Reach Out and Touch Someone's PC: The Windows Telephony API

Charles Mirho
Andrew Raffman

Charles Mirho is a software consultant specializing in multimedia and telephony. He is currently working for Intel Corp. and can be reached on CompuServe at 70563,2671. Andrew Raffman is a multimedia programmer with the Visual Caching Operating System group at AT&T Microelectronics.

About two years ago Intel began preparing the spec for a line of PC telephone boards. Most business customers use phones connected to private branch exchange (PBX) switches. PBXs support numerous advanced features, but there is little standardization among vendors. Since the hardware lacks standards, Intel and Microsoft began designing a standard telephone interface for the Windows operating system.

The two companies had been independently investigating these standardization problems. In mid-1992, Intel completed a first draft of what would become TAPI, the Windows[1]

Telephony API. The specification evolved jointly from there. In December 1992, Microsoft held a summit at which vendors of telecommunications equipment, voice recognition products, applications—anyone active in the telecommunications market—provided input. In May 1993, the public got an introduction to the preliminary TAPI specification.

TAPI is a first step in the integration of advanced telephony with the powerful capabilities of PCs. Without TAPI, every telephone application is inextricably bound to specific hardware, fragmenting a potentially huge business market into small incompatible pockets. Not only can features vary between different vendors' PBXs, the features of the same PBX can vary across international boundaries.

Applications written for TAPI are, in theory, portable to any computer connected to any communication network. Writing portable TAPI applications is a major undertaking, mainly due to the lack of business telephony standards. Fortunately, the partitioning of application-level functionality and implementation detail has been cleanly defined in the TAPI spec.

The first TAPI-compliant hardware and software will be introduced in the coming months and Microsoft plans to include TAPI support in the next major release of Windows. A developer's kit for TAPI device drivers and applications (yes—the TDK!) is scheduled for November 1993, so you'll be able to write TAPI apps now for Windows 3.1 and Windows for Workgroups 3.11. (This article is based on the October Beta II release of the TDK. Some functionality and terminology is updated in the final release.)

Architecture

TAPI is a Windows Open Systems Architecture (WOSA) component. WOSA components are derived from an API that specifies the methods of requesting a service and a service provider interface (SPI) that specifies the interface to the device driver (see Figure 1). This design contrasts with the architecture used in other operating systems, where the device driver is exposed directly to applications.

TAPI.DLL provides a standard interface to applications. It also coordinates multiple applications with multiple drivers. Applications don't need to worry about which driver is associated with which hardware, unless they use non-standard extensions provided by some drivers. Drivers do not have to allocate contexts for multiple client applications, since they only interact with TAPI.DLL. This article is chiefly concerned with the application interface.

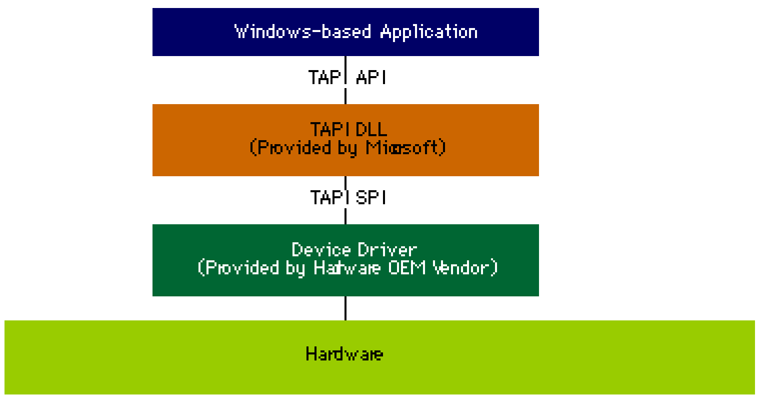
Windows-based Application
TAPI API
TAPI DLL (Provided by Microsoft)
TAPI SPI
Device Driver (Provided by Hardware OEM Vendor)
Hardware

Figure 1 TAPI Architecture

TAPI versus Media Streams APIs

Making or receiving phone calls involves two general steps: call setup and control, and data exchange. Call setup and control includes, for example, taking the phone off the hook, dialing a phone number, and answering an incoming call. These steps don't change much whether the call is voice, fax, modem, or video.

TAPI defines the means for applications to setup and control calls. It does not define a method for data exchange once the call is in progress. For instance, you can use TAPI to call your neighbor, but you can't use TAPI to talk to (or listen to) him or her when he or she answers the phone. TAPI by itself can place crank calls, and that's about all.

Why? Data exchange involves a media stream. The type of a media stream is determined by the type of data, giving rise to audio data streams, modem data streams, fax data streams, and so on. The interface between an application and a stream depends upon the type of data. Voice audio data requires a significantly different interface than modem data, for instance. Separate APIs for voice, modem, fax, and so on (see Figure 2) make TAPI easier to use.

To make a voice call, an application uses TAPI for call setup and control, then calls some other API to handle streaming. (Sometimes, the app uses no data API at all—for example, if the "data stream" is being handled by an external peripheral such as a telephone set.) The interface for modem streams is defined, for historical reasons, to resemble I/O to a serial port, where data transfer proceeds byte-by-byte. However, in some situations (such as file transfer) a more block-oriented API can yield performance advantages. A new block-oriented API could also work with TAPI. For example, block data transfers are used for disk I/O and in many high-speed transfer protocols.

So what are the APIs for voice, modem, and fax? The Windows wave audio API works well for voice data. Modem and fax APIs are not yet defined, although they are in the works and are planned for Chicago. While there is a de facto fax/modem API (using the serial port, the Hayes AT modem, and class 1 and 2 fax command sets), the standard is old and inadequate for advanced telephony environments. Also, the Hayes AT command set includes call control functions already found in TAPI.

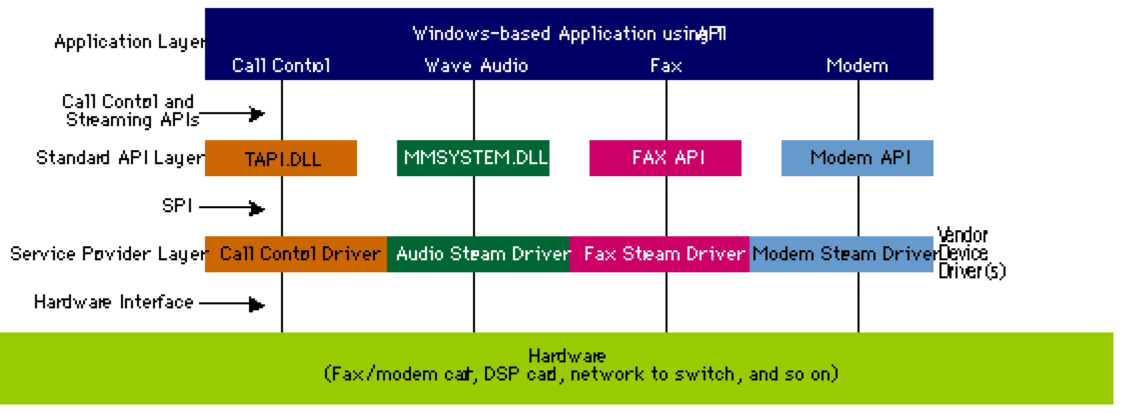

Figure 2 Multiple API's Are Used by a TAPI Application

Classes of TAPI Applications

Telephone programmers fall into two broad categories: those who want full control of the call, and those who just need simple features and can't be bothered with details. To support both, TAPI defines two interfaces, Full and Simple.

The Full interface includes functions and messages for all the details of making and managing a call, such as selecting a compatible line, monitoring call progress, and generating digits and tones. The Simple interface includes only three functions and one message. That allows the application to make and drop calls. There are no provisions for answering calls. (At the time this article was going to press, the TAPI terminology was still in flux. In upcoming TAPI documentation, "Windows TAPI" is synonymous with the Full interface referred to here, and Simple TAPI is known as "Assisted telephony"—Ed.)

If an application using Simple TAPI does not handle call setup and control, what does? A type of Full TAPI application, which I call a TAPI Server, does. Calls to Simple functions are forwarded to the TAPI Server, which uses the Full services to place and control the call (see Figure 3).

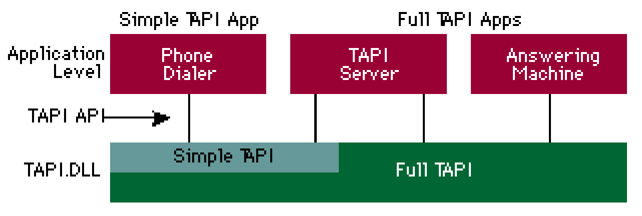

Figure 3 Simple Versus Full TAPI Applications

TAPI Concepts

Before taking the plunge into TAPI programming, an understanding of the central concepts is essential. The crucial concepts begin with logical devices, both lines and phones. Closely related to logical lines and phones are the concepts of addresses and calls.

The TAPI specification defines a line as any device that implements the line behavior of TAPI (swell, huh?). A logical line usually corresponds with the common perception of a physical phone line: a bundle of wires connecting the phone with some type of phone equipment, usually a switch. The TAPI definition is vague on this point for a reason. A line can do many different things, depending on the technology behind it. An ISDN line contains multiple channels, or information paths, for voice and data. The voice and data channels are so different in their behavior and features that they are properly modeled as two logical lines, even though they exist on the same physical set of wires. The TAPI model for lines is broad so that it covers LAN-based telephone servers and wireless networks.

TAPI defines a phone as anything that implements the phone behavior of TAPI. However, TAPI makes more specific requirements of phone devices than it does for line devices. In particular, a phone device will likely contain a hookswitch and transducer—that is, a means of audio input and output that can be switched on and off. A phone device may have several such devices attached to it, such as the traditional handset, speakerphone, and headset. A phone may provide volume control for the speakers and gain control for the microphone element. Virtually all phones employ some form of ringer, a means of notifying humans of incoming calls. Some digital phones use LCD displays to show call information and messages. For dialing, most phones contain buttons (you could imagine cases where a phone might not have buttons—voice-controlled phones might be one example). Many digital phones include lamps that flash on and off. Some advanced phones can be programmed remotely, and thus include instruction memories.

None of these features are required for logical phones under TAPI, although at least some of them must be present. The TAPI phone functions are oriented toward traditional analog and digital phone sets, although other devices could be supported with some ingenuity.

Addresses and Calls

TAPI refers to phone numbers as addresses. Those accustomed to a single phone number per line (the norm in residential homes) may have trouble with the concept of multiple addresses per line. It is not unusual in digital environments for a single phone line to be assigned multiple phone numbers (for example, one number for company internal calls, another for outside calls). Nor is it unusual for an address to handle multiple simultaneous calls. A call is defined simply as a connection between two addresses. Anyone familiar with call-waiting understands how two or more calls may be present on a single address at the same time. In call-waiting, an active call is interrupted by the arrival of a second call. The person with call-waiting flashes (presses and releases) the hookswitch to put the first call on hold and make the second call active.

Addresses are treated as static by TAPI. That is, a phone number never changes unless the phone company or switch administrator gets involved. Calls, however, are dynamic; they come and go with each conversation. Because of their dynamic nature, most of the interesting TAPI functions deal with calls.

Line-related Functions

TAPI defines over five dozen functions for working with logical lines. Explaining each function in detail is impossible in so short a space (the TAPI document uses around 120 pages for this purpose). Functions may, however, be grouped into broad categories (see Figures 4 and 5).

Figure 4 Line-related Functions (Includes Address and Call-related Functions)

Call Setup and Takedown Functions	**Assist in the process of dialing, answering, or dropping a phone call**
lineAccept	Closes an open line device
lineAnswer	Answers the offering call
lineClose	Closes an open line device
lineDial	Dials a number on a call
lineDrop	Drops or disconnects a call
lineInitialize	Initializes the application's use of the Telephony API DLL
lineMakeCall	Places a call on a line to the specified destination address
lineOpen	Opens a line specified by device ID and returns a line handle for the opened line
lineShutdown	Shuts down the application's usage of the line interface
lineDeallocateCall	Deallocates a call handle
Information Functions	Return information about the features or status of a line, address, or call
lineGetAddressCaps	Queries an address on a line to determine its telephony capabilities
lineGetAddressID	Returns the ID associated with an address
lineGetAddressStatus	Queries an address for its current status
lineGetCallInfo	Obtains static information about a call
lineGetCallStatus	Returns the current status of a call
lineGetConfRelatedCalls	Returns a list of calls that are part of a conference
lineGetDevCaps	Queries a line to determine its telephony capabilities
lineGetID	Returns a device ID for a device class associated with the specified line, address, or call
lineGetLineDevStatus	Queries an open line for its current status

lineGetNewCalls	Returns handles to calls on a line or address for which the application currently does not have handles
lineGetNumRings	Determine the number of times an inbound call on an address should ring prior to answering the call
lineGetRequest	Retrieves the next by-proxy request for the specified request mode
lineGetStatusMessages	Queries for which notification messages it is set to receive
Setup Functions	Manipulate the settings of a line, address, or call
lineSetAppSpecific	Sets the application-specific field of a call's call information record
lineSetCallParams	Sets the bearer mode and/or the rate parameters of an existing call
lineSetCallPrivilege	Sets the application's privileges to a call
lineSetMediaControl	Sets control actions on the media stream associated with the specified line, address, or call
lineSetNumRings	Sets the number of rings for an incoming call prior to answering the call
lineSetStatusMessages	Sets which status changes are reported on a line
lineSetTerminal	Sets the terminal to which information is to be routed
lineMonitorDigits	Sets the unbuffered detection of digits on the call
lineMonitorMedia	Sets the detection of media modes on a call
lineMonitorTone	Sets and disables the detection of inband tones on a call
Control Functions	Provide call control features
lineBlindTransfer	Performs a blind or single-step transfer of the call to the specified destination address
lineCompleteCall	Specifies how a call that could not be connected normally should be completed
lineCompleteTransfer	Completes the transfer of a call to the consultation call
lineForward	Forwards calls destined for an address on the line
lineHold	Places a call on hold
linePark	Parks a call
linePickup	Picks up a call
lineRedirect	Redirects an offering call to the specified destination address
lineSecureCall	Secures a call from any interruptions or interference that may affect its media stream
lineSetupTransfer	Initiates the transfer of a call
lineSwapHold	Swaps an active call with a call on consultation hold
lineUncompleteCall	Cancels a call completion request on a line
lineUnhold	Retrieves a held call
lineUnpark	Retrieves a parked call
lineGatherDigits	Initiates the buffered gathering of digits on a call
lineGenerateDigits	Initiates the generation of digits on a call as inband tones using the specified signaling mode
lineGenerateTone	Generates tones inband over a call
lineSendUserUserInfo	Sends user-to-user information to the remote party on a call
Conferencing Functions	Set up, control, and drop the parties of a conference call
lineAddToConference	Adds a call to a conference call
linePrepareAddToConference	Prepares a conference call for the addition of another party
lineRemoveFromConference	Removes a call from a conference
lineSetupConference	Sets up a conference call for the addition of the third party
Miscellaneous Functions	
lineDevSpecific	An extension mechanism to enable service providers to provide access to features not described in other lineDevSpecificFeature operations
lineHandoff	Used to give ownership of a call to another application
lineNegotiateAPIVersion	Negotiates an API version to use
lineNegotiateExtVersion	Negotiates an Extension version to use with a line device
lineRegisterRequestRecipient	Registers the invoking application as a recipient of requests for the specified request mode
lineTranslateAddress	Translates between an address in canonical format and its dialable address

