Speak Freely for Windows

by John Walker (kelvin@fourmilab.ch)
WWW Home page: http://www.fourmilab.ch/

Speak Freely is a Windows application that allows you to talk (actually send voice, not typed characters) over a network. If your network connection isn't fast enough to support real-time voice data, various forms of compression may allow you, assuming your computer is fast enough, to converse nonetheless. To enable secure communications, encryption with DES, IDEA, and/or a key file is available. If PGP is installed on your machine, it can be invoked automatically to exchange IDEA session keys for a given conversation. Speak Freely for Windows is compatible with Speak Freely for Unix, and users of the two programs can intercommunicate. Users can find one another by communicating with a "Look Who's Listening" phonebook server. You can designate a bitmap file to be sent to users who connect so they can see who they're talking to. Speak Freely supports Internet RTP protocol, allowing it to communicate with other Internet voice programs which use that protocol; in addition, Speak Freely can also communicate with programs which support the VAT (Visual Audio Tool) protocol.

Introduction

Speak Freely is a Microsoft Windows application that allows you, with appropriate hardware and software, to send and receive audio, in real time, over a computer network. If you're connected to the Internet by a sufficiently high-speed link, you can converse with anybody else similarly connected anywhere on Earth without paying long-distance phone charges. Users can find one another, even if they have dial-up connections to the Internet, by publishing and searching directory entries on a Look Who's Listening server. You can designate a bitmap file to be sent to users who connect so they can see who they're talking to.

Speak Freely not only because you aren't running up your phone bill, but also knowing your conversation is secure from eavesdroppers. Speak Freely provides three different kinds of encryption, including the same highly-secure IDEA algorithm PGP uses to encrypt message bodies. By using PGP to automatically exchange session keys, you can Speak Freely to total strangers, over public networks, with greater security than most readily available telephone scramblers provide.

Speak Freely for Windows is 100% compatible Speak Freely for Unix, currently available for a variety of Unix workstations. Windows users can converse, over the Internet, with users of those Unix machines. In addition, Speak Freely supports the Internet Real-Time Protocol (RTP) and the original protocol used by the Lawrence Berkeley Laboratory's Visual Audio Tool (VAT); by selecting the correct protocol, you can communicate with any other network voice program which conforms to one of these standards.

Multicasting is implemented, allowing those whose networks support the facility to create multi-party discussion groups to which users can subscribe and drop at will. For those without access to Multicasting, a rudimentary Broadcast capability allows transmission of an audio feed to multiple hosts on a fast local network.

Hardware and software requirements

In order to use Speak Freely, you need a personal computer with the following hardware and software:

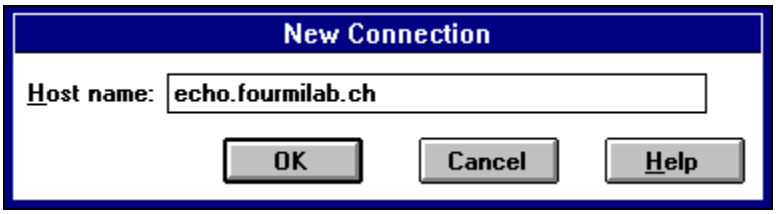
- Microsoft Windows 3.1 or above, in 386 Enhanced Mode
- Sound input/output card with Windows Multimedia driver
- Microphone and speaker(s) compatible with sound card
- Network interface with TCP/IP WINSOCK driver

Sending real-time audio over a data network is demanding on every component in the chain, and the performance required of your computer and network interact in complicated ways. For example, if you're communicating exclusively with other people over a high-speed local network and you aren't worried about eavesdropping, you don't need to enable either compression or encryption, both of which require a great deal of computation. For such an application a 386 machine is perfectly adequate. If your network link is slower, you'll have to compress the sound before it's transmitted. The most effective form of compression provided by Speak Freely, that used by GSM digital cellular telephones, reduces the data bandwidth requirement by almost a factor of five but is so computationally intense it can be done in real time only on a very fast 486 or Pentium machine. Encryption also takes time; the three methods available vary in the computation required. Compression reduces encryption overhead since there's less to encrypt.

Whether Speak Freely will work effectively for you depends upon your CPU speed, network bandwidth, load on the network, and the compression and encryption modes you select in a complicated and subtle manner. The best way to find out is to try it; if it works, great; if it doesn't, try again when you next upgrade your computer or network connection. You can experiment to determine which settings work best by connecting to an echo server which returns any sound you send to it after a 10 second delay.

Creating a new connection

To open a new connection, use the Connection/New... menu item. The new connection dialogue box will appear.



To initiate a network connection, enter the name of the host you wish to connect to, either as an Internet host name (for example, **stinky.dwarves.org**) or a numeric Internet address (IP address) such as **123.45.67.89**. Expert users can specify which Internet port number Speak Freely uses to communicate with the remote machine by appending it to the host name or IP address, separated by a slash (for example, **slimy.dwarves.org/5004**). Speak Freely's default port number is 2074.

When you press OK, an attempt will be made to establish the connection. Any errors which occur in the process will be reported in message boxes. If the connection is successfully established, a new connection window appears. If you select a host to which a connection is already established, its window will be activated. Once the connection is open, you can send live audio or sound files to the other party.

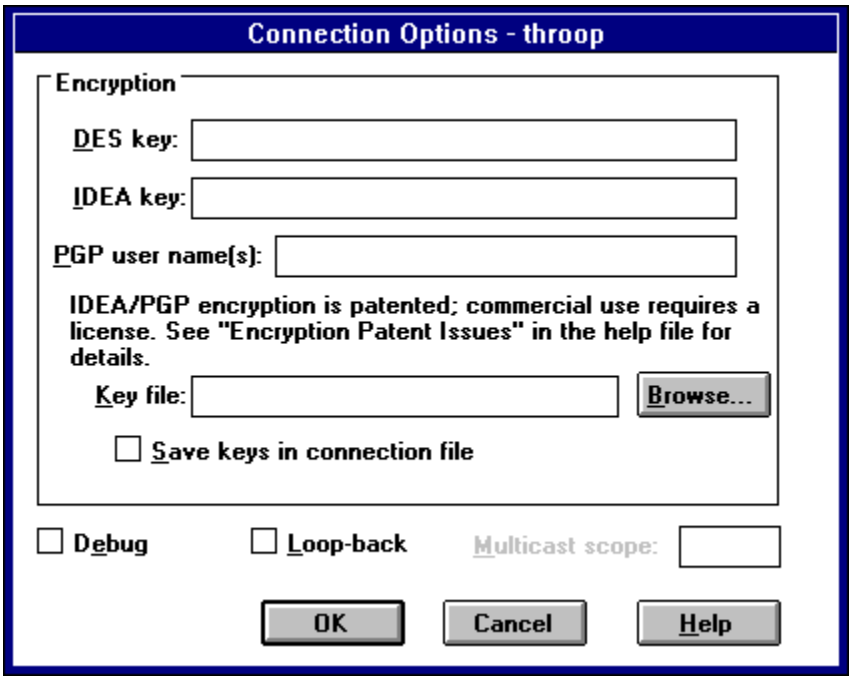
If audio is received from a host to which you don't have an active connection, a temporary connection is created. Unless you send audio or a sound file using that connection, it will be automatically closed after 30 seconds of inactivity. If you check the Options/Look Who's Talking menu item, if a new connection is established from a remote host while Speak Freely is minimised to an icon, it will pop open automatically so you can see who's just begun to talk to you.

If the person you connect to has provided an image of his or her face, it will appear in the connection window shortly after you receive the first sound from that user. Otherwise the user identity information (if any) published by the remote user is shown.

A new connection is created with default properties: no encryption or debugging selected. Use the Options/Connection menu item to select the modes you wish for the new connection.

For your initial experiments with Speak Freely, you may want to connect to an echo server which returns any sound you send to it after a 10 second delay.

Setting connection options



When a connection is active, you can use the Options/Connection menu item to display the Connection Properties dialogue, which allows you to specify encryption keys and debug modes for the connection. All of these modes are saved when you save a connection in a file.

Encryption keys

> Note: due to attempts by various governments to restrict the distribution and use of encryption, Speak Freely is distributed in two versions: with and without encryption capability. The non-encryption version (nicknamed "Spook Freely") can be posted on bulletin boards and on-line services in countries that impose restrictions on cryptographic software without fear of "imperial involvement". If you've obtained a non-encryption version of Speak Freely from such a source, you can replace it with a fully-functional version including encryption by downloading the software from the site listed in the Options/Connection dialogue box of the non-cryptographic version.

To encrypt audio you send and decrypt audio you receive with DES and/or IDEA, enter the key in the appropriate box; each key can be as long as 255 characters, perhaps a key phrase you've exchanged with the other party via PGP, or a "session key" generated automatically by Options/Create Key. To encrypt sound with a key given in a binary file, enter the full pathname of the file or use the Browse button to display an open file dialogue to select the file.

If PGP is installed on your machine, you can enter the names of one or more users on your PGP public keyring in the "PGP user name(s)" box. When you close the Connection Option dialogue, a random session key will be generated, PGP invoked to encrypt it with the public keys of the named individuals, and the encrypted session key transmitted. See "PGP Key Exchange" for additional information. The ability to encrypt a session key with more than one user's public key allows you to transmit securely to multiple subscribers to a multicast.

When you save a connection in a file, encryption keys are not, by default, written to the file; you must therefore re-enter them every time you reestablish the connection. If you check the "Save keys in connection file" box, they will be saved and restored automatically. This is very convenient, but consider that anybody who has access to files on your computer can obtain keys that permit them to listen in on your private conversations. It's up to you,

based on physical situation of your computer and personal trade-off of convenience versus security, to decide.

Debugging options

Checking the "Debug" box causes additional information to be displayed in the connection window when you send and receive sound. When you send sound from a connection with debugging enabled, debug information appears in the connection window on the receiving end as well. This is generally useful only to developers modifying the Speak Freely program.

Checking "Loop-back" causes the machine at the other end of the connection to immediately retransmit every packet of sound it receives back to your computer. You can use this to evaluate network performance and select an appropriate compression mode. In order to use loop-back effectively, your audio input/output hardware needs to be full-duplex--able to send and receive sound simultaneously. Many low-cost PC sound cards can't do this--whenever you're sending sound, any sound you receive is lost; if you have such hardware, there's no way to play the looped-back packets as they arrive.

Closing a connection

To close a connection, use the Connection/Close menu item. A connection can be closed at any time, even while you're sending or receiving sound (the transmission will be rudely interrupted). If additional sound subsequently arrives from the remote host, a new temporary connection will be automatically opened.

Closing a connection discards any changes you've made to the connection options. If you'd like to later reestablish the connection with the same options, save the connection to a file before closing it.

Saving a connection in a file

The Connection/Save and Connection/Save As... menu items create an **.SFX** file in which all the options of the connection are saved. You can later reestablish the connection with the same properties by loading the **.SFX** file with the Connection/Open... menu item.

The first time you save a new connection, use Save As... to specify the name of the **.SFX** file. Subsequently, you can use Save to write changes to the connection options to the file.

Opening a connection file

The Connection/Open... menu item opens a connection using the destination and connection options given in an **.SFX** file previously created by Connection/Save or Connection/Save As....

If a connection to the host named by the **.SFX** file is already open, its options are set to those found in the file.

You can also open a connection by dragging an **.SFX** file from the File Manager and dropping it in an open area of Speak Freely's window. If you make an association between the **.SFX** file type and Speak Freely, you can launch Speak Freely by double clicking an **.SFX** file in the File Manager. If a Speak Freely is already running when you double click on an **.SFX** file, it will appear as a new connection in the existing Speak Freely window.

If you'd like Speak Freely to start with one or more connections already open, you can specify their .SFX files on the Speak Freely command line. Other applications can initiate Speak Freely connections to remote hosts by writing a minimal **.SFX** file containing the host name, then launch Speak Freely with that **.SFX** file named on the command line. To connect to host **slimy.dwarves.org**, for example, you could use the following file:

[Host]
Name=slimy.dwarves.org

Receiving sound

As sound packets arrive, they're immediately sent to the audio output device. If a packet arrives from a host with which you haven't opened a connection, a new temporary connection to that host is automatically opened. If you respond to the host by either sending live audio or a sound file, the connection becomes permanent. Otherwise, a temporary connection is automatically closed if no sound is received from that host for 30 seconds.

If sound arrives while you're sending live audio and your audio hardware is half-duplex, Speak Freely, by default, simply discards the incoming sound and increments the number of lost input packets shown in the Extended Status dialogue. If you check the Options/Break Input menu item, sound that arrives while you're sending will, instead, interrupt your transmission, letting you know that the other person wants to say something.

Network traffic congestion and the fact that packets can travel on a variety of routes between two sites can lead to random pauses (jitter) in the sound you receive. To reduce the severity of the pauses, Speak Freely usually delays playback of the first in a sequence of sound packets to provide some margin for subsequent packets to arrive, even if slightly delayed. This improves the quality of the sound, but at the cost of introducing an additional delay before you start to hear a transmission from another user. The Options/Jitter Compensation menu allows you to select a variety of anti-jitter delays ranging from none at all to three seconds. If you're communicating across a local network, "None" is the best setting. The default, 1 second, generally gives much better results across Internet connections than no delay. If you have severe delay problems, you might want to try a higher setting. Lower jitter compensation times are usable when communicating between sites with high-bandwidth connectivity to the Internet.

If sound arrives simultaneously from more than one host, the packets are interleaved. This makes it difficult to understand, but it does permit interrupting a long winded speaker in a conference call.

Sending live audio

To send live audio to a connected host, move the mouse into its open connection window. When you do so, the mouse cursor changes into a button showing a telephone receiver. When you press and hold the left mouse button, the cursor changes to an ear and the legend "Transmitting" appears in the connection window, indicating you're now sending live audio from your audio input port to that host.

If you double click the left mouse button, live audio output is sent continuously until you next single click in the connection window. You can thus double click in multiple connection windows transmit simultaneously to a number of hosts (assuming your network is fast enough to handle the additional traffic). If your sound hardware is full-duplex, this lets you make "conference calls" where several people make connections to one another and take turns speaking. If your network supports it, multicasting may provide a more efficient way to conduct conference calls than directly transmitting to multiple hosts.

You can also use the space bar to toggle transmission of live audio; if you don't have a mouse, use Ctrl+Tab to activate the desired connection window, then press the space bar to begin transmitting. When you're done, a second press of the space bar ends the transmission.

After you've gotten familiar with sending in the normal "push to talk" mode, you may want to try out voice activated mode, which switches between transmit and receive automatically based on the sound level from your microphone.

Sending sound files

To send a prerecorded sound file to the active connection window, use the Connection/Send Sound File... menu item or drag the sound file icon from the File Manager and drop it in any connection window or icon, whether active or not.

Sound files can be in Windows **.WAV** format, monaural or stereophonic, 8 or 16 bits per sample, at 8000, 11025, 22050, or 44100 samples per second (if you're making a file expressly to be sent by Speak Freely, for best sound quality and efficiency select 8000 16 bit monaural samples per second). For compatibility with Unix and the World-Wide Web, you can also send files in Sun Audio File or raw Sun Audio formats (**.AU** files). Such files are assumed to be recorded at 8000 8 bit monaural samples per second, using mu-law encoding.

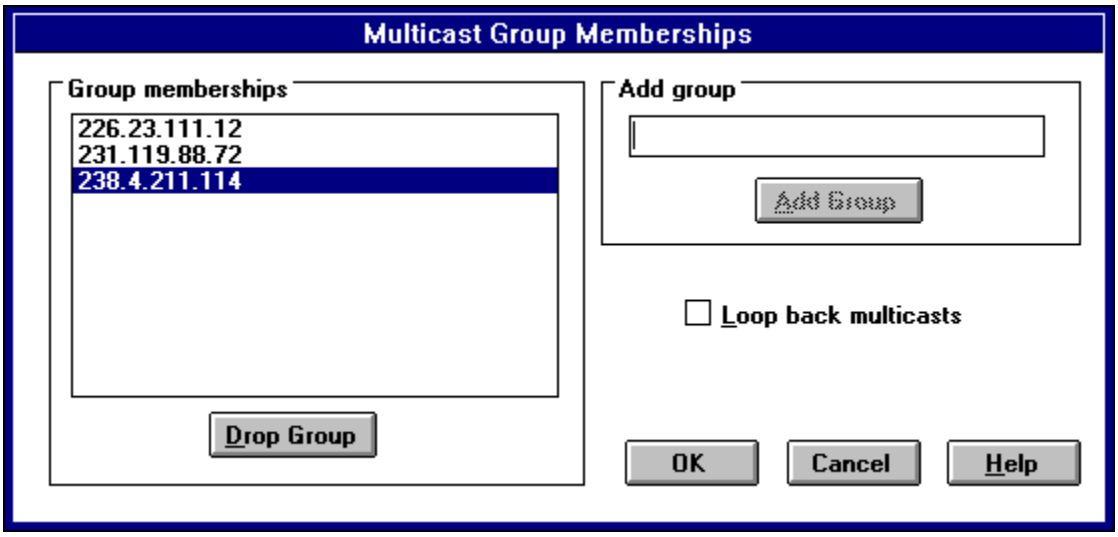
Ringing a remote user

Sometimes you're trying to establish a connection with a remote user who's running Speak Freely, but who's accidentally turned the volume down to zero or, as happens on Sun workstations, diverted output from the speaker to the headphones and forgotten to return it to the speaker when taking off the 'phones. Selecting the Connection/Ring menu item sends the "Ring sound file", the first packet of which will contain a flag which causes the receiving machine to try to get the user's attention. Speak Freely, upon receiving such a packet, sets the output volume to the maximum value if the Options/Workarounds/Audio/Set Maximum Volume on Ring menu item is checked; a Sun workstation diverts audio output to the speaker and sets output volume to the default level.

The Ring File can be any file you could send with Connection/Send Sound File.... The first time you use Ring, a dialogue appears which invites you to designate a sound file as the Ring file; subsequent Ring requests use that file without prompting you. You can change the Ring file at any time with the Options/Ring File Name... menu item.

It is impolite to ring a user without first seeing if a regular hail elicits a response.

Multicasting to a group



Some implementations of Windows Sockets support "IP Multicasting", a facility which allows the creation of conference groups which individual hosts can join and leave at will. A multicast conference is far more efficient than sending duplicate messages to all recipients, as actual replication of packets is done as close as possible to the actual recipient. If your network does not support Multicasting, you may be able to use the Speak Freely's Broadcast facility to transmit audio to multiple destinations.

If your Windows network software implements IP Multicasting, you can use the Connection/Multicast Groups dialogue to join and drop multicast conferences. To join a conference, enter its name or numeric IP address in the "Add group" edit box and press the eponymous button. If the address is a valid multicast address, it will be added to the "Group memberships" list at the left. To leave a conference, select its item in the Group memberships box and click the "Drop Group" button. If you've joined a multicast group and you send sound to it, the sound is normally sent back to your own machine. If you don't like this, or if it doesn't make sense because your sound hardware is half-duplex, uncheck the "Loop back multicasts" button to disable this action. Some Windows Sockets implementations don't allow you control over this behaviour; if that's the case, the Loop back button will be disabled.

You transmit to a multicast group as you would to any other host; create a new connection or open a connection file to the name or numeric IP address (and port number, if nonstandard) of the group. You can specify the extent of distribution of your multicast by entering a number in the "Multicast scope" field of the Options/Connection dialogue. The following are guidelines for multicast scope values:

Distribution	Multicast scope
Restricted to the same host	0
Restricted to the same subnet	1
Restricted to the same site	32
Restricted to the same region	64
Restricted to the same continent	128
Unrestricted	255

The distribution scopes given above should be taken *cum grano salis*. Their meaning depends entirely upon the implementation of the various intermediate links in the multicast network.

