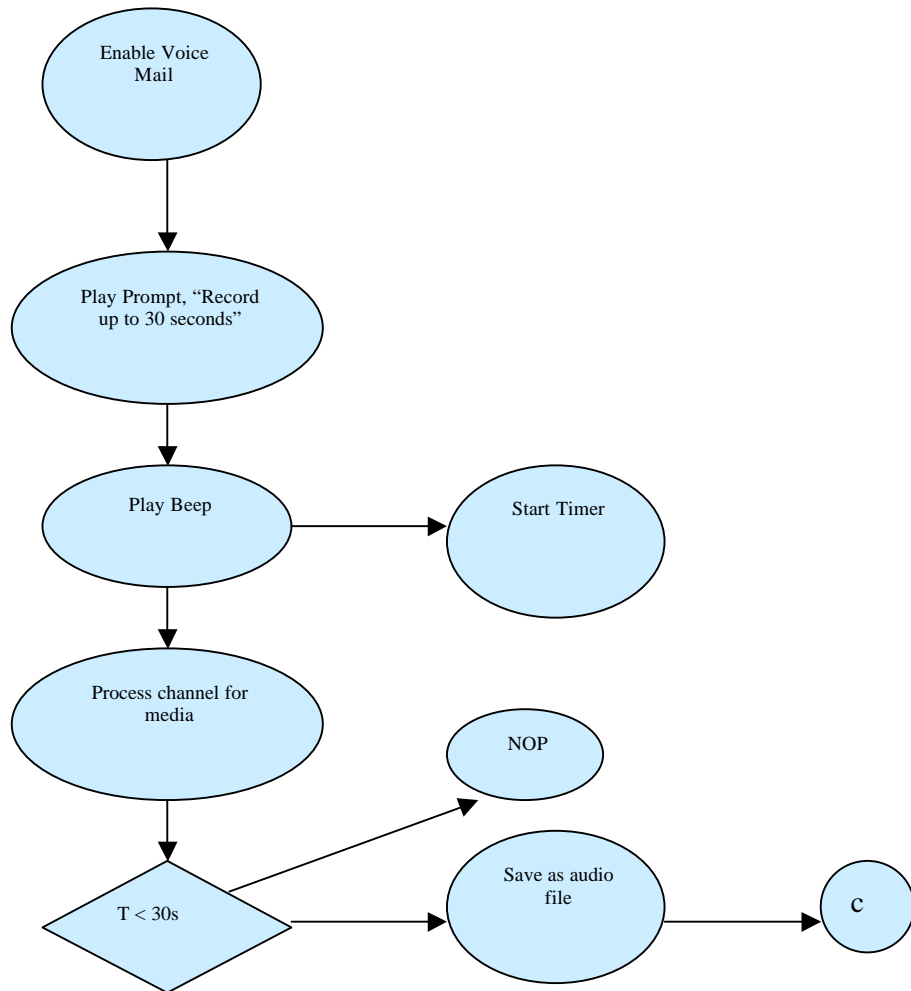
8.1.2 E-mail Download using POP3 Client



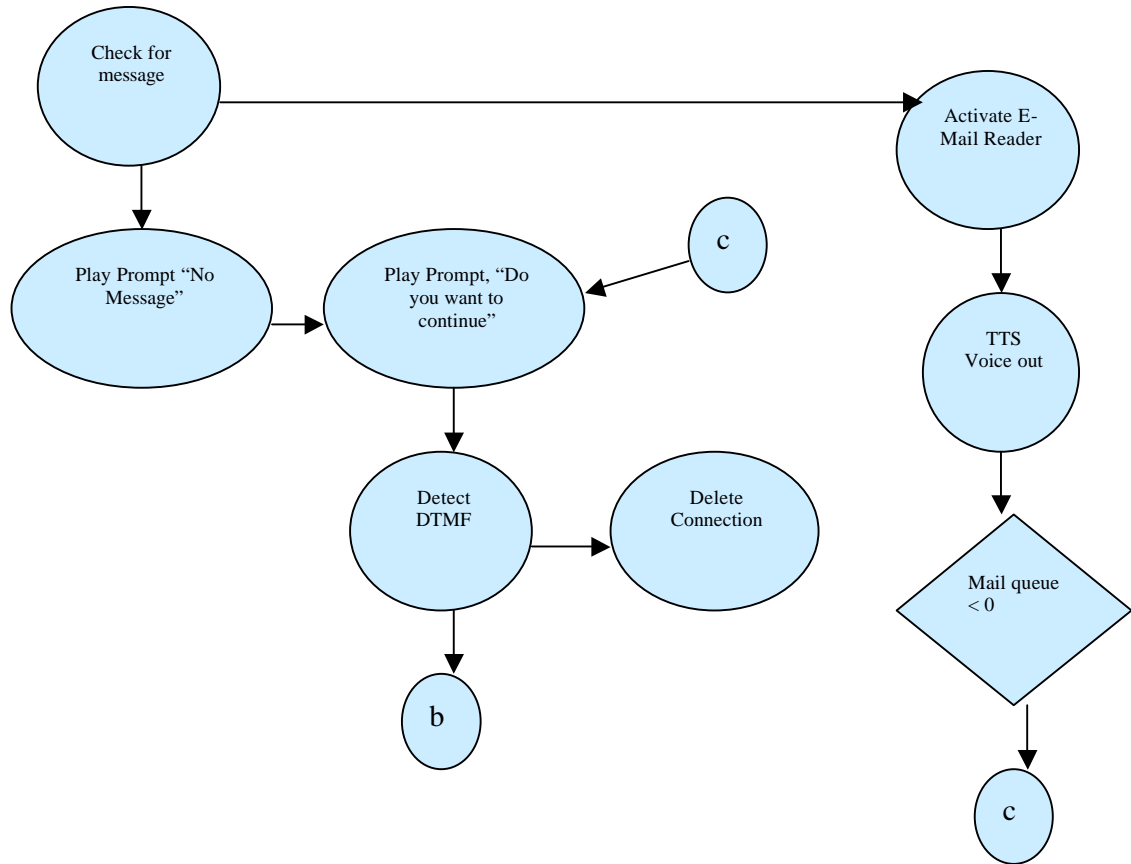
8.1.3 User Authorization/User Menu



8.1.4 Message Recording



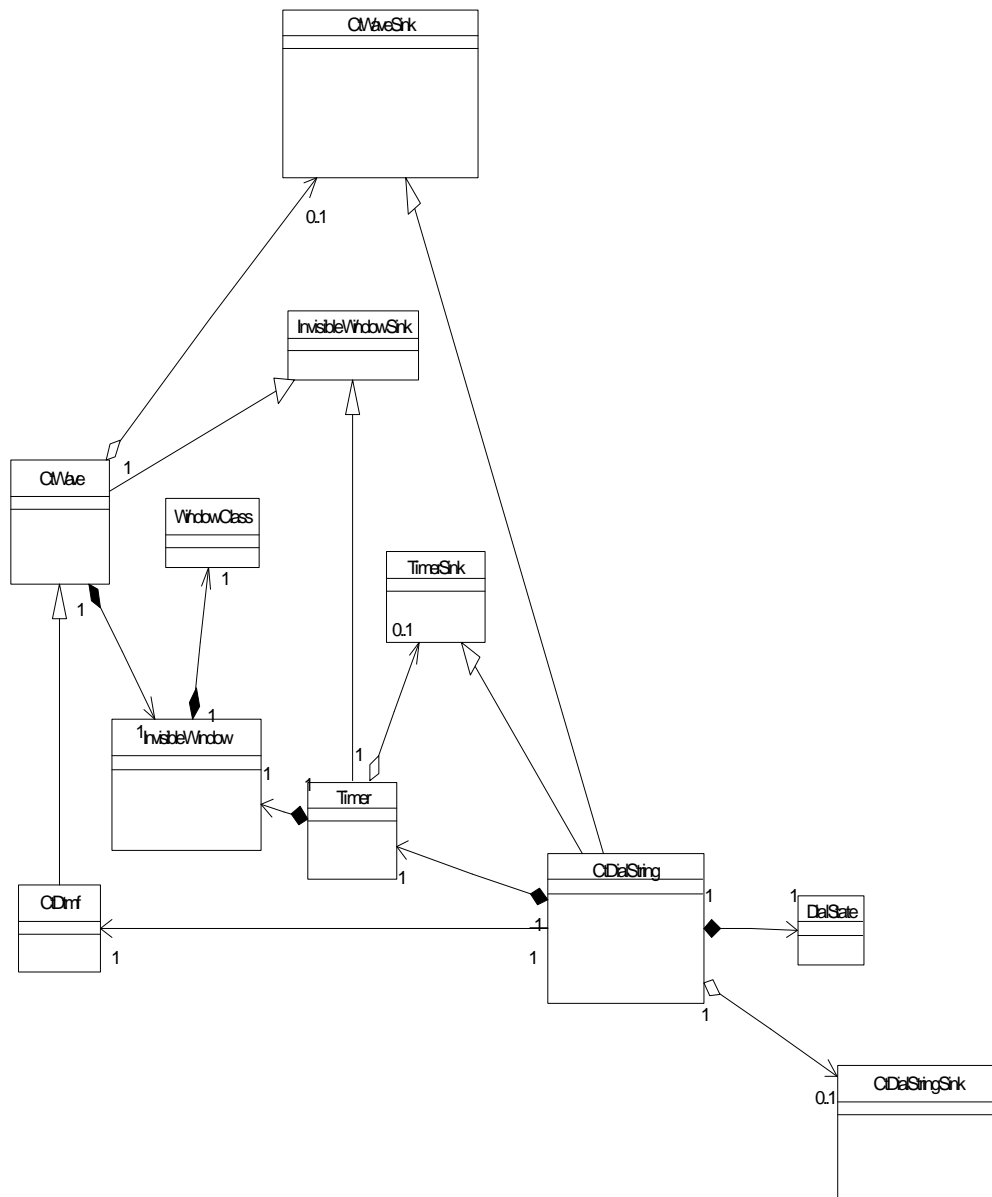
8.1.5 E-Mail Playback using TTS



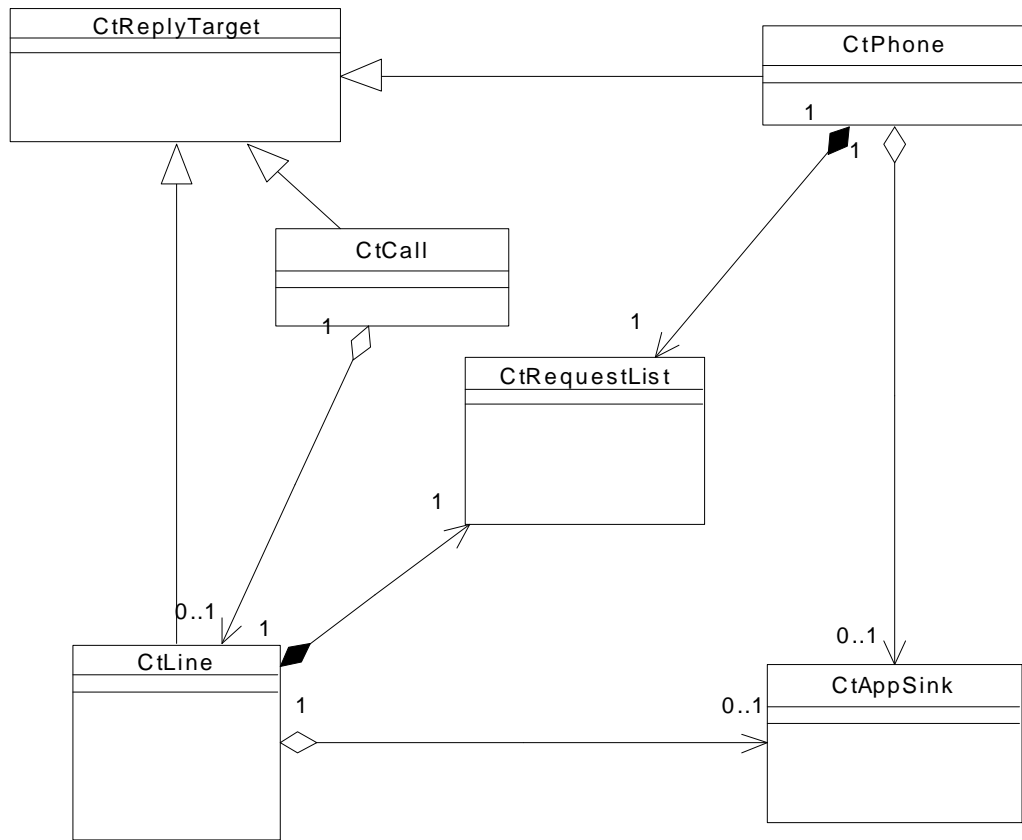
8.2 Class Diagrams

UML Class diagrams in this section explain the classes identified for implementation of the system in Visual C++. Their relationships; collaboration, association, realization and generalization, with other classes is represented using standard UML notations.