Figure 5 Phone-related Functions

There are fewer functions for logical phones than for logical lines, reflecting TAPI's narrower model for logical phones

Initialization/Shutdown Functions	Responsible for initiating or closing the use of a phone device
phoneOpen	Opens a phone device
phoneClose	Closes an open phone device
phoneInitialize	Initializes the application's use of the Telephony API DLL for use of the phone abstraction
phoneShutdown	Shuts down the application's usage of TAPI's phone abstraction
Information Functions	Return data on the features and settings of a phone
phoneGetButtonInfo	Returns information about a button
phoneGetData	Uploads the information from a data buffer in the open phone device
phoneGetDevCaps	Queries a phone to determine its telephony capabilities
phoneGetDisplay	Returns the current contents of the phone display
phoneGetGain	Returns the gain setting of a phone's hookswitch microphone
phoneGetHookSwitch	Returns the current hook switch mode of an open phone device
phoneGetID	Returns a device ID for the device class associated with a phone
phoneGetLamp	Returns the current lamp mode of a lamp
phoneGetRing	Queries a phone for its current ring mode
phoneGetStatus	Queries a phone for its overall status
phoneGetStatusMessages	Returns the phone state changes which will generate a message to the application
phoneGetVolume	Returns the volume setting of a phone's hookswitch speaker
Setup Functions	Manipulate the settings of a phone
phoneSetButtonInfo	Sets information about a button
phoneSetData	Downloads the information in the buffer to the phone
phoneSetDisplay	Displays the specified string on the phone
phoneSetGain	Sets the gain of the hookswitch microphone
phoneSetHookSwitch	Sets the state of the phone's hookswitch

phoneSetLamp	Causes the lamp to be lit in the specified lamp mode
phoneSetRing	Rings the phone using the specified ring mode and volume
phoneSetStatusMessages	Monitors the phone for selected status events
phoneSetVolume	Sets the volume of the speaker component of the phone hookswitch
Miscellaneous Functions	
phoneDevSpecific	A general extension mechanism to the phone API
phoneNegotiateAPIVersion	Negotiates an API version to use for a phone
phoneNegotiateExtVersion	Negotiates an Extension version to use with a phone

Canonical Addresses and Dialable Addresses

Imagine that you're a traveling salesperson. On the road you spend a great deal of time on the phone keeping in touch with clients. You decide to simplify this process by keeping a list of contacts and their phone numbers in a database on your portable PC. When you call a client, you look up their name in the database, hit a button, and the PC automatically dials the number.

You need to enter each client's name and the number in the database. However, the appropriate number to dial depends upon both the client's location and your own. To dial an out-of-state client from home, you must use an area code. If you find yourself in the client's home state, the area code is unnecessary. Additionally, the dialed number depends upon the telephone system you are connected to. Dialing from a hotel or business phone probably requires an initial 9 to get an outside line. An ISDN network may have its own codes for initiating a call.

It would be unacceptable to require updates to the client's number each time you change geographic location. Better to let TAPI automatically figure out the correct number to dial.

TAPI achieves this through the use of canonical and dialable addresses. A canonical address fully describes all possible aspects of a telephone number, and is meant to be a universal identifier independent of calling location and access method. The canonical address is what goes into the database. Canonical addresses take the form of strings and have the format shown in Figure 6.

To make a phone call, a dialable address must be formed from the canonical address. A dialable address contains the necessary dialing and routing information to place a call on a specific line at a specific location. Dialable addresses have the format shown in Figure 7.

Figure 6 Canonical Address Format

+ CountryCode Delimiter [AreaCodeDelimiter] SubscriberNumber [| Subaddress] [^ Name] [CRLF ...]

Where fields between [ ] are optional

+	ASCII Hex (2B). Indicates that the number that follows uses the canonical format.
CountryCode	A variably sized string containing only the digits 0–9. It identifies the country in which the address is located.
Delimiter	Any character except 0–9 + | ^ CRLF. It is used to delimit the end of the CountryCode part and the beginning of the AreaCode or SubscriberNumber, and the AreaCode and SubscriberNumber parts of an address. I suggest – as a delimiter.
AreaCode	A variably sized string containing only the digits 0–9. Optional. AreaCode is the area code part of the address. If present, it must be followed by a delimiter.
SubscriberNumber	A variably sized string containing only the digits 0–9 as well as delimiters, but excluding + $ | ^ CRLF. Delimiters embedded in the SubscriberNumber are ignored.
|	ASCII Hex (7C). Optional. If present, the information following it up to the next + | ^ CRLF, or the end of the canonical address string is treated as subaddress information, as for an ISDN subaddress.
Subaddress	A variably sized string containing a subaddress. The string is delimited by + | ^ CRLF or the end of the address string. During dialing, subaddress information is passed to the remote party. It can be such things as an ISDN subaddress or an email address.
^	ASCII Hex (5E). Optional. If present, the information following it up to the next CRLF or the end of the canonical address string is treated as an ISDN name.
Name	A variably sized string treated as name information. Name is delimited by CRLF or the end of the canonical address string and may contain other delimiters. During dialing, name information is passed to the remote party.
CRLF	ASCII Hex (0D) followed by ASCII Hex (0A). Optional. If present, it indicates that another canonical number is following this one. It is used to separate multiple canonical addresses as part of a single address string for use with inverse multiplexing. Inverse multiplexing is typically used to allow multiple channels to be combined into a single call to increase the call's bandwidth. This technique is infrequently used, so you can disregard it for now.

Figure 7 Dialable Address Format

DialableNumber [| Subaddress] [^ Name][CRLF ...]

Where:

DialableNumber Digits and modifiers 0–9 A–D * # , ! W w P p T t @ $? ; delimited by | ^ CRLF or the end of the dialable address string.

Within the DialableNumber, note the following definitions:

0–9 A–D * #	ASCII characters corresponding to the DTMF and/or pulse digits. Although most telephones support only 12 DTMF tones in a 3x4 array, the DTMF standard supports a 4x4 array with 16 tones. The characters A–D correspond to the extra 4 DTMF tones.
!	ASCII Hex (21). Indicates that a flash is to be inserted in the dial string.
P p	ASCII Hex (50) or Hex (70). Indicates that pulse dialing is to be used for the digits following it.
T t	ASCII Hex (54) or Hex (74). Indicates that tone (DTMF) dialing is to be used for the digits following it.
,	ASCII Hex (27). Indicates that dialing is to be paused. The duration of a pause is device-specific and can be retrieved from the line's device capabilities. Multiple commas can be used to provide longer pauses.
W w	ASCII Hex (57) or Hex (77). An uppercase or lowercase W indicates that dialing should proceed only after a dial tone has been detected.
@	ASCII Hex (40). Indicates that dialing is to "wait for quiet answer" before dialing the remainder of the dialable address. This means to wait for at least one ringback tone followed by several seconds of silence.
$	ASCII Hex (24). It indicates that dialing the billing information is to wait for a "billing signal" (such as a credit card prompt tone).
?	ASCII Hex (3F). It indicates that the user is to be prompted before continuing with dialing. The provider does not actually do the prompting, but the presence of the ? forces the provider to reject the string as invalid, alerting the app to the need to break it into pieces and prompt the user in-between.
;	ASCII Hex (3B). If placed at the end of a partially specified dialable address string, it indicates that the dialable number information is incomplete and more address information will be provided later. ; is only allowed in the DialableNumber portion of an address.
|	ASCII Hex (7C). Optional. If present, the information following it up to the next + | ^ CRLF, or the end of

	the dialable address string is treated as subaddress information (as for an ISDN subaddress).
Subaddress	A variable sized string containing a subaddress. The string is delimited by the next + | ^ CRLF or the end of the address string. When dialing, subaddress information is passed to the remote party. It can be an ISDN subaddress, an email address, and so on.
^	ASCII Hex (5E). Optional. If present, the information following it up to the next CRLF or the end of the dialable address string is treated as an ISDN name.
Name	A variable sized string treated as name information. Name is delimited by CRLF or the end of the dialable address string. When dialing, name information is passed to the remote party.
CRLF	ASCII Hex (0D) followed by ASCII Hex (0A). Optional. If present, this indicates that another dialable number is following this one. It is used to separate multiple canonical addresses as part of a single address string (inverse multiplexing).

TAPI supplies the function lineTranslateAddress to convert a canonical address to a dialable address. The function takes a canonical address, a line identifier, and information about the caller's current location, and synthesizes a dialable address. Information about the caller's current geographic location is typically specified by running the Telephony control panel applet.

For example, a valid canonical address for Manhattan telephone information would be "+1-212-555-1212", where the first 1 is the country code for the USA and 212 is the Manhattan area code. If this number is passed to lineTranslateAddress, and if you are calling from a business phone in Manhattan, the function would return a number similar to "t9w5551212" (depending on the features of the line you're using). The "t" in the dial string says to use DTMF (Dual-Tone Multifrequency signals—Touch Tones, as the lay folk call them) dialing. It first dials a 9 and waits for a dial tone, then dials 555-1212. (It's not smart enough to substitute 411.) Notice that the 212 area code was removed by TAPI, since you're calling within the 212 area code.

The lineTranslateAddress function does not perform any call processing; it is strictly informational. To make the call, the application must call lineMakeCall or lineDial. The application is not required to call lineTranslateAddress to form dialable addresses; it may create its own as long as the address conforms to the format for dialable addresses.

Dialdemo

The simplest of the Simple functions is tapiRequestMakeCall, which places a voice call. Figures 8 and 9 show Dialdemo, a sample application that demonstrates tapiRequestMakeCall. The Window menu contains a single Dial option. This brings up a dialog box that requests a name and telephone number. When the OK is pressed, the program places the call.