Broadcasting to multiple sites

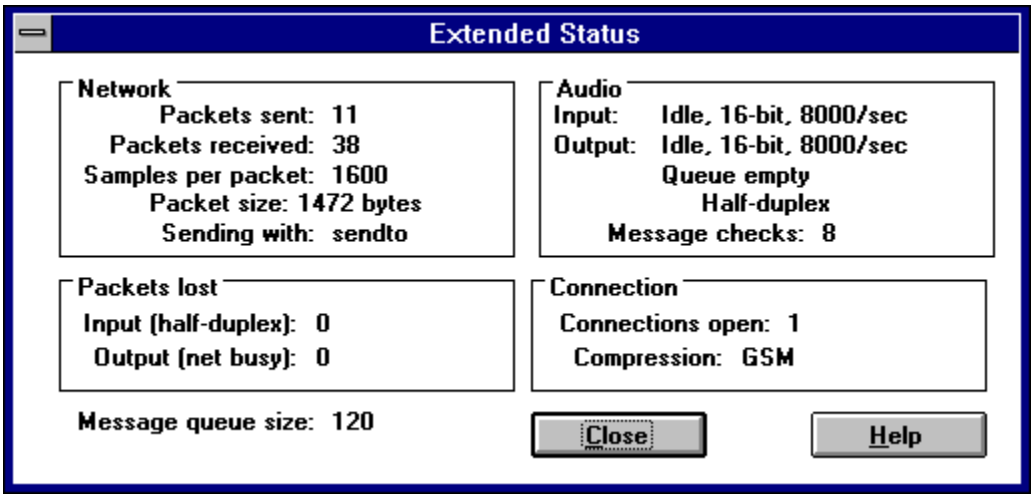
The most efficient way to transmit audio to a group of sites is "IP Multicasting", which allows the creation of conference groups that individual hosts can join and leave at will. A multicast conference is far more efficient than sending duplicate messages to all recipients. Unfortunately, many networks do not support, or have not enabled Multicasting, and often setting up Multicast groups requires the involvement of a site's system administrator, making it difficult to use for informal, *ad hoc* groups.

Speak Freely's Broadcast facility provides an alternative which requires no special network configuration. Without the benefit of Multicasting is it forced, however, to send duplicate packets to each recipient, which usually works only on fast local networks. Since many educational institutions and enterprises have such networks, broadcasting can be an effective way to transmit classes, seminars, and meetings to multiple destinations within the organisation.

Broadcasting is activated by checking the Connection/Broadcast menu item. Audio input is sent to all currently-open connections and the title bar displays a legend indicating a broadcast is in progress. When a broadcast is underway, other hosts can "subscribe" to the broadcast simply by making a connection to the broadcasting host and sending a short (say, one second) burst of sound to it. (The sound is discarded by the broadcasting host and will not affect the broadcast). This opens a connection to the new host, which will then begin to receive the broadcast. A host can unsubscribe from the broadcast by sending another short burst of sound. To prevent rapid toggling between subscribed and unsubscribed state, at least 10 seconds must elapse between subscribe and unsubscribe requests, and transmission to a host may continue for up to 10 seconds after it sends an unsubscribe request.

Toggling broadcasting off immediately ceases the transmission. Connections established during the broadcast will time out according to the normal rules unless additional sound is received from them or sound is explicitly sent to their connection. During a broadcast, mouse and keyboard input to connection windows is ignored; all connections remain in transmit mode, as indicated by the ear icon when the mouse is over a connection window.

Viewing extended status



The Help/Extended Status dialogue shows detailed information about the status of Speak Freely. Once displayed, the dialogue remains on the screen until you press the Close button and is updated in real time as events occur which change the status of Speak Freely. The status information is grouped into the following categories.

Network

Items in this box specify the number of sound packets sent to connected hosts and received from hosts since Speak Freely was launched. "Samples per packet" gives the number of original sound samples which can be stored in each 512 byte output packet, based on the compression modes currently selected. There are 8000 samples of audio per second; compression allows more original samples to fit in each packet. "Packet size" indicates the maximum size datagram packet your network can transmit; as long as this is more than the WINSOCK requirement of 512 bytes, everything should work fine. "Sending with:" indicates which socket write function is being used; to work around errors in certain network drivers, it's necessary to support both send() and sendto(). In case of trouble it's handy to know which one we're calling.

Packets lost

This box lets you know how many packets have been lost due to audio hardware or network bandwidth limitations. If your audio hardware is half-duplex, you won't be able to hear incoming sound while you're transmitting unless you check the Options/Break Input menu item. The "Input (half-duplex)" field increments every time a received packet is discarded because it arrived while you were transmitting. Input packets can also be lost if the size of the audio output queue exceeds 3/4 of the size of the message queue obtained from Windows; see the "Audio" and "Message queue" sections below for additional details. The "Output (net busy)" field increments every time a packet of sound you're attempting to transmit is discarded because your network connection was too slow to complete sending earlier packets. This is an indication you need to use a more efficient compression mode.

Audio

The current status, sample size, and sampling rate of the audio hardware are given for both input and output channels. "Idle" indicates the channel is not in use, "Active" that it's currently sending or receiving. The Output channel can also have modes of "Terminating", indicating shutdown in progress while Speak Freely is exiting, and "Transition": forced shutdown of output in progress when transmission is requested and the audio hardware is half-duplex. When output is in progress, a number of output packets can be in the transmission queue at once; the "Queue length" field indicates how many or "Queue empty" if no packets remain to play. The next line in the dialogue indicates whether the audio hardware is half- or full-duplex. Finally, "Message checks" gives the number of times a check was made to see if Speak Freely was falling behind due to the selected compression and encryption modes exceeding the

computer's ability to perform them in real time. If this number increases rapidly when you're sending, choose less demanding compression and encryption modes; if it increases rapidly while you're receiving sound, ask the person who's sending to select modes which don't overload your computer. A few message checks are normal; there's a problem only if the number increments continuously during transmission or reception.

Connection

The number of active connections (whether temporary or permanent) is given, along with the compression mode currently selected for audio sent to those connections.

Message queue size

Because Speak Freely can be doing many things at once, it needs to increase the size of the queue which receives messages from Windows to more than the default of 8 messages. When launched, Speak Freely attempts to increase the message queue to the maximum of 120 messages. The actual size obtained is shown in this field. The maximum number of packets in the audio output queue (see "Audio" above) is limited to 3/4 of the message queue size. If this limit is exceeded, the input packet will be discarded and the Input field in the "Packets lost" box incremented.

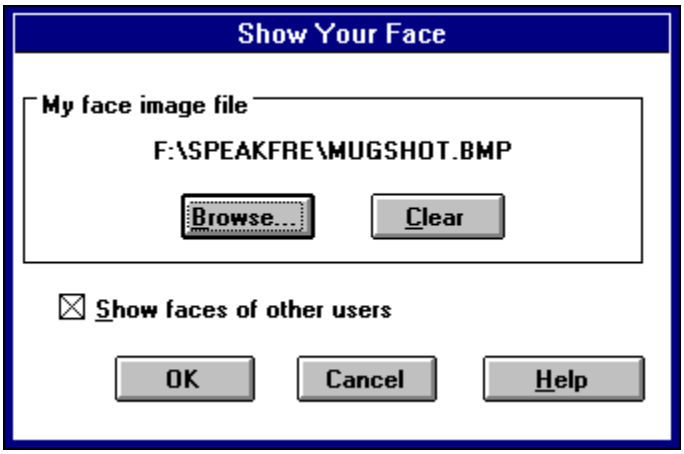
The Answering Machine

What if somebody calls you with Speak Freely when you're not at the computer? No problem! Enable the built-in answering machine, and everything will be saved for replay when you return.

The answering machine is initially disabled. To activate it, bring up the answering machine dialogue with the Connection/Answering Machine menu item and use the "Browse" button to specify a message file. This file should usually be kept on your hard disc; if you keep it on a RAM drive, all messages will be lost if you reboot the system or experience a power failure.

After selecting a message file name, check the "Record incoming messages" button and press "Close"; all subsequent messages will be saved in the message file. You can review the messages at any time by opening the Connection/Answering Machine dialogue. The Next, Previous, and Replay buttons function in the obvious fashion. Rewind returns to the start of the recorded messages. When you're done replaying messages, press "Erase All" to delete all the messages on the answering machine and reclaim the disc space used to store them.

Show your face



Speak Freely's Show Your Face mechanism lets people see who they're talking to. If you'd like to show people who connect an image of your face (or any other image, for that matter), create a GIF file or a 256 colour uncompressed Microsoft Windows Device-Independent Bitmap (**.BMP**) file containing the image. The image should not be larger than 128x128 pixels; otherwise it will take a long time to transfer and may interfere with the quality of audio while it's being sent. Use the Options/Show Your Face... dialogue to specify the bitmap file as "My face image file". The "Clear" button cancels a previously selected image file. If the "Show faces of other users" box is checked in this dialogue, face images will be retrieved, where available, and shown in the connection window.

When a face image is displayed in a connection window, transmission to the window is indicated by an arrow ("==>") before the site name in the window title.

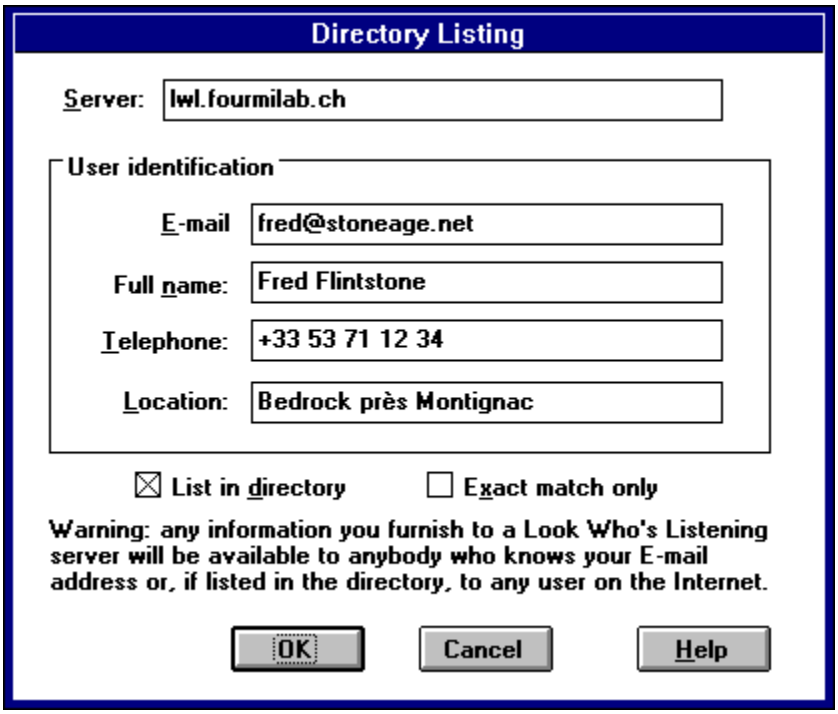
If you have trouble getting your face to appear on others' machines, the most likely cause is that you've made the image file as one of the many incompatible variants of **.BMP** format such as:

OS/2 instead of Windows
Monochrome, 16 colour, or 24-bit colour instead of 256 colour
RLE compressed instead of uncompressed

Make sure the image conforms to the format given above, and it should be transmitted with no difficulty. The rigid requirements on bitmap file format are imposed in order to make it easier for other, non-Windows systems, to display them.

All presently known variants of GIF files seem to work OK; since GIF files are compressed and hence somewhat smaller than the equivalent .BMP file, they're usually the best choice.

Publishing your directory entry



Speak Freely's "Look Who's Listening" facility allows you (at your sole discretion) to publish a directory entry on a server so other people on the Internet can see you're on line and willing to accept calls. If your Internet connection is via a dial-up line (SLIP or PPP connection) which assigns you as different Internet (IP) address and/or host name each time you connect, Look Who's Listening allows people to find your address and contact you any time you're dialed in.

If you'd like to publish your contact information on a Look Who's Listening server, use the Phonebook/Edit Listing... menu item to display the dialogue box shown above. Set the "Server" field to the host name or Internet address of the Unix site that is running the server. A list of servers operating at the time this version of Speak Freely was released appears in the section "Finding on-line users". The information in your directory entry will be visible only to users who query the same server on which you've published it. This allows private networks to set up an in-house Look Who's Listening server that is not accessible to everybody on the Internet.

Enter the information you wish to publish in the various "User identification" fields. The E-mail address is required; all the other fields may be left blank. **Think carefully before publishing your telephone number and location; you are potentially disclosing this information to every user on the Internet.** If you include a telephone number, please remember to include your international dialing country code and area code. The country code for Canada and the United States is 1.

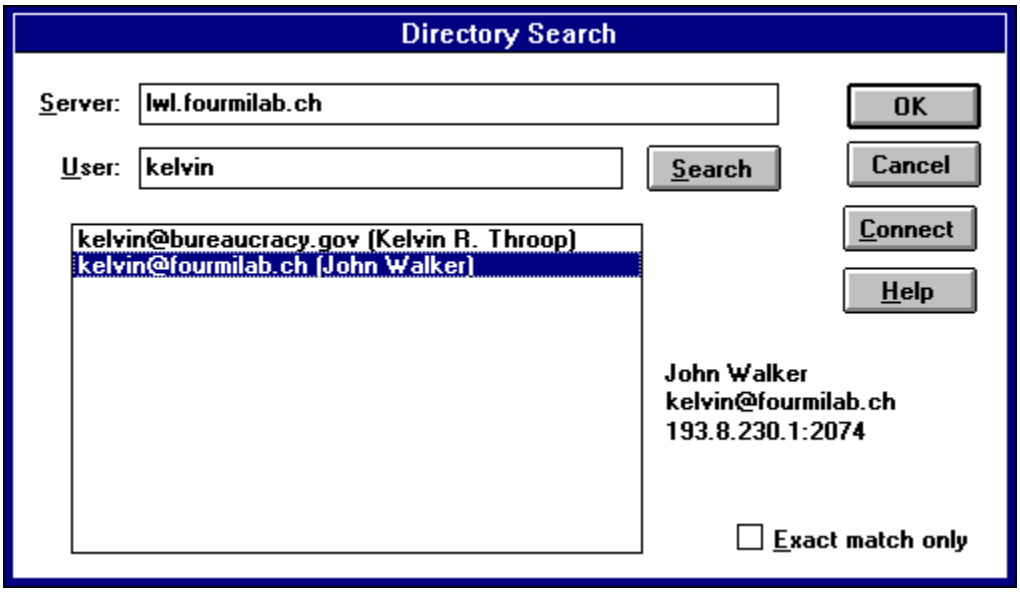
Your E-mail address is the primary means by which others contact you; enter the address you usually give to individuals who wish to contact you or include, for example, on your business card. It needn't have anything to do with the host and network on which you're running Speak Freely . For example, if you usually give out your E-mail address at work, you might specify **jetson@sprockets.com** even though you connect to the Internet at home as **george@slip3986.terra.ssol.net**.

The information you supply will be sent to the named server only if the "List in directory" box is checked. If you don't check "List in directory", your identity information will only be disclosed to people you connect to.

Normally, the server returns all active sites which contain the query string in either the E-mail address or full name

fields. If you check the "Exact match only" box, only queries which exactly match your E-mail address will return your contact information. Further, checking "Exact match only" excludes your entry from the World-Wide Web document published by the server listing all currently active sites. This allows dial-up users to permit those knowing their E-mail address to contact them while not informing any curious Internet user that they're on line. The security-conscious should note that this protection is provided by the Look Who's Listening server, and assumes the site you contact is running an unmodified version of the server program which is operating as intended.

Finding on-line users



Speak Freely's "Look Who's Listening" facility allows Speak Freely users to publish their E-mail and Internet addresses and optional additional information on a server accessible to other users. You can query a Look Who's Listening Server with the Phonebook/Search... menu item.

The "Server" specifies the host name or Internet address of a Unix site running the server you wish to query. If you've published your directory entry on a server, the same server is used as the default for searches. At the time this version of Speak Freely was released, servers were running at the following sites:

corona.itre.ncsu.edu	United States (North Carolina)
lwl.fourmilab.ch	Switzerland

As with everything on the Internet, servers are in a constant state of flux. For up to date information on available servers consult the Speak Freely World-Wide Web page <http://www.fourmilab.ch/speakfree/windows/>. It's generally best to publish your directory information on a nearby server. Remember that each server is independent; you only see users who have published their address on that server. If you want to communicate regularly with someone, it's best to agree to meet on the same server.

When you first connect to a server, the text box at the bottom left will show the server's welcome message, if any. This message usually identifies the server and indicates the location on the World-Wide Web where it publishes a list of all active Speak Freely users who have published directory entries there.

To look for an on-line user, enter the user's E-mail address in the "User" box and press "Search". If the user did not check "Exact match only" in the Phonebook/Edit Listing... dialogue and you did not check "Exact match only" in this dialogue box, all entries that contain the characters you entered in the "User" box within the E-mail address or Full name fields are returned, up to a limit of 5 to 10 depending on the length of the entries. The server is intended to find individual users, not provide a "wild card" list of a large number of active sites; consult the World-Wide Web document indicated by the server in its welcome message for a list of all active sites.

Mailing lists

To obtain additional information about Speak Freely, notification of new releases, and to meet other Speak Freely users to discuss problems and solutions, tips and tricks, and your experiences with the package, the following Internet mailing lists are available.

speak-freely@fourmilab.ch
: Unmoderated list for discussion of any topic related to Speak Freely. Each message posted to the list is immediately copied to all subscribers. To subscribe, send an electronic mail message containing the word "subscribe" in the message body (*no*t as the Subject) to **speak-freely-request@fourmilab.ch**. You can receive the same information in periodic digest form by subscribing to **speak-freely-digest@fourmilab.ch** described below, reducing the number of individual messages you receive.

speak-freely-digest@fourmilab.ch
: Periodic digest of messages sent to **speak-freely@fourmilab.ch**. The digest is updated every several days depending on the amount of traffic received; when traffic grows to a sufficient volume to warrant it, daily digests will be published. Subscribing to **speak-freely-digest** instead of **speak-freely** dramatically reduces the number of individual messages from the mailing list that arrive in your in-box, albeit at the cost of less timely delivery of information. To subscribe, send an electronic mail message containing the word "subscribe" in the message body (*not* as the Subject) to **speak-freely-digest-request@fourmilab.ch**.

speak-freely-announce@fourmilab.ch
: This list, moderated by Speak Freely author John Walker, is reserved for announcements of general interest to the Speak Freely user community such as:

  New releases of Speak Freely (Windows and Unix)
  New machine ports of Speak Freely for Unix
  Workarounds and patches for common problems
  Mirror sites for Speak Freely distribution
  New Look Who's Listening servers
  Periodic distribution of an FAQ containing all the above items.

  Announcements on **speak-freely-announce** are also posted to **speak-freely** (and hence **speak-freely-digest**), so if you subscribe to that list you don't need to separately subscribe to speak-freely-announce. To subscribe, send an electronic mail message containing the word "subscribe" in the message body (*not* as the Subject) to **speak-freely-announce-request@fourmilab.ch**.

Local loopback

Before you try contact other people or test over the network by contacting an echo server, it's a good idea to make sure your machine's audio hardware is set up properly. An easy way to verify this is Speak Freely's local loopback facility, which allows you to open a connection to your own machine that does not go over the network. Sound you transmit is stored in memory, then replayed shortly after you end the transmission. You can evaluate different compression modes and other options, and set your audio input and output levels optimally for the modes you're using.

To establish a local loopback connection, use the Help/Local Loopback menu item. Once the connection window appears, try transmitting short sequences of sound (like "Testing: one, two, three, four."). If all is well with your audio hardware, about a second after the end of each transmission you'll hear your voice replayed. If you don't hear anything, make sure your speaker is plugged into the right jack on the sound card and that the speaker volume is turned up. If you hear only a quiet hiss or hum, see if your microphone is plugged into the correct jack (a common error is to plug the microphone into a "Line in" jack designed for higher-level signals than the microphone generates). If you've checked these things and still can't hear anything in local loopback, try the suggestions in the list of frequently asked questions on setting up your audio hardware.

Since local loopback stores audio in memory rather than sending it over the network, the length of transmission it can store is limited. For short test messages, this isn't usually a problem unless your machine has extremely little free memory. If you're using voice activation, note that local loopback does not begin replay when silence is detected, but only when you end the transmission. The reason for this is that even though nothing is being sent, voice activation must still "listen" in order to resume transmission as soon as you resume speaking. With half-duplex sound hardware, this would prevent the looped-back sound from being played. Even with full-duplex hardware, the combination of voice activation and loopback would lead to an endless series of echoes if sound from the speaker triggered the microphone.

Echo servers

Setting up Speak Freely usually involves fiddling around with different compression modes, connection options, and, perhaps, workarounds for bugs in your network and audio drivers. Getting everything set right for your machine, network connection, and audio hardware usually requires testing various modes in real connections. It's irritating to get lots of "Hello, can you hear me?" calls which consume 10 or 15 minutes of your time each as a total stranger asks you to report on various settings on their end.