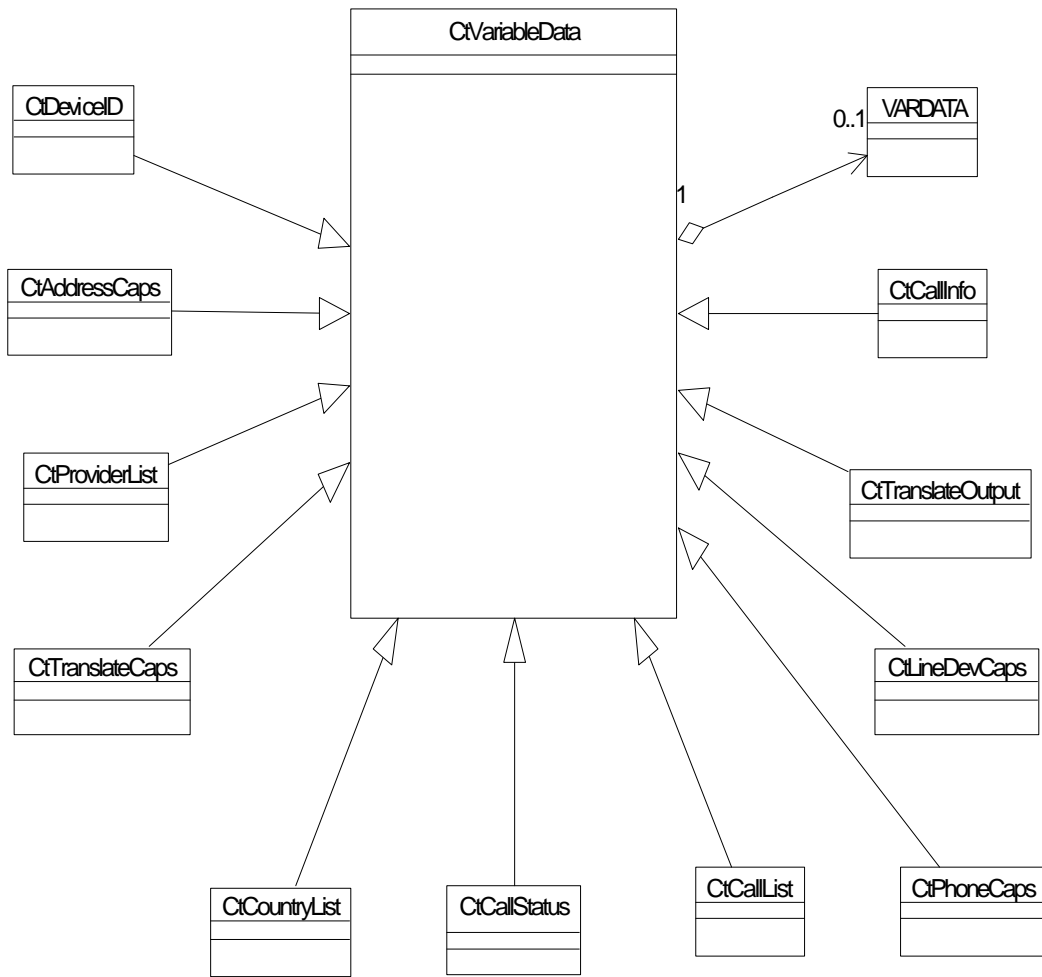
8.2.1 Timer Handler



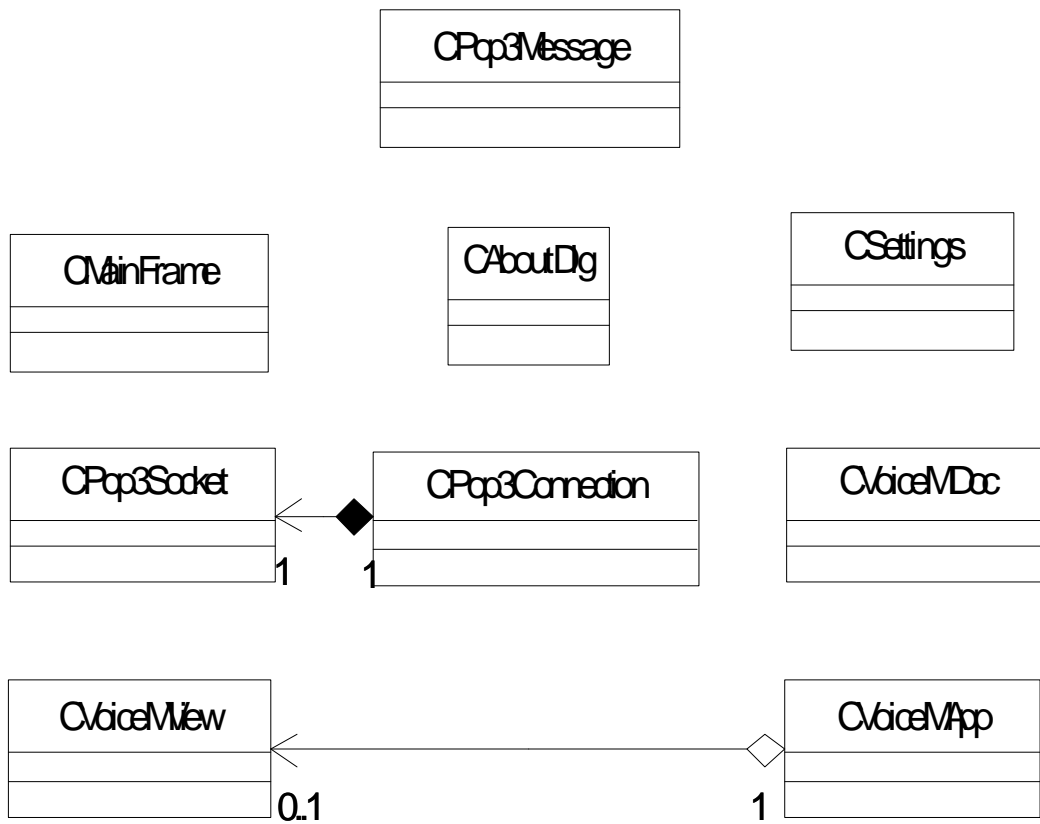
8.2.2 Call Handler I



8.2.4 Device Handler



8.2.5 POP3 Client



8.2.6 Data Types



8.3 Use Cases

Below is a description of a set of sequences of actions, including variants that our system performs to yield an observable result of values to an actor.

8.3.1 Connection Establishment

Overview: This use case explains how the connection is established between the user and the system.

Use Case View:

Use Case Name	Actor's Action	System Response
Connection Establishment	USER DIALS THE SERVER'S PHONE NUMBER.	After three rings the system picks up the call and plays a greeting.

8.3.2 User Authorization

Overview: This use case explains how security checks are observed during the authorization of a user.

Use Case View:

Use Case Name	Actor's Action	System Response
User Authorization.	USER PUNCHES HIS PIN CODE.	1. System detects the DTMF tones and stores in a buffer.
	USER WAITS FOR SERVER'S RESPONSE.	2. Server checks the ID in its database and plays a user menu in case of success or asks two more times for the PIN in case the ID is incorrect.

8.3.3 User Menu

Overview: User Menu use case describes the behaviour of the system when the menu is played to the user.

Use Case View:

Use Case Name	Actor's Action	System Response
User Menu	USER DIALS THE DIGITS CORRESPONDING TO THE DESIRED ACTION (READ/WRITE MESSAGE)	1. System detects the DTMF tone and plays another message.

8.3.4 Message Recorder

Overview: Message recorder analyses the situation when a message is played back to the user.

Use Case View:

Use Case Name	Actor's Action	System Response
User Menu	IF USER WANTS TO RECORD A MESSAGE.	1. System initialises the recorder and plays a beep.

8.3.5 Message Player

Overview: This use case explains how a TTS message is played to the user.

Use Case View:

Use Case Name	Actor's Action	System Response
Message Player	IF THE USER WANTS TO LISTEN TO HIS/HER EMAILS.	1. System initialises the TTS.

8.3.6 Delete Connection Menu

Overview: Connection deletion use case describes the situation when a connection is to be deleted.

Use Case View:

Use Case Name	Actor's Action	System Response
Delete Connection Menu.	AFTER THE EMAIL IS PLAYED OR A MESSAGE IS RECORDED BY THE USER, THE USER WAITS.	1. System detects the DTMF and initialises the TTS.

8.3.7 Connection Deletion

Overview: This use case view explains how a connection is deleted.

Use Case View:

Use Case Name	Actor's Action	System Response
Connection Deletion	THE USER DECIDES TO DROP THE CALL .	1. System frees system resources (buffers, timers etc) and deletes the connection.

8.3.8 Back to Basics

Overview: This use case view explains how a connection is deleted.

Use Case View:

Use Case Name	Actor's Action	System Response
Back to Basics	The user decides to Continue with the call.	1. System detects the DTMF tone and plays the welcome menu, like normal style.

8.3.9 Registration

Overview: This use case view explains how a user is registered at the web site.

Use Case View:

Use Case Name	Actor's Action	System Response
Registration	USER ENTERS HIS/HER NAME, EMAIL ADDRESS AND ITS PASSWORD AT THE E-MAIL VIA PHONE REGISTRATION WEB SITE.	1. The system stores the name and email address of the new user in its database and returns a new PIN code to the user. This PIN code is also stored in the database.

9 E-Mail via Phone: Implementation

This chapter outlines the implementation of E-Mail via Phone system including all three packages related to Telephony (TFX), Voice Messaging (VoiceM) and Integration. This implementation is based on the design explained in the last chapter. The packages explained in this chapter include the classes for POP3 client, Text-to-Speech conversion, message recording and playback. These packages are explained below.

9.1 TFX

TFX library provides all the functionality required for building a telephony applications. API includes listening for incoming calls, initiating calls and processing media from the line. Classes included in this package are described in the following sections.

9.1.1 WindowClass

9.1.1.1 Public Methods:

9.1.1.1.1 WindowClass (pszClassName : LPCTSTR, pfnWndProc : WNDPROC) :

WindowClass

9.1.1.1.2 ~WindowClass () :

9.1.1.1.3 ClassName () : LPCTSTR

9.1.1.1.4 IsRegistered () : bool

9.1.2 InvisibleWindowSink

9.1.2.1 Public Methods:

9.1.2.1.1 OnWindowMessage (hwnd : HWND, nMsg : UINT, wparam : WPARAM, lparam : LPARAM) : LRESULT

9.1.3 InvisibleWindow

9.1.3.1 Public Methods:

9.1.3.1.1 InvisibleWindow () : InvisibleWindow

9.1.3.1.2 ~InvisibleWindow () :

9.1.3.1.3 Create (pSink : InvisibleWindowSink*) : bool

9.1.3.1.4 Destroy () : void

9.1.3.1.5 GetHwnd () : HWND

9.1.3.2 Private Methods:

9.1.3.2.1 InvisibleWindowProc (hwnd : HWND, nMsg : UINT, wparam : WPARAM, lparam : LPARAM) : int

9.1.4 CtAddressCaps

Derived from CtVariableData

9.1.4.1 Public Methods:

9.1.4.1.1 GetAddressCaps (nLineID : DWORD, nAddressID : DWORD) :
TRESULT

9.1.4.1.2 operator const LPLINEADDRESSCAPS () : const
LPLINEADDRESSCAPS

9.1.4.1.3 GetLineDeviceID () : DWORD

9.1.4.1.4 GetAddress () : LPCSTR

9.1.4.1.5 GetAddressSharing () : DWORD

9.1.4.1.6 GetAddressStates () : DWORD

9.1.4.1.7 GetCallInfoStates () : DWORD

9.1.4.1.8 GetCallerIDFlags () : DWORD

9.1.4.1.9 GetCalledIDFlags () : DWORD

9.1.4.1.10 GetConnectedIDFlags () : DWORD

9.1.4.1.11 GetRedirectionIDFlags () : DWORD

9.1.4.1.12 GetRedirectingIDFlags () : DWORD

9.1.4.1.13 GetCallStates () : DWORD

9.1.4.1.14 GetDialToneModes () : DWORD

9.1.4.1.15 GetBusyModes () : DWORD

9.1.4.1.16 GetSpecialInfo () : DWORD

9.1.4.1.17 GetDisconnectModes () : DWORD

9.1.4.1.18 GetMaxNumActiveCalls () : DWORD

9.1.4.1.19 GetMaxNumOnHoldCalls () : DWORD

9.1.4.1.20 GetMaxNumOnHoldPendingCalls () : DWORD

9.1.4.1.21 GetMaxNumConference () : DWORD

9.1.4.1.22 GetMaxNumTransConf () : DWORD

9.1.4.1.23 GetAddrCapFlags () : DWORD

9.1.4.1.24 GetCallFeatures () : DWORD

9.1.4.1.25 GetRemoveFromConfCaps () : DWORD

9.1.4.1.26 GetRemoveFromConfState () : DWORD

9.1.4.1.27 GetTransferModes () : DWORD

9.1.4.1.28 GetParkModes () : DWORD

9.1.4.1.29 GetForwardModes () : DWORD

9.1.4.1.30 GetMaxForwardEntries () : DWORD

- 9.1.4.1.31 GetMaxSpecificEntries () : DWORD
- 9.1.4.1.32 GetMinFwdNumRings () : DWORD
- 9.1.4.1.33 GetMaxFwdNumRings () : DWORD
- 9.1.4.1.34 GetMaxCallCompletions () : DWORD
- 9.1.4.1.35 GetCallCompletionConds () : DWORD
- 9.1.4.1.36 GetCallCompletionModes () : DWORD
- 9.1.4.1.37 GetNumCompletionMessages () : DWORD
- 9.1.4.1.38 GetCompletionMsgText (nMsg : DWORD) : LPCSTR
- 9.1.4.1.39 GetAddressFeatures () : DWORD

9.1.4.2 Protected Methods:

- 9.1.4.2.1 FillBuffer () : TRESULT

9.1.4.3 Private Methods:

- 9.1.4.3.1 GetData () : const LPLINEADDRESSCAPS

9.1.5 CtVariableData

9.1.5.1 Public Methods:

- 9.1.5.1.1 CtVariableData () : CtVariableData
- 9.1.5.1.2 ~CtVariableData () :