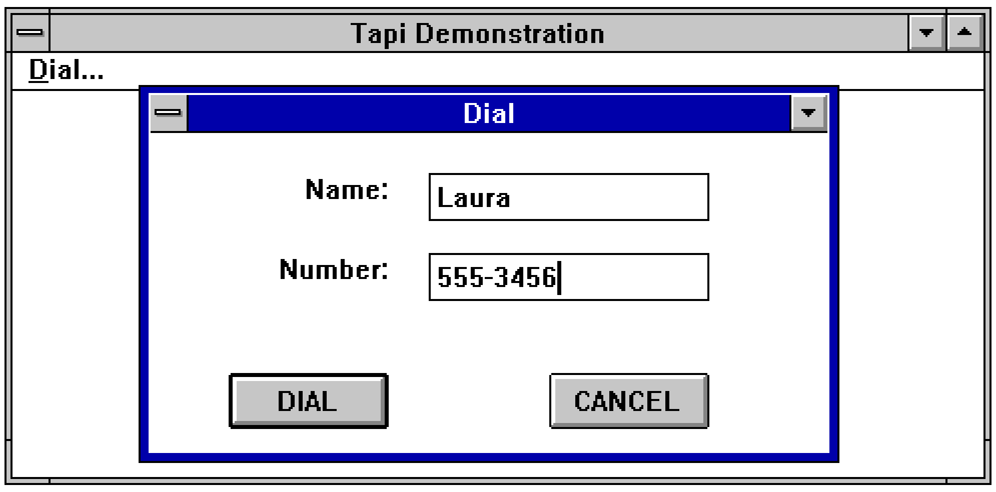

Figure 8 Dialdemo

Figure 9 Dialdemo

```
DIAL.H
#define IDM_DIAL    100

#define IDC_NAME    100
#define IDC_NUMBER    101

extern char szDialNumber[128];
extern char szDialName[128];

BOOL __export FAR PASCAL DialDialogProc(    HWND hWndDlg,
                                            WORD wMsgID,
                                            WORD wParam,
                                            LONG lParam );

TAPI.H
#define TAPI_REPLY 0

#define TAPIERR_CONNECTED           0
#define TAPIERR_NOREQUESTRECIPIENT  1
#define TAPIERR_REQUESTQUEUEFULL    2
#define TAPIERR_INVALDESTADDRESS    3

LONG tapiRequestMakeCall(   LPCSTR const lpszDestAddress,
                            LPCSTR const lpszAppName,
                            LPCSTR const lpszCalledParty,
                            LPCSTR const lpszComment );

LONG tapiRequestMediaCall(  HWND hWnd,
                            WORD wRequestId,
                            LPCSTR const lpszDeviceClass,
                            LPCSTR const lpDeviceID,
                            DWORD dwSize,
                            DWORD dwSecure,
                            LPCSTR const lpszDestAddress,
                            LPCSTR const lpszAppName,
                            LPCSTR const lpszCalledParty,
                            LPCSTR const lpszComment );

LONG tapiRequestDrop(HWND hWnd, WORD wRequestId);

DIAL.RC
#include "windows.h"

#include "dial.h"
```

```
MyMenu MENU
{
    MENUITEM "&Dial..."    IDM_DIAL
}


DialDialog DIALOG 10, 20, 170, 80
STYLE DS_MODALFRAME | WS_MINIMIZEBOX | WS_POPUP | WS_VISIBLE | WS_CAPTION |
    WS_SYSMENU
CAPTION "Dial"
BEGIN
    RTEXT           "Name:", -1,   10, 10, 50, 8
    EDITTEXT        IDC_NAME,      70, 10, 70, 12, ES_AUTOHSCROLL
    RTEXT           "Number:", -1, 10, 30, 50, 8
    EDITTEXT        IDC_NUMBER,    70, 30, 70, 12, ES_AUTOHSCROLL
    PUSHBUTTON      "DIAL",   IDOK,     20,  60, 40, 14
    PUSHBUTTON      "CANCEL", IDCANCEL, 100, 60, 40, 14
END
```

```
DIAL.C
#include "windows.h"
#include "dial.h"


char szDialNumber[128];
char szDialName[128];

/*********************************************************/
/* Dialog procedure to get a name & phone number */
BOOL __export FAR PASCAL DialDialogProc(    HWND hWndDlg,
                                            WORD wMsgID,
                                            WORD wParam,
                                            LONG lParam )
{
    switch (wMsgID)
    {
        case WM_INITDIALOG:
            SetFocus (GetDlgItem (hWndDlg, IDC_NAME));
            return FALSE;
        case WM_COMMAND:
            switch (wParam)
                {
                case IDOK:
                    GetDlgItemText(hWndDlg,IDC_NAME,  szDialName,  128);
                    GetDlgItemText(hWndDlg,IDC_NUMBER,szDialNumber,128);
                case IDCANCEL:
                    EndDialog (hWndDlg, wParam);
                    return TRUE;
                }
            break;
        }
    return FALSE;
}
```

```
DIALDEMO
OBJ=dialdemo.obj dial.obj
COPT=-c -AM -GAs -Od  -Zpei -W4 -I\l\tdk\inc
LIBS=libw mlibcew mmsystem tapi oldnames

all: dialdemo.exe

# Update the resource if necessary

dial.res: dial.rc dial.h
    rc -r dial.rc

# Update the object file if necessary

dialdemo.obj: dialdemo.c dialdemo.h dial.h
    cl $(COPT) dialdemo.c

dial.obj: dial.c dial.h
    cl $(COPT) dial.c

dialdemo.exe: $(OBJ) dialdemo.def dial.res
    link /NOD /CO $(OBJ),,, $(LIBS), dialdemo.def
    rc dial.res dialdemo.exe
```

```
DIALDEMO.DEF
NAME    DIALDEMO

DESCRIPTION 'A Simple TAPI Application'

EXETYPE WINDOWS

CODE    LOADONCALL DISCARDABLE PRELOAD
DATA    MOVEABLE MULTIPLE PRELOAD

HEAPSIZE    1024
STACKSIZE   8192

EXPORTS
```

```
DIALDEMO.H
LRESULT __export FAR PASCAL MainWndProc(    HWND hWnd,
                                            UINT wMsgID,
                                            WPARAM wParam,
                                            LPARAM lParam);

int MakeACall();
void perror(char *Error);

DIALDEMO.C
#define STRICT
#include <windows.h>
#include <windowsx.h>
#include <tapi.h>
#include "dialdemo.h"
#include "dial.h"

/* Global variables */
HINSTANCE    ghInstance;
HWND        ghWnd;

char szAppName[] = "TapiDemo";

/*********************************************************/
int PASCAL WinMain (    HINSTANCE    hInstance,
                        HINSTANCE    hPrevInstance,
                        LPSTR        lpszCmdLine,
                        int          nCmdShow    )
{
    WNDCLASS WndClass;
    MSG         msg;

    ghInstance=hInstance;

    /* Register class */
    if (!hPrevInstance)
    {
        WndClass.lpszClassName  = szAppName;
        WndClass.hInstance      = hInstance;
        WndClass.lpfnWndProc    = (WNDPROC)MainWndProc;
        WndClass.hCursor        = LoadCursor(NULL,IDC_ARROW);
        WndClass.hIcon          = LoadIcon(NULL,IDI_APPLICATION);
        WndClass.lpszMenuName   = "MyMenu";
        WndClass.hbrBackground  = (HBRUSH)(COLOR_WINDOW+1);
        WndClass.style          = CS_HREDRAW | CS_VREDRAW;
        WndClass.cbClsExtra     = 0;
        WndClass.cbWndExtra     = 0;

        RegisterClass(&WndClass);
    }

    /* Create main window */
    ghWnd = CreateWindow( szAppName, "Tapi Demonstration",
                          WS_OVERLAPPEDWINDOW,CW_USEDEFAULT, 0, CW_USEDEFAULT,
                          0, NULL, NULL, hInstance, 0L );

    ShowWindow(ghWnd, nCmdShow);

    /* Enter message loop */
    while(GetMessage(&msg,NULL,0,0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    return msg.wParam;
}

/*********************************************************/
LRESULT __export FAR PASCAL MainWndProc(    HWND    hWnd,
                                            UINT    wMsgID,
                                            WPARAM  wParam,
                                            LPARAM  lParam )
{
    switch(wMsgID)
    {   case WM_COMMAND:
            switch (wParam)
            {
                case IDM_DIAL:

                    /* Make a phone call */
                    if (MakeACall())
                    {
                        perror("Can't make call");
                    }
                    return 0L;
            }
            break;

        case WM_CLOSE:
            DestroyWindow(hWnd);
            return 0L;

        case WM_DESTROY:
            PostQuitMessage(0);
            return 0L;
    }
    return DefWindowProc(hWnd, wMsgID, wParam, lParam);
}

/*********************************************************/
/* Procedure to dial a phone call using TAPI */
```

```
int MakeACall()
{
    long tapiRetCode;

    /* Get name & number */
    if (DialogBox(ghInstance, "DialDialog", ghWnd, (DLGPROC)DialDialogProc) !=
        IDOK)
        return 0;

    tapiRetCode = tapiRequestMakeCall(szDialNumber,szAppName,szDialName,NULL);

    switch (tapiRetCode)
    {
        case 0:  perror("Call placed successfully");
                 break;
        case TAPIERR_NOREQUESTRECIPIENT:
                 perror("No call manager app running");
                 break;
        case TAPIERR_REQUESTQUEUEFULL:
                 perror("Request queue full; try again later");
                 break;
        case TAPIERR_INVALDESTADDRESS:
                 perror("Invalid destination address");
                 break;
        default: perror("Unrecognized error");
    }
    return (int)tapiRetCode;
}

void perror(char *Error)
{
    MessageBox(ghWnd, Error,    szAppName, MB_OK | MB_ICONEXCLAMATION);
}
```

Dialdemo uses two source files, DIALDEMO.C and DIAL.C. (To run this application, you'll need TAPI-compatible hardware, a service provider module, and the following modules from the TDK: DIALER.EXE, TAPI.DLL, TAPIEXE.EXE—a small program automatically launched and terminated by TAPI.DLL—and TELEPHON.INI.) Dialdemo is your basic Windows-based application: selecting Dial sends a WM_COMMAND message to MainWndProc with a wParam of IDM_DIAL. MainWndProc calls MakeACall, which puts up the dialog box. If the user hits Cancel, the DialogBox procedure in DIAL.C returns without action. Otherwise, tapiRequestMakeCall is called (see the MakeACall function in Figure 9). Each of the four arguments to tapiRequestMakeCall points to a NULL-terminated string. The first argument, lpszDestAddress, is required; it specifies the destination address of the called party, that is, the phone number to call. This address may be in canonical or dialable format. If the address is canonical, the TAPI Server translates it to a dialable address before placing the call. The other three arguments specify the name of the application making the call, the name of the called party, and a comment. Each of these arguments is optional, and NULL may be substituted if the information is unknown. I set the application name and called party, but left the comment NULL. If tapiRequestMakeCall returns 0, the call completed normally. Otherwise the return is one of the TAPIERR_XXX values specifying why the call failed. (For a list of TAPI return values, see TAPI.H in the TDK.) The TAPI Server controls the call from this point on.

Access to Media Streams

When programming with media streams, the first thing you need to know is how to associate a stream with a particular call. The tapiRequestMediaCall function allows the developer to specify the media device class and device ID to associate with a call. This ID is the same as the one used with the stream API.

Suppose you have been hired by ACME Telemarketing to develop a program to automatically dial up potential customers and deliver a short message advertising ACME's newest product. Playdemo, the next sample program, dials a number, waits for an answer, plays an audio WAV file over the phone line, and then hangs up. (Code for Playdemo can be found on any MSJ bulletin board—Ed.) It's based on Dialdemo with minor changes. The program first determines the wave device associated with the telephone, then it opens a WAV file and allocates data buffers. It places a call. Next it opens the wave device and plays the audio file. Finally it closes the wave device, hangs up the phone, and deallocates the data buffers.

A media device is defined by its device class and ID. The device class for wave audio devices is "wave". Determining the device ID is a little trickier. If your PC contains a Sound Blaster-style audio card and a TAPI-compatible telephone board, each will have its own device ID. If you use the wrong device ID with tapiRequestMediaCall, the result is a TAPI_REPLY message with lParam set to TAPIERR_DEVICEIDUNAVAIL, meaning the call failed. In addition, if you were to try to play the audio file to that device ID, the sound would stream to your local speakers (connected to your audio card), not the phone line. Therefore, it is imperative to choose the device ID associated with the telephone line.

Surprisingly, no simple method exists to determine this ID! In the sample programs I just hardcoded the ID in the header file along with the device class. A Full TAPI application can call lineGetID to return the correct device ID (we do this in the next example), but Simple TAPI is supposed to protect you from Full TAPI. Once the device ID is known, you can use the low-level audio functions to play the message. We'll come back to this shortly.

When Dial is selected, tapiRequestMediaCall places the call. This function takes 10 (count 'em) parameters. The first parameter specifies the window to which TAPI messages are routed. I set it to my main window. The second parameter is returned as wParam in all TAPI messages to MainWndProc. An application may use this parameter to differentiate among multiple calls. Since I'm placing only one call, I set this parameter to 0. The next two parameters specify the device class and device ID. Notice that lpszDeviceClass is a long pointer to a null-terminated string but lpDeviceId need not be zero-terminated. Why? The device ID may not be a string at all, but rather a short integer or long integer, or possibly a structure. The next parameter, dwSize, specifies the actual number of bytes in lpDeviceId. I set these to identify the appropriate wave output device. Parameter dwSecure requests that features like call-waiting not be allowed to disrupt the call. (The dwSecure flag prevents other phone switches from injecting unwanted tones into your data stream.) Values of 0 and 1 enable and disable this feature, respectively. The last four parameters, are identical to their tapiRequestMakeCall counterparts.

If the request is accepted by Playdemo, the function returns 0. At this point the application returns to Windows while the call completes. At some later time, the application receives a TAPI_REPLY message. lParam will specify one of a number of predefined TAPIERR_XXX values indicating the success or failure of the call. If all went well, lParam is set to TAPIERR_CONNECTED, indicating that the call completed and someone (or something) is waiting on the other end of the line. It's time to play our little advertisement.

The playSound function, implemented in PLAY.C, uses the Windows wave API to play a WAV file. This function takes as parameters the device ID of the wave device, the name of the WAV file to play, and the window handle to receive progress messages. playSound uses calls from the wave API to perform these tasks.

- Open the WAV file. The WAV file is read using the mmio_Xxx functions. These functions, which are part of the mmsystem library, are designed to allow easy navigation of RIFF files. WAV files are a subtype of RIFF (Resource Interchange File Format) files.
- Retrieve the format information for the audio data from the file.
- Allocate an audio data buffer and load the audio data from the file into the buffer.
- Create a wave header to send to the audio output device.
- Verify that the waveform output device specified by dw_devid can support the audio format.
- Open the waveform output device corresponding to dw_devid. The hWnd parameter is passed to waveOutOpen so that all messages

from the wave device will be sent to the main window procedure.

- Prepare the waveform header and send the header and audio data buffer to the wave output device.

For simplicity, playSound loads the entire audio data into a single data buffer. If the audio data will not fit into available physical memory, the function fails.

The audio buffer is queued for playing using waveOutWrite. From this point the wave output device plays the buffer through the telephone line in the background without any further intervention by the application. When it finishes playing the audio buffer, the wave device sends a MM_WOM_DONE message to MainWndProc. Upon receiving this message, it unprepares the wave header, closes the audio device, and frees the wave header and data buffer. It then disconnects the call using tapiRequestDrop. This function takes only two parameters, which must be the same hWnd and wRequestID passed originally to tapiRequestMediaCall.

At any point during the call the program may receive a TAPI_REPLY message with lParam set to TAPIERR_ DROPPED. This indicates that the call disconnected, either because the application called tapiRequestDrop, or because the person on the other end got tired of junk calls and hung up.

Full TAPI

Full TAPI is divided into three levels of service: basic, supplemental, and extended. Basic services include the minimal functionality required in all TAPI products. Supplemental services are optional functions for accessing advanced features. These include:

- Selection of call quality/protocol when a call is initiated or answered
- Ability to dynamically monitor a call to determine what type of data (voice, modem, fax, and so on) is being sent
- Ability to monitor and generate DTMF digits or other tones
- Call transfer, hold, conference, and so on

Extended services are specific to a particular service provider. TAPI does not specify what these services are, but does define the mechanism for calling them.

As a developer, you should remember that service providers may not support all or even any of the supplemental or extended services. To take advantage of these services, verify that they are supported through calls to lineGetDevCaps and phoneGetDevCaps. If you are marketing a program that requires certain supplemental or extended features, it would be wise to advertise that fact to alert your customers that they need a compatible service provider and hardware. Conversely, if you are in the market for TAPI hardware, you may want to compare different vendors' products on the basis of how far they go beyond the basic services.

Version Negotiation

Full TAPI applications use version negotiation to identify the versions of TAPI.DLL installed and the service provider. Negotiation anticipates future revisions to TAPI. Applications written for extended features of specific service providers must identify which revision of the provider is resident, so they know which extended features to use.