Echo servers allow you to run tests on your own, 24 hours a day, without disturbing others. An echo server is simply a machine running a special copy of Speak Freely for Unix which, rather than playing audio it receives on the speaker, stores it in memory for 10 seconds and then sends it back to the machine which sent it, using the same compression and encryption modes. Before you experiment with an echo server, you might want to try local loopback to verify that your sound hardware is working properly before venturing onto the network.

To run a test, create a new connection to one of the echo servers listed below. Unless you deliberately want to experiment with long distance transmission, it's usually best to connect to a nearby server. Then select whatever compression and other modes you want to try and transmit a short (less than 10 second) test message, such as the traditional "Testing: one, two three, four" and go back to receive mode. Ten seconds after the start of your test message, plus however long it takes the network to transmit the sound in both directions, you'll hear your test message returned by the echo server. If the audio is broken up, you may have to select different modes (or it may simply indicate traffic on the network between you and the server is so congested everything is being delayed).

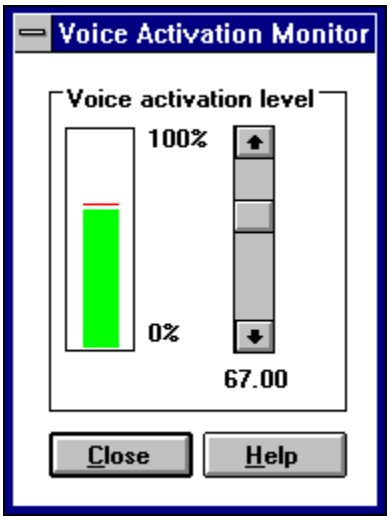
At the time this version of Speak Freely was released, echo servers were running at the following sites:

echo.fourmilab.ch	Switzerland (see note below)
corona.itre.ncsu.edu	United States (North Carolina)
rpcp.mit.edu	United States (Massachusetts)

As with everything on the Internet, servers are in a constant state of flux. For up to date information on available echo servers consult the Speak Freely World-Wide Web page <http://www.fourmilab.ch/speakfree/windows/>.

The **echo.fourmilab.ch** server shares a relatively slow connection to the Internet with the busy Web site **www.fourmilab.ch**. As a result, you may get break-ups when testing with that site purely because the Web traffic has saturated the capacity of the site's Internet connection. Other servers may have similar constraints, which often vary with the time of day and day of the week.

Voice activation



Network capacity (bandwidth) is finite and precious; it's important not to waste it. Even though users with full-duplex audio hardware could, in theory, transmit continuously, this would be irresponsible as it would double the load on the network with no real benefit. Speak Freely normally operates in "push to talk" mode like a handheld radio; you use the left mouse button or space bar to flip between sending and receiving; if your audio hardware is half-duplex, you have to switch modes somehow, since it can't send and receive simultaneously.

In radio communications, "voice activation" or "VOX" has been used for decades where convenience or the need for hands-free operation rule out push to talk. The idea is simple: monitor the microphone input and transmit only when the user is actually talking. In practice, voice activation has a number of subtleties that make implementing it and getting it to work well quite challenging. Fortunately, a Speak Freely user who is also an expert Windows application and driver developer, Dave Hawkes (daveh@cadlink.co.uk), took on the task of adding voice activation to Speak Freely and, fortunately for all of us, contributed the feature so we can benefit from it.

Voice activation can be tricky to set up; it's important to use a good microphone, well isolated from your speakers, set the input level correctly, and make sure echoes don't trigger transmission. Transmissions from Apollo astronauts on the Moon were voice activated, and if you listen to tapes of them, you'll hear the odd dropped word, echo feedback, and other inevitable artifacts of voice activation. Still, it got the job done and allowed the astronauts to concentrate on what they were doing rather than operating the radio. Speak Freely's voice activation can do the same for you.

It's best to become familiar with Speak Freely in push to talk mode; once you've mastered the basics of establishing connections, setting compression modes, and coping with the inevitable problems of sending voice over the network, you'll be ready to tackle the additional complexities of voice activation.

You enable voice activation with the Options/Voice Activation menu item. The default setting, "None", disables Voice Activation and selects the usual push to talk mode. To enable Voice Activation, check one of the three voice activation speed items, "Fast", "Medium", or "Slow". To avoid transmission breakups due to short pauses in speech, voice activation continues transmitting until a period of silence of a given duration has occurred; the choices refer to the length of silence that deems a transmission complete. For most purposes, "Medium" will work well.

Once you've selected voice activation, you need to adjust the level which causes Speak Freely to begin transmitting. This depends on your microphone, input gain setting, and the amount of background noise, so you have to set the level appropriate for your own environment. Use the Options/Voice Activation/Monitor menu item to display the Voice Activation Monitor dialogue box. Open a connection (perhaps to one of the echo servers),

and press the space bar to begin transmitting. The green bar graph at the left shows you, in real time, the sound level received from the microphone (while you're transmitting to any connection). The red line is the level above which voice activation enables transmission. You can move this level up and down with respect to the audio input level with the scroll bar. Adjust it so it's slightly above the level of the green bar when you're not speaking into the microphone. You'll see that when it's adjusted correctly transmission will stop (an X is drawn through the ear cursor) shortly after you stop speaking and resume (the X disappears) at your next utterance.

When using voice activated transmission, if your audio hardware is half-duplex, you should also select the Options/Break Input menu item to allow received sound to interrupt your transmissions. Otherwise, the continuous monitoring of the microphone would prevent your hearing remote users.

Communicating with other network voice programs

> Note: sending voice over the Internet is complicated enough even when the same application is used on both ends. Speak Freely supports the emerging standard protocol, RTP, and will eventually transition to using it as the default. Since the standard is so new, no program, Speak Freely included, can be absolutely guaranteed to support it completely correctly. So unless you are an experienced Speak Freely user who needs to communicate with users of other programs which support RTP or VAT protocols, you're well advised to ignore this section. As long as you're talking to another Speak Freely user, Speak Freely's own protocol (selected by default) will give you better performance and more secure encryption than are available with the standard interchange protocols. Further, many of the widely-publicised commercial Internet voice programs have their own proprietary protocols and do not yet support RTP. Do not assume you can communicate with a user of such a program unless its vendor explicitly certifies it to be RTP compliant.

As voice communication over the Internet moves from the exotic to the everyday, standards are emerging which allow users of a variety of Internet voice programs to communicate with one another. As users demand the ability to speak to one another regardless of which program they're using, vendors will be forced to conform to these standards.

Speak Freely supports the Internet Real-Time Transport Protocol (RTP) (RFC 1899) and RTP Profile for Audio and Video Conferences with Minimal Control (RFC 1890), as issued in January of 1996. See the Bookshelf for complete citations of these documents, including where you can download them on the Internet. In addition to RTP, Speak Freely also supports the protocol used by the Lawrence Berkeley Laboratory Visual Audio Tool (VAT), a widely used Unix conferencing program. (Starting with Version 4, VAT supports RTP in addition to its native protocol, so there's no reason to use the more limited VAT protocol except when communicating with an earlier version of VAT, or a VAT-compatible program which does not yet support RTP.)

When you receive sound from a remote site, Speak Freely automatically detects the protocol the user is transmitting and displays this in the connection window (unless a face image is displayed, but since only Speak Freely's native protocol supports face images, if a face appears you know you're connected to another copy of Speak Freely).

To transmit to the user of an RTP or VAT compatible application, you must select the appropriate protocol with the Options/Protocol menu item--Speak Freely does not automatically transmit in the protocol it's receiving, so if you are establishing a new connection to a site, you have to choose a protocol that site understands. (Since Speak Freely understands all three protocols, you can communicate with Speak Freely users regardless of which protocol you've chosen.)

Due to design differences in the various protocols, the following restrictions apply when using RTP and VAT protocols:

- Simple compression mode cannot be selected. RTP and VAT protocols do include this form of compression.
- The only encryption option available is DES. The other forms of encryption provided by Speak Freely are not presently a part of the specification of the other protocols.
- VAT protocol programs do not all use the same transformation of the key string into the DES key, so DES encryption may or may not work in VAT protocol, depending on which program you're talking to. It's best to use RTP protocol if at all possible.
- The DES encryption specified for RTP and VAT protocols includes the initial and final bit permutations, which most cryptographers believe serve only to deter software implementations. They do this quite effectively--DES encryption in RTP and VAT protocol require a substantially faster CPU to perform in real time than the permutation-free DES used by Speak Freely's own protocol.

- RTP and VAT use smaller packets than Speak Freely. This, coupled with the relatively poor real-time response of Windows, may result in "choppier" audio than with Speak Freely's own protocol. The stronger the compression mode you've selected (GSM and LPC are the strongest), the less the small packets will degrade the audio quality. A workaround allows you to transmit larger packets in RTP protocol, but this is not guaranteed to work with all RTP applications.

- RTP and VAT encrypt the entire transmitted packet, as opposed to including an in-the-clear prefix containing control information of no use to an eavesdropper as Speak Freely does. This has the consequence that if somebody is sending you encrypted packets in RTP or VAT protocol, there's no way to determine the identity of the sender or to discern what protocol they're sending.

Payload Types Supported

The "payload type" of a real-time protocol packet refers to the encoding and compression modes used to represent the data within it. All applications are not required to support all payload types, though support of a minimum subset is encouraged. Speak Freely supports all the payload types of VAT protocol and the following RTP payload types, using the nomenclature of RFC 1890.

DVI4, GSM, L16, LPC, PCMA, PCMU

To transmit in the protocol RTP and VAT refer to as DVI, select ADPCM Compression on the Options menu. PCMA is supported on receive only--since PCMU provides equivalent compression and fidelity and is one of the recommended minimum subset payload types, there is no need to implement PCMA transmission.

Compression modes

If you're talking to another user on the same high-speed local area network, or you're one of the lucky few with a high bandwidth connection to the Internet backbone, there's no need to bother compressing audio. The data rate of 8000 bytes per second is modest compared to other Internet applications such as file transfer and accessing graphics-intensive pages on the World-Wide Web.

The rest of us, faced with a bottleneck of anywhere from 14,400 to 65,536 bits per second between our machine and the rest of the world, have to find a way to squeeze 8000 bytes per second into a communications channel with a capacity between 1440 and 6500 bytes per second. Speak Freely provides four forms of compression, each with different trade-offs among efficiency of compression, loss of fidelity in the compression process, and the amount of computation required to compress and decompress.

Compression options

Compression is selected by checking one or more of the compression items on the Options menu. The chosen compression mode(s) apply to all sound transmitted to open connections: sound files as well as live audio. Compression modes cannot be changed while you're transmitting live audio; click the mouse in each transmitting connection window to pause transmission, change the compression mode, then click or double click to resume transmission.

If **no compression** is selected, Speak Freely requires your network to reliably transmit 8000 characters per second. If it's slower than that, the person you're talking to will hear pauses in the sound they receive and sound will be lost. Most local area networks, unless extremely heavily loaded, have no difficulty transmitting data at this rate--in fact, most are capable of speeds on the order of a million characters per second. It's when you leave your local network and venture into the worldwide Internet that compression becomes crucial. Very few Internet users today have connections faster than 64 kilobits per second, and many are using dial-up modem lines at 14.4 or 28.8 kilobits per second.

For asynchronous serial communication, the data rate in bytes per second is about one tenth the speed in bits per second so it's clear that even a 64 Kb line can't transmit uncompressed sound at 8000 bytes per second. Speak Freely provides three forms of compression which can be selected independently or in combination to reduce the data rate.

"**Simple compression**" discards every other sample and thereby halves the data rate to 4000 bytes per second, within the capability of a 64 Kb connection. On the receiving end, the elided samples are synthesised by averaging adjacent samples. Simple compression requires very little CPU time but it substantially degrades sound quality--high frequency components are lost and weird sampling aliasing can occur. Still, voice is generally intelligible and it's certainly better than random pauses and lost sound.

"**GSM**" compression (the default mode) employs the algorithm GSM (Global System Mobile) telephones use to reduce the data rate by a factor of almost five with little degradation of voice-grade audio. Enabling this option reduces the data rate from 8000 bytes per second to 1650 bytes per second, which renders a connection by 28.8 Kb modem usable. The catch is that GSM encoding is a very complicated process and, if your computer isn't fast enough, it won't be able to keep up with the audio coming in. (Decoding requires only about half the computation as encoding.) To use GSM compression, you'll need a fast 486, Pentium, or later generation processor. Thus, a slower network connection increases the demand on your computer.

"**ADPCM**" compression uses Adaptive Differential Pulse Code Modulation to halve the data rate to 4000 bytes per second. The compression is identical to that accomplished by Simple compression, but the loss in fidelity is much less; for voice grade audio, it's barely perceptible. ADPCM encoding and decoding requires more computation than Simple compression but enormously less than GSM; if your computer is too slow for GSM and the compression achieved by ADPCM is adequate for your network link, it's the best choice.

You can combine Simple and either GSM or ADPCM compression. The CPU requirement is only slightly greater

than for GSM or ADPCM compression alone and the sound quality is about the same as for Simple compression. Simple and GSM compression combined yield a data rate of 800 bytes per second, which a 14.4 Kb network link can handle. Simple and ADPCM compression together yield a data rate of 2000 bytes per second, within the capability of a 28.8 Kb link.

"**LPC**" compression uses Linear Predictive Coding to reduce the data rate by more than a factor of 12. This achieves the greatest degree of compression of any of the available options but, like GSM, it is extremely computationally intense. LPC requires many calculations to be done in floating point; if your machine does not have a math coprocessor, it will almost certainly be unable to do LPC compression and decompression in real time. LPC compression is extremely sensitive to high frequency noise and clipping caused by setting the audio input level too high. If you hear frequent bursts of loud static, try reducing the gain on the microphone or speaking further away from it. Also, try to avoid the pops that result from talking directly into the mike; they also create bursts of noise. Finally, users with high pitched voices may not be able to use LPC compression at all: it just loses too much high-frequency information. If GSM is a cellular phone, think of LPC as a shortwave radio. It doesn't always work, you have to be careful to get the best results, and even in the best of circumstances there will be some noise and distortion. But, like shortwave, it lets you communicate (or at least try) when nothing else will work. If your network link is so slow that none of the other forms of compression are usable, give it a try.

Only one of the compression modes GSM, ADPCM, and LPC may be selected at once. Choosing any of them turns off a previously-selected mode.

Here's a summary of the various compression options available to you:

Compression	**Bytes per second**	**Kilobits per second**	**Need fast CPU?**	**Sound fidelity**
No compression	8000	80000	No	Best
Simple	4000	40000	No	Poor
ADPCM	4000	40000	No	Good
Simple + ADPCM	2000	20000	No	Lousy
GSM	1650	16500	Yes	Good
Simple + GSM	825	8250	Yes	Lousy
LPC	650	6500	Yes	Depends

You can experiment to determine which settings work best by connecting to an echo server which returns any sound you send to it after a 10 second delay.

Why encryption?

Why bother with encryption? *Privacy!* When you talk over a data network, anybody connected to your network or with access to any of the links your sound data passes through on its way to a distant Internet site can potentially eavesdrop on your conversation. Encryption, based on a key known only to you and person you're talking to, protects against interception by third parties. No encryption scheme can be absolutely guaranteed to be 100% secure and even if it were, you'd still be at risk if somebody gained access to your key. But the encryption offered by Speak Freely, particularly the IDEA algorithm which is also used by PGP to encrypt message bodies, provides a great deal of security, indeed better than any generally available digital cellular telephone.

PGP key exchange

If PGP is installed on your computer, Speak Freely will cooperate with it to provide the convenience of public key encryption. To encrypt sound to one or more users on your PGP public keyring, enter enough of their user name(s) to uniquely identify them in the "PGP user name(s)" field of the Option/Connections dialogue. When you click OK to close the dialogue, Speak Freely generates a 128 bit random session key for subsequent communications, invokes PGP in an MS-DOS window to encrypt it with the public key(s) of the named user(s), and transmits it to the host or multicast group the connection addresses. Sound packets sent subsequently to that connection are IDEA encrypted (the same algorithm PGP uses for message bodies) using the session key. The ability to encrypt a session key with more than one user's public key allows you to transmit securely to multiple subscribers to a multicast.

When a session key is received from a remote host, PGP is invoked to decrypt it using your secret key. If you haven't specified your secret key pass phrase using the PGPPASS environment variable, you'll have to type the pass phrase in the MS-DOS window in which PGP is running. See the discussion of the security risks created by the undeniably convenient PGPPASS variable in Phil Zimmerman's *" The Official PGP User's Guide"* cited in the bookshelf.

Speak Freely invokes PGP via the SFPGP.PIF file in Speak Freely's release directory. If you'd like to change the modes used for running PGP (for example, to use full-screen mode instead of an MS-DOS window), edit this file with the PIF editor and select the modes you prefer. If no SFPGP.PIF file is found, Speak Freely attempts to run PGP directly, using whatever default modes you've set for MS-DOS programs launched by Windows applications.

The IDEA encryption algorithm used to encrypt audio following a PGP key exchange is patented and may not be used commercially without a license; see "Patent issues" for further details.

PGP key exchange and its subsequent IDEA encryption are independent of, and can be used in conjunction with, the other secret key encryption options provided by Speak Freely. There's little to be gained in security and everything to be lost in convenience by combining secret and public key encryption, but if you want to for some reason, you can.

Since the IDEA encryption performed by PGP key exchange is not specified as a part of the RTP and VAT protocols, PGP key exchange can be used only when transmitting in Speak Freely protocol.

DES encryption

If a DES key is specified in the Option/Connections dialogue, it will be is used to encrypt sound transmitted to that host using a slightly modified version of the Data Encryption Standard algorithm (the initial and final permutations, which do not contribute to the security of the algorithm and exist purely to deter software implementations of DES are not performed). In order to decrypt sound encoded with DES, the connection on the receiving machine must specify an identical DES key. The DES key phrase can be as long as 255 characters. The actual DES key is created by applying the **MD5** algorithm to the given key phrase, then folding the resulting 128 bit digest into 56 bits with XOR and AND.

Speak Freely will continue to correctly receive unencrypted sound from a given host even if a DES key is specified for the connection as long as the remote host is transmitting in Speak Freely protocol. RTP and VAT protocols do not permit this, so you must clear the DES key for the connection to receive unencrypted RTP and VAT transmissions.

IDEA encryption

If an IDEA key is specified in the Option/Connections dialogue, it will be is used to encrypt sound transmitted to that host with the International Data Encryption Algorithm (IDEA). In order to decrypt sound encoded with IDEA, the connection on the receiving machine specify an identical IDEA key. The IDEA key phrase can be as long as 255 characters. The actual IDEA key is created by applying the **MD5** algorithm to the given key phrase to create the 128 bit IDEA key.

IDEA encryption is substantially faster and generally considered to be much more secure than DES encryption. However, IDEA is newer, has not been formally adopted by governments, and is patented, restricting its commercial use. If your CPU is fast enough, you can enable any combination of IDEA, DES, and key file encryption. But since PGP uses IDEA to transmit message bodies, if you rely on PGP to exchange keys with other parties, the fundamental security of your voice link rests upon IDEA alone.

Cipher block chaining is used within each sound packet, but not from packet to packet. If that were done, loss of a single packet would render the entire rest of the conversation unintelligible.

Speak Freely will continue to correctly receive unencrypted sound from a given host even if an IDEA key is specified for the connection.

The IDEA encryption algorithm is patented and may not be used commercially without a license; see "Patent issues" for further details.

Since IDEA encryption is not specified as a part of the RTP and VAT protocols, it can be used only when transmitting in Speak Freely protocol.

Key file encryption

A key file is a binary file containing essentially random data as long as the individual sound packets being sent. Enter the full path name of the file in the "Key file" box of the Options/Connection dialogue box or use the "Browse" button to pop up a file open dialogue to select the file.

The file you specify as a key file should be at least 512 bytes long and consist of near-random (in the sense of incompressible) data. The "**+makerandom=***length filename*" option of PGP is an excellent way to generate such a file. To securely deliver a copy of the key file to the person you wish to talk to, encrypt it with their public key using PGP and send it to them, in ASCII armoured form, via electronic mail.

Key file encryption is by far the least demanding on your computer; it requires almost no additional computation. The level of security provided, however, is much less than the other encryption options, and should be viewed as a last resort alternative to transmitting in the clear if your machine is too slow to use any other form of encryption.

Speak Freely will continue to correctly receive unencrypted sound from a given host even if a key file is specified for the connection.

Since key file encryption is not specified as a part of the RTP and VAT protocols, it can be used only when transmitting in Speak Freely protocol.