9.1.5.2 Protected Methods:

9.1.5.2.1 UpdateData () : TRESULT

9.1.5.2.2 GetStringPtr (nOffset : DWORD, nSize : DWORD, dwStringFormat :
DWORD = STRINGFORMAT_ASCII) : LPCSTR

9.1.5.2.3 GetDataPtr (nOffset : DWORD) : void*

9.1.5.2.4 FillBuffer () : TRESULT

9.1.6 CtAppSink

9.1.6.1 Public Methods:

9.1.6.1.1 OnLineCreate (nLineID : DWORD) : void

9.1.6.1.2 OnPhoneCreate (nPhoneID : DWORD) : void

9.1.6.1.3 OnLineRequest (nRequestMode : DWORD, hRequestWnd : HWND,
nRequestID : TREQUEST) : void

9.1.7 CtCall

Derived from CtReplyTarget

9.1.8 Public Methods:

- 9.1.8.1.1 CtCall (pLine : CtLine*) : CtCall
- 9.1.8.1.2 CtCall (pLine : CtLine*, hCall : HCALL, pInitialSink : CtCallSink* = 0) : CtCall
- 9.1.8.1.3 GetLine () : CtLine*
- 9.1.8.1.4 ~CtCall () :
- 9.1.8.1.5 GetHandle () : HCALL
- 9.1.8.1.6 Attach (hCall : HCALL, pInitialSink : CtCallSink* = 0) : HCALL
- 9.1.8.1.7 Detach () : HCALL
- 9.1.8.1.8 AddSink (pSink : CtCallSink*) : void
- 9.1.8.1.9 RemoveSink (pSink : CtCallSink*) : void
- 9.1.8.1.10 IsRequestPending (nRequestID : TREQUEST = 0, pnRequestType : DWORD* = 0) : BOOL
- 9.1.8.1.11 IsRequestTypePending (nRequestType : DWORD) : BOOL
- 9.1.8.1.12 FromHandle (hCall : HCALL) : CtCall*
- 9.1.8.1.13 Accept (psUserUserInfo : LPCSTR = 0, nSize : DWORD = 0) : TRESULT
- 9.1.8.1.14 Answer (psUserUserInfo : LPCSTR = 0, nSize : DWORD = 0) : TRESULT
- 9.1.8.1.15 Dial (szDestAddress : LPCSTR, dwCountryCode : DWORD = 0) : TRESULT
- 9.1.8.1.16 Drop (psUserUserInfo : LPCSTR = 0, nSize : DWORD = 0) : TRESULT
- 9.1.8.1.17 GenerateDigits (szDigits : LPCSTR, nDuration : DWORD = 0, nDigitMode : DWORD = LINEDIGITMODE_DTMF) : TRESULT
- 9.1.8.1.18 GenerateTone (nToneMode : DWORD, nDuration : DWORD, nCustomTones : DWORD = 0, pCustomTones : LINEGENERATETONE* = 0) : TRESULT
- 9.1.8.1.19 Deallocate () : TRESULT
- 9.1.8.1.20 GatherDigits (pszDigits : LPSTR, nDigits : DWORD, pszTerminationDigits : LPCSTR = 0, nFirstDigitTimeout : DWORD = 5000, nInterDigitTimeout : DWORD = 5000, nDigitMode : DWORD = LINEDIGITMODE_DTMF) : TRESULT
- 9.1.8.1.21 Handoff (szFileName : LPCSTR) : TRESULT

9.1.8.1.22 Handoff (nMediaMode : DWORD) : TRESULT
9.1.8.1.23 MakeCall (szDestAddress : LPCSTR = 0, nCountryCode : DWORD = 0, pInitialSink : CtCallSink* = 0, pCallParams : LINECALLPARAMS* = 0) : TRESULT
9.1.8.1.24 MonitorDigits (dwDigitModes : DWORD = LINEDIGITMODE_DTMF) : TRESULT
9.1.8.1.25 OnInfo (nCallInfo : DWORD) : void
9.1.8.1.26 OnState (nCallState : DWORD, dwParam2 : DWORD, nCallPrivilege : DWORD) : void
9.1.8.1.27 OnGatherDigits (nGatherTerm : DWORD) : void
9.1.8.1.28 OnGenerate (nGenerateTerm : DWORD) : void
9.1.8.1.29 OnMonitorDigits (cDigit : char, nDigitMode : DWORD) : void
9.1.8.1.30 OnMonitorMedia (nMediaMode : DWORD) : void
9.1.8.1.31 OnMonitorTone (dwAppSpecific : DWORD) : void
9.1.8.1.32 OnReply (nRequestID : TREQUEST, nResult : TRESULT, nRequestType : DWORD) : void

9.1.8.2 Protected Methods:

9.1.8.2.1 AddToCalls (pCall : CtCall*) : void
9.1.8.2.2 RemoveFromCalls (pCall : CtCall*) : void

9.1.9 CtLine

Derived from CtReplyTarget

9.1.9.1 Public Methods:

- 9.1.9.1.1 CtLine () : CtLine
- 9.1.9.1.2 GetHandle () : HLINE
- 9.1.9.1.3 ~CtLine () :
- 9.1.9.1.4 GetDeviceID () : DWORD
- 9.1.9.1.5 AddSink (pSink : CtLineSink*) : void
- 9.1.9.1.6 RemoveSink (pSink : CtLineSink*) : void
- 9.1.9.1.7 IsRequestPending (nRequestID : TREQUEST = 0, pnRequestType : DWORD* = 0) : BOOL
- 9.1.9.1.8 IsRequestTypePending (nRequestType : DWORD) : BOOL
- 9.1.9.1.9 FromHandle (hLine : HLINE) : CtLine*
- 9.1.9.1.10 GetNumDevs () : DWORD
- 9.1.9.1.11 GetAppHandle () : HLINEAPP
- 9.1.9.1.12 GetAppVersion () : DWORD
- 9.1.9.1.13 SetAppVersion (dwLoVersion : DWORD, dwHiVersion : DWORD) : void
- 9.1.9.1.14 GetApiVersion (nLineID : DWORD) : DWORD
- 9.1.9.1.15 Initialize (pAppSink : CtAppSink*, szAppName : LPCSTR, hInst : HINSTANCE) : TRESULT
- 9.1.9.1.16 Shutdown () : TRESULT
- 9.1.9.1.17 GetMakeCallRequest (plmc : LPLINEREQMAKECALL) : TRESULT
- 9.1.9.1.18 GetMediaCallRequest (plmc : LPLINEREQMEDIACALL) : TRESULT
- 9.1.9.1.19 RegisterRequestRecipient (dwRequestMode : DWORD, bEnable : BOOL) : TRESULT
- 9.1.9.1.20 TranslateDialog (nLineID : DWORD, hwndOwner : HWND, szAddressIn : LPCSTR) : TRESULT
- 9.1.9.1.21 ConfigDialog (nLineID : DWORD, hwndOwner : HWND, pszDeviceClass : LPCSTR = 0) : TRESULT
- 9.1.9.1.22 GetIcon (nLineID : DWORD, phicon : LPHICON, pszDeviceClass : LPCSTR = 0) : TRESULT
- 9.1.9.1.23 SetCurrentLocation (nLocationID : DWORD) : TRESULT

9.1.9.1.24 Open (nLineID : DWORD, pInitialSink : CtLineSink* = 0, dwPrivileges : DWORD = LINECALLPRIVILEGE_NONE, dwMediaModes : DWORD = LINEMEDIAMODE_INTERACTIVEVOICE) : TRESULT

9.1.9.1.25 Close () : TRESULT

9.1.9.1.26 GetAddressID (pdwAddressID : LPDWORD, nAddressMode : DWORD, pszAddress : LPCSTR, nSize : DWORD) : TRESULT

9.1.9.1.27 GetNumRings (nAddressID : DWORD, pnRings : DWORD*) : TRESULT

9.1.9.1.28 SetNumRings (nAddressID : DWORD, nRings : DWORD) : TRESULT

9.1.9.1.29 ForwardAll (plfl : const LPLINEFORWARDLIST, nRings : DWORD) : TRESULT

9.1.9.1.30 ForwardAddress (nAddressID : DWORD, plfl : const LPLINEFORWARDLIST, nRings : DWORD) : TRESULT

9.1.9.2 Protected Methods:

- 9.1.9.2.1 NegotiateApiVersions () : void
- 9.1.9.2.2 AddToLines (pLine : CtLine*) : void
- 9.1.9.2.3 RemoveFromLines (pLine : CtLine*) : void
- 9.1.9.2.4 OnCreate (dwDeviceID : DWORD) : void
- 9.1.9.2.5 OnRequest (nRequestMode : DWORD, hRequestWnd : HWND, nRequestID : TREQUEST) : void
- 9.1.9.2.6 OnEvent (dwDevice : DWORD, nMsg : DWORD, dwParam1 : DWORD, dwParam2 : DWORD, dwParam3 : DWORD) : void
- 9.1.9.2.7 OnAddressState (nAddressID : DWORD, nAddressState : DWORD) : void
- 9.1.9.2.8 OnClose () : void
- 9.1.9.2.9 OnDevSpecific (dwDevice : DWORD, dwParam1 : DWORD, dwParam2 : DWORD, dwParam3 : DWORD) : void
- 9.1.9.2.10 OnDevSpecificFeature (dwDevice : DWORD, dwParam1 : DWORD, dwParam2 : DWORD, dwParam3 : DWORD) : void
- 9.1.9.2.11 OnDevState (nDevState : DWORD, dwParam2 : DWORD, dwParam3 : DWORD) : void
- 9.1.9.2.12 OnCallInfo (hCall : HCALL, nCallInfo : DWORD) : void
- 9.1.9.2.13 OnCallState (hCall : HCALL, nCallState : DWORD, dwParam2 : DWORD, nCallPrivilege : DWORD) : void
- 9.1.9.2.14 OnNewCall (nAddressID : DWORD, hCall : HCALL, nCallPrivilege : DWORD) : void
- 9.1.9.2.15 OnGatherDigits (hCall : HCALL, nGatherTerm : DWORD) : void
- 9.1.9.2.16 OnGenerate (hCall : HCALL, nGenerateTerm : DWORD) : void
- 9.1.9.2.17 OnMonitorDigits (hCall : HCALL, cDigit : char, nDigitMode : DWORD) : void
- 9.1.9.2.18 OnMonitorMedia (hCall : HCALL, nMediaMode : DWORD) : void
- 9.1.9.2.19 OnMonitorTone (hCall : HCALL, dwAppSpecific : DWORD) : void
- 9.1.9.2.20 OnReply (nRequestID : TREQUEST, nResult : TREQUEST, nRequestType : DWORD) : void
- 9.1.9.2.21 TfxLineCallback (dwDevice : DWORD, nMsg : DWORD, dwInstance : DWORD, dwParam1 : DWORD, dwParam2 : DWORD, dwParam3 : DWORD) : int

9.1.9.3 Private Methods:

9.1.9.3.1 CtLine (: const CtLine&) : CtLine

9.1.9.3.2 AddRequest (nRequestID : TREQUEST, pTarget : CtReplyTarget*,
dwRequestType : DWORD) : void

9.1.9.3.3 RemoveAllRequests (pTarget : CtReplyTarget* = 0) : void

9.1.9.3.4 IsCallRequestPending (nRequestID : TREQUEST, pnRequestType :
DWORD*) : BOOL

9.1.9.3.5 IsCallRequestTypePending (pCall : const CtCall*, nRequestType :
DWORD) : BOOL

9.1.9.3.6 operator = (: const CtLine&) : CtLine&

9.1.10 CtReplyTarget

9.1.10.1 Public Methods:

9.1.10.1.1 OnReply (nRequestID : TREQUEST, nResult : TRESULT,
nRequestType : DWORD) : void

9.1.11 CtCallInfo

Derived from CtVariableData

9.1.11.1 Public Methods:

- 9.1.11.1.1 GetCallInfo (pCall : const CtCall*) : TRESULT
- 9.1.11.1.2 GetCallInfo (hCall : const HCALL) : TRESULT
- 9.1.11.1.3 operator const LPLINECALLINFO () : const LPLINECALLINFO
- 9.1.11.1.4 GetLineHandle () : HLINE
- 9.1.11.1.5 GetLineID () : DWORD
- 9.1.11.1.6 GetAddressID () : DWORD
- 9.1.11.1.7 GetBearerMode () : DWORD
- 9.1.11.1.8 GetRate () : DWORD
- 9.1.11.1.9 GetMediaMode () : DWORD
- 9.1.11.1.10 GetAppSpecific () : DWORD
- 9.1.11.1.11 GetCallID () : DWORD
- 9.1.11.1.12 GetRelatedCallID () : DWORD
- 9.1.11.1.13 GetCallParamFlags () : DWORD
- 9.1.11.1.14 GetCallStates () : DWORD
- 9.1.11.1.15 GetMonitorDigitModes () : DWORD
- 9.1.11.1.16 GetMonitorMediaModes () : DWORD
- 9.1.11.1.17 GetDialParams () : const LPLINEDIALPARAMS
- 9.1.11.1.18 GetOrigin () : DWORD
- 9.1.11.1.19 GetReason () : DWORD
- 9.1.11.1.20 GetCompletionID () : DWORD
- 9.1.11.1.21 GetNumOwners () : DWORD
- 9.1.11.1.22 GetNumMonitors () : DWORD
- 9.1.11.1.23 GetCountryCode () : DWORD
- 9.1.11.1.24 GetTrunk () : DWORD
- 9.1.11.1.25 GetCallerIDFlags () : DWORD
- 9.1.11.1.26 GetCallerID () : LPCSTR
- 9.1.11.1.27 GetCallerIDName () : LPCSTR
- 9.1.11.1.28 GetCalledIDFlags () : DWORD
- 9.1.11.1.29 GetCalledID () : LPCSTR
- 9.1.11.1.30 GetCalledIDName () : LPCSTR
- 9.1.11.1.31 GetConnectedIDFlags () : DWORD
- 9.1.11.1.32 GetConnectedID () : LPCSTR

9.1.11.1.33 GetConnectedIDName () : LPCSTR
9.1.11.1.34 GetRedirectionIDFlags () : DWORD
9.1.11.1.35 GetRedirectionID () : LPCSTR
9.1.11.1.36 GetRedirectionIDName () : LPCSTR
9.1.11.1.37 GetRedirectingIDFlags () : DWORD
9.1.11.1.38 GetRedirectingID () : LPCSTR
9.1.11.1.39 GetRedirectingIDName () : LPCSTR
9.1.11.1.40 GetAppName () : LPCSTR
9.1.11.1.41 GetDisplayableAddress () : LPCSTR
9.1.11.1.42 GetCalledParty () : LPCSTR
9.1.11.1.43 GetComment () : LPCSTR
9.1.11.1.44 GetDisplay () : LPCSTR
9.1.11.1.45 GetUserUserInfoSize () : DWORD
9.1.11.1.46 GetUserUserInfo () : void*
9.1.11.1.47 GetHighLevelCompSize () : DWORD
9.1.11.1.48 GetHighLevelComp () : void*
9.1.11.1.49 GetLowLevelCompSize () : DWORD
9.1.11.1.50 GetLowLevelComp () : void*
9.1.11.1.51 GetChargingInfoSize () : DWORD
9.1.11.1.52 GetChargingInfo () : void*
9.1.11.1.53 GetNumTerminals () : DWORD
9.1.11.1.54 GetTerminalModes (nTermID : DWORD) : DWORD
9.1.11.1.55 GetDevSpecificSize () : DWORD
9.1.11.1.56 GetDevSpecificInfo () : void*