Let's assume an application is written to use the API in TAPI.DLL version 1.0, and that after thousands of users have acquired and installed the application, Microsoft releases TAPI.DLL version 1.1. This new TAPI.DLL includes extended data structures, flags, and messages not present in version 1.0. Since the application does not use the new enhancements, TAPI should not use them either. During version negotiation, the application and TAPI agree to use only structures, flags, and messages available in version 1.0.

Now let's assume an application supports integrated paging, where if you call someone from your PC but get no answer, you can click a button to page that person (via a PA system locally connected to his or her phone system). Integrated paging is a TAPI extension; it is not provided among either the basic or supplemental functions. Assume further that early versions of the service provider did not support integrated paging. In this case, the application must negotiate with the service provider to determine if integrated paging is supported. If not, the application knows to disallow this feature (perhaps by graying the corresponding check box).

Figure 10 demonstrates version negotiation. An application negotiates the API version with TAPI.DLL, as well as the version of the service provider. This code shows negotiation for the line functions only; the phone functions must be negotiated separately.

Figure 10 Version Negotiation

```
/* tapi.h */
typedef struct lineextensionid_tag {
  DWORD  dwExtensionID0;
  DWORD  dwExtensionID1;
  DWORD  dwExtensionID2;
  DWORD  dwExtensionID3;
} LINEEXTENSIONID, FAR *LPLINEEXTENSIONID;
o
o
o
/* application.h */
#define APIHIVERSION  0x00010014  /* 1.20 */
#define APILOWVERSION 0x00010000  /* 1.0 */
#define EXTHIVERSION  0x00010005  /* 1.5 */
#define EXTLOWVERSION 0x00000009  /* 0.9 */

/* application.c */
LINEEXTENSIONID ext_id, ext_ourid;
o
o
o
/* negotiate the line API version */
if (lineNegotiateAPIVersion(hApp, 1, APILOWVERSION, APIHIVERSION,
                            &dw_versionAPI, &ext_id)) < 0) {
     MessageBox (...);
     return -1;
} //End if (error negotiating API version)

/* no extensions until proven otherwise */
b_extensionsEnabled = FALSE;
/* can we use extended features? */
if (ext_id.dwExtensionID0 = = ext_ourid.dwExtensionID0 &&
    ext_id.dwExtensionID1 = = ext_ourid.dwExtensionID1 &&
    ext_id.dwExtensionID2 = = ext_ourid.dwExtensionID2 &&
    ext_id.dwExtensionID3 = = ext_ourid.dwExtensionID3) {
     /* negotiate extension version with service provider */
     if (lineNegotiateExtVersion(hApp, 1, dw_versionAPI,
                                 EXTLOWVERSION, EXTHIVERSION,
                                 &dw_versionExt)) = = 0) {
          b_extensionsEnabled = TRUE;
     } //End if (success negotiating extensions version)
} //End if (service provider supports our extended feature set)
```

The application first negotiates the version of the line API using lineNegotiateAPIVersion. Typically, this is the second TAPI function called. The first is lineInitialize, which returns an application handle (lineInitialize will be demonstrated later). lineNegotiateAPIVersion takes six parameters: the application handle, the ID of the line device, and the API version range supported by the application. In this case the application supports TAPI versions 1.0 through (currently nonexistent) 1.2 (see the definitions of APILOWVERSION and APIHIVERSION in Figure 10). The fifth parameter is a pointer to the negotiated version, determined by TAPI.DLL, and the sixth is a pointer to a LINEEXTENSIONID structure, which the service provider will fill with its unique extension ID. Different line devices can support different extensions, hence the need for the second parameter, the line device ID. If no extensions are supported for this device, the structure is filled with zeros.

The range of API versions supported by the application is defined by the range APILOWVERSION to APIHIVERSION. These constants are defined as longs. The low word of each long holds the minor version number; the high word holds the major version number. Thus

```
0x00010014
```

corresponds to a version of 1.20.

TAPI.DLL returns the negotiated version in dw_versionAPI. This value will be the highest value in the range APILOWVERSION to APIHIVERSION that TAPI.DLL supports. For example, if APILOWVERSION = 1.0 and APIHIVERSION = 1.20, and TAPI.DLL supports up to version 1.1, then TAPI.DLL returns 0x0001000A in dw_ versionAPI. The application must be careful not to use API features later than version 1.1. If TAPI.DLL is very old, it may not even support versions up to APILOWVERSION. For example, TAPI.DLL may only support versions 0.5 to 0.9. In this case lineNegotiateAPIVersion fails, and the user must install a later version of TAPI.DLL.

Finally, the service provider fills the ext_id structure with its own extension ID. Unless all four fields of the extension ID match those of the application’s extension ID exactly, the application cannot use extensions on that line. You must use the EXTIDGEN program that comes with the TDK to generate an extension ID.

Telephone Answering Machine

Now let’s go on to a more fun example, a telephone answering machine. To review how these work, for those of you who’ve been in Skylab since 1978, a telephone answering machine monitors the phone line for an incoming call. When a call arrives, the phone rings a preset number of times, and then the answering machine takes the phone off the hook. The answering machine plays out a prerecorded voice message, then begins recording whatever the caller wants to say. When the caller hangs up, the answering machine saves the message, if any, and cycles back to waiting for the next call. This behavior is normally implemented in hardwired circuits mounted in a housing that sits between the line and the phone. We’re going to implement the same behavior in software.

The answering machine (cleverly called TAM, for The Answering Machine) must first register to receive incoming voice calls. It can’t simply take control of the line as a normal answering machine would; other applications may be using it. By registering, the application asks TAPI for permission to answer incoming voice calls. This does not mean the application will actually get any calls; that depends on what other applications are running and what priorities they have in the answer sequence. The answer sequence is stored in TELEPHON.INI. For example, two applications may express an interest in interactive voice calls (a typical phone conversation), using the following entry:

```
[HandoffPriorities]
interactivevoice = app1, app2
```

app1 has the highest priority over incoming voice calls, since it appears first in the list. Of course, you usually don’t know whether a call is actually voice until after you answer it, as everyone who’s gotten a fax screech can attest. Anyway, the call must be answered and someone, either a human or machine, must make noise on the line before you can determine what type of call it is. This type of a call is referred to as the call’s media mode or media type. TAPI products will typically include enough intelligence to distinguish a screeching fax or a warbling modem from a human voice. So what do you do about answering calls before their type is known?

This is where TAPI gets clever. Applications can register to receive calls of an unknown media type. These applications are in effect telling TAPI, “I want to answer the phone when it rings, and I don’t care if it’s a digital fax or an analog fax or a modem or a human or whatever, just give me the darn call and I’ll decide what to do with it.”

```
[HandoffPriorities]
unknown = app1, app2
```

When the phone rings, the running application with the highest priority to answer calls of unknown type will be notified first. Some time later, after the call has been answered, either the application or the service provider will determine what media type it is (this is called classifying the call). If the service provider makes the determination, TAPI passes this information on to the application that answered the call, and the application decides what to do next. If the application can handle calls of that media type, it proceeds with whatever it’s been doing (playing out a voice message, in the case of the answering machine). Otherwise, the application does a handoff.

When the application discovers a call it can’t handle, it passes it to the next application in line. This scheme is probably the best one possible, given that the type of a call may be unknown until after the call is answered, and someone has to answer it. The answering machine includes code to hand off a call if it is not automated voice (automated voice involves prerecorded voice information, like the answering machine prompt).

Where does the call go once it is handed off? TAPI gives the call to the highest priority application that has registered for calls of that media type. If no application is running that’s willing to accept the call, then the application that answered the call is stuck with it. More than likely the app will simply hang up.

Implementing the State Machine

The answering machine code implements the state machine shown in Figure 11. TAM.EXE is divided into three source files (see Figure 12). TAM.C implements the state machine. MESSAGE.C contains the code for recording incoming voice messages. PLAY.C plays audio files. The window for the example contains only a single menu item, Wait for Call, which you select when you want the answering machine to answer calls.

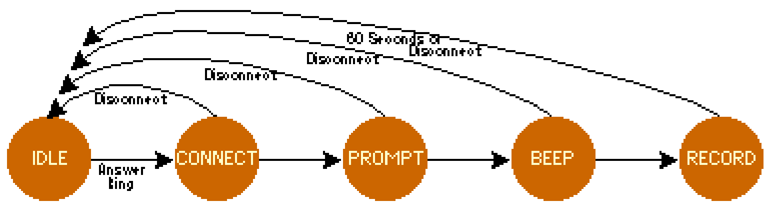

Figure 11 State Diagram for TAM.EXE

Figure 12 TAM

```
TAM
OBJ=tam.obj message.obj play.obj async.obj
COPT=-c -AM -GAs -Od  -Zpei -W4 -I\l\tdk\inc
LIBS=libw mlibcew mmsystem tapi oldnames

all: tam.exe

# Update the resource if necessary
tam.res: tam.rc tam.h
    rc -r tam.rc

# Update the object file if necessary
```

```
tam.obj: tam.c tam.h
    cl $(COPT) tam.c

message.obj: message.c tam.h
    cl $(COPT) message.c

play.obj: play.c
    cl $(COPT) play.c

async.obj: async.c
    cl $(COPT) async.c

tam.exe: $(OBJ) tam.def tam.res
    link /NOD /CO $(OBJ),,, $(LIBS), tam.def
    rc tam.res
```

TAM.H

```
#define IDM_AUTOANSWER      WM_USER+0

extern  HWAVE playSound(DWORD, char *, HWND);
extern  HWAVEIN recordMessage(DWORD dw_devid);
extern  int saveMessage(void);

extern  int __pascal WinMain(unsigned int hInstance,unsigned int hPrevInstance,
                             char __far *lpCmdLine,int nCmdShow);
extern  int InitApplication(unsigned int hInstance);
extern  int InitInstance(unsigned int hInstance,int nCmdShow);
extern  long __pascal __far __export MainWndProc(unsigned int hWnd,
                                                 unsigned int message,
                                                 unsigned short wParam,
                                                 long lParam);
extern  void __pascal __far __export lineCallback(unsigned long dwDevice,
                             unsigned long dwMsg,unsigned long dwCallbackInstance,
                             unsigned long dwParam1,unsigned long dwParam2,
                             unsigned long dwParam3);
extern  int doInitStuff (HLINEAPP *ph_app, DWORD *pdw_ver, DWORD *pdw_id);
extern HANDLE getMessageFile ();
extern int initOpenFile (char _far *szFileName, char _far *szTitle,
                         char _far *szExt, char _far *szFilter);
extern void lineError (LONG lrc);
extern LONG getWaveDeviceID (HLINE h_line);
extern void doneCall (void);
```

TAM.DEF

```
; module-definition file for tam
NAME         Test          ; application's module name
DESCRIPTION  'Sample TAPI Application'
EXETYPE      WINDOWS       ; required for all Windows applications
STUB         'WINSTUB.EXE' ; Generates error message if application
                           ; is run without Windows
;CODE can be moved in memory and discarded/reloaded
CODE  PRELOAD MOVEABLE DISCARDABLE
;DATA must be MULTIPLE if program can be invoked more than once
DATA  PRELOAD MOVEABLE MULTIPLE
HEAPSIZE     1024
STACKSIZE    30000
```

TAM.RC

```
#include <windows.h>
#include "tam.h"

tamMenu MENU
BEGIN
    MENUITEM "&Wait for call", IDM_AUTOANSWER
END
```