Key generation, management, and exchange

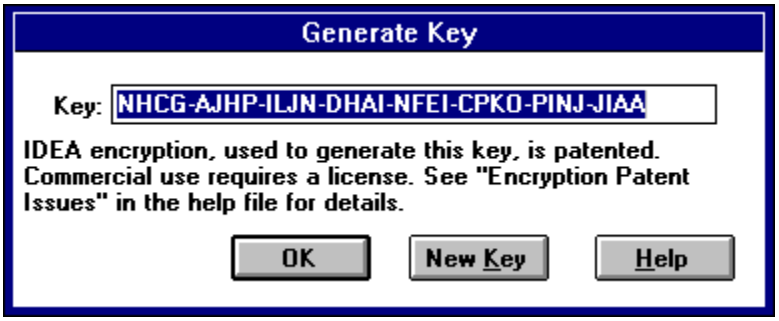
Unlike PGP, Speak Freely is a conventional secret key cryptographic system. Why? Because the RSA public key cryptosystem is the subject of U.S. Patent 4,405,929 and I have no desire to thread through the legal maze that PGP had to run in order to become both legal and freely available. Rather than replicating the public key functionality of PGP, Speak Freely cooperates with a copy of PGP installed on your machine, delegating the public key encryption of a session key to PGP. If you have PGP installed on your machine, please turn to the section on PGP key exchange. Users without access to PGP or who, for some reason or another, can't execute PGP from within Speak Freely can use the following technique to generate and exchange session keys.

First of all, make up a key. You could pick a long nonsense phrase like,

> The fribbits seem glinky today, don't you think?

or some gibberish pounding on the keyboard. (But watch out! That's often a lot less random that you might think. That's how the Russians used to make one-time pads for spies. The alternation of left and right hand keystrokes on mechanical typewriters was clearly evident from captured pads.)

But why go to the trouble when Speak Freely will make up a key for you on the spot? The Options/Create Key menu item generates a seed number from the time, date, and a variety of information about your computer and uses it to initialise a highly-secure IDEA-based random number generator. (Commercial users: read the "Patent issues" section before using this feature.) The key is displayed in a dialogue box:



Each time you press the "New Key" button, another key is generated, containing 128 bits of information, the same as an IDEA key and more than twice the 56 bits of a DES key. The text of the key is automatically selected and may be copied to the clipboard with Ctrl+C and pasted into a key field in the Options/Connection dialogue box and an electronic mail message to the person you want to speak with. The message might read something like:

> Hi! I'd like to talk to you around 21:00 your time tonight with Speak Freely. I've generated a key of:
>
> NHCG-AJHP-ILJN-DHAI-NFEI-CPKO-PINJ-JIAA
>
> for this conversation. Please paste this key into the IDEA encryption box when you set up the connection to my machine. I'm looking forward to chatting with you!

Encrypt the message with whatever tool you use to protect your electronic mail, and send it winging its way over the Internet.

When your friend receives the message and decrypts it, she will know the key to use for your forthcoming conversation. You can either generate a new random key for each conversation (as PGP does) or, if you trust the other person (and yourself!) to keep the key secret, use it for multiple conversations with that individual.

If you use a public key cryptosystem, this technique permits you to exchange keys for conversations with people you've never previously communicated with in any manner, as long as you trust their published public keys to

actually be theirs. Of course when you let Speak Freely generate the key, you're trusting me to not have installed a "back door" that allows me to know what key you'll get, or to have accidentally introduced a bug which makes the keys predictable.

Encryption legal issues

Certain governments attempt to restrict the availability, use, and exportation of software with cryptographic capabilities. Speak Freely was developed in Switzerland, which has no such restrictions. The DES, MD5, and IDEA packages it uses were obtained from an Internet site in another European country which has no restrictions on cryptographic software. If you import this software into a country which restricts cryptographic software, be sure to comply with whatever laws and regulations apply. The obligation to obey the law in your jurisdiction is entirely your own.

If you are concerned about the legal ramifications of using or redistributing software with encryption capability, a special version of Speak Freely (nicknamed "Spook Freely") with all cryptographic facilities removed is available.

The IDEA encryption algorithm is patented and may not be used commercially without a license; see "Patent issues" for further details.

Encryption patent issues

The IDEA[tm] block cipher is patented by Ascom-Tech AG. The Swiss patent number is PCT/CH91/00117, the European patent number is EP 0 482 154 B1, and the U.S. patent number is US005214703. IDEA[tm] is a trademark of Ascom-Tech AG. There is no license fee required for noncommercial use. Commercial users may obtain licensing details from Dr. Dieter Profos, Ascom-Tech AG, Solothurn Lab, Postfach 151, CH-4502 Solothurn, Switzerland, Tel +41 65 242 885, Fax +41 65 235 761.

You can use IDEA encryption for noncommercial communications without a license from Ascom-Tech; commercial use is prohibited without a license. If you don't want to obtain a license from Ascom-Tech, use DES or key file encryption instead.

Command line arguments

If you'd like Speak Freely to start with one or more connections already open, save the connections in **.SFX** files, and create a program item (Windows 3.x) or shortcut (Windows 95) which names the saved connections on the Speak Freely command line (target in Windows 95). Be sure to specify a complete path name for each connection file. If you'd like Speak Freely to start minimised rather than as an open window, add the "/S" switch to the command line or choose that mode when editing the properties of the program item/shortcut.

The following command line, which assumes you've installed Speak Freely in the directory C:\SPEAKFRE, starts the program minimised with two connections already open.

```
c:\speakfre\speakfre.exe /s c:\speakfre\alice.sfx c:\speakfre\bob.sfx
```

Problems: regular pauses in output

If you hear regular pauses in output you receive, or the person you're talking to reports the same in audio you send, the most likely causes are:

1. Your network connection isn't fast enough to send real-time audio with the compression modes you've chosen. Try additional compression to reduce the volume of data you're sending.

2. The compression mode you've selected (almost always GSM) requires more computation than your computer or the computer of the person you're talking to) can perform in real time. Choose a less efficient but faster form of compression.

3. You've chosen encryption mode(s) which require more computation than your computer can do in real time. Use fewer or less computationally intense modes of encryption. DES is the slowest form of encryption, IDEA (also used by PGP key exchange) is intermediate in speed. Key file encryption requires virtually no computation. If you've selected multiple encryption modes, the computation required is the sum of each of the individual modes. Note that DES encryption is much more time consuming in RTP and VAT protocols.

4. Your machine may be sufficiently slow that the mechanism Speak Freely uses to guard against system hangs due to overload is itself creating delays which cause packets to be lost. You can disable the system hang protection with the Options / Workarounds / Network/ Disable Message Loop Insurance menu item, but it's unwise to do this before you're absolutely sure the compression and encryption modes you're using don't overload your computer.

You can experiment with various compression and encryption modes without disturbing other users by connecting to one of the Speak Freely echo servers.

Problems: random pauses in output

Random pauses in output, for example when you've received sound for several minutes from a given connection, then lose three or four seconds of sound, or just random brief interruptions in the sound you hear are most probably due to:

1. The network you're communicating over, whether a local network in your office or the global Internet, is busy and sound packets are being delayed by other traffic. Unlike a file transfer which can proceed at any speed, intelligible audio requires not only adequate bandwidth (data rate) but consistent delivery time. The latter condition breaks down as a network approaches saturation. The only solution (other than connecting to a faster network) is to reschedule your conversation for a time when the network is less heavily loaded. You may be able to reduce the severity of the pauses in the sound you hear by increasing the jitter compensation time, which delays playback of a transmission to provide a margin for delayed packet to arrive.

2. Other tasks running in the background on your computer or that of your interlocutor may be stealing CPU cycles that Speak Freely needs in order to compress/decompress or encrypt/decrypt sound in real time. Terminate the background tasks.

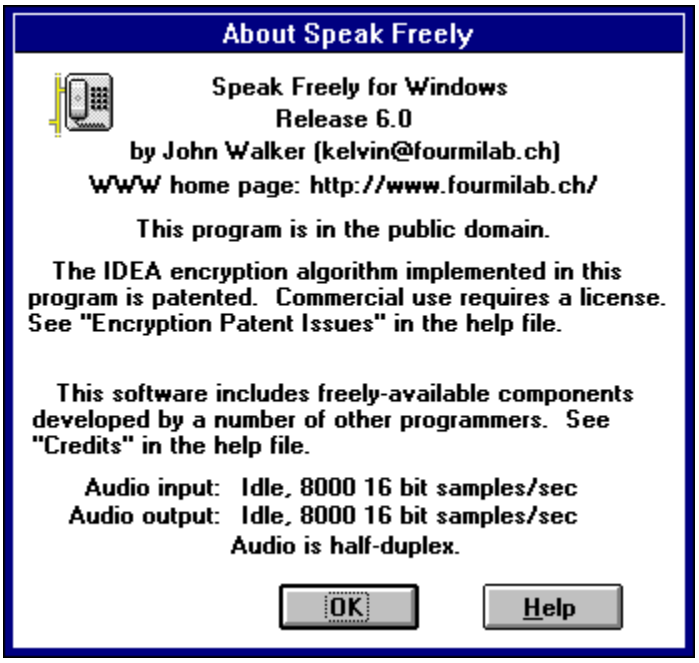
Problems: compression ***slows down*** connection

You've encountered regular pauses when sending or receiving sound and, to cope with the problem, enabled GSM compression to reduce the data rate. But the problem *got worse*-- more audio was lost with compression than without! What's going on?

This indicates that not only isn't your network connection fast enough to transmit audio in real time, your computer can't compress it into something your network can send quickly enough to keep up with the audio input.

Short of replacing your computer with a faster machine which can perform GSM compression in real-time or obtaining a faster connection to the Internet, the only options are to try ADPCM compression or to see if enabling both Simple and GSM compression with the Options/Connection menu item works. By discarding every other sample, Simple compression halves the amount of data the GSM algorithm must process in a given time period.

Viewing hardware configuration



The configuration of machine's audio input/output hardware is shown in the Help/About Speak Freely dialogue. The sample rate, bits per sample, and active/idle status are given for both input and output channels. If your machine's sound hardware can't send at the same time it's receiving (half-duplex), a line so indicating will appear.

More detailed information about configuration and the real time status of Speak Freely is available in the Help/Extended Status dialogue.

Sampling: 8 vs. 16 bit

For the best sound quality, 16 bit sampling is desirable. Speak Freely translates 16 bit Pulse Code Modulation (PCM) samples into an 8 bit logarithmic encoding known as mu-law, which effectively squeezes 13 bits of audio information into each 8 bit byte. If your sound hardware supports 16 bit samples, that mode is automatically chosen. If, for some reason, you want to run in 8 bit mode (which sounds worse and doesn't run any faster) you can do so by checking the Options/Use 8-bit Audio menu item.

If your hardware supports only 8 bit samples, this menu item is automatically checked and disabled, since you don't have the option of choosing 16 bits.

Half-duplex vs. full-duplex

On a regular telephone, you can talk and listen at the same time: the telephone is a *full-duplex* device. With a portable radio walkie-talkie, on the other hand, as long as you hold down the "Talk" button, you can't hear anybody else who's trying to talk to you--that's why radio users say "Over" at the end of a transmission--so the other person knows you've finished and they can talk now. The walkie-talkie is *half-duplex*: it can communicate in both directions but only one way at a time. (A radio broadcast station is *simplex:* you can't respond to its transmissions at all, except by calling the DJ on the phone.)

Audio hardware on the Sun and Silicon Graphics machines which run Speak Freely for Unix is full-duplex but, unfortunately, many inexpensive sound boards installed in Windows machines are half-duplex, intended for "recording" and "playing" like a tape recorder instead of real-time conversation.

Speak Freely copes with half-duplex audio hardware in the following manner. When launched, it immediately attempts to open both audio input and output simultaneously, input first. If the output open fails, but then succeeds on a second try after input has been closed, the hardware is marked half-duplex. You can see whether Speak Freely detected your hardware to be half- or full-duplex by displaying the Help/About Speak Freely or Help/Extended Status dialogue boxes.

If a half-duplex card is installed, pressing the mouse button to send live audio immediately mutes any sound you're receiving and discards any that arrives while you're talking unless you've checked the Options/Break Input menu item, in which case arriving sound disrupts your transmission. When you release the mouse button, output of sound from other hosts resumes. You can send sound files to hosts even while they're transmitting to you but, of course, if their own hardware is half-duplex they won't hear it.

If your hardware is half-duplex, use the double click feature with great care. If you accidentally leave a connection in constant transmission, output will remain muted and nobody will be able to speak to you. If an input-output conflict has caused your audio hardware to be treated as half-duplex, an indication to that effect appears in the Help/About Speak Freely dialogue box.

Since they all have full-duplex audio, some Unix workstation users of Speak Freely for Unix transmit continuously. If you're talking to somebody who does this, consider asking them to indicate, perhaps by saying "Over", when they're done speaking and expect a response from you and, ideally, use the push-to-talk mode provided in Speak Freely for Unix when communicating with you.

Bugs, features, and frequently-asked questions

I have a high-bandwidth connection to the Internet. Why do I get pauses and lost sound?

Unlike file transfers, transmitting intelligible audio requires not only adequate bandwidth but *consistent delivery time*. If one packet of sound takes a tenth of a second to arrive, the next two seconds, and the third half a second, not only will be there an audible pause but they'll be received and played out of order.

Network congestion and moment-to-moment re-routing can cause precisely this kind of inconsistent delivery time. If you're using a public network, there's nothing you can do other than try again later when there may be less traffic. Remember, the Internet was never intended to transmit real-time data such as audio; it's a miracle it works as well as it does.

When I send, nobody can hear me / I can't hear anybody else.

(The following was contributed by John Deters, `jad@dsddhc.com`). For those of you who aren't particularily familiar with Windows audio hardware but would like to use Speak Freely, here are some hints that might help get you started.

First, does your sound card play regular Windows sounds? When you start Windows, do you hear the "Ta-Da"? If not, make sure your speakers are plugged in to the "speaker" or "output" jack coming from your sound card. While you're at it, make sure your microphone is plugged into the "mic in" jack. If there is a volume knob on your sound card, make sure it's turned up. If your speakers have a power supply (such as batteries or a power transformer) make sure you have power, such as fresh batteries or the transformer is plugged into a live outlet. Make sure your speakers are turned on. You may need to refer to your sound card or speaker's documentation to get it set up correctly.

Some sound cards come with the microphone input not "turned on". What this means is that your sound card will not "listen" to your microphone until you tell it to. Included with your sound card was (probably) a "Mixer" application (if not, there may be a volume control application in your Accessories group.) Double click the Mixer to start it. With Sound Blaster cards, the mixer appears as columns of sliding volume knobs. Your sound card may have come with a different mixer.

If there is a volume control labeled "Microphone", this refers to how much of the sound from the microphone input will come out through your speakers. For now, turn it on (by checking the box or whatever) and set the volume to the same level as the volume control marked "Wave". If you're not sure, just turn it all the way up (you can always turn it down later). Make sure the switch on your microphone is turned on, and tap on it or talk in it. Do you hear yourself? If not, you need to find the input side of the mixer.

Hunt around for a menu option or button called "Recording Controls". When you select it, you'll see a similar looking screen that lists all of the inputs to your system. Turn the microphone input on by clicking the box, and if there is a "gain", set it to its maximum setting. Now, try tapping on the microphone. You should now hear the tapping coming from your speakers. Speak into the microphone, and compare the level of sound from your voice to that of the other sounds in your system.

You want your voice to come out about the same loudness as the "ta-da", and you want it to be intelligible. You may need to adjust the gain downward a bit, or find the correct place to hold your microphone. Some microphones need to be held almost to your lips, while others meant for mounting on your monitor need to be a foot away from your face before you sound good. Experiment with this for a while until the sound coming out of your speakers sounds good to you.

Once you have the microphone gain set, you should return to the output side of your mixer by finding the menu option or button labeled "Volume" or "Output". It might have opened a second window called "Recording Control" or "Input" that you will need to close. Once you get back there, you will probably want to turn the microphone OFF by un-checking the box. This will keep your side of the conversation off of your speaker, preventing nasty feedback squeals. When you've done that, you won't hear anything

else from your microphone coming out your speaker, but you now know that your microphone is set up for recording your voice. (Special note to headphone users: if you use headphones, leave the microphone *ON*. It will help your voice sound better to you, and your conversations will sound more natural. You might wish to adjust the microphone volume setting.)

For a good test, use the "Sound Recorder" program found in your "Accessories" group. You should be able to click on the "Record" dot, say something, click on stop, then rewind, then the play arrow, and hear yourself.

Once your audio hardware is correctly set up and working to your satisfaction, you will probably find that Speak Freely now works surprisingly well. You can use the local loopback facility to verify correct operation with Speak Freely, then proceed to experimenting with an echo server.

I tried to call you and nobody answered.

Tens of thousands of people have downloaded Speak Freely, and the first thing a depressingly large percentage of them do is immediately try to call me to "see if it works". If I allowed all these calls to interrupt me, development on Speak Freely (and everything else) would immediately cease. So while I occasionally accept calls, until and unless people become more considerate of my time, I mute the speaker whenever I'm doing serious work. Use one of the echo servers for testing; that way you won't bother total strangers with unsolicited calls.

My dial-up Internet connection gives me a different host name and IP address every time I connect. How can other people find me when I'm on-line?

Publish your E-mail address on a Look Who's Listening server. People who wish to call you can look you up on the server, see if you're currently connected and, if so, connect to the address for this session. You can use the same procedure to locate other users with dial-up connections. Since your E-mail address is unique and does not change from session to session, it allows others to find you regardless of how you are currently connected to the Internet.

Why do I connect to a machine (like furry.zoo.org) instead of a user (like panda@fuzzy.zoo.org)?

Because the audio hardware belongs to the machine, not the user. Think of it like a house with a single telephone; there may be several people living there, but they all share the same phone number and only one can use the telephone at a time. Your host name or IP address is just like a telephone number, and your computer the telephone. To find what host name a given user is connected to at the moment, look up their E-mail address on a Look Who's Listening server.

My machine hangs as soon as I try to transmit.

This is usually the result of failing to set a compression mode appropriate for the speed of your Internet connection. What should happen in this case is that sound that can't be sent in time is just discarded, but some implementations of WINSOCK seem to hang the machine when a program tries to send data faster than the network can accept it. In many cases (assuming you have a fast enough computer), setting Options/GSM Compression will cure the problem.

Is there / when will there be a Macintosh version?

As soon as somebody makes one. I have neither the knowledge nor the hardware to port Speak Freely to the Macintosh myself, but it would be a relatively straightforward project for a Mac developer experienced in both network and audio programming, I believe. The compression and encryption code should be usable with little or no modification. The bulk of the work would be in the user interface, network driver, and audio input/output which, due to great differences between Windows and the Macintosh, would have to be essentially rewritten. If you're interested in making a Mac version of Speak Freely, please let me know by E-mail (kelvin@fourmilab.ch) so I can let you know if anybody else has already undertaken the task.

I don't get it. I've installed Speak Freely and it runs OK, but nothing happens when I connect to the IRC server. What gives?

Speak Freely is a *telephone*, not a party-line chat program like IRC. You run a copy on your machine, the other person runs a copy on theirs, and then you talk to one another person-to-person--there's no server in the loop nor any need for one. Somebody could certainly *make* a voice chat program, but this isn't it. If your network supports multicasting, you can use that facility to organise conference groups individuals can join and leave at will. The Phonebook/Search... menu item permits you to access a Look Who's Listening server to locate other people running Speak Freely.

Can Speak Freely talk to *other network voice program*?

Yes, as long as the program supports either Internet Real-Time Protocol (RTP) or the protocol used by the Lawrence Berkeley Laboratory's Visual Audio Tool (VAT). Speak Freely automatically detects which protocol a program is sending and displays the protocol in the connection window. When transmitting to a user running an RTP or VAT compatible program, be sure to set Options/Protocol to the protocol that program requires. Many commercial Internet voice programs use proprietary protocols to guarantee users can only talk to others with the same program. Until the vendors of these products adopt RTP as a means of communicating with other voice applications, it will not be possible to communicate with users of such products.

How can I make sure Speak Freely is always running on my machine?

Put a copy of the Speak Freely icon in your StartUp program group (or whatever it's called if you're running a non-English edition of Windows). You'll probably want to check the "Run Minimized" box so it starts as an icon. If you check the Look Who's Talking menu item, the Speak Freely window will automatically pop open whenever a remote machine makes a connection to yours. If you'd like to automatically establish one or more ready-to-use connections, specify the names of the connection files describing them on the command line, separated by spaces.

Help! I'm trapped behind a firewall and can't talk to people at other Internet sites. What can I do?