9.1.11.2 Protected Methods:

9.1.11.2.1 FillBuffer () : TRESULT

9.1.11.3 Private Methods:

9.1.11.3.1 GetData () : const LPLINECALLINFO

9.1.12 CtCallList

Derived from CtVariableData

9.1.12.1 Public Methods:

9.1.12.1.1 GetNewCalls (hLine : HLINE) : TRESULT

9.1.12.1.2 GetNewCalls (nAddress : DWORD) : TRESULT

9.1.12.1.3 operator const LPLINECALLLIST () : const LPLINECALLLIST

9.1.12.1.4 GetNumCalls () : DWORD

9.1.12.1.5 GetCall (nCall : DWORD) : HCALL

9.1.12.2 Protected Methods:

9.1.12.2.1 FillBuffer () : TRESULT

9.1.12.3 Private Methods:

9.1.12.3.1 GetData () : const LPLINECALLLIST

9.1.13 CtCallSink

9.1.13.1 Public Methods:

9.1.13.1.1 OnCallInfo (pCall : CtCall*, nCallInfo : DWORD) : void

9.1.13.1.2 OnCallState (pCall : CtCall*, nCallState : DWORD, dwParam2 :
DWORD, nCallPrivilege : DWORD) : void

9.1.13.1.3 OnCallGatherDigits (pCall : CtCall*, nGatherTerm : DWORD) : void

9.1.13.1.4 OnCallGenerate (pCall : CtCall*, nGenerateTerm : DWORD) : void

9.1.13.1.5 OnCallMonitorDigits (pCall : CtCall*, cDigit : char, nDigitMode :
DWORD) : void

9.1.13.1.6 OnCallMonitorMedia (pCall : CtCall*, nMediaMode : DWORD) : void

9.1.13.1.7 OnCallMonitorTone (pCall : CtCall*, dwAppSpecific : DWORD) : void

9.1.13.1.8 OnCallReply (pCall : CtCall*, nRequestID : TREQUEST, tr : TRESULT,
nRequestType : DWORD) : void

9.1.14 CtCallStatus

Derived from CtVariableData

9.1.14.1 Public Methods:

9.1.14.1.1 GetCallStatus (pCall : const CtCall*) : TRESULT

9.1.14.1.2 GetCallStatus (hCall : const HCALL) : TRESULT

9.1.14.1.3 operator const LPLINECALLSTATUS () : const LPLINECALLSTATUS

9.1.14.1.4 GetCallState () : DWORD

9.1.14.1.5 GetCallStateMode () : DWORD

9.1.14.1.6 GetCallPrivilege () : DWORD

9.1.14.1.7 GetCallFeatures () : DWORD

9.1.14.1.8 GetDevSpecificSize () : DWORD

9.1.14.1.9 GetDevSpecificInfo () : void*

9.1.14.2 Protected Methods:

9.1.14.2.1 FillBuffer () : TRESULT

9.1.14.3 Private Methods:

9.1.14.3.1 GetData () : const LPLINECALLSTATUS

9.1.15 CtCountryList

Derived from CtVariableData

9.1.15.1 Public Methods:

9.1.15.1.1 GetCountryList () : TRESULT

9.1.15.1.2 operator const LPLINECOUNTRYLIST () : const
LPLINECOUNTRYLIST

9.1.15.1.3 GetNumCountries () : DWORD

9.1.15.1.4 GetCountryCode (nCountry : DWORD) : DWORD

9.1.15.1.5 GetCountryName (nCountry : DWORD) : LPCSTR

9.1.15.1.6 GetSameAreaRule (nCountry : DWORD) : LPCSTR

9.1.15.1.7 GetLongDistanceRule (nCountry : DWORD) : LPCSTR

9.1.15.1.8 GetInternationalRule (nCountry : DWORD) : LPCSTR

9.1.15.2 Protected Methods:

9.1.15.2.1 FillBuffer () : TRESULT

9.1.15.3 Private Methods:

9.1.15.3.1 GetData () : const LPLINECOUNTRYLIST

9.1.16 CtDeviceID

Derived from CtVariableData

9.1.16.1 Public Methods:

9.1.16.1.1 GetID (szDeviceClass : LPCSTR, hPhone : HPHONE) : TRESULT

9.1.16.1.2 GetID (szDeviceClass : LPCSTR, hLine : HLINE) : TRESULT

9.1.16.1.3 GetID (szDeviceClass : LPCSTR, hLine : HLINE, nAddressID :
DWORD) : TRESULT

9.1.16.1.4 GetID (szDeviceClass : LPCSTR, hCall : HCALL) : TRESULT

9.1.16.1.5 operator const LPVARSTRING () : const LPVARSTRING

9.1.16.1.6 GetString () : LPCSTR

9.1.16.1.7 GetHandleAndString (ph : HANDLE*) : LPCSTR

9.1.16.1.8 GetDeviceID () : DWORD

9.1.16.1.9 GetDeviceIDs () : DWORD*

9.1.16.2 Protected Methods:

9.1.16.2.1 FillBuffer () : TRESULT

9.1.16.3 Private Methods:

9.1.16.3.1 GetData () : const LPVARSTRING

9.1.17 CtDialStringSink

9.1.17.1 Public Methods:

9.1.17.1.1 OnDialDone () : void

9.1.17.1.2 OnDialError () : void

9.1.18 CtDialString

Derived from CtWaveSink, TimerSink

9.1.18.1 Public Methods:

9.1.18.1.1 CtDialString (pSink : CtDialStringSink* = 0, pszDigits : const TCHAR* = 0) : CtDialString

9.1.18.1.2 ~CtDialString () :

9.1.18.1.3 operator = (pszDigits : const TCHAR*) : CtDialString&

9.1.18.1.4 Dial (nWaveOut : UINT, nDigitDuration : UINT, nCommaDelay : UINT) : bool

9.1.18.1.5 operator const TCHAR* () : const TCHAR*

9.1.18.1.6 Cancel () : void

9.1.18.2 Private Methods:

9.1.18.2.1 PlayDigit () : bool

9.1.18.2.2 OnTimer () : void

9.1.18.2.3 OnWaveOutOpen () : void

9.1.18.2.4 OnWaveOutDone () : void

9.1.19 CtDtmf

Derived from CtWave

9.1.19.1 Public Methods:

9.1.19.1.1 CtDtmf (pSink : CtWaveSink* = 0) : CtDtmf

9.1.19.1.2 SetTone (cTone : char) : bool

9.1.19.2 Private Methods:

9.1.19.2.1 Load (prgDtmf : BYTE*, nSize : size_t) : bool

9.1.20 Timer

Derived from InvisibleWindowSink

9.1.20.1 Public Methods:

9.1.20.1.1 Timer (pSink : TimerSink*) : Timer

9.1.20.1.2 Start (nElapse : UINT, nMinDelta : UINT = 0) : bool

9.1.20.1.3 ~Timer () :

9.1.20.1.4 Running () : bool

9.1.20.1.5 Stop () : void

9.1.20.2 Private Methods:

9.1.20.2.1 OnWindowMessage (hwnd : HWND, nMsg : UINT, wparam : WPARAM, lparam : LPARAM) : LRESULT

9.1.21 CtWaveSink

9.1.21.1 Public Methods:

9.1.21.1.1 OnWaveOutOpen () : void

9.1.21.1.2 OnWaveOutDone () : void

9.1.21.1.3 OnWaveOutClose () : void

9.1.21.1.4 OnWaveInOpen () : void

9.1.21.1.5 OnWaveInData () : void

9.1.21.1.6 OnWaveInClose () : void

9.1.22 TimerSink

9.1.22.1 Public Methods:

9.1.22.1.1 OnTimer () : void

9.1.23 CtWave

Derived from InvisibleWindowSink

9.1.23.1 Public Methods:

- 9.1.23.1.1 CtWave (pSink : CtWaveSink* = 0) : CtWave
- 9.1.23.1.2 Load (hinst : HINSTANCE, nID : UINT) : bool
- 9.1.23.1.3 Load (hinst : HINSTANCE, pszID : LPCTSTR) : bool
- 9.1.23.1.4 Load (pszFileName : LPCSTR) : bool
- 9.1.23.1.5 ~CtWave () :
- 9.1.23.1.6 Save (pszFileName : LPCSTR) : bool
- 9.1.23.1.7 Play (nWaveOut : UINT, bLoop : bool = false) : bool
- 9.1.23.1.8 Record (nWaveIn : UINT, nSecs : UINT) : bool
- 9.1.23.1.9 Stop () : bool
- 9.1.23.1.10 Close () : bool
- 9.1.23.1.11 AddSink (pSink : CtWaveSink*) : void

9.1.23.2 Protected Methods:

- 9.1.23.2.1 Load (hmmio : HMMIO) : bool
- 9.1.23.2.2 OnWindowMessage (hwnd : HWND, nMsg : UINT, wparam : WPARAM, lparam : LPARAM) : LRESULT

9.1.24 TapiRecover

9.1.24.1 Public Methods:

- 9.1.24.1.1 PreInitialize () : void
- 9.1.24.1.2 Initialize (happ : void*) : void
- 9.1.24.1.3 Shutdown () : void

9.1.24.2 Private Methods:

9.1.24.2.1 SubKeyName () : const char*

9.1.24.2.2 ShutdownApp (happ : void*) : long

9.1.25 LineTapiRecover

Derived from TapiRecover

9.1.25.1 Private Methods:

9.1.25.1.1 SubKeyName () : const char*

9.1.25.1.2 ShutdownApp (happ : void*) : long

9.1.26 PhoneTapiRecover

Derived from TapiRecover

9.1.26.1 Private Methods:

9.1.26.1.1 SubKeyName () : const char*

9.1.26.1.2 ShutdownApp (happ : void*) : long

9.1.27 CtRequestList

9.1.27.1 Public Methods:

9.1.27.1.1 AddRequest (nRequestID : TREQUEST, pTarget : CtReplyTarget*, nRequestType : DWORD) : void

9.1.27.1.2 ~CtRequestList () :

9.1.27.1.3 IsRequestPending (nRequestID : TREQUEST = 0, pnRequestType : DWORD* = 0) : BOOL

9.1.27.1.4 IsRequestTypePending (nRequestType : DWORD, pTarget : const CtReplyTarget* = 0) : BOOL

9.1.27.1.5 RemoveRequest (nRequestID : TREQUEST, ppTarget : CtReplyTarget** = 0, pnRequestType : DWORD* = 0) : BOOL

9.1.27.1.6 RemoveAllRequests (pTarget : CtReplyTarget* = 0) : void

9.1.27.2 Private Methods:

9.1.27.2.1 FindRequest (nRequestID : TREQUEST, ppar : CtRequestList::AsyncRequest** = 0, ppos : POSITION* = 0) : BOOL

9.1.28 CtLineDevCaps

Derived from CtVariableData

9.1.28.1 Public Methods:

- 9.1.28.1.1 GetDevCaps (nLineID : DWORD) : TRESULT
- 9.1.28.1.2 operator const LPLINEDEVCAPS () : const LPLINEDEVCAPS
- 9.1.28.1.3 GetProviderInfo () : LPCSTR
- 9.1.28.1.4 GetSwitchInfo () : LPCSTR
- 9.1.28.1.5 GetPermanentLineID () : DWORD
- 9.1.28.1.6 GetLineName () : LPCSTR
- 9.1.28.1.7 GetAddressModes () : DWORD
- 9.1.28.1.8 GetNumAddresses () : DWORD
- 9.1.28.1.9 GetBearerModes () : DWORD
- 9.1.28.1.10 GetMaxRate () : DWORD
- 9.1.28.1.11 GetMediaModes () : DWORD
- 9.1.28.1.12 GetGenerateToneModes () : DWORD
- 9.1.28.1.13 GetGenerateToneMaxNumFreq () : DWORD
- 9.1.28.1.14 GetGenerateDigitModes () : DWORD
- 9.1.28.1.15 GetMonitorToneMaxNumFreq () : DWORD
- 9.1.28.1.16 GetMonitorToneMaxNumEntries () : DWORD
- 9.1.28.1.17 GetMonitorDigitModes () : DWORD
- 9.1.28.1.18 GetGatherDigitsMinTimeout () : DWORD
- 9.1.28.1.19 GetGatherDigitsMaxTimeout () : DWORD
- 9.1.28.1.20 GetMedCtlDigitMaxListSize () : DWORD
- 9.1.28.1.21 GetMedCtlMediaMaxListSize () : DWORD
- 9.1.28.1.22 GetMedCtlToneMaxListSize () : DWORD
- 9.1.28.1.23 GetMedCtlCallStateMaxListSize () : DWORD
- 9.1.28.1.24 GetDevCapFlags () : DWORD
- 9.1.28.1.25 GetMaxNumActiveCalls () : DWORD
- 9.1.28.1.26 GetAnswerMode () : DWORD
- 9.1.28.1.27 GetRingModes () : DWORD
- 9.1.28.1.28 GetLineStates () : DWORD
- 9.1.28.1.29 GetUIIAcceptSize () : DWORD
- 9.1.28.1.30 GetUIIAnswerSize () : DWORD
- 9.1.28.1.31 GetUIIMakeCallSize () : DWORD
- 9.1.28.1.32 GetUIDropSize () : DWORD

- 9.1.28.1.33 GetUUISendUserUserInfoSize () : DWORD
- 9.1.28.1.34 GetUUICallInfoSize () : DWORD
- 9.1.28.1.35 GetMinDialParams () : const LPLINEDIALPARAMS
- 9.1.28.1.36 GetMaxDialParams () : const LPLINEDIALPARAMS
- 9.1.28.1.37 GetDefaultDialParams () : const LPLINEDIALPARAMS
- 9.1.28.1.38 GetNumTerminals () : DWORD
- 9.1.28.1.39 GetTerminalText (nTermID : DWORD) : LPCSTR
- 9.1.28.1.40 GetTermCaps (nTermID : DWORD) : const LPLINETERMCAPS
- 9.1.28.1.41 GetLineFeatures () : DWORD

9.1.28.2 Protected Methods:

- 9.1.28.2.1 FillBuffer () : TRESULT

9.1.28.3 Private Methods:

- 9.1.28.3.1 GetData () : const LPLINEDEVCAPS

9.1.29 CtLineSink

9.1.29.1 Public Methods:

9.1.29.1.1 OnLineAddressState (pLine : CtLine*, nAddressID : DWORD, nAddressState : DWORD) : void

9.1.29.1.2 OnLineNewCall (pLine : CtLine*, hCall : HCALL, nAddressID : DWORD, nCallPriviledge : DWORD) : void

9.1.29.1.3 OnLineClose (pLine : CtLine*) : void

9.1.29.1.4 OnLineDevSpecific (pLine : CtLine*, dwDevice : DWORD, dwParam1 : DWORD, dwParam2 : DWORD, dwParam3 : DWORD) : void