TAM.C

```
#include <windows.h>
#include <windowsx.h>
#include <mmsystem.h>
#include <tapi.h>
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include "tam.h"
#include "async.h"

/* flags */
BOOL b_playingmessage;
BOOL b_playingbeep;
BOOL b_recording;

/* handles to wave devices */
HWAVE h_wave;
HWAVEIN h_wavein;

HANDLE hInst;        //current instance
HANDLE h_wndmain;    //handle to main window

int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow);
BOOL InitApplication(HANDLE hInstance);
BOOL InitInstance(HANDLE hInstance, int nCmdShow);
long FAR PASCAL MainWndProc(HWND hWnd, unsigned message, WORD wParam, LONG lParam);

HLINE hLine=NULL;
DWORD dw_waveid=0;
HCALL h_call;           /* handle to active call */
```

```
/* async ids */
LONG l_ansid;           /* answer id */
LONG l_dropid;          /* drop id */

/***************************************************************************
    FUNCTION: WinMain(HANDLE, HANDLE, LPSTR, int)
    PURPOSE: calls initialization function, processes message loop
***************************************************************************/
int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow)
{
MSG msg;

    lpCmdLine;

    if (!hPrevInstance)                  //Other instances of app running?
    if (!InitApplication(hInstance))     //Initialize shared things
        return (FALSE);                  //Exits if unable to initialize
    //Perform initializations that apply to a specific instance
    if (!InitInstance(hInstance, nCmdShow))
        return (FALSE);
    //Acquire and dispatch messages until a WM_QUIT message is received.
    while (GetMessage(&msg,    //message structure
                      NULL,    //handle of window receiving the message
                      NULL,    //lowest message to examine
                      NULL))   //highest message to examine
    {
        TranslateMessage(&msg);    //Translates virtual key codes
        DispatchMessage(&msg);     //Dispatches message to window
    } //End while (not a quit message)
    return (msg.wParam);           //Returns the value from PostQuitMessage
} /* end function (WinMain) */
/***************************************************************************
    FUNCTION: InitApplication(HANDLE)
    PURPOSE: Initializes window data and registers window class
***************************************************************************/
BOOL InitApplication(HANDLE hInstance)
{
WNDCLASS  wc;

    //Fill in window class structure with parameters that describe the
    //main window.
    wc.style = NULL;                         //Class style(s).
    wc.lpfnWndProc = (WNDPROC)MainWndProc;  //Function to retrieve messages for
                                             //windows of this class.
    wc.cbClsExtra = 0;        //No per-class extra data.
    wc.cbWndExtra = 0;        //No per-window extra data.
    wc.hInstance = hInstance; //Application that owns the class.
    wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName =  "tamMenu";   //Name of menu resource in .RC file.
    wc.lpszClassName = "TAMWClass"; //Name used in call to CreateWindow.

    //Register the window class and return success/failure code.
    return (RegisterClass(&wc));
} //End function (InitApplication)
/***************************************************************************
    FUNCTION:  InitInstance(HANDLE, int)
    PURPOSE:  Saves instance handle and creates main window
***************************************************************************/
BOOL InitInstance(HANDLE hInstance, int nCmdShow)
{
HWND hWnd;

    //Save the instance handle in static variable, which will be used in
    //many subsequence calls from this application to Windows.
    hInst = hInstance;
    //Create a main window for this application instance.
    hWnd = h_wndmain = CreateWindow(
    "TAMWClass",                  // See RegisterClass() call.
    "Telephone Answering Machine",  // Text for window title bar.
    WS_OVERLAPPEDWINDOW,            // Window style.
    CW_USEDEFAULT,                  // Default horizontal position.
    CW_USEDEFAULT,                  // Default vertical position.
    CW_USEDEFAULT,                  // Default width.
    CW_USEDEFAULT,                  // Default height.
    NULL,                           // Overlapped windows have no parent.
    NULL,                           // Use the window class menu.
    hInstance,                      // This instance owns this window.
    NULL                            // Pointer not needed.
    );
    //If window could not be created, return "failure"
    if (!hWnd)
    return (FALSE);
    //Make the window visible; update its client area; and return "success"
    ShowWindow(hWnd, nCmdShow);  //Show the window
    UpdateWindow(hWnd);          //Sends WM_PAINT message

    return (TRUE);               //Returns the value from PostQuitMessage
} //End function (InitInstance)

/***************************************************************************
    FUNCTION: MainWndProc(HWND, unsigned, WORD, LONG)
    PURPOSE:  Processes messages
***************************************************************************/
long FAR PASCAL MainWndProc(HWND hWnd, unsigned message, WORD wParam, LONG lParam)
{
```

```
static HLINEAPP hApp=0;
static DWORD dwVerAPI;
static DWORD dwLineID;
HDC hDC;
PAINTSTRUCT ps;
LONG lrc;

    switch (message)
    {
     /* the window message function is notified when the buffer has played out */
        case MM_WOM_DONE:
            /* Unprepare the header */
            waveOutUnprepareHeader((HWAVEOUT)wParam,(LPWAVEHDR)lParam,
                                   sizeof(WAVEHDR));
            waveOutClose((HWAVEOUT)wParam); /* close wave device */
            GlobalFreePtr (((LPWAVEHDR)lParam)->lpData); /* free the wave data */
            GlobalFreePtr ((LPWAVEHDR)lParam); /* free the header */
            if (b_playingmessage) {
                b_playingmessage = FALSE;
                if (h_wave=playSound (dw_waveid,"beep.wav", h_wndmain)) /* beep! */
                    b_playingbeep = TRUE;
                break;
            }
            if (b_playingbeep) {
                b_playingbeep = FALSE;
                if (h_wavein=recordMessage (dw_waveid)) /* start recording msg */
                    b_recording = TRUE;
                break;
            }
            break;

        case MM_WIM_DATA:
            waveInUnprepareHeader (h_wavein, (LPWAVEHDR)lParam, sizeof(WAVEHDR));
            waveInClose (h_wavein); /* close wave device */
            /* drop the call */
            if (h_call) {
                if ((l_dropid = lineDrop (h_call, NULL, 0)) < 0) {
                    lineError (l_dropid);
                    MessageBox (h_wndmain, "Error dropping call", "", MB_OK);
                }
                WaitForAsyncReturn (&l_dropid);
            }
            saveMessage (); /* save the message */
            hfree (((LPWAVEHDR)lParam)->lpData); /* free 60s message buffer */
            b_recording = FALSE;
            break;

        case WM_PAINT:
            hDC=BeginPaint(hWnd,&ps);
            EndPaint (hWnd, &ps);
            break;

        case WM_COMMAND:
            switch (wParam) {
                case IDM_AUTOANSWER:
                    if ((lrc = lineOpen(hApp, dwLineID, &hLine, dwVerAPI,
                                        0 /* no extensions */,
                                        NULL /* instance data */,
                                        LINECALLPRIVILEGE_OWNER,
                                        LINEMEDIAMODE_UNKNOWN |
                                        LINEMEDIAMODE_AUTOMATEDVOICE,
                            NULL)) < 0) {
                     lineError (lrc);
                     PostQuitMessage (0);
                     break;
                 }

            /* get the id of the wave audio device associated with the line */
           if ((dw_waveid = getWaveDeviceID(hLine)) < 0) {
                break;
           }
           break;
     default:
            return (DefWindowProc(hWnd, message, wParam, lParam));
      } //End switch (wParam)
        break;

case WM_CREATE:
     /* initialize line device, neg. versions, etc...    */
     if ((doInitStuff (&hApp, &dwVerAPI, &dwLineID) < 0) ||
         ((dwVerAPI || dwLineID)= =0)) {
          MessageBox (hWnd, "Error initializing line", "", MB_OK);
          PostQuitMessage (0);
      }
      break;

case WM_DESTROY:
     if (h_call) { /* drop the call */
          if ((l_dropid = lineDrop (h_call, NULL, 0)) < 0) {
               lineError (l_dropid);
               MessageBox (h_wndmain, "Error dropping call", "", MB_OK);
          }
          WaitForAsyncReturn (&l_dropid);
     }
     if (hLine)
          lineClose (hLine);
     if (hApp)
          lineShutdown (hApp);
     PostQuitMessage(0);
     break;

case WM_TIMER:
     break;

default:
     return (DefWindowProc(hWnd, message, wParam, lParam));

} //End switch (message)
return (NULL);

} //End function (MainWndProc)
/*****************************************************************************
    FUNCTION: lineCallback
    PURPOSE:  callback function handles line events
*****************************************************************************/
VOID FAR PASCAL __export lineCallback (DWORD dwDevice, DWORD dwMsg,
                                       DWORD dwCallbackInst, DWORD dwParam1,
                                       DWORD dwParam2,DWORD dwParam3)
{
    dwCallbackInst;

    switch (dwMsg)
    {
     case LINE_MONITORMEDIA:
          /* if it's a fax call, hand it off to fax application */
          if (dwParam1 & LINEMEDIAMODE_G3FAX) {
              /* classify the call */
              lineSetMediaMode (h_call, LINEMEDIAMODE_G3FAX);
              /* hand it off to fax application */
              if (lineHandoff (h_call, NULL, LINEMEDIAMODE_G3FAX) < 0) {
                  /* could not hand off the fax - hang up in frustration */
                  /* drop the call */
                  if ((l_dropid = lineDrop (h_call, NULL, 0)) < 0) {
                      lineError (l_dropid);
                      MessageBox (h_wndmain, "Error dropping call", "", MB_OK);
                  }
                  WaitForAsyncReturn (&l_dropid);
              }
              /* clean up after call */
              /* since we don't drop the call, we must explicitly deallocate it
              instead of relying on LINE_REPLY to do it */
              if (h_call)
                  lineDeallocateCall (h_call);
              doneCall ();
          }
          break;
     case LINE_REPLY:
          /* call drop completion */
          if ((LONG)dwParam1 = = l_dropid) {
              l_dropid = 0;
              if (h_call)
                  lineDeallocateCall (h_call);
              h_call = NULL;
          }
          /* call answer completion */
          if ((LONG)dwParam1 = = l_ansid)
              l_ansid = 0;
          /* error */
          if ((LONG)dwParam2  < 0)
             lineError ((LONG)dwParam2);
          AsyncReturn (dwParam1, dwParam2);
          break;

     case LINE_CALLSTATE:
          switch (dwParam1)
          {
           case LINECALLSTATE_DIALTONE:
           case LINECALLSTATE_IDLE:
               doneCall (); /* clean up after call */
               break;
           case LINECALLSTATE_CONNECTED:
               if (h_wave=playSound (dw_waveid,"msg.wav", h_wndmain))
                   b_playingmessage = TRUE;
               break;

            case LINECALLSTATE_OFFERING:
               /* The call is being offered, signaling the arrival
               of a  new call.  check if we are the owner of the call, and if
               we are, then answer it. */
               if (dwParam3 = = LINECALLPRIVILEGE_OWNER) {
                  if ((l_ansid=lineAnswer(h_call = (HCALL)dwDevice,NULL,0))<0){
                      lineError (l_ansid);
                      MessageBox (h_wndmain, "Error answering call", "", MB_OK);
                      return;
                  }
                  WaitForAsyncReturn (&l_ansid);
                  /* set notification of fax detection for later handoff */
                  lineMonitorMedia (h_call, LINEMEDIAMODE_G3FAX);
               } /* end if (we own this call) */
               break;
        } /* end switch (on call state) */
    } /* end switch (on message) */
} /* end function (lineCallback) */

/*****************************************************************************
    FUNCTION: doInitStuff
    PURPOSE:  perform initializaions
```

```
*************************************************************************/
#define APIHIVERSION    0x00010001              /* 1.01 */
#define APILOWVERSION   0x00010001              /* 1.01 */

int doInitStuff (HLINEAPP *ph_app, DWORD *pdw_ver, DWORD *pdw_lineid)
{
LINEEXTENSIONID ext_id;
DWORD dw_numdevs, i;
LONG lrc;
LINEDEVCAPS *p_lineDevCaps;

    *pdw_lineid = *pdw_ver = 0;
    p_lineDevCaps = (LINEDEVCAPS *) calloc (1, sizeof(LINEDEVCAPS)+1000);

    /* initialize application's use of the telephone API */
    if (lineInitialize(ph_app, hInst, lineCallback, NULL, 0, &dw_numdevs) < 0) {
        /* return error */
        MessageBox (h_wndmain, "Error initializing line", "", MB_OK);
        return -1;
    }

    /* negotiate the line API version */
    if ((lrc = lineNegotiateAPIVersion(*ph_app,0 /* how do i know line id?? */,
                                       APILOWVERSION, APIHIVERSION, pdw_ver,
                                &ext_id)) < 0)      {
        lineError (lrc);
        lineShutdown (*ph_app);
        *ph_app = NULL;
        MessageBox (h_wndmain, "Error negotiating version", "", MB_OK);
        return -1;
    }

    /* for each line device, get capabilities until we find one that supports
    interactive voice */
    p_lineDevCaps->dwTotalSize = sizeof(LINEDEVCAPS) + 1000;
    for (i=0; i < dw_numdevs; i++) {
        lineGetDevCaps(*ph_app, i, *pdw_ver, 0, p_lineDevCaps);
        if (p_lineDevCaps->dwMediaModes & LINEMEDIAMODE_INTERACTIVEVOICE) {
            *pdw_lineid = i;
            break;
        }
    }
    free (p_lineDevCaps);

    return 0;
} /* end function (doInitStuff) */

/*************************************************************************
    FUNCTION: lineError
    PURPOSE:  line error messages
*************************************************************************/
void lineError (LONG lrc)
{
 switch (lrc) {
    case LINEERR_INVALAPPHANDLE:
       MessageBox (h_wndmain, "Invalid app handle.", "", MB_OK);
       break;
    case LINEERR_BADDEVICEID:
       MessageBox (h_wndmain,"The specified line device ID is out of range.", "",
                   MB_OK);