Little or nothing, unfortunately. Speak Freely communicates using Internet UDP protocol on non-privileged ports 2074 through 2076. Most firewalls block all non-privileged (and hence unknown) port number packets, since there's no way to know they aren't being used by a mole or a Trojan Horse application to transmit sensitive data to a remote site. Speak Freely uses a nonprivileged port precisely to avoid the need for involving your system administrator in installing the program, but if you're behind a firewall you have no alternative. If you can persuade your jovial sysadmin to allow UDP packets on ports 2074 through 2076 to pass both directions through the firewall, you'll be all set, but the odds of this are extremely slim--I certainly wouldn't permit it were I your site manager. Once there's a known port out of your system, any program, not just Speak Freely, can transmit anything it likes accessible on your system to any other host on the Internet; if you permit that, why have a firewall in the first place? Basically, you'll just have to wait until the demand for network voice conferencing becomes strong enough at your site that your administrator installs a secure proxy tool to create a bridge across the firewall that Speak Freely can cross. In the short term, there's nothing you or I can do to get across the firewall. If you do decide to create a bridge across your firewall for network voice, you should probably also allow packets on ports 5004 and 5005 to pass--this is the standard port pair for RTP protocol.

Aren't you going to wreck the Internet by clogging it with all this sound traffic?

This is a legitimate concern in regard to *video* conferencing programs, but not with a voice-only tool like Speak Freely. With GSM compression, real-time audio requires a bandwidth of just 1650 bytes per second. This is far less than a typical FTP session, and people regularly make multi-megabyte FTP archives available without fear of clogging the Internet. A user accessing one of the many graphics-rich World-Wide Web sites can easily consume more Internet bandwidth than Speak Freely. In addition, the load created by real-time audio is inherently self-limiting. As a network link approaches saturation, the consistency of packet delivery time becomes dramatically worse, while non-real-time applications such as FTP and Web transfers simply begin to smoothly slow down. Long before the links between two sites reach their bandwidth limit, the audio users will have given up in frustration with break-ups and lost sound, to try again, perhaps, when the network is less busy.

Why do face images go all weird when more than one person is connected at once?

This occurs when your system has a 256 colour display board, and each face has its own set of 256 colours. The active connection will be displayed with the correct colours but other connections must use a default colour table. If you upgrade your display adaptor to full-colour, multiple face images will display correctly.

How secure is the encryption?

I've used the best algorithms I know of and applied them in the best ways I could given the constraints of real-time audio, fallible communication networks, and the computing power of contemporary personal computers. But I'm not a professional cryptographer or cryptanalyst and even if I were, you shouldn't believe something is secure just because the author claims it to be. One of the reasons I'm releasing complete source code for Speak Freely is to permit independent evaluation of the implementation and application of encryption within the program by experts in the community. With time, a consensus will emerge as to the degree of security Speak Freely provides and how to remedy any perceived weaknesses in future releases. Key file encryption is very insecure, but I already warned you about that; it's intended purely as a last-ditch alternative for users with computers too slow to run any of the other forms of encryption, yet who prefer any protection, however weak, to transmitting entirely in the clear.

Why do you use datagram protocol with no end-to-end acknowledgment that would permit detecting and correcting errors?

Network transmission delays rule out end-to-end acknowledgment. Audio has to be delivered in real time to be intelligible. Waiting for an ack from the other end would, in many cases, require delaying up to a second before sending the next packet. The best we can do is blindly spray packets at the other end in the hope enough will arrive with sufficiently consistent delivery time to be intelligible. Over slow links to distant sites, several packets will be flowing through the network toward the destination at a given time. Providing connections with guaranteed bandwidth and consistent delivery time is one of the main challenges in extending the Internet to accommodate real time audio and video communication.

I run the Windows debug kernel and I get *whatever* messages when I do...

I've sweated blood to try to make this thing debug kernel clean, and as of this writing, with half-duplex audio hardware, attempting to open input while output is open causes several "GlobalReAlloc failed" and "LocalAlloc failed" messages from the Kernel, which it presumably handles correctly since no apparent harm is done. But, as a battle-weary Windows warrior, I have no illusions that I've "found the last one". If you see warnings or errors unrelated to half-duplex output to input transitions, please let me know by E-mail (kelvin@fourmilab.ch) and provide as much information as you can about precisely what you were doing, what message(s) you got, whether you could reproduce the problem, and which sound card, network interface, and network (WINSOCK) software you're using. Thanks in advance.

Workarounds for driver bugs

One of the special joys of working with Windows is the never-ending challenge of discovering and working around ratty bugs in crummy drivers. With Windows late to arrive with a resounding thud in the worlds of networking and multimedia, Speak Freely finds itself close to the frontier, as it were, where frequent raids by marauding bands of byte bandits are the price one pays for the privilege of pioneering. The defensive programmer finds himself transcending that time-proven style of software development and becoming truly paranoid, always looking over his shoulder for the next incoming arrow of misfortune. This is, after all, a system on which functions such as GetTextMetrics() can return an error status.

The Options/Workarounds menu tree allows you to select workarounds for various errors in audio and network drivers, and variants of Speak Freely, RTP, and VAT protocols which may allow you to communicate with other Internet voice programs which implement those protocols in an eccentric manner. In an ideal world none of these would be needed, but in an ideal world Windows wouldn't *exist*. All workaround settings are remembered from session to session. Many workarounds can be selected only when no connections are active, and some workarounds take effect only when Speak Freely is restarted; a message box will appear to let you know if this is the case.

Available workarounds are described in the following paragraphs, along with suggestions as to when enabling them may be necessary.

Audio

Assume Half-duplex

Assumes the sound card is half-duplex without requiring it to fail an output open while input is open. Accommodates cards which are actually half-duplex but don't indicate so by failing when one attempts to open input and output simultaneously. Also handles cards which crash the system or application when you try to open them in full-duplex mode.

Assume 11025 Samples/sec

Assumes the card is capable only of 11025 samples per second mode, not our preferred 8000 samples per second. Permits correct operation on cards which don't fail when opened with a sample rate of 8000 samples per second but which can't actually run at that rate.

Set Maximum Volume on Ring

If this item is checked, when the first packet of a remote ring request is received, the output volume is set (if the sound card has that capability) to maximum. If other programs on your machine mute the speaker or turn down the volume to a low level, enabling this item may keep you from missing a call. But beware: some sound cards don't correctly handle setting the output volume--I've even encountered one which *mutes the microphone* when output volume is changed! So if you enable this mode, be sure to run some tests to make sure it's behaving as intended on your machine.

Network

Always Bind Socket

When a network socket is created for transmitting sound, there's no reason to bind it to an address, but some network drivers are reputed to fail if a socket isn't bound. Checking this menu item binds transmit sockets to persuade such feeble minded networks to let us use them.

Never Connect Outbound Socket

Don't connect() the output sockets. This implies we'll always use sendto() to write to those sockets. Clears "Use send(), Not sendto()" mode if set. Some WINSOCK implementations, notably Microsoft's own in Windows NT and Windows 95, blatantly diverge from the Berkeley sockets practice of treating connect() on a datagram socket as merely specifying a default address, not prohibiting subsequent use of sendto() with an explicit address. Checking this item disables the call on connect() entirely, just in case there's some driver which becomes entirely befuddled if it is used.

Use send(), Not sendto()

Always use send() to write to outbound sockets; don't wait for a sendto() to fail first. Accommodates drivers where a sendto() on a connected socket crashes the application or system. Clears "Never Connect Outbound Socket" mode if set. We normally auto-detect the failure of sendto() on a connected socket and fall back to send(). This item allows entirely bypassing the sendto() just in case it wreaks havoc when used on a connected socket.

Multicast TTL Argument Is char

The arguments for the multicast setsockopt() calls IP_MULTICAST_TTL and IP_MULTICAST_LOOP are documented as type "char" in every Unix Socket implementation I've seen. The Windows Sockets 1.1 specification does not contain these calls, as multicast was not a part of WINSOCK at the time. Microsoft's application note on multicast support in Windows NT (and now Windows 95) shows the argument for these two calls as "int" and, sure enough, if you pass a char the call errors with WSAEFAULT (bad address). Speak Freely conforms to the Microsoft specification and passes int arguments to these two calls but, just in case there's a more Unix-like WINSOCK out there which requires a char argument, provides you this workaround to use char instead. Even though we're running on a little-endian machine, since the length of the argument is passed in the setsockopt() call, the two cases are distinguishable.

Disable Output Overflow Recovery

Some versions of WINSOCK appear to crash the machine rather than throwing away UDP packets when the user selects a compression mode which transmits faster than the outbound network connection can accommodate. Speak Freely attempts to detect this and discard packets itself when this situation occurs, since losing data is better than a hung machine. The mechanism used to detect and recover from output overflow ventures into poorly-lit regions of WINSOCK where I suspect may lurk many bugs in various implementations, given how many of the *easy* things so many manage to get wrong. This workaround turns off output overflow detection and recovery code for such buggy WINSOCKs, running the risk of a crash due to output overflow if they are also buggy in that regard.

Disable Message Loop Insurance

If the selected compression and encryption modes (or the modes in packets being received) exceed the ability of the CPU to process in real time, there's a risk Speak Freely will hang Windows since sound buffers or packets from the network, combined with the computing to process them, result in Speak Freely never going idle and relinquishing control to other applications. To avoid this, there's a mechanism in Speak Freely which detects if 350 milliseconds or more have elapsed since the last opportunity for other applications to run and, if so, explicitly yields control to any waiting application(s). On slower machines, the very mechanism which saves them from hanging may, itself, cause pauses in sound. So, you can disable the message loop check (restoring the potential for a hang), if necessary, with this workaround. Don't disable it until you're confident your machine is working well with the compression and encryption modes you've settled on.

Get Host Name Synchronously

When you enter a numeric IP address (for example, 127.112.201.14) rather than a host name, Speak Freely attempts to look up the host name to display it in the connection window. It does this the recommended way, with the non-blocking call WSAAsyncGetHostByAddr. Unfortunately, this does not work correctly on every WINSOCK. The one that comes with Sun Select PC-NFS 5.1, for example, returns the correct results but plants a time bomb that can explode when you finally exit Speak Freely. This workaround uses the blocking call gethostbyaddr() which does not seem to trigger the bomb. Not that it works perfectly--it forgets to null terminate the host string it returns, but that's just ugly, not catastrophic. If the WINSOCK identifies itself as PC-NFS, this workaround is enabled by default.

Protocol

No Speak Freely Heartbeat

Disable the periodic Speak Freely protocol heartbeat on the control channel. This is primarily intended

as a last resort if the (less than 1%) added bandwidth saturates a close to the edge connection, and also in case the control channel packets awake something horrid lurking on the next higher port.

Large RTP Protocol Packets

Uses Speak Freely's preferred packet sizes for GSM and LPC compression rather than those typically sent by RTP programs. Most RTP programs were developed on fast workstations with high bandwidth network connectivity. Speak Freely users generally have slower machines and network links which benefit from larger packets. Try this if the person you're talking to reports halting audio in RTP protocol.

Disable VAT Protocol Detection

VAT protocol will never be automatically selected as a result of receiving a message on the control channel which resembles a VAT control message. Enable this if you never receive VAT protocol messages and are annoyed at how long it takes to identify the protocol of encrypted RTP messages.

Disable RTP Protocol Detection

RTP protocol will never be automatically selected as a result of receiving a message on the control channel which resembles a RTP control message. Enable this if you never receive RTP protocol messages and are annoyed at how long it takes to identify the protocol of encrypted VAT messages.

No Encryption of RTP Control Packets

RTP control packets can, according to the standard, be sent either encrypted or in the clear. Most RTP programs I've encountered encrypt their control packets, so this is the default Speak Freely sends (it accepts both encrypted and clear packets). If you set this workaround, control packets are sent in the clear.

Speak Freely for Unix

Speak Freely for Unix is currently available for a variety of Unix workstations. It allows network communications compatible with Speak Freely for Windows. The Unix version lacks the graphical user interface of the Windows edition, but supports all its compression and encryption modes. The Unix edition includes software which allows you to operate Look Who's Listening and echo servers.

The current version of Speak Freely for Unix is always posted in the directory:

ftp://ftp.fourmilab.ch/pub/kelvin/speakfree/unix

Documentation for the Unix version is included in the archive available from the above directory, and on the World-Wide Web as:

http://www.fourmilab.ch/speakfree/unix/

Credits

Like most free software, Speak Freely exists in large part because I was able to stand on the shoulders of other authors of generally available software. The following software components, either incorporated into Speak Freely or providing a model for how to develop similar software, tremendously reduced the blood, sweat, toil, and tears, not to mention man-months required to complete this software. Any restrictions on the use and distribution of these software components are noted below.

The **GSM compression and decompression code** was developed by Jutta Degener (jutta@cs.tu-berlin.de) and Carsten Bormann (cabo@cs.tu-berlin.de) of the Communications and Operating Systems Research Group, Technische Universität Berlin: Fax: +49.30.31425156, Phone: +49.30.31424315. They note that **THERE IS ABSOLUTELY NO WARRANTY FOR THIS SOFTWARE**. Please see the **README** and **COPYRITE** files in the **gsm** directory of the source code distribution for further details.

The **ADPCM compression and decompression code** was developed by Jack Jansen of the Centre for Mathematics and Computer Science, Amsterdam, The Netherlands. Please see the **README** and **COPYRITE** files in the **adpcm** directory of the source code distribution for further details.

The **DES encryption code** used with Speak Freely protocol was developed by Phil Karn, KA9Q. Please see the **README** file in the **des** directory of the source code distribution for further details.

The DES encryption library used for encrypting and decrypting VAT and RTP protocol packets was developed by Eric Young (eay@mincom.oz.au or eay@psych.psy.uq.oz.au). Please see the **README** and **COPYRITE** files in the **libdes** directory of the source code distribution for further details.

The **IDEA algorithm** was developed by Xuejia Lai and James L. Massey, of ETH Zürich. The implementation used in Speak Freely was modified and derived from original C code developed by Xuejia Lai and optimized for speed by Colin Plumb <colin@nsq.gts.org>. The IDEA encryption algorithm is patented and may not be used commercially without a license; see "Patent issues" for further details.

The **MD5 message-digest algorithm** implementation is based on a public domain version written by Colin Plumb in 1993. The algorithm is due to Ron Rivest.

The experimental **Linear Predictive Coding (LPC)** compression code was developed by Ron Frederick of Xerox PARC.

The **Voice Activation** code, remote **Break-in** feature, the ability to **open additional connections by clicking .SFX files** while Speak Freely is already running, and a **work-around for Speak Freely hanging the machine** when the user has selected compression and encryption modes which overload the CPU were contributed by Dave Hawkes (daveh@cadlink.co.uk), who also discovered an elegant way to get Windows to do most of the work in **jitter compensation**.

Bookshelf

The following references will help you understand the design, implementation, and use of Speak Freely.

Allard, J., Keith Moore, and David Treadwell. *Plug into Serious Network Programming with the Windows Sockets API*. Microsoft Systems Journal, **Vol 8, No 7**, 35, (July 1993). Excellent introduction to the Windows Sockets (WINSOCK) API used by Speak Freely for network communications. The **WORMHOLE** sample application presented in this article provided the model for Speak Freely, in particular suggesting that the Multiple Document Interface (MDI) was an excellent way to represent multiple simultaneous connections.

Davis, Ralph. *Windows Network Programming*. Reading (Mass.): Addison-Wesley, 1993. Documents the programming interface of a variety of networks, including a network-independent interface module for each. The chapter covering the Windows Sockets (WINSOCK) API is one of the clearest expositions of that facility I've encountered.

Denning, Dorothy E. *Cryptography and Data Security.* Reading (Mass.): Addison-Wesley, 1987. Thorough technical reference on the design and application of various methods. Includes an analysis of the strengths and weaknesses of DES.

Microsoft. *Microsoft Windows Multimedia Programmer's Reference*. Redmond, Washington: Microsoft Press, 1991. Documents the Windows Multimedia API, including the **waveIn***xxx* and **waveOut***xxx* functions used to receive and send audio, and the **mmio***Xxx* functions used to read **.WAV** files. This book is useful only if you want to modify the source code of Speak Freely.

Schneier, Bruce. *The IDEA Encryption Algorithm*. Dr. Dobb's Journal, **208**, 50, (December 1993). Detailed information on the design, cryptographic security, and implementation of IDEA, used by both PGP and Speak Freely. The source code included in this article was adapted to implement Speak Freely 's IDEA encryption.

Schulzrinne. H., R. Frederick, and V. Jacobson. *RTP: A Transport Protocol for Real-Time Applications*. Internet RFC 1889 (January 1996). Standards track specification of RTP, the proposed protocol for all forms of real-time data on the Internet. This document is available on the Internet as ftp://ds.internic.net/rfc/rfc1889.txt.

Schulzrinne. H. *RTP Profile for Audio and Video Conferences with Minimal Control*. Internet RFC 1890 (January 1996). Specifies audio and video encodings (compression and encryption modes) used within RTP packets. Speak Freely's RTP support conforms to RFCs 1889 and 1890. This document is available on the Internet as ftp://ds.internic.net/rfc/rfc1890.txt.

Zimmerman, Philip R. *The Official PGP User's Guide*. Cambridge (Mass.): MIT Press, 1995. Written by the creator of PGP, this book provides practical information on how to obtain, install, and use PGP to securely exchange information (including Speak Freely keys) even with strangers, and discusses the strengths and weaknesses of the DES and IDEA cryptographic algorithms and the legal issues associated with secure communication between individuals.

About the author

John Walker founded Autodesk, Inc. in 1982, was its president through 1986 and chairman until 1988.

Autodesk (ACAD-NASDAQ), one of the five largest personal computer software companies, has become a leader in the computer aided design industry; its first product, *AutoCAD*, is the *de facto* worldwide standard for computer aided design and drafting.

John Walker is co-author of *AutoCAD* and other Autodesk products, including *AutoSketch*, *AutoShade*, and *Cellular Automata Laboratory*. He is also author of various public domain programs including *Home Planet*, *Moontool; Moontool for Windows*; *Speak Freely for Unix*; *CODEGROUP; STEGO; SETTIME; XD*; *BGET*; *ATLAST*; *DICTOOL*; *PSTAMPR*; *RANDOM*; *DIESEL*; *SMARTALLOC*; and the PBMPLUS utilities *ppmforge*, *pgmcrater*, *sldtoppm*, and *ppmtoacad*. He has been recognised in *Scientific American* as having created, in 1975, the first (benign) computer virus. He was smeared by *Wall Street Journal* hatchetman G. P. Zachary in a front-page profile on May 28, 1992, and in reply produced and directed the video *Reporter At Work*, offering unique uncut coverage of a high-stakes boardroom confrontation between an entrepreneur and a reporter sent to ruin him.

Walker's first book, *The Autodesk File*, was published in 1989 by New Riders Publishing. It chronicles Autodesk's growth from $60,000 pooled by a bunch of programmers to a billion dollar company in less than eight years. The fourth edition of *The Autodesk File*, updated through the end of 1993, is available on the World-Wide Web at <http://www.fourmilab.ch/autofile/www/autofile.html>.

His second book, *The Hacker's Diet: How To Lose Weight and Hair Through Stress and Poor Nutrition* is also available on the Web: <http://www.fourmilab.ch/hackdiet/www/hackdiet.html>.

For access to all of Walker's public domain software and writings, visit his home page: <http://www.fourmilab.ch/>.

If his diet book doesn't make the bestseller list and land him a guest shot with Oprah, Walker is entirely prepared to complete his manuscript-in-progress: *CatSports---How to Improve Any Game by Replacing the Ball with a Cat*.

Fore!

John Walker
kelvin@fourmilab.ch
Neuchâtel, Switzerland
March, 1996
16,584 lines of original code

Development Log

23 August 1995

Initial announcement of Speak Freely Release 5.0.

24 August 1995

Peter Claus Gutman (pgut01@cs.auckland.az.nz), developer of a very nice encryption library, wrote to suggest his library might prove useful. In the source code for the library, I found a clever 80x86 assembly-language implementation of IDEA, made freely available by its author, who is identified in the source code only as "Bryan". Whoever you are, Bryan, great piece of work! If you, or somebody who knows who you are, happens to read this, let me know so I can give complete attribution.

I integrated the assembly language loop into IDEA\IDEA.C, modifying it slightly to work with Microsoft Visual C's inline assembler, so you don't need a separate assembler to take advantage of the optimised code. Whether the assembler or original C code is used depends upon whether USE_ASM is defined, so you can use the original loop for reference or if, for example, your compiler doesn't support inline assembly code or is incompatible with the way Microsoft do it.