9.1.29.1.5 OnLineDevSpecificFeature (pLine : CtLine*, dwDevice : DWORD, dwParam1 : DWORD, dwParam2 : DWORD, dwParam3 : DWORD) : void

9.1.29.1.6 OnLineDevState (pLine : CtLine*, nDevState : DWORD, dwParam2 : DWORD, dwParam3 : DWORD) : void

9.1.29.1.7 OnLineReply (pLine : CtLine*, nRequestID : TREQUEST, nResult : TRESULT, dwRequestType : DWORD) : void

9.1.30 CtPhone

Derived from CtReplyTarget

9.1.30.1 Public Methods:

- 9.1.30.1.1 CtPhone () : CtPhone
- 9.1.30.1.2 GetHandle () : HPHONE
- 9.1.30.1.3 ~CtPhone () :
- 9.1.30.1.4 GetDeviceID () : DWORD
- 9.1.30.1.5 AddSink (pSink : CtPhoneSink*) : void
- 9.1.30.1.6 RemoveSink (pSink : CtPhoneSink*) : void
- 9.1.30.1.7 IsRequestPending (nRequestID : TREQUEST, pnRequestType : DWORD* = 0) : BOOL
- 9.1.30.1.8 IsRequestTypePending (nRequestType : DWORD) : BOOL
- 9.1.30.1.9 FromHandle (HPHONE : HPHONE) : CtPhone*
- 9.1.30.1.10 GetNumDevs () : DWORD
- 9.1.30.1.11 GetAppHandle () : HPHONEAPP
- 9.1.30.1.12 GetAppVersion () : DWORD
- 9.1.30.1.13 SetAppVersion (dwLoVersion : DWORD, dwHiVersion : DWORD) : void
- 9.1.30.1.14 GetApiVersion (nPhoneID : DWORD) : DWORD
- 9.1.30.1.15 Initialize (pAppSink : CtAppSink*, szAppName : LPCSTR, hInst : HINSTANCE) : TRESULT
- 9.1.30.1.16 Shutdown () : TRESULT
- 9.1.30.1.17 GetIcon (nPhoneID : DWORD, phicon : LPHICON, pszDeviceClass : LPCSTR) : TRESULT
- 9.1.30.1.18 Open (nPhoneID : DWORD, pInitialSink : CtPhoneSink* = 0, dwPriviledges : DWORD = PHONEPRIVILEGE_OWNER) : TRESULT
- 9.1.30.1.19 Close () : TRESULT
- 9.1.30.1.20 SetHookSwitch (dwHookSwitchDevs : DWORD, nHookSwitchMode : DWORD) : TRESULT

9.1.30.2 Protected Methods:

- 9.1.30.2.1 NegotiateApiVersions () : void
- 9.1.30.2.2 AddToPhones (pPhone : CtPhone*) : void
- 9.1.30.2.3 RemoveFromPhones (pPhone : CtPhone*) : void
- 9.1.30.2.4 OnCreate (dwDeviceID : DWORD) : void
- 9.1.30.2.5 OnEvent (dwDevice : DWORD, nMsg : DWORD, dwParam1 : DWORD, dwParam2 : DWORD, dwParam3 : DWORD) : void
- 9.1.30.2.6 OnButton (nButtonOrLampID : DWORD, nButtonMode : DWORD, nButtonState : DWORD) : void
- 9.1.30.2.7 OnClose () : void
- 9.1.30.2.8 OnDevSpecific (dwDevice : DWORD, dwParam1 : DWORD, dwParam2 : DWORD, dwParam3 : DWORD) : void
- 9.1.30.2.9 OnState (dwPhoneStates : DWORD, dwPhoneStateDetails : DWORD) : void
- 9.1.30.2.10 OnReply (nRequestID : TREQUEST, nResult : TREQUEST, nRequestType : DWORD) : void
- 9.1.30.2.11 TfxPhoneCallback (dwDevice : DWORD, nMsg : DWORD, dwInstance : DWORD, dwParam1 : DWORD, dwParam2 : DWORD, dwParam3 : DWORD) : int

9.1.30.3 Private Methods:

- 9.1.30.3.1 CtPhone (: const CtPhone&) : CtPhone
- 9.1.30.3.2 AddRequest (nRequestID : TREQUEST, pTarget : CtReplyTarget*, dwRequestType : DWORD) : void
- 9.1.30.3.3 RemoveAllRequests (pTarget : CtReplyTarget* = 0) : void
- 9.1.30.3.4 operator = (: const CtPhone&) : CtPhone&

9.1.31 CtPhoneCaps

Derived from CtVariableData

9.1.31.1 Public Methods:

- 9.1.31.1.1 GetDevCaps (nPhoneID : DWORD) : TRESULT
- 9.1.31.1.2 operator const LPPHONECAPS () : const LPPHONECAPS
- 9.1.31.1.3 GetProviderInfo () : LPCSTR
- 9.1.31.1.4 GetPhoneInfo () : LPCSTR
- 9.1.31.1.5 GetPermanentPhoneID () : DWORD
- 9.1.31.1.6 GetPhoneName () : LPCSTR
- 9.1.31.1.7 GetPhoneStates () : DWORD
- 9.1.31.1.8 GetHookSwitchDevs () : DWORD
- 9.1.31.1.9 GetHandsetHookSwitchModes () : DWORD
- 9.1.31.1.10 GetSpeakerHookSwitchModes () : DWORD
- 9.1.31.1.11 GetHeadsetHookSwitchModes () : DWORD
- 9.1.31.1.12 GetVolumeFlags () : DWORD
- 9.1.31.1.13 GetGainFlags () : DWORD
- 9.1.31.1.14 GetDisplayNumRows () : DWORD
- 9.1.31.1.15 GetDisplayNumColumns () : DWORD
- 9.1.31.1.16 GetNumRingModes () : DWORD
- 9.1.31.1.17 GetNumButtonLamps () : DWORD
- 9.1.31.1.18 GetButtonModes (nButton : DWORD) : DWORD
- 9.1.31.1.19 GetButtonFunction (nButton : DWORD) : DWORD
- 9.1.31.1.20 GetLampModes (nLamp : DWORD) : DWORD
- 9.1.31.1.21 GetNumSetData () : DWORD
- 9.1.31.1.22 GetSetData (nDatum : DWORD) : DWORD
- 9.1.31.1.23 GetNumGetData () : DWORD
- 9.1.31.1.24 GetGetData (nDatum : DWORD) : DWORD
- 9.1.31.1.25 GetDevSpecificSize () : DWORD
- 9.1.31.1.26 GetDevSpecificData () : void*

9.1.31.2 Protected Methods:

9.1.31.2.1 FillBuffer () : TRESULT

9.1.31.3 Private Methods:

9.1.31.3.1 GetData () : const LPPHONECAPS

9.1.32 CtPhoneNo

9.1.32.1 Public Methods:

9.1.32.1.1 CtPhoneNo () : CtPhoneNo

9.1.32.1.2 CtPhoneNo (szWholePhoneNo : LPCSTR) : CtPhoneNo

9.1.32.1.3 CtPhoneNo (szCountryCode : LPCSTR, szAreaCode : LPCSTR, szPhoneNo : LPCSTR) : CtPhoneNo

9.1.32.1.4 CtPhoneNo (nCountryCode : DWORD, szAreaCode : LPCSTR, szPhoneNo : LPCSTR) : CtPhoneNo

9.1.32.1.5 CtPhoneNo (pno : const CtPhoneNo&) : CtPhoneNo

9.1.32.1.6 operator = (pno : const CtPhoneNo&) : CtPhoneNo&

9.1.32.1.7 GetCountryCode () : LPCSTR

9.1.32.1.8 ~CtPhoneNo () :

9.1.32.1.9 GetCountryCodeNum () : DWORD

9.1.32.1.10 GetAreaCode () : LPCSTR

9.1.32.1.11 GetPhoneNo () : LPCSTR

9.1.32.1.12 GetCanonical () : LPCSTR

9.1.32.1.13 GetDisplayable () : LPCSTR

9.1.32.1.14 GetTranslatable (pszMap : LPCSTR = "4442447") : LPCSTR

9.1.32.1.15 SetWholePhoneNo (szWholePhoneNo : LPCSTR) : void

9.1.32.1.16 SetCanonical (szCanonical : LPCSTR) : void

9.1.32.1.17 SetCanonical (szCountryCode : LPCSTR, szAreaCode : LPCSTR, szPhoneNo : LPCSTR) : void

9.1.32.1.18 SetCanonical (nCountryCode : DWORD, szAreaCode : LPCSTR, szPhoneNo : LPCSTR) : void

9.1.32.1.19 ResetToLocation () : void

9.1.32.1.20 SetCountryCode (szCountryCode : LPCSTR) : void

9.1.32.1.21 SetCountryCode (nCountryCode : DWORD) : void

9.1.32.1.22 SetAreaCode (szAreaCode : LPCSTR) : void

9.1.32.1.23 SetPhoneNo (szPhoneNo : LPCSTR) : void

9.1.32.2 Private Methods:

9.1.32.2.1 ClearConstructs () : void

9.1.32.2.2 CheckDefaults () : void

9.1.32.2.3 ResetAll () : void

9.1.32.2.4 Copy (pno : const CtPhoneNo&) : void

9.1.33 CtPhoneSink

9.1.33.1 Public Methods:

9.1.33.1.1 OnPhoneButton (pPhone : CtPhone*, nButtonOrLampID : DWORD, nButtonMode : DWORD, nButtonState : DWORD) : void

9.1.33.1.2 OnPhoneClose (pPhone : CtPhone*) : void

9.1.33.1.3 OnPhoneDevSpecific (pPhone : CtPhone*, dwDevice : DWORD, dwParam1 : DWORD, dwParam2 : DWORD, dwParam3 : DWORD) : void

9.1.33.1.4 OnPhoneReply (pPhone : CtPhone*, nRequestID : TREQUEST, nResult : TRESULT, dwRequestType : DWORD) : void

9.1.33.1.5 OnPhoneState (pPhone : CtPhone*, dwPhoneStates : DWORD, dwPhoneStateDetails : DWORD) : void

9.1.34 CtProviderList

Derived from CtVariableData

9.1.34.1 Public Methods:

9.1.34.1.1 GetProviderList () : LONG

9.1.34.1.2 operator const LPLINEPROVIDERLIST () : const
LPLINEPROVIDERLIST

9.1.34.1.3 GetNumProviders () : DWORD

9.1.34.1.4 GetProviderPermanentID (nProvider : DWORD) : DWORD

9.1.34.1.5 GetProviderFilename (nProvider : DWORD) : LPCSTR

9.1.34.2 Protected Methods:

9.1.34.2.1 FillBuffer () : TRESULT

9.1.34.3 Private Methods:

9.1.34.3.1 GetData () : const LPLINEPROVIDERLIST

9.1.35 CtTranslateCaps

Derived from CtVariableData

9.1.35.1 Public Methods:

9.1.35.1.1 GetTranslateCaps () : TRESULT

9.1.35.1.2 operator const LPLINETRANSLATECAPS () : const
LPLINETRANSLATECAPS

9.1.35.1.3 GetCurrentLocationID () : DWORD

9.1.35.1.4 GetNumLocations () : DWORD

9.1.35.1.5 GetPermanentLocationID (nLocation : DWORD) : DWORD

9.1.35.1.6 GetLocationName (nLocation : DWORD) : LPCSTR

9.1.35.1.7 GetCountryCode (nLocation : DWORD) : DWORD

9.1.35.1.8 GetAreaCode (nLocation : DWORD) : LPCSTR

9.1.35.1.9 GetCityCode (nLocation : DWORD) : LPCSTR

9.1.35.1.10 GetPreferredCardID (nLocation : DWORD) : DWORD

9.1.35.1.11 GetLocalAccessCode (nLocation : DWORD) : LPCSTR

9.1.35.1.12 GetLongDistanceAccessCode (nLocation : DWORD) : LPCSTR

9.1.35.1.13 GetTollPrefixList (nLocation : DWORD) : LPCSTR

9.1.35.1.14 GetCountryID (nLocation : DWORD) : DWORD

9.1.35.1.15 GetLocationOptions (nLocation : DWORD) : DWORD

9.1.35.1.16 GetCancelCallWaiting (nLocation : DWORD) : LPCSTR

9.1.35.1.17 GetCurrentPreferredCardID () : DWORD

9.1.35.1.18 GetNumCards () : DWORD

9.1.35.1.19 GetPermanentCardID (nCard : DWORD) : DWORD

9.1.35.1.20 GetCardName (nCard : DWORD) : LPCSTR

9.1.35.1.21 GetCardNumberDigits (nCard : DWORD) : DWORD

9.1.35.1.22 GetSameAreaRule (nCard : DWORD) : LPCSTR

9.1.35.1.23 GetLongDistanceRule (nCard : DWORD) : LPCSTR

9.1.35.1.24 GetInternationalRule (nCard : DWORD) : LPCSTR

9.1.35.1.25 GetCardOptions (nCard : DWORD) : DWORD

9.1.35.2 Protected Methods:

9.1.35.2.1 FillBuffer () : TRESULT

9.1.35.3 Private Methods:

9.1.35.3.1 GetData () : const LPLINETRANSLATECAPS

9.1.36 CtTranslateOutput

Derived from CtVariableData

9.1.36.1 Public Methods:

9.1.36.1.1 TranslateAddress (nLineID : DWORD, pszAddressIn : LPCSTR, nCardID : DWORD = 0, dwTranslateOptions : DWORD = 0) : TRESULT

9.1.36.1.2 operator const LPLINETRANSLATEOUTPUT () : const LPLINETRANSLATEOUTPUT

9.1.36.1.3 GetDialableString () : LPCSTR

9.1.36.1.4 GetDisplayableString () : LPCSTR

9.1.36.1.5 GetCurrentCountry () : DWORD

9.1.36.1.6 GetDestCountry () : DWORD

9.1.36.1.7 GetTranslateResults () : DWORD

9.1.36.2 Protected Methods:

9.1.36.2.1 FillBuffer () : TRESULT

9.1.36.3 Private Methods:

9.1.36.3.1 GetData () : const LPLINETRANSLATEOUTPUT

9.2 Integration

This section includes all the datatypes that have been used for the integration of the telephony, TTS and POP3 client modules. These datatypes include:

9.2.1 CPtrList

9.2.2 HCALL

9.2.3 CPtrArray

9.2.4 HLINE

9.2.5 HPHONE

9.2.6 HLINEAPP

9.2.7 HPHONEAPP

9.2.8 WAVEHDR

9.2.9 WAVEFORMATEX

9.2.10 HWAVEIN

9.2.11 HWAVEOUT

9.3 VoiceM

VoiceM package embodies the application that downloads the e-mails using POP3 and plays it using Text-to-Speech conversion. It also listens for incoming calls using TFX. Classes included in this package are explained below.