       break;
    case LINEERR_INCOMPATIBLEAPIVERSION:
       MessageBox (h_wndmain, "Incompatible API version.", "", MB_OK);
       break;
    case LINEERR_INCOMPATIBLEEXTVERSION:
       MessageBox (h_wndmain, "Incompatible extension version.","", MB_OK);
       break;
    case LINEERR_NOMEM:
       MessageBox (h_wndmain, "No memory","", MB_OK);
       break;
    case LINEERR_NODRIVER:
       MessageBox (h_wndmain, "No driver loaded", "", MB_OK);
       break;
    case LINEERR_RESOURCEUNAVAIL:
       MessageBox (h_wndmain, "Resource overcommittment", "", MB_OK);
       break;
    case LINEERR_INVALPRIVSELECT:
       MessageBox (h_wndmain, "Requested invalid priviledges", "", MB_OK);
       break;
    case LINEERR_INVALMEDIAMODE:
       MessageBox (h_wndmain, "Requested invalid media mode", "", MB_OK);
       break;
    case LINEERR_LINEMAPPERFAILED:
       MessageBox (h_wndmain, "Line mapper failed", "", MB_OK);
       break;
    case LINEERR_INVALPOINTER:
       MessageBox (h_wndmain, "The specified pointer parameter is invalid.",
                   "", MB_OK);
       break;
    case LINEERR_OPERATIONFAILED:
       MessageBox (h_wndmain, "Operation failed.", "", MB_OK);
       break;
    case LINEERR_INVALLINEHANDLE:
       MessageBox (h_wndmain, "Invalid line handle", "", MB_OK);
       break;
    case LINEERR_INVALADDRESSID:
       MessageBox (h_wndmain, "Invalid address id", "", MB_OK);
       break;
    case LINEERR_INVALCALLHANDLE:
       MessageBox (h_wndmain, "Invalid call handle", "", MB_OK);
       break;
    case LINEERR_INVALCALLSELECT:
       MessageBox (h_wndmain, "Invalid call selection", "", MB_OK);
       break;
    case LINEERR_NODEVICE:
       MessageBox (h_wndmain, "No associated device for given class",
                   "", MB_OK);
       break;
       }
}

/*****************************************************************************
    FUNCTION: getWaveDeviceID
    PURPOSE:  get the id of the wave device corresponding to the line
    COMMENTS: wave devices use an unsigned integer id.  we must allocate enough
              space for this id to be returned from the driver.
*****************************************************************************/
LONG getWaveDeviceID (HLINE h_line)
{
LONG lrc;
WORD w_id;
/* allocate 2 extra bytes for wave id. */
VARSTRING *p_varstr = (VARSTRING *)calloc (sizeof(VARSTRING) + 2, 1);

    p_varstr->dwTotalSize =  sizeof (VARSTRING) + 2;
    if ((lrc = lineGetID (h_line, 0 /* not an address */, 0 /* not a call */,
                          LINECALLSELECT_LINE /* it's a line! */, p_varstr,
                          "wave" /* wave device */)) < 0) {
        lineError (lrc);
        PostQuitMessage (0);
        return -1;
    }
    /* if the returned id was not binary, something is wrong */
    if (p_varstr->dwStringFormat != STRINGFORMAT_BINARY) {
        MessageBox (h_wndmain, "Bad wave device id", "", MB_OK);
        PostQuitMessage (0);
        return -1;
    }
    w_id = (WORD) *((BYTE *)p_varstr+sizeof(VARSTRING)); /* get id */
    free (p_varstr);

    return (LONG)w_id;
} /* end function (getWaveDeviceID) */

/*****************************************************************************
    FUNCTION: doneCall
    PURPOSE:  end-of-call processing
*****************************************************************************/
void doneCall ()
{
    /* if playing or recording a message, close up */
    if (b_playingmessage || b_playingbeep) {
        b_playingmessage = b_playingbeep = FALSE;
        waveOutReset (h_wave); /* reset wave device */
        /* drop the call */
        if ((l_dropid = lineDrop (h_call, NULL, 0)) < 0) {
            lineError (l_dropid);
            MessageBox (h_wndmain, "Error dropping call", "", MB_OK);
            return;
        }
        WaitForAsyncReturn (&l_dropid);
    }
    else if (b_recording) { /* close up if recording */
        b_recording = FALSE;
        waveInStop (h_wavein);
    }

} /* end function (doneCall) */

MESSAGE.C
#include <windows.h>
#include <windowsx.h>
#include <mmsystem.h>
#include <commdlg.h>
#include <tapi.h>
#include <io.h>
#include <stdlib.h>
#include <malloc.h>
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include "tam.h"


/* quick and dirty RIFF chunk */
BYTE ab_riffchunk[] = {'R','I','F','F',  0,0,0,0, 'W','A','V','E'};
/* quick and dirty format chunk tag */
BYTE ab_formatchunktag[] = {'f','m','t',' ',  0,0,0,0};
/* quick and dirty data chunk header */
BYTE ab_datachunktag[] = {'d','a','t','a',  0,0,0,0};

/* message headers */
WAVEHDR inMessageHeader = {NULL, 0, 0, NULL, 0 /* no flags */, 0 /* no loops */,
                           NULL, 0};

/* handle to main window */
```

```
extern HWND h_wndmain;

/* format of in-message */
PCMWAVEFORMAT messageFormat = {WAVE_FORMAT_PCM, 1, 22050, 22050, 1, 8};

/*****************************************************************************
    FUNCTION: recordMessage
    PURPOSE:  begin recording voice message
*****************************************************************************/
/* record the voice message  - 60s max */
HWAVEIN recordMessage (DWORD dw_devid)
{
HWAVEIN h_wavein;
HDRVR hdrv;
DWORD dw_60sbufsz = 60L * (LONG)messageFormat.wf.nSamplesPerSec *
                            (LONG)messageFormat.wBitsPerSample/8L;

 /* open recorder. */
 if (waveInOpen( &h_wavein, (WORD) dw_devid, (LPWAVEFORMAT)&messageFormat,
                 (DWORD)h_wndmain, NULL, CALLBACK_WINDOW)) {
     MessageBox (NULL, "Error opening wave record device.", "", MB_OK);
     return 0;
 }

 /* allocate 60s message buffer */
 if ((inMessageHeader.lpData = (HPSTR) halloc (dw_60sbufsz, 1)) = = NULL) {
     MessageBox (NULL, "Not enough memory.", "", MB_OK);
     return 0;
 }
 inMessageHeader.dwBufferLength = dw_60sbufsz;
 if (waveInPrepareHeader(h_wavein, &inMessageHeader, sizeof(WAVEHDR))) {
     MessageBox (NULL, "Error preparing message header.", "", MB_OK);
     return 0;
 }

 /* pass down a 60s buffer to record into */
 if (waveInAddBuffer (h_wavein, &inMessageHeader, sizeof(WAVEHDR))) {
     MessageBox (NULL, "Error adding buffer.", "", MB_OK);
     return 0;
 }
 waveInStart (h_wavein); /* start recording */
 return h_wavein;

} /* end function (record message) */
/*****************************************************************************
    FUNCTION: saveMessage
    PURPOSE:  save a recorded voice message
*****************************************************************************/
/* save the message after the caller hangs up, or after 60s */
int saveMessage ()
{
HANDLE h_file;
static int MsgNum=1;
char sz_fname[_MAX_PATH+1];

    wsprintf(sz_fname,"call-%d.wav",MsgNum++);
    h_file = open (sz_fname, O_CREAT | O_BINARY | O_WRONLY);
    if (h_file= =-1) {
        MessageBox (NULL, "Error opening file.", "", MB_OK);
        return -1;
    }

    /* write out the RIFF chunk */
    *((DWORD *)&ab_riffchunk[4])=4 + sizeof(ab_formatchunktag) +
                                 sizeof(messageFormat) + sizeof(ab_datachunktag) +
                                 inMessageHeader.dwBytesRecorded;
    write(h_file, ab_riffchunk, sizeof(ab_riffchunk));
    *((DWORD *)&ab_formatchunktag[4]) = sizeof(messageFormat); /* write  tag */
    write(h_file, ab_formatchunktag, sizeof(ab_formatchunktag));

    /* write out the canned format header */
    write (h_file, &messageFormat, sizeof(messageFormat));

    /* write out the data chunk tag */
    *((DWORD *)&ab_datachunktag[4]) = inMessageHeader.dwBytesRecorded;
    write(h_file, ab_datachunktag, sizeof(ab_datachunktag));

    /* write out the data chunk */
    _hwrite (h_file, inMessageHeader.lpData, inMessageHeader.dwBytesRecorded);
    close (h_file); /* close message file */
    return 0;

} /* end function (save message) */

PLAY.C
#include <windows.h>
#include <windowsx.h>
#include <mmsystem.h>
#include <tapi.h>
#include “tam.h”

/*****************************************************************************
    FUNCTION:
    PURPOSE:
*****************************************************************************/
/* begin playing out the message */
HWAVEOUT playSound (DWORD dw_devid, char *szWaveFile, HWND hWnd)
{
HWAVEOUT        hWave;
```

```
HDRVR             hdrv;
HMMIO            hmmio;
MMCKINFO        mmckinfoParent;
MMCKINFO        mmckinfoSubchunk;
DWORD            dwFmtSize;
DWORD            dwDataSize;
HPSTR            lpWaveData;
WAVEHDR FAR*    lpWaveHdr;
PCMWAVEFORMAT FAR *lpWaveFormat;

    /* Open wave file */
    hmmio = mmioOpen(szWaveFile, NULL, MMIO_READ | MMIO_ALLOCBUF);
    if(!hmmio)
    {
        return 0;
    }

    /* Locate a ‘RIFF’ chunk with a ‘WAVE’ form type */
    mmckinfoParent.fccType = mmioFOURCC(‘W’, ‘A’, ‘V’, ‘E’);
    if (mmioDescend(hmmio, (LPMMCKINFO) &mmckinfoParent, NULL, MMIO_FINDRIFF))
    {
        mmioClose(hmmio, 0);
        return 0;
    }

    /* Find the format chunk */
    mmckinfoSubchunk.ckid = mmioFOURCC(‘f’, ‘m’, ‘t’, ‘ ‘);
    if (mmioDescend(hmmio, &mmckinfoSubchunk, &mmckinfoParent, MMIO_FINDCHUNK))
    {
        mmioClose(hmmio, 0);
        return 0;
    }

    /* Get the size of the format chunk, allocate memory for it */
    dwFmtSize = mmckinfoSubchunk.cksize;
    lpWaveFormat = (PCMWAVEFORMAT FAR *) GlobalAllocPtr(GMEM_MOVEABLE, dwFmtSize);
    if (!lpWaveFormat)
    {
        mmioClose(hmmio, 0);
        return 0;
    }

    /* Read the format chunk */
    if (mmioRead(hmmio, (HPSTR) lpWaveFormat, dwFmtSize) != (LONG) dwFmtSize)
    {
        GlobalFreePtr( lpWaveFormat );
        mmioClose(hmmio, 0);
        return 0;
    }

    /* Ascend out of the format subchunk */
    mmioAscend(hmmio, &mmckinfoSubchunk, 0);

    /* Find the data subchunk */
    mmckinfoSubchunk.ckid = mmioFOURCC(‘d’, ‘a’, ‘t’, ‘a’);
    if (mmioDescend(hmmio, &mmckinfoSubchunk, &mmckinfoParent, MMIO_FINDCHUNK))
    {
        GlobalFreePtr( lpWaveFormat );
        mmioClose(hmmio, 0);
        return 0;
    }

    /* Get the size of the data subchunk */
    dwDataSize = mmckinfoSubchunk.cksize;
    if (dwDataSize = = 0L)
    {
        GlobalFreePtr( lpWaveFormat );
        mmioClose(hmmio, 0);
        return 0;
    }

    /* Allocate and lock memory for the waveform data. */
    lpWaveData = (HPSTR) GlobalAllocPtr(GMEM_MOVEABLE | GMEM_SHARE, dwDataSize );
    if (!lpWaveData)
    {
        GlobalFreePtr( lpWaveFormat );
        mmioClose(hmmio, 0);
        return 0;
    }

    /* Read the waveform data subchunk */
    if(mmioRead(hmmio, (HPSTR) lpWaveData, dwDataSize) != (LONG) dwDataSize)
    {
        GlobalFreePtr(lpWaveFormat);
        GlobalFreePtr(lpWaveData);
        mmioClose(hmmio, 0);
        return 0;
    }

    /* We’re done with the file, close it */
    mmioClose(hmmio, 0);

    /* Allocate a waveform data header */
    lpWaveHdr = (WAVEHDR FAR*) GlobalAllocPtr(GMEM_MOVEABLE | GMEM_SHARE,
                 (DWORD)sizeof(WAVEHDR));
    if (!lpWaveHdr)
    {
        GlobalFreePtr(lpWaveData);
        GlobalFreePtr(lpWaveFormat);
        return 0;
    }

    /* Set up WAVEHDR structure and prepare it to be written to wave device */
    lpWaveHdr->lpData = lpWaveData;
    lpWaveHdr->dwBufferLength = dwDataSize;
    lpWaveHdr->dwFlags = 0L;
    lpWaveHdr->dwLoops = 0L;
    lpWaveHdr->dwUser  = 0L;

    /* make sure wave device can play our format */
    if (waveOutOpen((LPHWAVEOUT)NULL,
                    (WORD)dw_devid,
                    (LPWAVEFORMAT)lpWaveFormat,
                    0L, 0L, WAVE_FORMAT_QUERY)) {
        GlobalFreePtr( lpWaveFormat );
        MessageBox (NULL, "Unsupported wave format.", "", MB_OK);
        return 0;
    }

    /* open the wave device corresponding to the line */
    if (waveOutOpen (    &hWave,
                        (WORD)dw_devid,
                        (LPWAVEFORMAT)lpWaveFormat,
                        (DWORD)hWnd,
                        0L,
                        CALLBACK_WINDOW))
    {
        GlobalFreePtr( lpWaveFormat );
        MessageBox (NULL, "Error opening wave device.", "", MB_OK);
        return 0;
    }

    GlobalFreePtr( lpWaveFormat );

     /* prepare the message header for playing */
    if (waveOutPrepareHeader (hWave, lpWaveHdr, sizeof(WAVEHDR))) {
       MessageBox (NULL, "Error preparing message header.", "", MB_OK);
        return 0;
    }

    /* play the message right from the data segment;  set the play message flag */
    if (waveOutWrite (hWave, lpWaveHdr, sizeof (WAVEHDR))) {
        MessageBox (NULL, "Error writing wave message.", "", MB_OK);
        return 0;
    }


    return hWave;

} /* end function (play message) */

ASYNC.H
int WaitForAsyncReturn(long *pRetCode);
int AsyncReturn(long RequestID, long RetCode);

ASYNC.C
#include <windows.h>
#include <stdlib.h>

typedef struct tagWaitListElement
{
    struct tagWaitListElement *pNext;
    long *pRetCode;
} WLE;

WLE *pWaitList=NULL;

/****************************************************************************
    FUNCTION: WaitForAsyncReturn
    PURPOSE:  wait for asyn function to complete
****************************************************************************/
int WaitForAsyncReturn(long *pRetCode)
{
MSG msg;
    if (*pRetCode>0)
    {
        WLE *pNewWLE;

        pNewWLE = malloc(sizeof(WLE));
        if (!pNewWLE)
            return -1;
        pNewWLE->pRetCode=pRetCode;
        pNewWLE->pNext=pWaitList;
        pWaitList=pNewWLE;


        while (GetMessage(&msg, NULL, NULL, NULL)) /*loop til request id is reset*/
            {
            TranslateMessage(&msg);    //Translates virtual key codes
            DispatchMessage(&msg);     //Dispatches message to window
            if (*pRetCode <= 0)
                break;
            } //End while (not a quit message)
    }
    return 0;
} /* end function (WaitForAsyncReturn) */
```

```
/**************************************************************************
    FUNCTION: AsyncReturn
    PURPOSE:  called to remove asyn function from list
**************************************************************************/
int AsyncReturn(long RequestID, long RetCode)
{
WLE *pFoundWLE, **ppPrevWLE;

    ppPrevWLE=&pWaitList;
    pFoundWLE=pWaitList;

    while (pFoundWLE)
    {
        if (*pFoundWLE->pRetCode= =RequestID)
        {    // update return code that application is spinning on
            *pFoundWLE->pRetCode=RetCode;
            *ppPrevWLE = pFoundWLE->pNext; // remove element from list
            free(pFoundWLE); // free memory associated with element
            return 0; // return success
        }
        ppPrevWLE=&pFoundWLE->pNext;
        pFoundWLE=pFoundWLE->pNext;
    }

    return -1;    // return failure- no element found
} /* end function (AsyncReturn) */
```