Enabling the assembly language code increased the speed of IDEA encryption and decryption on my 486/50 machine from 152,000 bytes per second to 242,000 bytes per second--well worth the trouble of integrating the code.

30 August 1995

Completed a massive revision to avoid all packet fragmentation and thus work with WINSOCK drivers such as Trumpet WINSOCK. The changes were so great and ubiquitous there's no point in trying to describe them. In debugging the changes, one of the mysteries that has been dogging me was finally solved--the random hangs, loss of synchronism, failure to release resources, etc. etc. etc. were the result of Windows discarding messages to the main window as a result of overflows of the default 8 message queue. Speak Freely juggles a lot of balls in the air at once, and it's very easy to hit this limit. At initialisation time, we now try to expand the queue to its maximum size of 120 messages or whatever lower maximum the system we're running on supports.

Added the "Extended Status" (Propeller Head) dialogue.

Made compression modes global rather than per-connection. This means compression only has to be done once, which speeds up party line transmissions. The change is necessary in any case so that packet size can be optimised.

1 September 1995

Update release 5.1.

8 September 1995

CreateSocket() in UTILITY.C contained a "defensive bind()" to address zero as a work-around for some defective WINSOCK implementations. Unfortunately, this work-around causes other
that built into Windows NT to fail. I made the nugatory bind conditional on a new Options/Workarounds/Always Bind Socket menu item which is, of course, saved in the .INI file.

Update release 5.1a.

9 September 1995

Discovered that the reason the socket write was failing on Windows NT and Windows 95 is that Microsoft's built-in WINSOCK, entirely incompatible with Unix and every other WINSOCK I have encountered, refuses sendto() once

a datagram socket has been connect()ed. The sole function of connect() on a datagram socket is to specify a default address so subsequent writes can be done with send() (or, on Unix, write()), and there is no prohibition of overriding this default address with a subsequent sendto(). The WINSOCK specification nowhere mentions such a restriction as a Windows-specific change. I modified the socket write code in CONNECT.C and the loop-back socket write in FRAME.C to first try sendto(). If it fails, send() is then tried and if that works all subsequent socket writes for the rest of the session are done using send(). This code has been verified to work on both Windows NT and Windows 95 (first customer shipment edition). Special thanks to John Deters (jad@DHDSC.MN.ORG) who both identified the source of this problem on Windows NT and tested innumerable versions slowly converging toward the actual fix.

Added an item to the Propeller Head dialogue to indicate whether sendto() or send() is being used to write to outbound sockets; it's "Sending with" in the "Network" box.

Tested with the WINSOCK implementation included with Sun PC-NFS 5.1. Works fine.

Update release 5.1b.

10 September 1995

After last week's experience I decided to indulge in some preemptive workarounds for crummy network and sound card drivers which fail in obvious ways which haven't bitten me yet. I expanded the Options/Workarounds menu to include:

Audio

Assume Half-duplex

Assumes the sound card is half-duplex without requiring it to fail an output open while input is open. Accommodates cards which are actually half-duplex but don't indicate this by failing a simultaneous input and output open. Also handles cards which crash the system or application when you try to open them in full-duplex mode.

Assume 11025 Samples/sec

Assumes the card is capable only of 11025 samples per second mode, not our preferred 8000 samples per second. Permits correct operation on cards which don't fail when opened with a sample rate of 8000 samples per second but which can't actually run at that rate.

Network

Always Bind Socket

As before; bind outbound sockets, even though there's no need to do so.

Never Connect Outbound Socket

Don't connect() the output sockets. This implies we'll always use sendto() to write to those sockets. Clears "Use send(), Not sendto()" mode if set.

Use send(), Not sendto()

Always use send() to write to outbound sockets; don't wait for a sendto() to fail first. Accommodates drivers where a sendto() on a connected socket crashes the application or system. Clears "Never Connect Outbound Socket" mode if set.

Multicast TTL Argument Is char

Certain Winsock implementations by a *soi-disant* "setter of standards" headquartered east of Seattle, Washington in the United States flagrantly ignore the long-established convention that Boolean arguments to multicast setsockopt() calls are of type int. Their code errors such requests with a "bad address" fault, and accept them only if the argument is passed as a character (incompatible with Unix). This workaround is set if we empirically discover this to be the case on the system on which we're running, and can be set by the user to preempt dastardly behaviour by systems that don't have the courtesy to inform us of their

incompatibilities with contemporary community standards.

All the workaround modes are saved in the SPEAKFRE.INI file and apply to subsequent executions. The menu items are disabled when a connection is active.

As suggested by John Deters (jad@DHDSC.MN.ORG), I added the ability to automatically open an iconised version of Speak Freely whenever a new inbound connection is established. This lets you see the site that's just started talking to you. Since some people might find such an unsolicited pop-up irritating, this only happens if you check the new Options menu item "Look Who's Talking".

Tested under Windows 95 final build. Works fine when using the standard built-in WINSOCK, but doesn't resolve host names when Sun PC-NFS is overloaded on top of Windows networking. This appears to be a general problem of this configuration; other programs fail on gethostbyname() in precisely the same way. When you configure Windows 95, be sure to install the TCP/IP driver; if you don't you'll get nowhere fast.

12 September 1995

Sending a stereo .WAV file in ADPCM compression mode crashed the Unix speaker program. The code in READWAVE.C which calculates the number of bytes of .WAV file needed to fill a packet was incorrectly assuming the nBlockAlign field was the size of an individual sample, not the frame of samples for all channels. Fixed.

Closing a connection while a .WAV file was being sent orphaned the MMIO handle used to read the file. Fixed in CONNECT.C.

13 September 1995

Added the ability to drop saved connection (.SFX) files in the MDI frame window and thereby open (or activate, if already open) connections to the hosts given in the files. You can drop multiple connection files in a multiple selection and each will be opened.

CONNECT.C had its own implementation of DragAcceptFiles() which directly twiddled WS_EX_ACCEPTFILES. It doesn't any more.

If a connection file is named on the command line when the program is launched, it is opened once the application is initialised. This permits making an association between the .SFX extension and Speak Freely in the File Manager and launching the program for a given connection by double clicking the connection file. You can specify multiple connection files on the command line, space separated. This allows making a program item icon which opens a collection of connections, a handy thing to put in your StartUp folder. (Suggested by John Gilmore (gnu@toad.com)).

John also pointed out that the program wasn't usable without a mouse since the left mouse button was the only way to push to talk. I added logic in CONNECT.C that permits the space bar to be used to toggle push to talk, just as in the Unix mike program. You can cycle between open connections with Ctrl+Tab and use the space bar to select any set to which you wish to transmit.

Mouseless users who push to talk with the space bar don't have the
benefit of the cursor change to indicate which connections are transmitting. I added a "Transmitting" status indicator in the connection window which appears whenever live audio is being sent to the window.

If you make a .WAV sound file with the (nonstandard) sampling rate of 8000 samples per second, it is now played correctly by READWAVE.C, not forced to the closest standard sampling rate of 11025 samples per second. Conversion of stereo .WAV files into mono is still performed for 8000 sample per second files. If the user has the ability to make 8000 sample/sec .WAVs, this reduces file size, improves sound quality, and eliminates CPU overhead when sending such files. .AU files remain the fastest, since they're already mu-law encoded.

14 September 1995

Update release 5.1c.

20 September 1995

Began work on answering machine. Defined structure for data in file, added a new ANSWER.C module with a function to save a sound buffer in an answer file in that format.

25 September 1995

Modified the new connection dialogue handler to allow numeric IP addresses which can't be resolved into host names. If the host name lookup fails, the dotted IP number from inet_ntoa is used as the host name.

Good ole' Trumpet Winsock returns an error status if gethostname() is called with a buffer too small to hold the entire name, as opposed to truncating it as Unix does. I changed the two calls in CONNECT.C to get the host name in a temporary buffer, then copy as much as will fit into the sendinghost field of the sound buffer.

Added the ability to set the multicast scope with a new item in the Options/Connection dialogue. This item is enabled only if the IP address is a valid multicast group number.

Bad ole' Windows 95 WINSOCK returns a WSAEFAULT error if you pass a single byte argument for the IP_MULTICAST_TTL setsockopt() call. This is incompatible with all Unix documentation I have seen. Trumpet works correctly with the single byte argument, and accepts the 2 byte short required by Windows 95. Given the likelihood there's some other WINSOCK that requires a one byte argument, in goes another Options/Workaround/Network item: "Multicast TTL Argument Is char" which does it the Unix way, not as required by Windows 95.

26 September 1995

Added a new Connection/Multicast Groups dialogue which allows adding and dropping membership in multicast groups. Groups can be specified by DNS-resolvable name or by IP address. A check box controls multicast loop-back of locally sent packets to groups in which this host has added membership. The loopback box is disabled on systems (such as Windows 95) which do not implement the IP_MULTICAST_LOOP setsockopt() option.

1 October 1995

Discovered the multicast tear-down code in the WM_DESTROY message handler of FRAME.C wasn't testing for a NULL multiName[], resulting in bad GlobalFree() calls when we failed to initialise a multicast port. Fixed.

FRAME.C wasn't killing the main timeout timer at WM_DESTROY. Fixed.

If the attempt to drop a multicast membership at WM_DESTROY time failed, a message box was displayed as a child window of the one frame being destroyed. This is apparently (yet another of the billions and billgatesillions) undocumented no-no--in any case, if you do it, you get an "err USER: Attempt to activate destroyed window" at the time the WM_DESTROY returns. I changed the parent of the message box in this case to be NULL and it seems to be happy now. (In FRAME.C).

Finished implementation of the answering machine, ANSWER.C. I'll probably be back before long to make it more message-oriented (select message from a list box of sites and times, individually delete messages, etc.) but at least it now has basic functionality.

2 October 1995

Added keyboard accelerator (CTRL+T) for answering machine, and a new connection menu item that lets you

toggle whether incoming messages are recorded without having to pop up the answering machine dialogue. Fixed a bug in which checking or unchecking the record incoming messages box in the answering machine dialogue didn't take effect until you closed the dialogue; now it takes effect immediately.

Added code to overwrite the 16 byte session key exchanged via PGP before closing the file on disc. Unfortunately, since we can't transmit and receive the with a pipe, as we do on Unix, there's still a window while PGP is running during which the session key is visible, but at least this keeps it from lying around in unallocated disc space for an indeterminate time.

If no answering machine message file was configured, the answering machine dialogue in ANSWER.C called scanMessageFile anyway. Unfortunately, that routine didn't test for answerFile being NULL and proceeded to stomp all over memory. Fixed in ANSWER.C scanMessageFile().

Moved all translatable strings and formats from the .C modules into the string table of the resource file, using the rstring(), rfilter() functions and the Format() macro as intermediaries. Strings that aren't to be translated,such as fopen() mode strings, formats that contain only a field editing code, etc. continue to appear as strings in the source code. Banishing these strings to the resource file reclaimed almost 4K of data space, enough to give us some breathing room should it prove necessary to introduce another static full-size sound buffer for some reason.

3 October 1995

The enabling and disabling of buttons in the answering machine was befuddling Windows' dialogue box keyboard accelerator logic. I added code at the end of a message replay to restore the input focus to the button last pressed or its logical successor if that button has become disabled as a result of the message we just completed.

Keyboard accelerators in the answering machine were less than optimally chosen due to renaming of buttons during its development. I rationalised them so the most commonly used buttons have the most obvious keyboard shortcuts.

Pressing the Close button in the answering machine gave a debug kernel "err: window destroyed in window callback". Why, I know not. It uses the standard code for modeless dialogues right out of Petzold, which identical code works perfectly in the propeller-head modeless dialogue. Changing the DestroyWindow() to a PostMessage of WM_CLOSE to ourself made the message go away. I changed the propeller-head dialogue in DIALOGS.C to use the same logic.

Several modal dialogues needlessly included the system menu in their title bar. Eliminated. (The modal dialogues such as the answering machine and propeller-head continue to display the system menu.)

Installed help buttons in all the dialogues, linked to the topic in the help file which describes the dialogue.

Moved the names of our help file and the base Windows help file into the resource string table.

I removed the "How to use help" menu item, which has fallen out of fashion.

Changed "Help/Search..." to "Help/Search for Help on..." as used in current Microsoft applications.

4 October 1995

Completed moving all section and item titles for the main .INI file and saved connection files to the string table in the resource file. Whether these should be translated isn't clear: a normal user won't ever examine these files and translating renders them incompatible between different language editions. But the saving in data segment size by elminiating duplication of the section titles alone justifies the work.

Added two new string constants kS0[1] = "0" and kS1 = "1" to FRAME.C and changed all references to the explicit constants in profile file I/O to use them. This eliminates redundant string constants in the data space.

Found a few string constants I'd missed somehow in READWAVE.C. Banished.

Fixed the answering machine to update the host name when a definitive name (one not displayed in parentheses) is seen, replacing any previously displayed name.

Added help butttons to all the file open dialogues, linked to the appropriate topics in the help file.

Added a pleasant default ring file. I haven't found a suitable (well-recorded and public domain) telephone bell, so I decided to pioneer non-irritating notification of an incoming call with this wind chime derived sound. The original appeared on the CD-ROM (N° 5) accompanying "News Windows" N° 26 (octobre 1995) as the file WINDBELL.WAV. I used Silicon Graphics' soundfiler to convert this from an 11025 kHz PCM stereo file to an 8 kHz monaural .AU file for optimal transmission.

Substantial data space was being wasted by repeated constant references to the profile (.INI) file name. I moved this string to the string table in the resource file and changed the code that loads and saves the global configuration to load it once into a string on the stack and reference that temporary copy in all the [Read|Write]PrivateProfile... calls that follow.

Added a new MAKEBIN.BAT file in the home directory which builds the binary release archive. Now that the release includes more than the .EXE and .HLP file, something more archival than my fallible memory is needed to make sure

5 October 1995

Remade all screen shots for help file, the addition of the help buttons required updating all the dialogue bitmaps.

Added logic to the invocations of PGP in FRAME.C and DIALOGS.C to first try to use the SFPGP.PIF file from the Speak Freely release directory (obtained with GetModuleFileName) and then, if that fails, fall back to call on PGP counting on path search to find it. Going through the PIF allows the user to override the default modes for a WinExec call to a DOS program such as PGP, in particular, to run it in a window, which is much less disruptive of the user's equanimity than blasting out to a DOS prompt.

6 October 1995

Feature release 5.3.

30 October 1995

All of the Look Who's Listening functionality is working, at least if you don't push it into reentrancy into Winsock by trying one of the LWL dialogues while sending or receiving sound. I'll have to go back and review the appropriate locks to keep from befuddling Winsock with actual multitasking. Essentially all the code is in the new module **lwl.c**.

7 November 1995

Added support for RTP-compatible LPC compression (the Xerox PARC algorithm developed by Ron Frederick). This algorithm does a *lot* of floating point computation (forget it if you don't have a math coprocessor), and it sometimes mangles sound, especially if you drive the audio input into clipping or have a high-pitched voice. But when it works, it achieves better than 12 to 1 compression, and allows running over 9600 baud lines. The LPC code is in a new **lpc** subdirectory.

13 November 1995

Added a first cut "broadcast" facility to permit transmission of material to multiple hosts (over a suitably fast,

probably local network) without the need to install multicast. The facility is relatively crude but should be adequate for uses applications such as broadcasting meetings across a local network.

The site performing the broadcast simply checks Connection/Broadcast. Any audio which arrives while Broadcast is checked is sent to every connected host. All input events are ignored in connection windows while a broadcast is in progress, and remotely initiated connections will not time out during a broadcast. A user can subscribe to a broadcast from a given host by initiating a connection to it and sending a short burst of sound (a second's worth, say). This opens a connection on the broadcasting host to which the broadcast will be sent. A remote host can unsubscribe from the broadcast by sending a similar short burst of sound any time after 10 seconds into the broadcast; the site's connection on the broadcasting host will be closed 10 seconds later. The 10 second delay is to prevent toggling of the broadcast state due to multiple packets being received from the remote site. Whilst broadcasting, the application title indicates "- Broadcasting" and the cursor is always the ear when over a connection window. When broadcasting is toggled off, all connection windows are marked as not being transmitted to and remotely-opened connections resume the timeout process.

Using short bursts of sound to subscribe and unsubscribe is ugly but it gets the job done. Once we have a proper RTP packet exchange delimiting the connection, it can be replaced.

14 November 1995

If somebody is already blasting sound at us when Speak Freely is launched, it got all befuddled due to packets arriving before initialisation was fully complete. I changed the WM_CREATE logic in FRAME.C to not enable input on the socket until initialisation is entirely done.

16 November 1995

Feature release 5.5

22 November 1995

People seem to get floating divide by zero errors if they try to use LPC compression on Windows 95. I added a call to _control87() in the initialisation code to disable all floating point error interrupts. This should allow the LPC code to just bumble along with infinities and NANs like it does on Unix, which doesn't seem to do any harm (my suspicion is that this happens only when the LPC code is fed dead silence). This will have no effect anywhere else, since floating point is used only in the LPC code.

I made GSM compression the default out of the box. I'm deeply weary of explaining how to enable GSM compression to hundreds of people a day who can't be bothered to read the help file.

The Phonebook/Search host didn't default to lwl.fourmilab.ch unless you'd previously made a directory listing. Fixed.

The Phonebook/Search box wasn't quite wide enough to hold the longest line of a typical server message and didn't have a horizontal scroll bar. This caused perplexed people to send hundreds of E-mails when they couldn't access the truncated URL the LWL server published. I made the box a few characters wider and enable horizontal scrolling.

There was also some ragged logic in the default server for publishing directory entries. Fixed to correctly default to lwl.fourmilab.ch.

Update release 5.5a.

28 November 1995

Integrated server-side support for the "show your face" feature. The new file FACE.C contains a dialogue that

allows the user to designate a 256 colour .BMP file (the format is verified) as his or her face image and a function invoked from FRAME.C that delivers blocks of the image as requested by a remote host. Processing of face image requests occurs before audio output is acquired (but after creating a connection), so half-duplex systems can still transfer face images while sending audio.

30 November 1995

Integrated client-side handler for "show your face". Face data packets are assembled in FACE.C into an in-memory bitmap in the connection structure. If a complete bitmap is available, the WM_PAINT handler in CONNECT.C for the connection window displays the bitmap instead of the usual status information. When a bitmap is displayed in the connection window, transmit state is indicated by preceding the host name with a small ASCII-art arrow.

After adding hundreds of lines of bullshit Windows code trying to swap the palette intelligently when two face images are simultaneously displayed on a colour-mapped display, I decided to exercise that time-proven prerogative of the Windows developer and just give up. The vast majority of users won't connect to more than one person at a time. I fixed it (with even more bullshit code) so that the active window is always shown with the correct palette and inactive windows with the default palette. If you this this is easy to fix, baby, just go and try it before you write me some smart-ass E-mail. Hint: everything you read in the Windows API documentation about palettes is a lie or worse when you start to talk about MDI child windows. There's a sample application on the Developer CD which claims to do this, but a glance at it leads me to estimate at least a week to integrate and test all the crap they went through trying to do what, on any vaguely competently designed windowing system, should be essentially transparent to the application. Users of high-colour and true-colour display boards will be blithely unaware of any problem with multiple simultaneous face images.

1 December 1995

Mycal (mycal@monitor.net) reported that messages using simple (2X) compression were played back at twice normal speed by the answering machine. Fixed.

Feature release 5.6.

20 December 1995

Added logic in CONNECT.C to set outputSocketBusy if the send() or sendto() returns less than the number of bytes we attempted to write. The WSAEWOULDBLOCK error status is treated as a truncated buffer and sets outputSocketBusy. All of this is disabled if the new workaround "Disable Output Overflow Recovery" is set just, as always, in case.

When outputSocketBusy is set, we're guaranteed by the Winsock spec that we'll receive a the FD_WRITE notification we requested in the call on WSAAsyncSelect for the socket. Right. So in the timer, should we discover the socket has become unblocked for output and the fink didn't tell us, clear outputSocketBusy so things don't hang up.

To avoid output overruns, I changed the logic that responds to face image requests to ignore requests received while audio output is active. This should keep face data from pushing an 14.4 modem connection that is barely keeping up with GSM over the edge. The transmission of the face will be resumed by the timeout on the receiving end when the audio transmission is done with no harm done. I also tweaked the timeouts so they're less likely to collide with one another.