9.3.1 CMainFrame

Derived from CFrameWnd

9.3.1.1 Public Properties:

9.3.1.1.1 __CLOSE_AFX_VIRTUAL :

9.3.1.2 Public Methods:

9.3.1.2.1 __OPEN_AFX_VIRTUAL (: CMainFrame) : int

9.3.1.2.2 PreCreateWindow (cs : CREATESTRUCT&) : BOOL

9.3.1.2.3 ~CMainFrame () :

9.3.1.3 Protected Methods:

9.3.1.3.1 DECLARE_DYNCREATE (: CMainFrame) : int

9.3.1.3.2 CMainFrame () : CMainFrame

9.3.1.3.3 __OPEN_AFX_MSG (: CMainFrame) : int

9.3.1.3.4 OnCreate (lpCreateStruct : LPCREATESTRUCT) : afx_msg

9.3.1.3.5 DECLARE_MESSAGE_MAP () : __CLOSE_AFX_MSG

9.3.2 CPop3Message

9.3.2.1 Public Properties:

9.3.2.1.1 m_pszMessage : char*

9.3.2.2 Public Methods:

9.3.2.2.1 CPop3Message () : CPop3Message

9.3.2.2.2 GetMessageText () : LPCSTR

9.3.2.2.3 ~CPop3Message () :

9.3.2.2.4 GetHeader () : CString

9.3.2.2.5 GetHeaderItem (sName : const CString&, nItem : int = 0) : CString

9.3.2.2.6 GetBody () : CString

9.3.2.2.7 GetRawBody () : LPCSTR

9.3.2.2.8 GetSubject () : CString

9.3.2.2.9 GetFrom () : CString

9.3.2.2.10 GetDate () : CString

9.3.2.2.11 GetReplyTo () : CString

9.3.3 CPop3Socket

9.3.3.1 Public Methods:

9.3.3.1.1 CPop3Socket () : CPop3Socket

9.3.3.1.2 ~CPop3Socket () :

9.3.3.1.3 Create () : BOOL

9.3.3.1.4 Connect (pszHostAddress : LPCTSTR, nPort : int = 110) : BOOL

9.3.3.1.5 Send (pszBuf : LPCSTR, nBuf : int) : BOOL

9.3.3.1.6 Close () : void

9.3.3.1.7 Receive (pszBuf : LPSTR, nBuf : int) : int

9.3.3.1.8 IsReadable (bReadable : BOOL&) : BOOL

9.3.3.2 Protected Methods:

9.3.3.2.1 Connect (IpSockAddr : const SOCKADDR*, nSockAddrLen : int) : BOOL

9.3.4 CPop3Connection

9.3.4.1 Protected Properties:

9.3.4.1.1 m_nNumberOfMails : int

9.3.4.2 Public Methods:

9.3.4.2.1 CPop3Connection () : CPop3Connection

9.3.4.2.2 ~CPop3Connection () :

9.3.4.2.3 Connect (pszHostName : LPCTSTR, pszUser : LPCTSTR, pszPassword : LPCTSTR, nPort : int = 110) : BOOL

9.3.4.2.4 Disconnect () : BOOL

9.3.4.2.5 Statistics (nNumberOfMails : int&, nTotalMailSize : int&) : BOOL

9.3.4.2.6 Delete (nMsg : int) : BOOL

9.3.4.2.7 GetMessageSize (nMsg : int, dwSize : DWORD&) : BOOL

9.3.4.2.8 GetMessageID (nMsg : int, sID : CString&) : BOOL

9.3.4.2.9 Retrieve (nMsg : int, message : CPop3Message&) : BOOL

9.3.4.2.10 GetMessageHeader (nMsg : int, message : CPop3Message&) : BOOL

9.3.4.2.11 Reset () : BOOL

9.3.4.2.12 UIDL () : BOOL

9.3.4.2.13 Noop () : BOOL

9.3.4.2.14 GetLastCommandResponse () : CString

9.3.4.2.15 GetTimeout () : DWORD

9.3.4.2.16 SetTimeout (dwTimeout : DWORD) : void

9.3.4.3 Protected Methods:

- 9.3.4.3.1 ReadStatResponse (nNumberOfMails : int&, nTotalMailSize : int&) :
BOOL
- 9.3.4.3.2 ReadCommandResponse () : BOOL
- 9.3.4.3.3 ReadListResponse (nNumberOfMails : int) : BOOL
- 9.3.4.3.4 ReadUIDLResponse (nNumberOfMails : int) : BOOL
- 9.3.4.3.5 ReadReturnResponse (message : CPop3Message&, dwSize : DWORD) :
BOOL
- 9.3.4.3.6 ReadResponse (pszBuffer : LPSTR, nInitialBufSize : int, pszTerminator :
LPSTR, ppszOverFlowBuffer : LPSTR*, nGrowBy : int = 4096) : BOOL
- 9.3.4.3.7 List () : BOOL
- 9.3.4.3.8 GetFirstCharInResponse (pszData : LPSTR) : LPSTR

9.3.5 CSettings

Derived from CDialog

9.3.5.1 Private Properties:

- 9.3.5.1.1 __NOTE_AFX_INSERT_LOCATION :

9.3.5.2 Public Methods:

- 9.3.5.2.1 CSettings (pParent : CWnd* = NULL) : CSettings

9.3.6 CVoiceMApp

Derived from CWinApp, CtCallSink, CtLineSink, CtWaveSink

9.3.6.1 Public Properties:

9.3.6.1.1 m_Rings : int

9.3.6.1.2 m_Seconds : int

9.3.6.2 Protected Properties:

9.3.6.2.1 AutoLineID : int

9.3.6.3 Public Methods:

- 9.3.6.3.1 `__OPEN_AFX_VIRTUAL` (: CVoiceMApp) : int
- 9.3.6.3.2 `CVoiceMApp` () : CVoiceMApp
- 9.3.6.3.3 `InitInstance` () : BOOL
- 9.3.6.3.4 `ExitInstance` () : int
- 9.3.6.3.5 `OnCallInfo` (pCall : CtCall*, nCallInfo : DWORD) :
`__CLOSE_AFX_VIRTUAL`
- 9.3.6.3.6 `OnCallState` (pCall : CtCall*, nCallState : DWORD, dwParam2 :
DWORD, nCallPrivilege : DWORD) : void
- 9.3.6.3.7 `OnCallMonitorDigits` (pCall : CtCall*, cDigit : char, nDigitMode :
DWORD) : void
- 9.3.6.3.8 `virtual void OnCallMonitorMedia`(CtCall* pCall, DWORD nMediaMode);
- 9.3.6.3.9 `virtual void OnCallMonitorTone`(CtCall* pCall, DWORD
dwAppSpecific);
- 9.3.6.3.10 `OnCallReply` (pCall : CtCall*, nRequestID : TREQUEST, tr : TRESULT,
nRequestType : DWORD) : void
- 9.3.6.3.11 `virtual void OnLineAddressState`(CtLine* pLine, DWORD
nAddressID, DWORD nAddressState);
- 9.3.6.3.12 `OnLineNewCall` (pLine : CtLine*, hCall : HCALL, nAddressID :
DWORD, nCallPrivilege : DWORD) : void
- 9.3.6.3.13 `OnLineDevState` (pLine : CtLine*, nDevState : DWORD, dwParam2 :
DWORD, dwParam3 : DWORD) : void
- 9.3.6.3.14 `OnWaveOutDone` () : void
- 9.3.6.3.15 `OnWaveInData` () : void

9.3.6.4 Protected Methods:

- 9.3.6.4.1 `__OPEN_AFX_MSG` (: CVoiceMApp) : int
- 9.3.6.4.2 `OnAppAbout` () : afx_msg
- 9.3.6.4.3 `DECLARE_MESSAGE_MAP` () : `__CLOSE_AFX_MSG`

9.3.7 CVoiceMView

Derived from CFormView

9.3.7.1 Private Properties:

9.3.7.1.1 __NOTE_AFX_INSERT_LOCATION :

9.3.7.2 Protected Methods:

9.3.7.2.1 CVoiceMView () : CVoiceMView

9.3.7.3 Private Methods:

9.3.7.3.1 GetDocument () : CVoiceMDoc*

9.3.8 CVoiceMDoc

Derived from CDocument

9.3.8.1 Public Properties:

9.3.8.1.1 __CLOSE_AFX_VIRTUAL :

9.3.8.2 Public Methods:

9.3.8.2.1 `__OPEN_AFX_VIRTUAL (: CVoiceMDoc) : int`

9.3.8.2.2 `OnNewDocument () : BOOL`

9.3.8.2.3 `Serialize (ar : CArchive&) : void`

9.3.8.2.4 `~CVoiceMDoc () :`

9.3.8.3 Protected Methods:

9.3.8.3.1 `DECLARE_DYNCREATE (: CVoiceMDoc) : int`

9.3.8.3.2 `CVoiceMDoc () : CVoiceMDoc`

9.3.8.3.3 `__OPEN_AFX_MSG (: CVoiceMDoc) : int`

9.3.8.3.4 `DECLARE_MESSAGE_MAP () : __CLOSE_AFX_MSG`

9.3.9 CAboutDlg

Derived from `CDialog`

9.3.9.1 Public Methods:

9.3.9.1.1 `CAboutDlg () : CAboutDlg`

9.3.9.2 Private Methods:

9.3.9.2.1 `CAboutDlg () : CAboutDlg`

9.3.9.2.2 `DoDataExchange (pDX : CDataExchange*) : void`

10 Future Expansion Possibilities

There is a lot of room for future expansions in this system. Some of the other services that can be added to the E-Mail via Phone system include the following:

10.1 News Update

The News update will allow the users to listen to latest news via any touch-tone phone from anywhere. The news will be updated at the server regularly via Internet. The text-to-speech technology will be incorporated to convert the news in text form to speech, enabling the user to listen to the latest news update.

10.2 Weather Forecast

Similarly the weather forecast will keep the users in touch with the latest weather forecast. The user will dial the service provider's phone number and will choose the weather forecast option from the available options. Weather forecast will be constantly updated at the PSP server. Text-to-speech technology will make the weather forecast available to the user in audible form via telephone.

10.3 Sports News

The sports update service will allow the users to listen to latest sports news and live commentary of important sports events. Again the user will dial the PSP's number and will choose the sports option. The server will establish connection with an online radio/sports station and will allow the user to listen to live commentary or sports update being broadcasted from that station.

10.4 Flight Timings

Flight timings service will inform the users about the flight schedules of the local as well as international airlines. The access and lookup procedure for this service will be more or less the same as in case of the news and sports update.

10.5 On Line Transaction Processing

Online Transaction Processing (OLTP) service will allow the users to purchase or order products via any touch-tone phone. Online Transaction Processing will involve ordering products from a list of offered products by just dialling the PSP's number. These may include placing orders to a restaurant or purchasing an item from a departmental store.

11 Conclusion

On the whole, the project has been a success as it gave us a valuable chance to gain knowledge in the latest fields of Internet telephony and e-commerce. The experience gained during the course of this project will help us in our future endeavours. This project will indeed be a milestone in our academic and professional careers.

While analysing, designing and implementing this project, we tried to apply all we had studied in different courses through out this undergraduate degree program. This exercise not only allowed us to implement the theoretical concepts of Computer Science, but also provided us with an excellent opportunity to revise and refresh everything in great detail, making us ready for the challenges of practical life and abreast of the cutting edge technological breakthroughs.

12 Bibliography

1. Allen, J., Hunnicutt, M. S. and Klatt, D. H. (1987). *From text to speech: The MITalk system*, Cambridge: Cambridge University Press.
2. Amundsen, Michael C. (1996), *MAPI, SAPI &TAPI Developer's Guide*, New York: Sams Publishing.
3. Beckman, M., Hertz, S., and Fujimura, O. (1983). "SRS Pitch Rules for Japanese", *Working Papers of the Cornell Phonetics Laboratory* 1, 1-16.
4. Chomsky, N. and Halle, M. (1968). *The Sound Pattern of English*, New York: Harper and Row.
5. Clements, G. N., Hertz, S. R., Lauret, B. (1995). A representational basis for modeling English vowel duration, *Proceedings of the XIIIth International Congress of Phonetic Sciences*.
6. Clements, G. N., Hertz, S. R. (1966). An integrated approach to phonology and phonetics, in J. Durand and B. Laks (eds.), *Current Trends in Phonology: Models and Methods*, CNRS, Paris X and University of Salford Publications.
7. Campbell, N. and A. Black (1997). Prosody and the selection of source units for concatenative synthesis, in J. van Santen, R. Sprout, J. Olive and J. Hirshberg (eds.), *Progress in Speech Synthesis*, Berlin: Springer Verlag, 279-292.

8. Dutoit, T. (1997). *An Introduction to Text-to-Speech Synthesis*, Dordrecht: Kluwer.
9. Hertz, S. R. (1979). Appropriateness of different rule types in speech synthesis, in J. J. Wolf and D. H. Klatt (eds.), *ASA*50 Speech Communication Papers*, 511-514.
10. Hertz, S. R. (1982). From text to speech with SRS, *Journal of the Acoustical Society of America* 72, 1155-1170.
11. Hertz, S. R. (1990a). A modular approach to multi-dialect and multi-language speech synthesis using the Delta System, *Proceedings of the Workshop on Speech Synthesis*, European Speech Communication Association, 225-228.
12. Hertz, S. R. (1990b). The Delta programming language: an integrated approach to non-linear phonology, phonetics, and speech synthesis, in J. Kingston and M. Beckman (eds.), *Papers in Laboratory Phonology I: Between the Grammar and the Physics of Speech*, Cambridge University Press, 215-257.
13. Hertz, S. R. (1991). Streams, phones, and transitions: toward a phonological and phonetic model of formant timing, *Journal of Phonetics* 19, 91-109.
14. Hertz, S. R. and M. K. Huffman (1992). A nucleus-based timing model applied to multi-dialect speech synthesis by rule, *Proceedings of the International Conference on Spoken Language Processing* 2, 1171-1174.