During the processing of WM_CREATE, before the main window is displayed, TAM registers as a TAPI application and negotiates a compatible API version. It also searches out a line that can answer voice calls. The function lineInitialize tells TAPI.DLL that the program is interested in using telephone services.

```
/* initialize app’s use of the telephone API */
if (lineInitialize(ph_app, hInst, lineCallback, NULL,
                   &dw_numdevs) < 0) {
    /* return error */
    MessageBox (h_wndmain, "Error initializing line",
                "", MB_OK);
    return -1;
}
```

Our application is uniquely identified by its instance handle. The third parameter of lineInitialize is the address of TAM’s callback function, where TAPI can notify it of changes to lines, addresses, or calls. The fourth parameter specifies a user-friendly name for the application, a name available to other running TAPI applications, such as Answering Machine. We set it to NULL. The last parameter is filled by TAPI and specifies the number of line devices available to the application. TAPI fills the first parameter with a unique application handle to identify the application in future TAPI calls.

If successful in registering and negotiating an API version, lineInitialize polls each line device until it finds one that carries voice.

```
for (i=0; i < dw_numdevs; i++) {
    lineGetDevCaps(*ph_app, i, *pdw_ver, 0, p_lineDevCaps);
    if (p_lineDevCaps->dwMediaModes &           LINEMEDIAMODE_INTERACTIVEVOICE) {
        *pdw_lineid = i;
        break;
    }
```

When it finds a line, the function saves the ID and returns. The code that finds the line calls lineGetDevCaps and checks the dwMediaModes field for voice. Notice anything strange about the last parameter to this function? It’s a pointer to a LINEDEVCAPS structure allocated as

```
p_lineDevCaps = (LINEDEVCAPS *) calloc(1,
                               sizeof(LINEDEVCAPS)+1000);
```

What’s going on here? The pointer references a buffer with the structure of LINEDEVCAPS, but with 1000 extra bytes tacked on to the end. This is common when working with TAPI. At least some of the extra bytes are required (We’ll explain why in a minute). Nothing more happens in the application until the user clicks Wait for Call.

When the user clicks Wait for Call, two things happen. First, TAM opens the line and indicates interest in unknown-type calls and calls that can be handled by automated voice applications like an answering machine. Why both? The reason goes back to the handoff mechanism. If the application is first to answer calls, it is possible the media mode will be unknown at call offering (ring) time. If TAM is not first in line, the call is answered by another application that may hand it off to TAM to take a message. So to be safe, TAM registers for both unknown and automated voice calls.

```
if ((lrc = lineOpen(hApp, dwLineID, &hLine, dwVerAPI,
                    0    /* no extensions */,
                    NULL /* instance data */,
                    LINECALLPRIVILEGE_OWNER,
                    LINEMEDIAMODE_UNKNOWN |
                    LINEMEDIAMODE_AUTOMATEDVOICE,
                    NULL)) < 0) {
    lineError (lrc);
    PostQuitMessage (0);
    break;
}
```

The first parameter to lineOpen is the application handle retrieved from lineInitialize. Next comes the line number to open, followed by a pointer to a line handle. TAPI fills this with a handle used in future line-related calls. (TAPI uses handles to identify everything—applications, lines, addresses, and calls.) Next is passed the negotiated API version, the extension version (no extensions are used, so 0 is passed), followed by additional information for TAPI to pass to TAM’s callback function. Since the callback doesn’t use additional information, this parameter is set NULL. The seventh parameter specifies the privileges for calls on the line. We want to own the calls (as opposed to just monitoring or snooping on them), so LINECALLPRIVILEGE_OWNER is specified (see the TAPI spec or TAPI.H for its privilege flags). Then the two media modes of interest are specified. The last parameter indicates additional call parameters and is not used.

The next step is to determine the ID of the wave device associated with the line just opened. Remember, TAPI is designed to setup, take down, and control calls, not to transfer data. Playing and recording voice messages requires an API like the wave API in MMSYSTEM.DLL. So the ID of the associated wave device is retrieved to play and record messages later. To do this, TAM calls lineGetID. This useful function searches the system for the wave device associated with the line just opened.

```
/* allocate 2 extra bytes for wave id. */
VARSTRING *p_varstr = (VARSTRING *)calloc(
                                  sizeof(VARSTRING)+2,
                                  1);

p_varstr->dwTotalSize = sizeof (VARSTRING) + 2;
if ((lrc=lineGetID(h_line, 0 /* not an address */,
                    0 /* not a call */,
                    LINECALLSELECT_LINE /*it’s a line*/,
                    p_varstr,
                    "wave" /* wave device */)) < 0) {
     lineError (lrc);
     PostQuitMessage (0);
     return -1;
}
```

The call to lineGetID also uses a structure with extra bytes allocated on the end. Here’s the explanation for the extra bytes. TAPI uses structures this way whenever the parameters to functions can have many different formats. lineGetID is a good example. You can call this function to get the ID of wave devices, COMM devices, network devices, MIDI devices, and devices yet to be defined. Each device codes IDs in its own way. Some return short integers, some long integers, some may return strings, and some may even return structures. Microsoft wanted a single function to return the IDs for all devices, so the extended-structure scheme was implemented.

The extended structure works like a suitcase with a false bottom. There is a well-defined space everyone knows about, the common fields of the structure. Then the bottom opens up, and there’s more space. You may allocate as much space as you need in the false bottom of the structure; just make sure it’s enough or TAPI will complain. Because wave devices return a short integer ID, we allocate two extra bytes at the end of the structure. We tell TAPI about the extra bytes by setting the dwTotalSize field of the structure to the size of the structure plus two bytes. Then it’s time to pass the line handle—since the function may also retrieve the ID of devices associated with addresses and calls, we set a flag to indicate that TAM is interested in the line (LINECALLSELECT_LINE). We pass a pointer to the extended structure, and the string “wave” to indicate the wave device. The function returns the wave device ID in the extra bytes of the structure, which are validated and saved for later.

In the earlier call to lineGetDevCaps, we tacked 1000 extra bytes onto the end of the LINEDEVCAPS structure. 1000, because it is almost certainly more than we need. For lineGetID, the exact number of extra bytes required is well known (2 bytes). In the case of lineGetDevCaps, the number of extra bytes varies depending on the service provider, and there is no way to know ahead of time how many are needed. The TAPI specification recommends a different method for determining the proper size: call the function once with the size set to sizeof (structure). The function will return LINEERR_STRUCTURE_TOOSMALL, with the dwNeededSize field of the structure set to the required number of bytes. Now allocate dwNeededSize bytes for the structure and call the function again. We prefer to allocate more than we need and call the function once. Of course, if you plan to keep the structure around for awhile after calling the function, this can lead to a considerable waste of space. Choose the method you prefer.

Once again, the application sits idle, this time waiting for the arrival of a call. This situation corresponds to the IDLE state in the diagram. All that work just to get to the IDLE state!

A Call

When a call arrives, TAPI notifies the callback function with a LINECALLSTATE_OFFERING message. TAM checks parameter three to ensure that it owns the call, then proceeds to answer it with a call to lineAnswer. The first parameter to the callback function dwDevice is either a line handle, address handle, or call handle, depending on the message. For LINE_CALLSTATE messages, dwDevice contains a call handle. The call handle is passed to lineAnswer, with user-to-user information. User-to-user information is anything you want to pass to the remote party when the call is answered, such as the string “Hi, you have reached Joe’s answering machine!” Unfortunately, only advanced telephone environments such as ISDN support neat features like user-to-user info, so I can’t take advantage of it here. The user-to-user info parameter (parameter two) is set to NULL, and the size of the info is set to 0 (third parameter).

```
case LINECALLSTATE_OFFERING:
   /* The call is being offered, signaling the arrival
   of a new call. check if we are the owner of the
   call, and if we are, then answer it. */
   if (dwParam3 = = LINECALLPRIVILEGE_OWNER) {
      if ((l_ansid=lineAnswer (h_call=(HCALL)dwDevice,
                               NULL, 0)) < 0) {
         lineError (lrc);
         MessageBox (h_wndmain, "Error answering call",
                     "", MB_OK);
         return;
      }
   WaitForAsyncReturn (&l_ansid);
   } /* end if (we own this call) */
/* set notification of fax detect for later handoff */
lineMonitorMedia (h_call, LINEMEDIAMODE_G3FAX);
```

A nonzero number will be returned in lineAnswer. A negative value indicates an error; a positive value indicates asynchronous execution—lineAnswer returned before the call was actually answered. Later the callback function will receive a LINE_REPLY message indicating whether or not the answer was successful. TAM waits for this message to be sent by calling WaitForAsyncReturn (see below). It then calls lineMonitorMedia to inform TAPI that it’s interested in knowing if the call turns out to be a fax, so that it can hand it off to a fax application. Once the call is connected, things happen quickly. TAPI sends a LINECALLSTATE_CONNECTED message to indicate the transition to the connected state. TAM traps this message to call playSound, which plays out the answering machine message. The prompt plays using the wave API, and when it completes, the wave driver sends a MM_ WOM_DONE message to TAM’s window procedure. TAM frees the resources associated with the prompt message, which completes the transition through the PROMPT state in Figure 11. Next TAM calls playSound again to play the beep, which tells the caller to leave a message. playSound again concludes with a MM_WOM_DONE to TAM’s window procedure, at which time it frees the resources associated with the beep, which completes the BEEP state. Next TAM begins recording the message by calling recordMessage, which forces it into the RECORD state. The RECORD state can terminate one of two ways. First, the sixty-second maximum message length may expire, in which case MMSYSTEM.DLL sends a MM_WIM_ DATA message to TAM’s window procedure. It saves the recorded message and goes back to the idle state. Or the caller can hang up, in which case the callback function may receive either a LINECALLSTATE_DIALTONE or LINECALLSTATE_IDLE message for the call. Upon receipt of either message we call TAM’s doneCall function, which, depending on the state it’s in, will take appropriate action and transition back to the IDLE state (see Figure 13).

Figure 13 doneCall

```
void doneCall()
{
/* if playing or recording a message, close up */
if (b_playingmessage || b_playingbeep) {
     b_playingmessage = b_playingbeep = FALSE;
     /* reset wave device */
     waveOutReset (h_wave);
     /* drop the call */
     if ((l_dropid = lineDrop (h_call, NULL, 0)) < 0) {
   lineError (l_dropid);
   MessageBox (h_wndmain, "Error dropping call", "", MB_OK);
   return;
   }
     WaitForAsyncReturn (&l_dropid);
}
/* close up if recording */
else if (b_recording) {
     b_recording = FALSE;
     waveInStop (h_wavein);
}
} /* end function (doneCall)*/
```

Look at the internals of doneCall. If the answering machine played out the message or the beep, it calls waveOutReset. This forces the wave device to send it an MM_WOM_DONE message, which cleans up play buffers and closes the device as explained above. It also calls lineDrop, which drops the current call. Notice that lineDrop, like lineAnswer, is asynchronous. TAM saves the return value and then waits for asynchronous completion by calling WaitForAsyncReturn. When lineDrop completes, TAPI sends the callback a LINE_REPLY message, at which point TAM calls the function lineDeallocateCall. This function frees the resources (such as allocated memory) associated with a call after the call has been dropped. If you examine the listing for TAM.C, you will notice that lineDrop is called in three different places, whereas lineDeallocateCall is called only once, during the processing of LINE_REPLY (a special case occurs if the call turns out to be a fax, which we shall explain shortly). Since TAM always deallocates a call after dropping it, the code for LINE_REPLY is a convenient place to put lineDeallocateCall.

If the answering machine was recording a message when the call disconnected, it calls waveInStop. This stops the recording process and forces the wave device to send the program an MM_WIM_DATA message, which saves the message and transitions to IDLE. All the functions for recording and saving incoming messages can be found in MESSAGE.C. Incoming messages are recorded using a monophonic, 22 KHz, 8 bit/sample format, which is specified by the messageFormat structure at the top of the file. This format is playable by most sound cards for Windows on the market.

To record the incoming message, TAM first calls waveInOpen to open the waveform input driver. It passes the handle to its main window so that when it is finished recording, the driver will send it an MM_WIM_DONE message. Next, the program allocates a buffer large enough to hold 60 seconds of audio data, and call waveInPrepareHeader to ready the buffer for recording. The record buffer is passed to the driver using waveInAddBuffer, and recording is started using waveInStart. Once recording is started, the wave device will record all audio on the phone line without any further intervention on the part of the application. Upon returning from recordMessage, TAM sets the b_recording variable to TRUE to indicate that recording is in progress.

Upon receiving an MM_WIM_DONE message, the program closes the wave device using waveInClose and drops (hangs-up) the phone call using lineDrop. lineDrop will not free resources allocated to the call, so lineDeallocateCall is immediately invoked, which cleans up the call’s resources. Finally, TAM saves the message to disk using the saveMessage function. saveMessage saves each call in a file named CALL–XX.WAV, where XX represents the number of the call. These files may be played back using the Windows Media Player application. saveMessage uses a quick-and-dirty approach to creating RIFF files. It bypasses the mmio functions and directly writes the WAV header and wave data to a file. This saves space, but in general you should call the mmio function when reading or writing RIFF files. The answering machine is now ready to accept the next call.

Handling Fax Calls

If a fax machine is on the other end of the line, TAM doesn’t want to record a message, but it also doesn’t want to ignore the call, nor drop it if another running application can handle a fax. So it hands-off the call to the fax application (see Figure 14).

Figure 14 Handling Fax Calls

```
case LINE_MONITORMEDIA:
     /* if it's a fax call, hand it off to fax application */
     if (dwParam1 & LINEMEDIAMODE_G3FAX) {
          /* classify the call */
          lineSetMediaMode (h_call, LINEMEDIAMODE_G3FAX);
   /* hand it off to fax application */
   if (lineHandoff (h_call, NULL, LINEMEDIAMODE_G3FAX) < 0) {
        /* could not hand off the fax - hang up in frustration */
        /* drop the call */
        if ((l_dropid = lineDrop (h_call, NULL, 0)) < 0) {
             lineError (l_dropid);
             MessageBox (h_wndmain, "Error dropping call", "", MB_OK);
             }
        WaitForAsyncReturn (&l_dropid);
        }
   /* clean up after call */
   /* since we don't drop the call, we must explicitly deallocate it
   instead of relying on LINE_REPLY to do it */
   if (h_call)
        lineDeallocateCall (h_call);
   doneCall ();
   }
     break;
```

The answering machine expressed an interest in fax calls immediately after it answered the call. The call to lineMonitorMedia asks TAPI to inform us later on if the phone call is from a fax machine. If the service provider detects the CNG tone (that piercing tone fax machines generate), it will inform TAPI, which sends a LINE_ MONITORMEDIA message to TAM’s callback. The service provider checks the first parameter to see if the new media mode for the call is fax. Note that the service provider will not detect fax calls unless another application has done a lineOpen with the LINEMEDIAMODE_ G3FAX flag set; in other words, there must be a fax application running.