23 December 1995

Face bitmap exchange seems to hold the potential for bad medicine when stirred in the pot with multicasting. When sending to a multicast port, CONNECT.C now never offers a face to the subscribers, and face requests received from multicast ports (shouldn't happen, but who knows: it's Windows!) are ignored and the face retrieval

status set to Abandoned.

Strengthened the .BMP file format verification in FACE.C. We now verify that the bitmap has one plane and <= 256 colours, as must be the case for any compliant bitmap.

Added more stringent verification of the format of received bitmaps in FACE.C before passing them on to CONNECT.C to display. This gives us some protection against rogues who let bogus bitmaps through, and weird errors in transmission of the the bitmap that corrupt it.

31 December 1996

Added an explicit +armor=off to all invocations of PGP to guarantee the session key is encrypted in binary mode even if the user has modified the PGP configuration file to make ASCII armour the default.

16 January 1996

Integrated the VOX, Break-In, VOX GSM compression, and anti-hangup code from Dave Hawkes (daveh@cadlink.co.uk). In the process of testing the integrated version, I made the following (perhaps temporary) changes:

* For some reason, the fix that places the read-only data in ULAW.C in the code segment causes Speak Freely to crash with a GPF on the first reference to the tables in CONNECT.C, but only when I compile in DEBUG mode. I also needed to include the definition of the new tag CONST_DATA to be able to recompile the ADPCM and LPC libraries, which contain references to the ULAW.C tables. As a stopgap, to get things running, I changed the definition of CONST_DATA back to FAR. I'll look at this in more detail when I get a chance and see if I can't get that data back in the code segment.

* The anti-hangup code which calls DefaultMessageLoop() every time a network packet or wave audio input buffer arrives causes unacceptable break-ups of sound on my 486/50; apparently PeekMessage takes too long even in the normal case. I modified MessageLoop() and DefaultMessageLoop() in NETFONE.C to save the time (GetTickCount()) of each pass through the message loop and only run the PeekMessage loop if 10 milliseconds or more (constrained, of course, by the fundamental resolution of the timer) have elapsed. I'm hoping this will run the risk of the delay only when an actual message loop backlog occurs, which runs the graver risk of a lock-up. I've verified that the PeekMessage loop only rarely runs on my machine (usually when I block the window by moving it or pulling down menus), but since I still cannot reproduce the actual lock-up, even when I run my machine at 25 MHz, I can't verify the other side of the equation: whether the lock up is still avoided.

* I changed the "VOX" menu item to "Voice Activation" to avoid jargon which might befuddle the radio-naïve.

* Changed the title of the "VOX" monitor dialogue to "VOX Monitor". The dialogue isn't wide enough to avoid the "VOX" abbreviation, but adding "Monitor" makes it look a little less stark. It might make sense to go to a horizontal meter to justify more room for a longer title.

Some people would like to be able to launch Speak Freely from another application, pointing it a given host with various preset options on the connection. Writing an .SFX file naming the host and specifying the options, then invoking SPEAKFRE.EXE with the .SFX file on the command line permits this, but the .SFX file had to specify both the host name and IP address, forcing the calling application to look up the host name. I modified the newConnection() code in FRAME.C to automatically look up the IP address if an .SFX file specifies only a host name.

A typo in the Connection/Save code could have led to nugatory void entries in the .SFX file. Fixed.

17 January 1996

Added a visual indication when VOX is squelching transmission. The ear cursor now changes to an ear with a big X through it (that's the best I can think of right now, but it's better than no indication at all). This makes it a lot easier to evaluate the effect of VOX speed, especially if you don't have a local machine to run tests.

Figured what was wrong with the ULAW.C data in the code segment trick. Apparently CONNECT.C and the libraries didn't get recompiled after the definition was changed to CONST_DATA, and continued to reference the Ulaw tables as FAR (as they must). I changed ULAW.C to explicitly place the tables in the code segment, but continue to reference them with a FAR declaration in ULAW.H.

19 January 1996

As suggested by Enoch Wexler (wexler@datasrv.co.il), I added the ability to send and receive Show Your Face images in GIF as well as BMP format. GIFs offer substantial compression compared to even the compressed variants of BMP, which reduces the time it takes to transfer a face image and the likelihood of disrupting audio transmission in the process. Received GIF files are converted in-memory to BMP format by the new module GIFTOBMP which is based on the NETPBM utility GIFTOPNM. GIFTOPNM is copyright 1990, 1991, 1993, by David Koblas (koblas@netcom.com), who notes:

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. This software is provided "as is" without express or implied warranty.

GIF file decompression requires substantial storage for the LZW decompression buffers and colour map tables. I modified the decompression code to move all large buffers to a dynamically allocated global storage block to avoid overlow of DGROUP and/or the inclusion of static storage which would block execution of multiple instances.

The face image selection dialogue in FACE.C was modified to allow selection of GIF images as well as BMP files.

Added a new VOX menu item which calls a new function in VOX.C, vox_reset_parameters() to restore all the VOX level adjustment parameters to their original defaults. I also added a Reset button to the VOX Monitor dialogue which does the same thing. In the process, I rearranged the contents of the Monitor dialogue to create enough room to spell out its title.

Added a new workaround that totally disables the DefaultMessageLoop lockup-prevention mechanism. While I think my 10 millisecond trigger based on GetTickCount() should be enough, this provides an escape hatch in case it isn't.

23 January 1996

Based on reports that receiving a Ring message can screw up the sound card (for example, muting the microphone), I demoted the previous default call on waveOutSetVolume() in SPEAKER.C which attempts to set maximum output volume when a ring is received to a Workaround which is off by default, "Set Maximum Volume on Ring". Only in the world of Windows would you suppose that something as innocent as a volume control would conceal sharp edges and booby traps awaiting the unsuspecting developer.

Based on input from Dave Hawkes, I revised the DefaultMessageLoop code once again. This time it keeps track of the last time the program potentially yielded control to another application (by doing a PeekMessage with the PM_NOYIELD and PM_NOREMOVE flags in the main message loop in NETFONE.C, which seems to return fast enough to avoid pauses, and saving the GetTickCount() value if there is no message in the queue). Then, if DefaultMessageLoop() discovers 350 milliseconds or more have elapsed since the last yield, flushes the message queue using PeekMessage(), which will allow other applications to gain control of the CPU.

Dave also pointed out that my pointy-headed code that changes the cursor when VOX muting occurs changed the cursor even if it was outside the window. Fixed.

25 January 1996

The workaround that disables the DefaultMessageLoop() insurance did not actually turn off all traces of the code--the PeekMessage in the main message loop in NETFONE.C still remained. Since I'm sure this will cause unspeakable horrors when it triggers some booby trap Billy-boy has hidden in one of his existing products or is in store for us in the future, I made sure it's disabled when DefaultMessageLoop() is turned off.

6 February 1996

Made the inclusion of encryption conditional on the tag CRYPTO being defined in NETFONE.H. If CRYPTO is not defined, the version number in the About dialogue will have a suffix of " (no crypto)" and the IDEA patent notice will be replaced by an explanation of where to obtain a version including full encryption. The "Encryption" box in the Options/Connection dialogue will contain a more detailed explanation of the no-crypto edition. The Options/Create Key menu item is disabled in no-crypto builds. What's the rationale for this? Simple: a number of CD-ROM publishers and sound card manufacturers are interested in distributing Speak Freely. But since many will be shipping from the U.S. and other countries which attempt to restrict the export of "munitions" like Speak Freely they're afraid, and rightly so, that putting Speak Freely in the box might result in an all-expenses paid extended vacation at Club Fed. The non-crypto version allows them to include Speak Freely without such worries. Once a user has installed Speak Freely and is ready to start using encryption, they can simply follow the instructions in the dialogue boxes (and, soon, the help file) and download a full-encryption version from a site in a country which does not restrict cryptographic software. The non-crypto version can also be posted on bulletin board and commercial online services without risking government-initiated unpleasantness.

Note that undefining CRYPTO does not just block access to encryption and decryption; it totally removes the code from the program--the encryption libraries are never referenced and therefore not included by the linker in the executable. Thus there is no risk of a non-crypto build being deemed a munition. The only "cryppish" code that remains is MD5, and it is widely used (for example, in the export edition of Netscape) in non-encryption roles. In non-crypto Speak Freely, the RTP SSRC, timestamp, and packet sequence numbers are generated directly with MD5 rather than the somewhat more random idearand() used in crypto builds. Since we're not going to encrypt the RTP packets anyway, this doesn't compromise anything.

As the first step in integrating the RTP support code from the Unix version, replaced RTPACKET.C with the fully-functional one and verified the new rtcp_make_sdes() and rtcp_make_bye() didn't break LWL support.

Here's where we stand at the end of first day of the campaign to integrate RTP and VAT support into Speak Freely for Windows. The two procotol translation modules, RTPACKET.C and VATPKT.C and all their support files have been included in the program and fixed to compile without errors or warnings. As noted above, the new RTPACKET.C continues to generate valid packets for the LWL server. Sending both VAT and RTP protocol works for all compression modes, testing with VAT on another machine. VAT correctly recognises the VAT ID and RTCP SDES message we send on the control channel.

7 February 1996

Trying to integrate the LIBDES encryption package need for VAT and RTP encryption blew the data segment, so it's time to run another sweep for excess baryon particles. Using the same CONSTANT_DATA trick as in ULAW.H, I moved the large constant tables in LIBDES\FCRYPT.C and DES\DES.C into the code segment. Result: still over the brink.

The biggest memory hog, LPC\LPC.C, was unfortunately not so easily fixed, since its four large floating point analysis vectors are read/write and cannot be hidden in the code segment. I integrated a modified version of the LPC.C from NeVoT, in which the state of the decoder is in a dynamically allocated buffer. I obtain this buffer with GlobalAlloc, getting it out of the static data segment. This of course required FARs all over the place in LPC\LPC.C, but it did the trick. This will also allow, if we decide it's worth doing, maintaining a separate LPC state for each inbound connection.

It was intensely irritating to have to constantly answer E-mail from people who tried to build Speak Freely from source code but whose Winsocks not only didn't support multicast, their WINSOCK.H didn't even include the definitions for multicast. I fixed FRAME.C, DIALOGS.C, and NETFONE.H to, if IP_MAX_MEMBERSHIPS isn't defined, silently delete multicast from the build. If the user's Winsock doesn't define the variables we need to generate the code, there's no way he's going to be able to use the feature anyway. The Unix version uses the same trick to adapt to pre-multicast sockets implementations.

Swept through the program and added the fProtocol flag to all places Speak Freely protocol packets are generated. This flag helps receivers distinguish Speak Freely packets from VAT and RTP messages.

Went a long way toward implementing DES encryption of outbound RTP and VAT packets. It pretty much works--I'll make the final round of tests when DES works in both directions and I can verify correct operation in both directions.

8 February 1996

I modified code in CONNECT.C and FRAME.C to zero the SDES resend timer when the transmit protocol or encryption key is changed. This causes an immediate resend of the SDES/VAT ID in the new mode, which will help the receiver to "sync up" with the change.

I discovered that the clever way I integrated VAT and RTP encryption into sendpkt() in CONNECT.C had completely screwed up encryption for Speak Freely protocol. In the process of fixing this, I cleaned up some of the rather tangled logic in that function.

Added code to the Options/Connection dialogue to disable the fields and captions for IDEA, PGP, and Key file compression if the protocol is RTP or VAT. These protocols currently specify only DES as a standard mode.

Set the "talk spurt" flag for the first packet of a sound file.

Added VAT packet translation to sound file output in CONNECT.C.

Under certain circumstances, sending a sound file to a connection after sending a ring would just re-send the ring. Fixed in FRAME.C.

Disabled direct modem connections. This feature, which fell into the trapdoor called Windows serial port (8250) support, compounded the incoming E-mail pain due to idiots confusing direct dial-up modem connections with SLIP/PPP Internet access. Besides, since Serial I/O is near the top of the Redmond Kiddies' "API of the Year" list, the time it takes to debug it on all the ratty drivers out there exceeds the product life cycle. Users who wish to use Speak Freely as a phone scrambler on direct calls should establish a peer-to-peer TCP/IP connection and use Speak Freely in network mode. Since it will probably take Billy's bozos 20 or 30 years to debug Windows to Windows TCP/IP links, the fact that they'll blame their screwups on other vulnerable applications as well as Speak Freely will deflect a significant percentage of flames, albeit not to the flamers responsible for the mess in the first place.

9 February 1996

Voice activation didn't work with RTP and VAT protocols because load_vox_type_params() in VOX.C didn't know about the new packet sizes used by those protocols. Now it does. Independently, LPC and VOX don't seem to be getting along very well together, regardless of protocol. I'll have to look into this later on.

Coming to terms with the fact that I'll be chasing "bugs" in this program as long as there are Kode Kiddies in Redmond, I integrated the hex dump module from the Unix version. It lets me dump packets on the debug stream to see where Winsock wants to go today.

Transmission of non-encrypted VAT and RTP packets now seems to be working. That's not to say that DES

encryption doesn't work, just that I haven't tested it yet. The initial tests of bouncing VAT messages off the echo server failed due to byte order dependencies in VATPKT.C. These are now fixed; the changes must now be integrated into the Unix stream to handle little-endian boxes.

10 February 1996

Added code to the WM_DESTROY handler in CONNECT.C to transmit an RTP or VAT BYE message to indicate the user has closed the connection. In the process, I added a new sendSessionControl() function which is used by both this logic and the periodic RTCP/VAT ID transmission code in the timer.

Discovered that the sockets were getting closed in WM_CLOSE rather than WM_DESTROY, which kept the BYE transmission from working. I moved the socket close to after the BYE is sent in WM_CLOSE.

11 February 1996

Well, I think I finally found out why weird things occasionally happened when you quit Speak Freely with one or more connection windows open. This is really getting too depressing to document, but here we go. Windows, with its unerring instinct for doing things in the most idiotic way possible, sends a WM_DESTROY to the application's outer window procedure, and then *later* sends WM_DESTROY to each of the child windows. Suppose one or more of those child windows need to do something--send a BYE message, for example--using one of the resources that get freed by the application's outermost WM_DESTROY? Blooie. But don't think you can get away with just sending WM_DESTROY to each of the child windows: nopey, nopey, no. If you try that you fall into the toilet because calling WM_DESTROY doesn't actually make them go away, and as a result they get destroyed twice and all kinds of other horrors ensue. So, once again we are forced into subterfuge by the quintessential inelegance of Windows. We dig out of this particular hole by sending a custom WM_CLEAN_UP_YOUR_ACT message to the child window to tell it we're terminating. The child will then do its regular WM_DESTROY cleanup, including releasing the client data pointer and zeroing the window word to it. When the actual WM_DESTROY arrives, it will discover the client data pointer is zero and avoid executing the cleanup twice.

12 February 1996

Integrated the new LWL\LWL.C library from the Unix version, which allows a separate decoder state for each receiver and contains numerous fixes for subtle coder gotchas such as dividing by zero if total silence is received. This, of course, ran squarely into one of the innumerable floating point code generation bugs in Visual C++ 1.5 (they call it "Visual" because only if you're seeing things would you confuse it for a production compiler for floating-point intensive code). After a fine afternoon of trying various compiler options and workarounds, I found a combination of restructuring of the loop which caused the "Stack overfl" (a very Redmond kind of error message, don't you think?) and optimisation options which got around the error. Until the next "improvement" of the compiler, I'm sure.

Found another pair of missing htons() in the LPC packet handlers in VATPKT.C and RTPACKET.C, when stuffing the decoded length into the first two bytes of the sound buffer.

Memo to file. Windows' real-time response is so pitiful that machines which are perfectly adequate to run Speak Freely protocol in all compression and encryption modes (a 486/50, for example) can't cope with the smaller packet sizes used by RTP and VAT, particularly on reception. If you want to talk to somebody who can only sent RTP or VAT, you'd better make sure you have enough Intel inside to cope with Billy Boy's idea of process switch latency.

13 February 1996

Integrated a fix to rtpout() in RTPACKET.C from the Unix version. The packet length for outbound RTP ADPCM packets was 2 bytes short, which caused "gravelly" speech and horrible ticking when encryption was enabled.

It's possible for the predicted value in the ADPCM coder (ADPCM/ADPCM_U.C) to exceed the range of a signed

16 bit linear sample. Clamping code limits the range when this happens, but needed to declare the unclamped sample as a long rather than int to work on a 16 bit architecture. Fixed.

After many, many hours of painful, unremunerated toil I finally figured out what was causing the Debug kernel warnings and fatal errors due to bad pointers at the time we call WSACleanup at application termination time. The essential clue was that it only happens if one or more connection windows have been created by the receipt of a packet from a remote site not already connected. If all connections were created locally it never happened. And, of course, this only happened under the Winsock supplied with Sun PC-NFS 5.1--the problem never occurred under any circumstances on other Winsocks I've tried.

I'm sure by now you will be shocked and stunned to learn that Sun NFS 5.1 doesn't correctly implement the WSAAsyncGetHostByAddr function. Oh, you can make the call, and you even get back a valid host name. But doing so plants a time bomb which will kill you (at least under the debug kernel) much, much later when you call WSACleanup() right before exiting the program. At that time, depending on where the random pointer inside their so-called WSHELPER points, you get either two invalid global pointer errors or a fatal error due to an object usage count underflow in the (bogus) global block. If the waNetSynchronousGetHostnameAction is set, we eschew the asynchronous request and make the user wait for a blocking gethostbyaddr() which has the merit, at least, of not blowing us away at program termination time.

waNetSynchronousGetHostnameAction is set, in turn, based on the workaround waNetSynchronousGetHostname, which can take on the values 0, 1, and 2. If 2, the default, asynchronous host name retrieval is disabled if the Winsock identifies itself in the szDescription field of the WSAData structure returned by WSAStartup() as "Sun Select PC-NFS Windows Sockets Implementation". This automatic selection can be overridden by the user explicitly checking or unchecking the Options / Workarounds / Network / Get &Host Name Synchronously menu item. Thereafter, the user's selection will be used regardless of the identity the Winsock reports.

In the process of adding profile variable support for the above workaround, I observed the number of workarounds was about to exceed the rstring() cache in the name of the workarounds section remained. Since, unlike the developers of Microsoft tools, I do not feed off human suffering and take joy in setting booby traps, I modified all the profile read and write code to copy the section name to a stack string variable rather than rely on the pointer within rstring()'s retrieval area to remain valid while all variables in the section are accessed.

14 February 1996

Integrated the new RTPACKET.C, RTPACKET.H, and DESKEY.C from the Unix version. These include the facilities we'll need for parsing SDES packets, recognising BYEs, and creating RTP keys from key strings compliant with RFC 1890.f

Adjusted packet sizes returned by inputSampleCount() (FRAME.C) for VAT to the maximum permitted within both the experience base of VAT and the 512 byte guaranteed MTU of Winsock.

Integrated a fix from the Unix version to guarantee (in our context) that pad bytes added to VAT and RTP packets are zeroed.

Modified makeVATid() in VATPKT.C to, as VAT does, prefer the user's full name to the E-mail address if both are available.

Added recognition of RTP and VAT SDES/ID packets in FRAME.C. The title of the connection window will now show the user's name, if supplied by the sender. The changeAudioState() function in CONNECT.C also uses the user name, if available, in preference to the host name when it updates the window title to indicate transmit state.

RTP and VAT BYE/DONE packets now cause the receive protocol to be reset to PROTOCOL_UNKNOWN. This expedites recognition of a new protocol if the sender switches on the fly. Changed in FRAME.C.

Integrated generation of RFC 1890 RTP key and separate old-protocol VAT key. I've still to integrate automatic

protocol and key sensing.

Added code to encrypt outbound RTP and VAT packets with the appropriate key. The inbound side remains to be done.

Nailed another encryption packet size rounding error in CONNECT.C, this time affecting ADPCM encoded outbound packets in VAT protocol.

15 February 1995

I remembered that there was one more place the PC-NFS 5.1 WSAAsyncGetHostByAddr() bug could stab us in the back--in the case where the user enters a numeric IP address in the Connection/New dialogue. I added a gethostbyaddr() alternative to this call if waNetSynchronousGetHostnameAction is set.

Finished integrating automatic protocol sensing and key selection for encrypted inbound RTP and VAT packets. The logic was a little less tangled than in the Unix version since we process control and data packets in different callback functions.

Did a non-CRYPTO build to make sure all the RTP and VAT changes didn't break something or suck in verboten bits. Sure enough, all of DESKEY.C needed #ifdef CRYPTO, as well as the encryption code in CONNECT.C's sendSessionControl(). Fixed.