15. Hertz, S. R. and L. Zsiga (1995). The Delta System with Syllt: increased capabilities for teaching and research in phonetics, *Proceedings ICPhS 95 Stockholm 2*, 322-325.
16. Hertz, S. R. (1997). The technology of text-to-speech, *Speech Technology*, April/May, 18-21.
17. Hertz, S. R., R. J. Younes and N. Zinovieva (1999). Language-universal and language-specific components in the multi-language ETI-Eloquence text-to-speech system, *Proceedings of the XIV International Congress of Phonetic Sciences*, 2283-2286.
18. Holmes, J. (1973). Influence of the glottal waveform on the naturalness of speech from a parallel formant synthesizer, *IEEE Transactions on Audio and Electroacoustics*, AU-21, 298-305.
19. Hunt, A and A. Black (1996). Unit selection in a concatenative speech synthesis system using a large speech database, *ICASSP 1*, 373-376.
20. Klatt, D. H. and L. C. Klatt (1990). Analysis, synthesis, and perception of voice quality variations among female and male talkers, *Journal of the Acoustical Society of America* 87(2), 820-857.
21. McCormick, S. and Hertz, S. R. (1989). "A new approach to English text-to-phoneme conversion using Delta Version 2", *Journal of the Acoustical Society of America*, Supplement 1 85, S124.

22. Moulines, E. and F. Charpentier (1990). Pitch synchronous waveform processing techniques for text-to-speech synthesis using diphones, *Speech Communication* 9, no. 5-6.
23. [RFC821] Postel, J., "Simple Mail Transfer Protocol", STD 10, RFC821, USC/Information Sciences Institute, August 1982.
24. [RFC822] Crocker, D., "Standard for the Format of ARPA-Internet TextMessages", STD 11, RFC 822, University of Delaware, August 1982.
25. [RFC1321] Rivest, R. "The MD5 Message-Digest Algorithm", RFC 1321, MIT Laboratory for Computer Science, April, 1992.
26. Sagisaka, Y., C. d'Alessandro, J. S. Liénard, R. Sproat, K. McKeown and J. Moore (1995). Spoken output technologies, in R. Cole (ed.), *Survey of the State of the Art in Human Language Technology*, Center for Spoken Language Understanding, Oregon Graduate Institute, 189-226.
27. Sproat, R. (ed.) (1990). *Multilingual Text-to-Speech Synthesis: the Bell Labs Approach*, Dordrecht: Kluwer.
28. Zsiga, E. C. (1994). *Syllt User's Manual* (Eloquent Technology, Inc., Ithaca, New York).

13 Appendix

13.1 Post Office Protocol – Version 3 (POP3) RFC

Network Working Group
Request for Comments: 1725
Obsoletes: 1460
Category: Standards Track
November 1994

J. Myers
Carnegie Mellon
M. Rose
Dover Beach Consulting, Inc.

Post Office Protocol - Version 3

Status of this Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Overview

This memo is a revision to RFC 1460, a Draft Standard. It makes the following changes from that document:

- removed text regarding "split-UA model", which didn't add anything to the understanding of POP
- clarified syntax of commands, keywords, and arguments
- clarified behavior on broken connection
- explicitly permitted an inactivity autologout timer
- clarified the requirements of the "exclusive-access lock"
- removed implementation-specific wording regarding the parsing of the mail drop
- allowed servers to close the connection after a failed authentication command
- removed the LAST command
- fixed typo in example of TOP command
- clarified that the second argument to the TOP command is non-negative
- added the optional UIDL command
- added warning regarding length of shared secrets with APOP

- added additional warnings to the security considerations section

1. Introduction

On certain types of smaller nodes in the Internet it is often impractical to maintain a message transport system (MTS). For example, a workstation may not have sufficient resources (cycles, disk space) in order to permit a SMTP server [RFC821] and associated local mail delivery system to be kept resident and continuously running. Similarly, it may be expensive (or impossible) to keep a personal computer interconnected to an IP-style network for long amounts of time (the node is lacking the resource known as "connectivity").

Despite this, it is often very useful to be able to manage mail on these smaller nodes, and they often support a user agent (UA) to aid the tasks of mail handling. To solve this problem, a node which can support an MTS entity offers a mail drop service to these less endowed

nodes. The Post Office Protocol - Version 3 (POP3) is intended to permit a workstation to dynamically access a mail drop on a server host in a useful fashion. Usually, this means that the POP3 is used to allow a workstation to retrieve mail that the server is holding for it.

For the remainder of this memo, the term "client host" refers to a host making use of the POP3 service, while the term "server host" refers to a host which offers the POP3 service.

2. A Short Digression

This memo does not specify how a client host enters mail into the transport system, although a method consistent with the philosophy of this memo is presented here:

When the user agent on a client host wishes to enter a message into the transport system, it establishes an SMTP connection to its relay host (this relay host could be, but need not be, the POP3 server host for the client host).

3. Basic Operation

Initially, the server host starts the POP3 service by listening on TCP port 110. When a client host wishes to make use of the service, it establishes a TCP connection with the server host. When the connection is established, the POP3 server sends a greeting. The client and POP3 server then exchange commands and responses (respectively) until the connection is closed or aborted.

Commands in the POP3 consist of a keyword, possibly followed by one or more arguments. All commands are terminated by a CRLF pair. Keywords and arguments consist of printable ASCII characters. Keywords and arguments are each separated by a single SPACE character. Keywords are three or four characters long. Each argument may be up to 40 characters long.

Responses in the POP3 consist of a status indicator and a keyword possibly followed by additional information. All responses are terminated by a CRLF pair. There are currently two status indicators: positive ("OK") and negative ("-ERR").

Responses to certain commands are multi-line. In these cases, which are clearly indicated below, after sending the first line of the response and a CRLF, any additional lines are sent, each terminated by a CRLF pair. When all lines of the response have been sent, a final line is sent, consisting of a termination octet (decimal code 046, ".") and a CRLF pair. If any line of the multi-line response begins with the termination octet, the line is "byte-stuffed" by pre-pending the termination octet to that line of the response. Hence a multi-line response is terminated with the five octets "CRLF.CRLF". When examining a multi-line response, the client checks to see if the line begins with the termination octet. If so and if octets other than CRLF follow, the the first octet of the line (the termination octet) is stripped away. If so and if CRLF immediately follows the termination character, then the response from the POP server is ended and the line containing ".CRLF" is not considered part of the multi-line response.

A POP3 session progresses through a number of states during its lifetime. Once the TCP connection has been opened and the POP3 server has sent the greeting, the session enters the AUTHORIZATION state. In this state, the client must identify itself to the POP3 server. Once the client has successfully done this, the server acquires resources associated with the client's mail drop, and the session enters the TRANSACTION state. In this state, the client requests actions on the part of the POP3 server. When the client has issued the QUIT command, the session enters the UPDATE state. In this state, the POP3 server releases any resources acquired during the TRANSACTION state and says goodbye. The TCP connection is then closed.

A POP3 server MAY have an inactivity autologout timer. Such a timer MUST be of at least 10 minutes' duration. The receipt of any command from the client during that interval should suffice to reset the autologout timer. When the timer expires, the session does NOT enter the UPDATE state--the server should close the TCP connection without removing any messages or sending any response to the client.

4. The AUTHORIZATION State

Once the TCP connection has been opened by a POP3 client, the POP3 server issues a one line greeting. This can be any string terminated by CRLF. An example might be:

```
S: +OK POP3 server ready
```

Note that this greeting is a POP3 reply. The POP3 server should always give a positive response as the greeting.

The POP3 session is now in the AUTHORIZATION state. The client must now identify and authenticate itself to the POP3 server. Two possible mechanisms for doing this are described in this document, the USER and PASS command combination and the APOP command. The APOP command is described later in this document.

To authenticate using the USER and PASS command combination, the client must first issue the USER command. If the POP3 server responds with a positive status indicator ("OK"), then the client may issue either the PASS command to complete the authentication, or the QUIT command to terminate the POP3 session. If the POP3 server responds with a negative status indicator ("-ERR") to the USER

command, then the client may either issue a new authentication command or may issue the QUIT command.

When the client issues the PASS command, the POP3 server uses the argument pair from the USER and PASS commands to determine if the client should be given access to the appropriate mail drop.

Once the POP3 server has determined through the use of any authentication command that the client should be given access to the appropriate mail drop, the POP3 server then acquires an exclusive-access lock on the mail drop, as necessary to prevent messages from being modified or removed before the session enters the UPDATE state. If the lock is successfully acquired, the POP3 server responds with positive status indicator. The POP3 session now enters the TRANSACTION state, with no messages marked as deleted. If the mail drop cannot be opened for some reason (for example, a lock can not be acquired, the client is denied access to the appropriate mail drop, or the mail drop cannot be parsed), the POP3 server responds with a negative status indicator. (If a lock was acquired but the POP3 server intends to respond with a negative status indicator, the POP3 server must release the lock prior to rejecting the command.) After returning a negative status indicator, the server may close the connection. If the server does not close the connection, the client may either issue a new authentication command and start again, or the client may issue the QUIT command.

After the POP3 server has opened the mail drop, it assigns a message-number to each message, and notes the size of each message in octets.

The first message in the mail drop is assigned a message-number of "1", the second is assigned "2", and so on, so that the n'th message in a mail drop is assigned a message-number of "n". In POP3 commands and responses, all message-number's and message sizes are expressed in base-10 (i.e., decimal).

Here are summaries for the three POP3 commands discussed thus far:

USER name

Arguments:

a string identifying a mailbox (required), which is of significance ONLY to the server

Restrictions:

may only be given in the AUTHORIZATION state after the POP3 greeting or after an unsuccessful USER or PASS command

Possible Responses:

+OK name is a valid mailbox
-ERR never heard of mailbox name

Examples:

```
C: USER mrose
S: +OK mrose is a real hoopy frood
...
C: USER frated
S: -ERR sorry, no mailbox for frated here
```


PASS string

Arguments:

a server/mailbox-specific password (required)

Restrictions:

may only be given in the AUTHORIZATION state after a successful USER command

Discussion:

Since the PASS command has exactly one argument, a POP3 server may treat spaces in the argument as part of the password, instead of as argument separators.

Possible Responses:

+OK mail drop locked and ready

-ERR invalid password

-ERR unable to lock mail drop

Examples:

C: USER mrose

S: +OK mrose is a real hoopy frood

C: PASS secret

S: +OK mrose's mail drop has 2 messages (320 octets)

...

C: USER mrose

S: +OK mrose is a real hoopy frood

C: PASS secret

S: -ERR mail drop already locked

QUIT

Arguments: none

Restrictions: none

Possible Responses:

+OK

Examples:

C: QUIT

S: +OK dewey POP3 server signing off

5. The TRANSACTION State

Once the client has successfully identified itself to the POP3 server and the POP3 server has locked and opened the appropriate mail drop, the POP3 session is now in the TRANSACTION state. The client may now issue any of the following POP3 commands repeatedly. After each command, the POP3 server issues a response. Eventually, the client issues the QUIT command and the POP3 session enters the UPDATE state.

Here are the POP3 commands valid in the TRANSACTION state:

STAT

Arguments: none

Restrictions:

may only be given in the TRANSACTION state

Discussion:

The POP3 server issues a positive response with a line containing information for the mail drop. This line is called a "drop listing" for that mail drop.

In order to simplify parsing, all POP3 servers required to use a certain format for drop listings. The positive response consists of "+OK" followed by a single space, the number of messages in the mail drop, a single space, and the size of the mail drop in octets. This memo makes no requirement on what follows the mail drop size. Minimal implementations should just end that line of the response with a CRLF pair. More advanced implementations may include other information.

NOTE: This memo STRONGLY discourages implementations from supplying additional information in the drop listing. Other, optional, facilities are discussed later on which permit the client to parse the messages in the mail drop.

Note that messages marked as deleted are not counted in either total.

Possible Responses:

+OK nn mm

Examples:

C: STAT

S: +OK 2 320

LIST [msg]

Arguments:

a message-number (optional), which, if present, may NOT refer to a message marked as deleted

Restrictions:

may only be given in the TRANSACTION state

Discussion:

If an argument was given and the POP3 server issues a positive response with a line containing information for that message. This line is called a "scan listing" for that message.

If no argument was given and the POP3 server issues a positive response, then the response given is multi-line. After the initial +OK, for each message in the mail drop, the POP3 server responds with a line containing information for that message. This line is also called a "scan listing" for that message.

In order to simplify parsing, all POP3 servers are required to use a certain format for scan listings. A scan listing consists of the message-number of the message, followed by a single space and the exact size of the message in octets. This memo makes no requirement on what follows the message

size in the scan listing. Minimal implementations should just end that line of the response with a CRLF pair. More advanced implementations may include other information, as parsed from the message.

NOTE: This memo STRONGLY discourages implementations from supplying additional information in the scan listing. Other, optional, facilities are discussed later on which permit the client to parse the messages in the mail drop.

Note that messages marked as deleted are not listed.

Possible Responses:
+OK scan listing follows
-ERR no such message

Examples:
C: LIST
S: +OK 2 messages (320 octets)
S: 1 120
S: 2 200
S: .
...
C: LIST 2
S: +OK 2 200
...
C: LIST 3
S: -ERR no such message, only 2 messages in mail drop

RETR msg

Arguments:
a message-number (required) which may not refer to a message marked as deleted

Restrictions:
may only be given in the TRANSACTION state

Discussion:
If the POP3 server issues a positive response, then the response given is multi-line. After the initial +OK, the POP3 server sends the message corresponding to the given message-number, being careful to byte-stuff the termination character (as with all multi-line responses).

Possible Responses:
+OK message follows
-ERR no such message

Examples:
C: RETR 1
S: +OK 120 octets
S: <the POP3 server sends the entire message here>
S: .