If the call is a fax, TAM tells TAPI to classify it as a fax call by calling lineSetMediaMode with the LINEMEDIAMODE_G3FAX flag set. Yes, the application must explicitly classify the call. It then calls lineHandoff to handoff the call to the highest-priority fax application. We could have specified a particular application name in the second parameter; setting this parameter to NULL tells TAPI to handoff by priority. If the handoff fails, TAM hangs up. Otherwise it calls lineDeallocateCall without dropping the call. TAM then calls our own doneCall function to close any media devices and free up buffers.

Synchronous/Asynchronous Interface

One of the peculiarities of the TAPI function call interface is that it’s both synchronous and asynchronous. When an application calls a TAPI function, that function will return one of three types of return values: positive, negative, or 0. A return value of 0 means the function succeeded, and a negative return value indicates an error occurred. (Individual TAPI functions are either synchronous or asynchronous, but not both. Only asynchronous functions can return positive values.) In either case the function completes before returning to the calling application.

A positive value indicates neither success or failure. It in effect says, “This may take awhile. I’ll get back to you later.” This is referred to as the asynchronous interface. The return code in these situations is a Request ID. At some later time, the application receives a message including this Request ID with details about the success or failure of the function. This wMsg param is set to either PHONE_REPLY or LINE_REPLY, dwParam1 is set to the Request ID, and dwParam2 holds the result code. The result code is either a 0 or a negative number representing success or an error code, respectively.

A number of programming strategies may be devised for handling asynchronous function calls. Perhaps the simplest is the nonblocking wait strategy. This makes asynchronous returns resemble synchronous returns by waiting for the function to complete.

In Figure 15, the program calls the WaitForAsyncReturn function after a TAPI function, passing a pointer to the return code. If the return code is nonpositive, WaitForAsync returns immediately. Otherwise, the return code is a Request ID, and WaitForAsyncReturn queues the pointer to the return code in a list of request IDs. This list contains all request IDs for outstanding TAPI function requests. It then uses a while loop to wait for the return code to turn nonpositive. The while loop incorporates a GetMessage call to allow other tasks in the system to continue running.

Figure 15 Implementing a Non-blocking Wait

```
// Code that calls a TAPI function
o
o
o
long RetCode;
RetCode=tapiFunc(...);
WaitForAsyncReturn(&RetCode);
// at this point, RetCode will either be 0 or negative
o
o
o

// Application's TAPI callback routine
MyTapiCallback(...)
{
...
long *pRetCode;
switch(dwMsg)
     {
o
o
o
     case PHONE_REPLY:
     case LINE_REPLY:
          AsyncReturn(dwParam1, dwParam2);
   break;
o
o
o
     }
return;
}


typedef struct tagWaitListElement
{
   struct tagWaitListElement *pNext;
   long *pRetCode;
} WLE;

WLE *pWaitList=NULL;

WaitForAsyncReturn(long *pRetCode)
{
MSG msg;
if (*pRetCode>0)
     {
     WLE *pNewWLE;

     pNewWLE = malloc(sizeof(WLE));
     if (!pNewWLE)
          return -1;
     pNewWLE->pRetCode=pRetCode;
     pNewWLE->pNext=pWaitList;
     pWaitList=pNewWLE;

     /* spin on message loop until request id is reset */
     while (GetMessage(&msg, NULL, NULL, NULL))
    {
     TranslateMessage(&msg);  //Translates virtual key codes
     DispatchMessage(&msg);   //Dispatches message to window
     if (*pRetCode <= 0)
          break;
     } //End while (not a quit message)
     }
     return 0;
}

int AsyncReturn(long RequestID, long RetCode)
{
     WLE *pFoundWLE, **ppPrevWLE;

     ppPrevWLE=&pWaitList;
     pFoundWLE=pWaitList;

     while (pFoundWLE)
          {
   if (*pFoundWLE->pRetCode= =RequestID)
        {
        // update return code that application is spinning on
        *pFoundWLE->pRetCode=RetCode;

        *ppPrevWLE = pFoundWLE->pNext // remove element from list
               free(pFoundWLE);        // free memory associated with element
        return 0;               // return success
        }
          ppPrevWLE=&pFoundWLE->pNext;
   pFoundWLE=pFoundWLE->pNext;
   }

     return -1; // return failure- no element found
}
```

When the TAPI function finally completes, MyTapiCallback is called with dwMsg set to either PHONE_REPLY or LINE_REPLY, dwParam1 set to the original request ID, and dwParam2 set to the success/error code. MyTapiCallback then calls the AsyncReturn function, which searches the list for a pointer to the request ID being satisfied. It replaces this entry with the actual return value and deletes the pointer from the list. The next time the while loop is executed, it sees the return code go nonpositive, and exits.

Conclusion

Writing TAPI applications will probably test the skills of even the most seasoned programmers. The key is to keep track of the state your program is in and rely on call state changes to drive the program state changes wherever possible. Also, don't try to get too fancy. Don't try to write an answering machine that is also a fax machine and a digital modem. Instead, break it up into three smaller programs and rely on call handoffs. Bear in mind that the TAPI specification, though thorough and well thought out, is still evolving. Niggling traps will spring up to frustrate you. For example, in TAM we negotiated an API version for the line before the loop that determines if the line could handle interactive voice. We assumed that the API version negotiated for line 0 would be valid for all lines, because lineGetDevCaps requires a version number. A better approach would be to move the call to lineNegotiateAPIVersion into the for loop along with lineGetDevCaps. That way, we could determine the version number of each line just prior to retrieving its capabilities. Since we stop looping when we find an interactive voice line, the negotiated version number would always match the chosen line.

Despite the challenges it presents, TAPI is one of the best attempts yet to create a standard interface for telephone applications. Its endorsement by many of the major players in the telecommunications market indicates that the standard will enjoy at least moderate success in the coming years.

TAPI Hardware

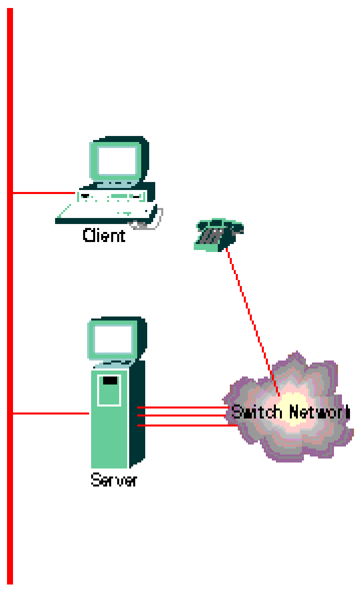

Figure A LAN-based TAPI Implementation

Many of the vendors supporting TAPI are unwilling to discuss their product plans right now. However, a few have made general announcements about their intentions. Lotus plans to incorporate telephony into its workgroup computing products. Northern Telecom will use TAPI in their VISIT Multimedia product line. Siemens and ROLM are planning products to integrate PBXs and PCs. Intel will use TAPI to enhance the SatisFAXtion® product line. Microsoft plans to provide applications, Contact Software International (CSI) will enhance its ACT! contact management software, Delrina will spruce up WinFax, Ericsson plans products to integrate PBXs and PCs, and Octus will use TAPI in its CTI (computer-telephone integration) applications.

TAPI supports a wide range of hardware configurations and capabilities. In PC-based plug-in cards, TAPI can and will be implemented on everything from low-end Hayes compatible fax/modems up to high-end DSP (Digital Signal Processor)-based cards supporting full TAPI functionality.

On a Hayes-compatible fax/modem, the Telephone API is mapped by the Service Provider Interface (SPI) layer into the AT command set. The connection between the PC and the TAPI hardware is, in this case, the PC's serial connection. (Even when the fax/modem is implemented as a plug-in card, internally the interface is still a serial port.) Using the AT command set, TAPI can support the full set of fax and modem features, but is limited in its handling of voice calls because the AT command set was not designed for voice calls.

TAPI-compatible plug-in cards will support the full range of TAPI features. These cards may continue to support the older serial port interface and AT command set for backward compatiblity, but will also implement higher performance hardware and software interfaces with greater functionality. Some cards may even eliminate the need for a local telephone handset by emulating a handset or speakerphone in hardware.

TAPI doesn't make assumptions about how the PC is connected to the telephone network. Figure A shows a LAN-based implementation in which the server is linked to the telephone switch via multiple telephone lines. TAPI requests from the client are forwarded to the server for processing. The server may also indirectly control the telephone at the client's desk via a third-party call control mechanism in the switch. The connection between the server and the switch may also exist via a single high-speed switch-to-host link, rather than using multiple standard telephone lines.

TAPI is independent of the telephone network. In addition to the analog POTS (Plain Old Telephone Service) network common in most residential homes, TAPI supports digital networks such as ISDN or even wireless networks.

Glossary of Telephony and TAPI-related Terms

analog line A standard 2500-type phone line. Signals on an analog line use a set of standard tones for call progress and DTMF signaling.

BRI-ISDN A basic rate ISDN connection consisting of two data-bearing channels of 64 kbps each for voice or data, and one data channel of 16 kbps for control.

call Two or more parties exchanging information (or attempting to exchange information) using telephony equipment. Most of the functions in TAPI operate on calls.

call progress Setting up a telephone call goes through several phases. Taking the phone offhook returns dialtone to indicate that a number can be dialed. Hearing the dial tone, the user dials the desired number. When the call reaches the destination phone, the caller will either receive a busy indication, indicating the called number is busy, or a ring back indication, indicating the dialed party is being alerted. Call progress is the process of monitoring the progress of a call through the various stages.

In analog networks, audible tones generated by the network provide the call progress indications to the user. Different tones allow the human ear to interpret the progress of the call. On digital networks (such as PBX or ISDN), the network may send indication messages to the phone to indicate the status of the call, and the phone may generate most tones locally, driven by those messages.

called-ID An identification (number, name) of the party being called. This identification is of interest when you transfer or forward a call. For example, when you forward an unanswered call to a voice messaging system, you use the called-ID of the original call to locate the mailbox of the called party.

caller-ID An identification (number, name) of the party initiating a call, as displayed to the called party prior to answering the call. A caller-ID may also be either unknown (due to telephone switch limitations) or blocked (concealed by the caller).

central office A public switch, directly connected to a number of telephones in a given geographical area.

Centrex A service provided by central offices that provides a virtual PBX to a set of extensions. It offers features such as transfer, conference, forward, etc., within that set of extensions.

client PC The host environment for which TAPI is defined. This is the PC running Windows.

CNG or fax tone A medium-pitched tone at 1,100Hz that lasts for half a second and repeats every 3.5 seconds.

desktop The logical pairing of a user’s PC and telephone.

digital line A digital station line on a PBX or digital key system. Signaling on a digital line uses vendor-specific (proprietary) protocols that exchange messages between the switch and the phone. A digital line typically requires a “matched” phone set.

DTMF Dual Tone Multiple Frequency. Pressing a button on the keypad of a Touchtone phone generates a pair of tones of specified frequency. The network or the equipment at the other end of the connection (such as a remote control for a phone answering machine), detects and interprets these tones.

ISDN Integrated Services Digital Network. A proposed standard for digital connectivity over standard phone lines.

key system A switching system where the phones have multiple buttons that permit the user to directly select incoming or outgoing lines. Key systems can typically support fewer users than can PBXs.

line A pool of resources (that is, communication channels) used by the application in performing telephony functions through the API. A PC may provide its applications with access to multiple lines and each line may provide different capabilities. It is the choice of the service provider for a line to decide how to model its resources. Note that a line need not correspond to a physical connection from the switch to the PC. For example, the realization of a line may involve a LAN-based server and a shared control link to the switch.

In essence, a line is any device that implements the line behavior defined by the API as the set of functions and messages for lines.

PBX Private Branch Exchange. A digital switch on the customer’s premises that provides switching (including a full set of switching features) for an office or campus. PBXs typically use proprietary digital line protocols, although the user features provided by the different PBX vendors are generally similar.

phone A device that behaves as a telephone set. This is usually, although not necessarily, the phone already on the user’s desk located “next to” the PC. Phones need not be physically connected to a PC. A LAN-based server with appropriate access to the switch may provide this logical connection. Note that TAPI treats the phone and the line as separate devices that can be independently controlled by the application.

In essence, a phone is any device that implements the phone behavior defined by this API as the set of functions and messages for phones.

POTS Plain Old Telephone Service. Basic single-line telephone service for the public switched telephone network. Only supports making and receiving calls.

station A peripheral device of the switch. This is any piece of equipment connected to a switch via a phone line. Examples are telephone sets, a fax machine, a PC with an add-in telephony card, an answering machine.

switch Telephone switch. A piece of equipment capable of establishing telephone calls. Within the context of the API, a switch can be a PBX, a key system, or a central office.

TAPI application Any software that uses TAPI. The term “application” is used in its broadest sense possible; it need not necessarily be a user level program, but can also be a DLL that uses the API and provides higher level services via its interface.

TAPI Service Provider The conglomerate of software code (DLLs, device drivers, firmware) and hardware (add-on hardware, phone set, switch, network) that jointly implement the TAPI SPI.

SPI Service Provider Interface. The interface that a service provider must implement to make its telephony services available to applications via the API. The SPI is a collection of C language function definitions, message definitions, types, and data structure definitions, along with enough English verbiage to describe their meanings completely and unambiguously.

1 For ease of reading, “Windows” refers to the Microsoft Windows operating system. “Windows” is a trademark that refers only to this Microsoft product.