Implemented the guts of the local loopback facility--it works, but tuning and a nice user interface remain to be done. Why local loopback? So users can debug their audio hardware problems before venturing onto the net, which will be one of the steps in the "beginner's guide to Speak Freely" I'll get around to writing one of these days. Eventually there will be a Help menu item which creates a local echo connection, but for the moment you activate such a connection by making a new connection to "localhost" (or, if your Winsock doesn't know localhost from Casper the Friendly Ghost, 127.0.0.1). Any packets you send are saved in memory until the end of your transmission and then returned, after a short delay, as if echoed by a remote site. This is, then, an echo server that doesn't use the network. Not only does it let users experiment with audio hardware locally, it allows isolating network-induced problems from those which inhere in the CPU or audio hardware.

16 February 1996

Modified changeAudioState to invalidate the connection window without erasing the background. This makes it quicker to repaint the "Transmitting" or blank status when the audio state changes.

Added support for Speak Freely SDES messages on the control port. When a Speak Freely connection is open, RTCP SDES messages with a protocol ID of 1 (the old RTP, used by no application I know of) are sent on the control port. These messages allow unambiguous recognition of Speak Freely protocol, transfer of user information, and disconnect notification.

The connection window paint code in CONNECT.C now displays the current sending protocol, user name, and E-mail address of the connected user. The connection window is resized depending on the number of lines currently displayed.

17 February 1996

Added the ability to specify a port number for a connection. This required changes all over the place:

> A port number can now be entered in the Connection/New dialogue after the host name or IP number, delimited by a slash.
>
> The port number (even if standard) is saved in an .SFX file by Connection/Save / Save As.

A port number, if specified, is restored when a connection file is loaded. If no port number appears in the file, the default of 2074 is used.

The "Connect" button in the Look Who's Listening dialogue now passes both the IP address and port number, separated by a slash, as the known host argment to newConnection in FRAME.C, which was modified to recognise that syntax.

FRAME.C now maintains a list of auxiliary receive sockets, asList. When input arrives, a new function, findPort() searches the list to identify, from the socket number, which port the input arrived from. This is used, if we're creating a new connection based on the input, to set the port to which we'll respond.

CONNECT.C now uses the "port" field in the connection structure as the port to which messages are sent rather than the canned value of 2074.

findClientByHost() in FRAME.C now considers two connections identical only if both the IP address and port numbers are identical.

When creating a new connection with a nonstandard port number, CONNECT.C calls the new function monitorPort() which creates an auxiliary socket pair for that port. If the port is already monitored, the reference count on the auxiliary socket is simply incremented.

When destroying a connection to a nonstandard port, CONNECT.C decrements the reference count on the auxiliary socket and if it's zero closes the socket pair.

Note that auxiliary sockets are not bound to a specific host; once a connection is established with a given port, connections from any host can be remotely initiated on that port. This means that if you want to accept connections on a given port as a matter of course, you can do so simply by opening a dummy connection (to a nonexistent address on your subnet, for example) with that port. I'll probably eventually add a separate dialogue that lets you specify ports to monitor automatically but for the moment this gets the job done. Most users will be specifying ports to connect to remote RTP and VAT conferences anyway, not accepting calls on nonstandard ports.

Guess what? Sun PC-NFS 5.1 Winsock will not only blow you away if you call WSAAsyncGetHostByAddr(), the blocking version, gethostbyaddr() has a bug in it as well--it forgets to null-terminate the host name in the h_name field, so if you retrieve a host with a name shorter than the last one part of the last host name still sticks out. Working around this by zeroing the host name after you retrieve it is a blatant violation of the Winsock spec which states (section 4.2.1) "The application must never attempt to modify this structure or to free any of its components.". I for one, am not going to add my name to the list of millions who ignore the Winsock spec, so there isn't a damned thing I can do this other than tell people to get a better Winsock. Fortunately, it's purely an ugliness that doesn't do any damage since we're just retrieving the host name to display in the connection window.

Oops! In Speak Freely protocol, control channel messages aren't supposed to be encrypted but they were. Fixed.

What the world needs now, is lots more workarounds, they're the only thing that drive the bugs to ground.... So, some more anticipatory retaliation: the following are available on the new Options/Workarounds/Protocol submenu. All are, of course, saved in the .INI file and otherwise treated as respectable citizens.

No Speak Freely Heartbeat

Disable the periodic Speak Freely protocol heartbeat on the control channel. This is primarily intended as a last resort if the (less than 1%) added bandwidth saturates a close to the edge connection, and also in case the control channel packets awake something horrid lurking on the next higher channel.

Large RTP Protocol Packets

Uses Speak Freely's preferred packet sizes for GSM and LPC compression rather than those typically sent by RTP programs. Most RTP programs were developed on fast workstations with high bandwidth network connectivity. Speak Freely users generally have slower machines and network links which benefit from

larger packets. Try this if the person you're talking to reports halting audio in RTP protocol.

Disable VAT Protocol Detection

VAT protocol will never be automatically selected as a result of receiving a message on the control channel which resembles a VAT control message. Enable this if you never receive VAT protocol messages and are annoyed at how long it takes to identify the protocol of encrypted RTP messages.

Disable RTP Protocol Detection

RTP protocol will never be automatically selected as a result of receiving a message on the control channel which resembles a RTP control message. Enable this if you never receive RTP protocol messages and are annoyed at how long it takes to identify the protocol of encrypted VAT messages.

No Encryption of RTP Control Packets

RTP control packets can, according to the standard, be sent either encrypted or in the clear. Most RTP programs I've encountered encrypt their control packets, so this is the default Speak Freely sends (it accepts both encrypted and clear packets). If you set this workaround, control packets are sent in the clear.

19 February 1996

After further deliberations, I decided not to implement automatic protocol switching to the protocol received from the active window, although much of the infrastructure to do so is in place. The reason is that simply adding the new dimension of multiple protocols has the potential for inducing further confusion among users who don't understand the distinction between the compression mode received and that used in transmission. Trying to explain all the possible conditions one could get into with automatic protocol switching is probably futile. I may eventually put in a warning that pops up if the user tries to transmit to a connection which has sent us packets in a different protocol than the current transmit protocol. Protocol mismatch is never a problem when communicating with other copies of Speak Freely, since it auto-senses the protocol. Since initially relatively few users will be talking to other programs, those cutting-edge users are probably best encouraged to operate in "manual transmission" mode to avoid confusion.

The gimmick that forces immediate transmission of the identity message on the control channel (rather than waiting for the next timer interval) wasn't doing so when the protocol is set to Speak Freely. Fixed in FRAME.C.

Drat! When I added the port number criterion to decide whether a connection was already open, I forgot to handle local loopback. Fixed.

Added direct pointers from the Help menu to the FAQ and Mailing List sections of the Help file, and to create local loopback connection directly.

Direct access to local loopback required a tweak in CONNECT.C to not attempt to turn the loopback IP address into a host name.

20 February 1996

sendSessionControl() in CONNECT.C wasn't incrementing packetsSent for the extended status dialogue. Fixed.

loop_sendto() in LOOPBACK.C wasn't returning SOCKET_ERROR and setting the last error code as it should if it can't allocate the loopback buffer. Fixed.

To eliminate the choppiness that afflicted local loopback, particularly with the small packets sent by VAT protocol, I modified loopback replay to adopt a strategy of keeping the output queue stuffed with packets up to a limit of 10, and refilling the queue to that length every 10 milliseconds (Hah!! More like when Windows gets around to us.) rather than attempting to time each packet to a time resolution Windows just can't handle. (If I used the multimedia timer, it probably could, but that requires interrupt code call-backs into a DLL and imposes restrictions on what we can do from the call-back that Speak Freely couldn't live with.)

Naturally, this straightforward approach walked squarely into the jaws of disaster. The message loop insurance code was causing the timer code to be re-entered while it was playing back loopback packets, setting off a spectacular riot of recursion. I disabled the message loop insurance for loopback packet playback. Since we're strictly controlling the rate of packet arrival from loopback and the length of the stream is limited anyway, we don't really need the message loop insurance, which is intended to keep packets arriving from the network from hanging us.

21 February 1996

I added some code to the MM_WIM_DATA message handler in FRAME.C to soften the impact of the anti-lockup code on outbound audio quality. If a machine is right on the ragged edge of being able to compress in real time (a 486/50 sending GSM, for example), occasional Windows-induced delays will trigger the anti-lockup code and cause a sound packet to be dropped. I added code that allows recovery from one re-entry to the message handler by saving the packet and processing it immediately after the already-underway packet. Re-entries while an already saved packet awaits processing continue to discard packets.

The anti-lockup code in MM_WIM_DATA and socketInput did not increment the appropriate PacketLost counters. Fixed.

Added an item to the Help menu that points people directly to the echo server topic in the help file.

Building on Dave Hawkes' insight that the Windows wave audio output pause and restart could be used to implement a de-jittering replay delay with no buffering logic within Speak Freely, I implemented a first cut at de-jittering. Any input packet from the network which causes us to acquire audio output is considered the start of a "talk spurt". (Once we've transitioned to RTP, we can use the packet header bit for this, but we have to get there somehow.) When such a packet is received, if the jitterBuf has a nonzero replay delay in milliseconds, a timer with that expiration is launched to trigger the replay and wave audio output is paused. Wave audio output is restarted when the timer expires, or if the number of packets queued for replay exceeeds half the number of messages in the input queue (detected in SPEAKER.C). A new Options/Jitter Compensation menu item allows specifying the initial delay for a talk spurt. The longer the delay, the greater the suppression of jitter, but at the cost of a greater time parallax between the reception of the packet and its being played on the speaker.

When shutting down audio input, the final partial packet of sound before the shutdown could be lost. Fixing this little buglet naturally required massive changes to how audio input is torn down, since Windows likes to return packets from the queue in any old order at shutdown time, but imposes on the application a rigid order in which the API must be called. The terminateWaveInput() function in FRAME.C now actually does no such thing. In fact, it just resets audio input, causing any partial packet and the rest of the input queue to be returned to the message loop. Code for the MM_WIM_DATA message now processes any partial packets, padding them if necessary to the length prescribed by the protocol (with the correct pad depending on whether audio is 8 or 16 bits--gosh this is fun!) and sends any non-zero-length packets. If termination is underway, packets are unprepared and released, and an allocated packet counter decremented. When that counter goes to zero, wave audio is finally actually closed.

The above fix of course broke how Options/Break Input manages the transition between input and output mode for half-duplex audio hardware. Fixed (I think, pending reports to the contrary from the field).

22 February 1995

One more tweak to Break Input--if inputPaused is set, the WM_MIM_DATA handler in FRAME.C now immediately discards any partial packets that are returned during input termination. This speeds up the transition to playing the packets arriving from the socket.

26 February 1996

Port numbers greater than 32767 were not accepted in Connection/New due to being scanned as a signed short rather

than unsigned. Fixed.

The supposedly private bits used by the answering machine to mark the start of a transmission conflicted with the fProtocol flag bit, resulting in each Speak Freely protocol packet being considered the start of a separate message by the answering machine. Fixed.

29 February 1996

Dodging another intracardial dagger from our south-of-the-equator purveyors of what purport to be WINSOCK drivers introduces another layer of unnecessary and unwarranted complexity. The WINSOCK spec allows mutant windsuckers to abort any call that "re-enters" WINSOCK with a WSAEINPROGRESS call. Fine: what would you expect from the "vision of the future is a reboot in the face forever" people? But could you imagine, even in your wildest fantasy, that the most innocent of all socket calls, the one which *places* a socket in non-blocking mode, could *itself* blow off if a so-called blocking call (and many calls so-deemed have no non-blocking variants) is in progress? So, I turned the code that sends the heartbeat to the Look Who's Listening server inside out to cope with this crap, and thereby avoid the dreaded "Operation already in progress" puke-o-rama which, on some WINSOCKs poisons all future network accesses. I am sure this will have ugly consequences on other buggy platforms which will become apparent in the weeks and months to come.

Failure to look up the host name corresponding to an IP address after a connection was made displayed an error dialogue. Little did I know that 95% of all Windows 95 users do not have a valid domain name server configured, and each and every one of them E-mail me when this message appears. Warning message deleted; don't keep those cards and letters coming.

1 March 1996

Integrated the changes to VATPKT.C from the Unix version to recognise IDLIST packets as valid to to provide, in the future, the ability to include a conference ID in the packets we send.

Integrated the fixes from the Unix version into FRAME.C to recognise VAT IDLIST (3) packets and correctly parse the user names therein. This allows us to connect in VAT protocol to CU-SeeMe reflectors. As part of this change, the user name field (uname) in the connection structure was made a dynamically allocated buffer to permit long lists of participants in a conference.

I modified the connection creation logic in controlInput() in FRAME.C to never create a new VAT protocol connection unless an ID (1) message is received. This keeps IDLISTS (3) which rain in from conferences you've just left from re-opening the conference connection. Also, it was inelegant to open a connection based on a VAT BYE from the blue. A remote VAT connection will be opened only upon the receipt of an unencrypted ID packet. Note that we still have a problem with VAT conference Lazarus connections which result from VAT packets which arrive on the data port and are mistaken for pre-6.0 Speak Freely sound packets--they won't be played, but they'll still open the connection. This will go away once we've gotten everybody on 6.0 and require control port session control unless a special workaround is set to communicate with older versions.

Added logic in the connection window WM_PAINT handler in CONNECT.C to distinguish a VAT IDLIST (and one of these days, a multi-party RTCP SDES) from a simple ID and list the users in the conference one one per line.

After an afternoon of flailing around that sacrified the requisite number of neurons, I finally figured out a way to handle a user quitting the program while audio output is active which does not lead to sudden death. This involves the usual flags, mushy timers, countdowns and activity tests which Windows requires to do even the simplest things, and is much too depressing to discuss here. If you must see it, start at the WM_CLOSE handler in FRAME.C and follow the trail of slime through the rest of that file.

Echo, voice on demand, and reflector servers all have a tendency to create "Lazarus connections" which, seconds after you close them, pop back into existence when a packet comes back from the other end. To prevent this, I added a special anti-Lazarus mechanism in FRAME.C and CONNECT.C. When the user closes a connection window, the

WM_CLEAN_UP_YOUR_ACT message handler in CONNECT.C saves the IP address of the host in a new global Lazarus and sets the timeout counter LazarusLong to LazarusLength (15 seconds as presently configured), which is decremented by the main one-second timer in FRAME.C. If a packet arrives from the last connection window to be closed while LazarusLong has not yet counted down to zero, is it discarded by code in socketInput and controlInput in FRAME.C before it causes the connection to be reestablished. This provides a "decent interval" to allow postmortem packets arriving from the remote host to be discarded without re-opening the connection window.

To avoid the ignominy of shipping a release containing OutputDebugString diagnostic output, I added an updated version of the check for debug output in a production build that's used in Home Planet. The new version provides a primate-readable description of the error that points to the offending line and doesn't interfere with compilation of the rest of the file.

2 March 1996

Fixed a place in isHalfDuplex() in FRAME.C where in the case of an error in the process of determining whether audio is half duplex, the wave format buffer could be freed twice.

For some reason, trying to compile the program on a Pentium with twice as much free RAM as the 486, the resource compiler dies in the middle of windowsx.h with "Out of far heap". I excluded this file from resource compiler builds, and the little pointy head now deigns to work.

The VAT IDLIST packet parsing in FRAME.C's controlInput() had an off-by-one error when the name string length was a multiple of 4 and the terminating '\0' fell into the next 4 byte segment. Fixed.

One final gratuitous Gatesian gutshot this fine Saturday night--Virtual (if you confuse it for a real compiler, you're seeing things) C 1.52c generates bad code for the VAT IDLIST packet parser in controlInput() (FRAME.C) when full optimisation is selected. (I'm sure, gentle reader, this will surprise you, having come this far with me down the rathole.) So, I #pragma'ed off all optimisation in that function, wishing there were were some way I could #fragma the "making it all too much" crowd spewing their ghastly gigabytes into an industry I was once proud to be a part of.

6.0-Alpha 4 prerelease.

3 March 1996

Integrated lots of fixes all over the place for 32 bit compile problems. I made these changes in a very conservative manner--what the compiler sees when compiling the 16 bit version should be identical to the code before the fixes were installed.

4 March 1996

People were having so much trouble getting the automatic adaptive VOX to work that I disabled it and replaced it with a manual set-the-level yourself mechanism. The VOX Monitor dialogue now allows you move the red threshold indicator with the scroll bar with complete freedom, while (if audio input is live) showing the VU meter as before. Slow, Medium, and Fast continue to regulate the number of samples of silence which must be seen before transmission mutes. There is no need with a manual VOX adjustment for a Reset facility, so the menu item and button in the monitor dialogue for that function were removed. I also disabled VOX GSM compression mode, since I'm afraid explaining its interaction with manual VOX would only confuse people. All the code is still there for adaptive VOX and VOX GSM compression--if you compile with VOX_GSM defined, it will all come back.

Just as I suspected, the workaround for Trumpet Winsock's WSAEINPROGRESS bugs in contacting the Look Who's Listening server walked right into the jaws of another flaky Winsock--this time the one that comes with Windows 95. You apparently can't count on it to always notify you when a socket is closed, even if you've requested such notification via WSAAsyncSelect(). This led to timeout warning messages, attempts to make socket calls on already-closed sockets, and other horrors. Throwing up my virtual hands in disgust, I ripped out all message-based event sequencing of the LWL socket in FRAME.C and replaced it with timer logic. This is a crazy

way of doing what should be trivial, but it's the only way to guarantee (to the extent one can ever use that word in conjunction with something associated with Windows) we won't be nailed by timing windows, lost notifications, or other flaky behaviour on the part of Winsock.

5 March 1996

Well, that didn't work on a production build under PC-NFS, because a blocking connect (to a slow-to-respond LWL server) could interfere with data transmission. So (and I have a very, very bad feeling about this), I made LWL transmission entirely non-blocking. When we receive the FD_CONNECT notification, that triggers the send() and sets the timer to close the socket 8 seconds later (since we don't dare count on linger mode to work properly). Wanna bet we need a timer to back up the FD_CONNECT notification in case Winsock forgets to send us one? We'll see.

Added some additional paranoia in LWL.C to make sure no traffic is sent to the LWL server while a non-blocking transmission is in progress.

6.0-Alpha 5 prerelease.

7 March 1996

It's reported that connecting to a VAT conference with 35 people active causes a "runtime error 202 at 0001:1E". I don't have the vaguest idea what is causing this, what the error number means, or even where the error message is coming from, and I can't reproduce it since I don't have access to multicast to get on such a conference. I am not going to let nonsense like this deny the 6.0 update to tens of thousands of users who will never go near anything like this. I just hammered a test in CONNECT.C that limits the number of participants displayed to 8 and puts an ellipsis at the end of the list if there are more than that.

Added code to multicastJoin() in FRAME.C to join or drop any auxiliary sockets to the current multicast list, as well as the default port sockets.

Modified monitorPort in CONNECT.C to call multicastJoin to drop and then reacquire the multicast memberships whenever a new auxiliary socket is created.

8 March 1996

Once again we see how the poor design of Windows turns what is conceptually an easy task into a snowdrift, nay polar caps, of tangled and tricky code that one is never really sure will work everywhere. This time it's double clicking in the connection window to begin continuous transmission (which a lot more people will be doing now that we have VOX and Break Input). Remember back on the 21st of February when I redid the logic of how audio input is switched off so as dodge various bullets in the multimedia complex? Well guess what...ever since then double clicking hasn't worked (I didn't notice this since I generally use the space bar). Now, the double click logic in CONNECT.C couldn't have been simpler or more compliant with the Windows interface guidelines--the second click merely extends the scope of what the first one does, and does so simply by not switching off input on the button up event following a double click. Ahhh, but recall that you can't just turn audio input on and off like a light switch. It *takes a while* for the input buffers to rattle through the message queue and all the assorted dust to settle. So when we received the double click and subsequent button up event, audio input was still in the process of being terminated (as a result of the first button up event), and that's no time to go and re-open it.

What to do? Well, here comes another brutal hack necessitated by irrational Microsoft design. When we process an WM_LBUTTONUP in CONNECT.C, we no longer close audio input to the connection. Instead, we set a *timer* (does this sound familiar?) running to expire GetDoubleClickTime() after the button was released. If we see a WM_LBUTTONDBLCLK an its subsequent WM_LBUTTONUP before the timer expires, we revoke the time with KillTimer() and leave audio input active. If the timer goes off without our having seen further mouse action, audio input to the connection is shut off at that time through the expedient of faking a space bar input to the message loop. This is spectacularly ugly but it gets the job done.