DELE msg

Arguments:
a message-number (required) which may not refer to a message marked as deleted

Restrictions:
may only be given in the TRANSACTION state

Discussion:
The POP3 server marks the message as deleted. Any future reference to the message-number associated with the message in a POP3 command generates an error. The POP3 server does not actually delete the message until the POP3 session enters the UPDATE state.

Possible Responses:
+OK message deleted
-ERR no such message

Examples:
C: DELE 1
S: +OK message 1 deleted
...
C: DELE 2
S: -ERR message 2 already deleted

NOOP

Arguments: none

Restrictions:
may only be given in the TRANSACTION state

Discussion:
The POP3 server does nothing, it merely replies with a positive response.

Possible Responses:
+OK

Examples:
C: NOOP
S: +OK

RSET

Arguments: none

Restrictions:
may only be given in the TRANSACTION state

Discussion:
If any messages have been marked as deleted by the POP3 server, they are unmarked. The POP3 server then replies with a positive response.

Possible Responses:
+OK

Examples:
C: RSET
S: +OK mail drop has 2 messages (320 octets)

6. The UPDATE State

When the client issues the QUIT command from the TRANSACTION state, the POP3 session enters the UPDATE state. (Note that if the client issues the QUIT command from the AUTHORIZATION state, the POP3 session terminates but does NOT enter the UPDATE state.)

If a session terminates for some reason other than a client-issued QUIT command, the POP3 session does NOT enter the UPDATE state and MUST not remove any messages from the mail drop.

QUIT

Arguments: none

Restrictions: none

Discussion:

The POP3 server removes all messages marked as deleted from the mail drop. It then releases any exclusive-access lock on the mail drop and replies as to the status of these operations. The TCP connection is then closed.

Possible Responses:

+OK

Examples:

C: QUIT

S: +OK dewey POP3 server signing off (mail drop empty)

...

C: QUIT

S: +OK dewey POP3 server signing off (2 messages left)

...

7. Optional POP3 Commands

The POP3 commands discussed above must be supported by all minimal implementations of POP3 servers.

The optional POP3 commands described below permit a POP3 client greater freedom in message handling, while preserving a simple POP3 server implementation.

NOTE: This memo STRONGLY encourages implementations to support these commands in lieu of developing augmented drop and scan listings. In short, the philosophy of this memo is to put intelligence in the part of the POP3 client and not the POP3 server.

TOP msg n

Arguments:

a message-number (required) which may NOT refer to a message marked as deleted, and a non-negative number (required)

Restrictions:

may only be given in the TRANSACTION state

Discussion:

If the POP3 server issues a positive response, then the response given is multi-line. After the initial +OK, the

POP3 server sends the headers of the message, the blank line separating the headers from the body, and then the number of lines indicated message's body, being careful to byte-stuff the termination character (as with all multi-line responses).

Note that if the number of lines requested by the POP3 client is greater than the number of lines in the body, then the POP3 server sends the entire message.

Possible Responses:

+OK top of message follows
-ERR no such message

Examples:

```
C: TOP 1 10
S: +OK
S: <the POP3 server sends the headers of the
message, a blank line, and the first 10 lines
of the body of the message>
S: .
...
C: TOP 100 3
S: -ERR no such message
```

UIDL [msg]

Arguments:

a message-number (optionally) If a message-number is given, it may NOT refer to a message marked as deleted.

Restrictions:

may only be given in the TRANSACTION state.

Discussion:

If an argument was given and the POP3 server issues a positive response with a line containing information for that message. This line is called a "unique-id listing" for that message.

If no argument was given and the POP3 server issues a positive response, then the response given is multi-line. After the initial +OK, for each message in the mail drop, the POP3 server responds with a line containing information for that message. This line is called a "unique-id listing" for that message.

In order to simplify parsing, all POP3 servers are required to use a certain format for unique-id listings. A unique-id listing consists of the message-number of the message, followed by a single space and the unique-id of the message.

No information follows the unique-id in the unique-id listing.

The unique-id of a message is an arbitrary server-determined string, consisting of characters in the range 0x21 to 0x7E, which uniquely identifies a message within a mail drop and which persists across sessions. The server should never reuse an unique-id in a given mail drop, for as long as the entity using the unique-id exists.

Note that messages marked as deleted are not listed.

Possible Responses:
+OK unique-id listing follows
-ERR no such message

Examples:

```
C: UIDL
S: +OK
S: 1 whqtsw00WBw418f9t5JxYwZ
S: 2 QhdPYR:00WBw1Ph7x7
S: .
...
C: UIDL 2
S: +OK 2 QhdPYR:00WBw1Ph7x7
...
C: UIDL 3
S: -ERR no such message, only 2 messages in mail drop
```

APOP name digest

Arguments:

a string identifying a mailbox and a MD5 digest string
(both required)

Restrictions:

may only be given in the AUTHORIZATION state after the POP3 greeting

Discussion:

Normally, each POP3 session starts with a USER/PASS exchange. This results in a server/user-id specific password being sent in the clear on the network. For intermittent use of POP3, this may not introduce a sizable risk. However, many POP3 client implementations connect to the POP3 server on a regular basis -- to check for new mail. Further the interval of session initiation may be on the order of five minutes. Hence, the risk of password capture is greatly enhanced.

An alternate method of authentication is required which provides for both origin authentication and replay protection, but which does not involve sending a password in the clear over the network. The APOP command provides this functionality.

A POP3 server which implements the APOP command will include a timestamp in its banner greeting. The syntax of the timestamp corresponds to the `msg-id' in [RFC822], and MUST be different each time the POP3 server issues a banner greeting. For example, on a UNIX implementation in which a separate UNIX process is used for each instance of a POP3 server, the syntax of the timestamp might be:

```
<process-ID.clock@hostname>
```

where `process-ID' is the decimal value of the process's PID, clock is the decimal value of the system clock, and hostname is the fully-qualified domain-name corresponding to the host where the POP3 server is running.

The POP3 client makes note of this timestamp, and then issues the APOP command. The `name' parameter has identical semantics to the `name' parameter of the USER command. The `digest' parameter is calculated by applying the MD5 algorithm [RFC1321] to a string consisting of the timestamp (including angle-brackets) followed by a shared secret. This shared secret is a string known only to the POP3 client and server. Great care should be taken to prevent unauthorized disclosure of the secret, as knowledge of the secret will allow any entity to successfully masquerade as the named user. The `digest' parameter itself is a 16-octet value which is sent in hexadecimal format, using lower-case ASCII characters.

When the POP3 server receives the APOP command, it verifies the digest provided. If the digest is correct, the POP3 server issues a positive response, and the POP3 session enters the TRANSACTION state. Otherwise, a negative response is issued and the POP3 session remains in the AUTHORIZATION state.

Note that as the length of the shared secret increases, so does the difficulty of deriving it. As such, shared secrets should be long strings (considerably longer than the 8-character example shown below).

Possible Responses:

```
+OK mail drop locked and ready  
-ERR permission denied
```

Examples:

```
S: +OK POP3 server ready <1896.697170952@dbc.mtview.ca.us>  
C: APOP mrose c4c9334bac560ecc979e58001b3e22fb  
S: +OK mail drop has 1 message (369 octets)
```

In this example, the shared secret is the string `tanstaaf'. Hence, the MD5 algorithm is applied to the string

```
<1896.697170952@dbc.mtview.ca.us>tanstaaf
```


which produces a digest value of

c4c9334bac560ecc979e58001b3e22fb

8. POP3 Command Summary

Minimal POP3 Commands:

USER name valid in the AUTHORIZATION state
PASS string
QUIT

STAT valid in the TRANSACTION state
LIST [msg]
RETR msg
DELE msg
NOOP
RSET

QUIT valid in the UPDATE state

Optional POP3 Commands:

APOP name digest valid in the AUTHORIZATION state

TOP msg n valid in the TRANSACTION state
UIDL [msg]

POP3 Replies:

+OK
-ERR

Note that with the exception of the STAT, LIST, and UIDL commands, the reply given by the POP3 server to any command is significant only to "+OK" and "-ERR". Any text occurring after this reply may be ignored by the client.

9. Example POP3 Session

```
S: <wait for connection on TCP port 110>
C: <open connection>
S:   +OK POP3 server ready <1896.697170952@dbc.mtview.ca.us>
C:   APOP mrose c4c9334bac560ecc979e58001b3e22fb
S:   +OK mrose's mail drop has 2 messages (320 octets)
C:   STAT
S:   +OK 2 320
C:   LIST
S:   +OK 2 messages (320 octets)
S:   1 120
S:   2 200
S:   .
C:   RETR 1
S:   +OK 120 octets
S:   <the POP3 server sends message 1>
S:   .
C:   DELE 1
S:   +OK message 1 deleted
C:   RETR 2
S:   +OK 200 octets
S:   <the POP3 server sends message 2>
```

```
S: .
C: DELE 2
S: +OK message 2 deleted
C: QUIT
S: +OK dewey POP3 server signing off (mail drop empty)
C: <close connection>
S: <wait for next connection>
```

10. Message Format

All messages transmitted during a POP3 session are assumed to conform to the standard for the format of Internet text messages [RFC822].

It is important to note that the octet count for a message on the server host may differ from the octet count assigned to that message due to local conventions for designating end-of-line. Usually, during the AUTHORIZATION state of the POP3 session, the POP3 server can calculate the size of each message in octets when it opens the mail drop. For example, if the POP3 server host internally represents end-of-line as a single character, then the POP3 server simply counts each occurrence of this character in a message as two octets. Note that lines in the message which start with the termination octet need not be counted twice, since the POP3 client will remove all byte-stuffed termination characters when it receives a multi-line response.

11. References

[RFC821] Postel, J., "Simple Mail Transfer Protocol", STD 10, RFC 821, USC/Information Sciences Institute, August 1982.

[RFC822] Crocker, D., "Standard for the Format of ARPA-Internet Text Messages", STD 11, RFC 822, University of Delaware, August 1982.

[RFC1321] Rivest, R. "The MD5 Message-Digest Algorithm", RFC 1321, MIT Laboratory for Computer Science, April, 1992.

12. Security Considerations

It is conjectured that use of the APOP command provides origin identification and replay protection for a POP3 session. Accordingly, a POP3 server which implements both the PASS and APOP commands must not allow both methods of access for a given user; that is, for a given "USER name" either the PASS or APOP command is allowed, but not both.

Further, note that as the length of the shared secret increases, so does the difficulty of deriving it.

Servers that answer -ERR to the USER command are giving potential attackers clues about which names are valid

Use of the PASS command sends passwords in the clear over the network.

Use of the RETR and TOP commands sends mail in the clear over the network.

Otherwise, security issues are not discussed in this memo.

13. Acknowledgements

The POP family has a long and checkered history. Although primarily a minor revision to RFC 1460, POP3 is based on the ideas presented in RFCs 918, 937, and 1081.

In addition, Alfred Grimstad, Keith McCloghrie, and Neil Ostroff provided significant comments on the APOP command.

14. Authors' Addresses

John G. Myers
Carnegie-Mellon University
5000 Forbes Ave
Pittsburgh, PA 15213

E-Mail: jgm+@cmu.edu

Marshall T. Rose
Dover Beach Consulting, Inc.
420 Whisman Court
Mountain View, CA 94043-2186

E-Mail: mrose@dbc.mtview.ca.us

14 Index

A

ActiveX 180
Actor 181
Answering Machine 60
Architecture 34
ASP 5, 86, 87, 88, 89
Authorization 49, 50, 94, 104

C

C++ 5, 34, 87
CallWave 181
Class 97
Client 46, 47, 49, 93
Computer Telephony 13, 62, 63
CoolSpeak 181

D

DELE 55, 56
Design 6, 62, 112
DFT 180
Diagrams 97
DLL 180
DTMF 40, 60, 70, 71, 72, 74,
77, 78, 80, 81, 82, 104, 105, 106, 107, 108,
109, 110, 118, 119

E

E1 181
eCommerce 181
Edutainment 19
eFax 181
E-mail 4, 11, 12, 15, 26, 60, 91, 93
eVoice 181

F

FFT 180
Flight Timings 181
FSM 180

G

Games 19
GUI 181

H

Homograph Disambiguation 21
HTML 5, 87

I

IIS 87, 88
Implementation 1, 5, 15, 29, 89, 91
Intelligent Network 68, 69
Internet 13, 62, 66, 86, 87, 88, 158, 160
Internet Telephony 13, 66
ISDN 181
IVR 63

J

JFax 181

L

LIST 52, 53, 54

M

MailCall 181
Memory 19, 61
Message Click 181
Mispronunciation 181
Modem 13, 67

N

News Update 181
NOOP 181

O

OLTP 181
OneBox 181
OOD 6
Operating System 181

P

Pagoo 181
PBX 181
Platform 88
POP3 47, 48, 49, 50, 51, 52, 53, 54,
55, 56, 57, 58, 93
POTS 181
Processor 13, 19, 61
Proof Reading 181
Prosody 20, 24
PSP 181
PSTN 13, 62, 63, 64, 66, 69
PTCL 180

Q

QUIT 50, 51, 57, 58

R

Rational ROSE6
 Recording 95, 97, 98, 99, 100, 101, 102
 Register 181
 Registration 14, 85, 86, 111
 RETR54, 55
 RSET.....56, 57

S

SAPI..... 17, 29, 32, 34, 35
 Security14
 Server 5, 86, 87, 88, 104
 ShoutMail..... 181
 Sink 41, 42, 43, 44, 45
 SIP 181
 Speech.....5, 13, 16, 17, 18, 20, 26, 27,
 28, 29, 31, 33, 34, 35, 36, 37, 38, 40, 42, 45
 Speech Recognition..... 31, 36, 37
 Sports 181
 SS7.....63, 65
 STAT51, 52
 Synthesis 181

T

T1..... 181
 TAPI5
 Telephony... 13, 40, 59, 60, 61, 62, 63, 64, 66
 Text Normalization20
 Time Shared 180
 Transaction 51, 57, 159
 TTS... 16, 17, 29, 34, 36, 38, 39, 43, 96, 107,
 108

U

UML6
 Unified6
 Update.....57, 158
 uReach 181
 Use Case..103, 104, 105, 106, 107,
 108, 109, 110, 111

V

Voice Mail.....60
 Voice Text..... 36, 39, 40

W

Weather Forecast 181
 Web Site..... 86, 89
 WOSA..... 180