Create Communications Programs for Windows 95 with the Win32 Comm API

Charles Mirho and Andy Terrice

Charles Mirho specializes in multimedia and telephony at Intel Corp. He can be reached on CompuServe at 70563,2671. Andy Terrice also specializes in telephony at Intel, and may be contacted via CompuServe: 74364,771.

Click to open or copy the TTYCHIC project files

Microsoft has equipped Windows™ 95 with an updated communications architecture. Windows 95 provides APIs for PC-based e-mail, fax, file transfer, interactive terminals, office networking, and remote network access (logging into an office network from outside the office). Although most of these APIs were available in earlier versions of the Windows™ operating system, they have never been integrated to this degree with the operating system and each other.

Why should you write communications programs for Windows 95? Well, there are the obvious reasons—you can take advantage of its 32-bit interfaces, flat address space, and tighter kernel code. Preemptive multitasking means that communications programs can run with greater reliability, because other applications cannot hog the processor by doing things like reading and writing huge files in a tight loop. Processor-hogging applications can cause problems in earlier nonpreemptive versions of Windows because they wreak havoc with the timers and timeouts common in communications programs. Other, less obvious, reasons for using Windows 95 concern modem “sharing” and incoming call routing. We’ll get to these later on.

The framework for communications in Windows 95 is shown in **Figure 1**. This is not a pretty picture, but at least you can begin to appreciate the interdependence of the components that make up Windows 95 communications. Unlike earlier versions of Windows, all of these pieces come standard with the operating system. **Figure 1** also makes it clear why no one article can adequately explain all the details of Windows 95 communications, so we’ll focus on one central component, the Win32® Communications API (here, Win32 Comm for short).

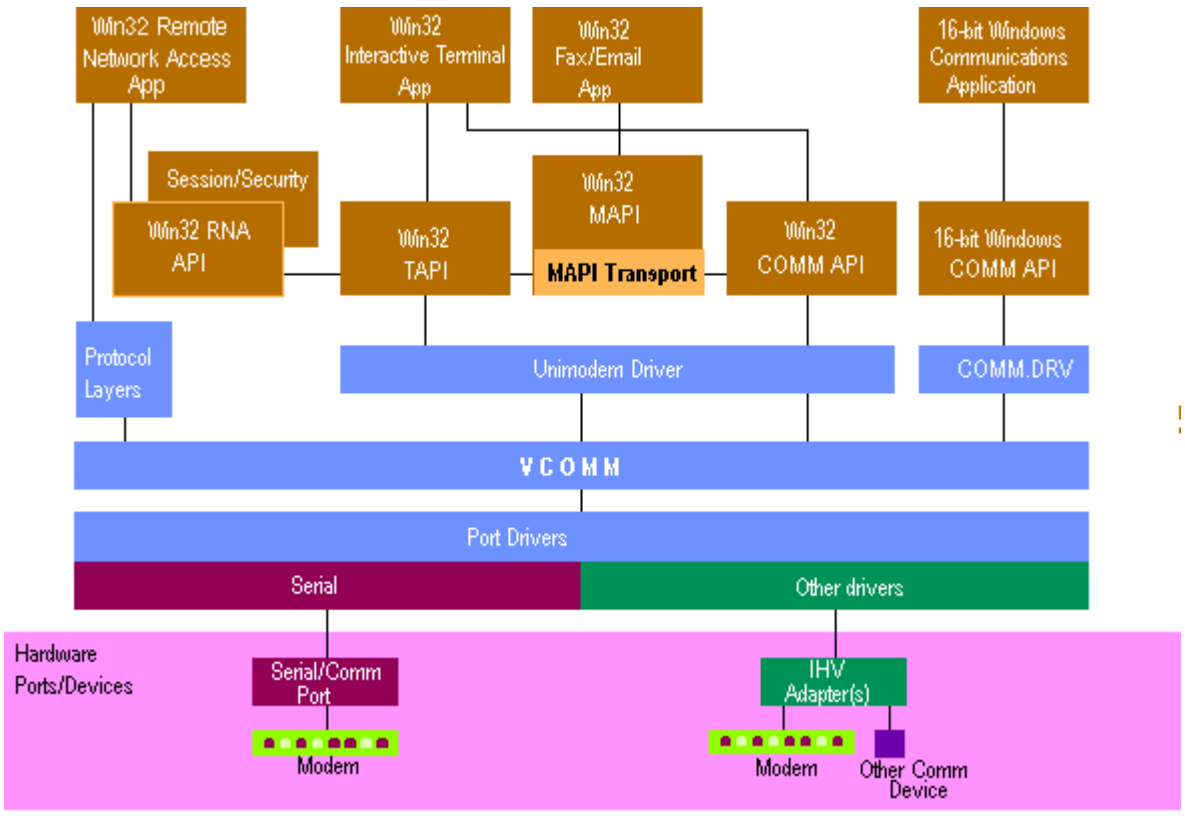

Figure 1 Windows 95 communications Architecture

Win32 Comm Uses

Generally, Win32 Comm should be used whenever the data will pass through a serial port or modem. Win32 Comm traces its roots to 16-bit Windows and the Windows 3.x COMM API, which is highly specialized for serial ports and modems. This means that Win32 Comm is not well suited for communications on a LAN. True, LANs often serialize data bits before sending them out on the wire just as a serial port does, but LANs use fewer wires than serial ports do, and LANs utilize access, routing, security, and error-correcting protocols that have little in common with serial port protocols. Even if you are communicating over a serial LAN (one made up of computers connected by their serial ports), Win32 Comm is not the way to go. Partly for historical reasons, partly for practical ones, Win32 Comm is just not good for LAN communications.

Win32 Comm won’t work for communicating with sound, either. For example, if you wanted to play a WAV sound file containing a recording of your voice over the phone line (for the greeting in a telephone answering machine application), you couldn’t use Win32 Comm. More generally, if you need to transfer data in real time, without the risk of potential delays, retransmissions, or interruptions, don’t use Win32 Comm. Win32 Comm does not implement the low-level ring-buffering mechanisms that real time data requires to be transmitted properly. To transmit or receive sound data in real time, use the WAVE API and a WAVE driver specially written to handle real-time communications. You will also need special hardware such as a modem that can handle voice data. Win32 Comm could be used to, say, download a sound file from CompuServe® and then play it using Sound Recorder (one of the Windows 95 applets). It is not the sound data per se that precludes Win32 Comm, but rather the requirement that it be transmitted in real time.

Ruling out LAN and sound-based communications still leaves a wide variety of applications for Win32 Comm. As mentioned earlier, anything that requires a serial port or modem is a candidate. This includes e-mail, Class I fax, file transfer, terminal emulators, and the front-ends for online services such as CompuServe. It also includes automation/control type applications that use the serial port to control an external piece of equipment, such as a laser-disc player. Remote network access shouldn’t be implemented with Win32 Comm even though it may use the serial port and a modem to connect to the remote network because of the issues surrounding LAN communications explained previously.

For faxes and e-mail, you can use a higher-level API. In Windows 95, the Messaging API (MAPI) utilizes the services of Win32 Comm at a low level. MAPI abstracts communications into messages between computers. MAPI’s communications specialty is store-and-forward; a message or document is created ahead of time, it is stored somewhere, and then forwarded (sent) to the remote computer, often in the background. It is easy to use MAPI for fax- and e-mail-based apps because many of the implementation details such as address books, spooling, and so on have been handled for you. If you write directly to Win32 Comm, you’ll need to do much more work, and you may actually interfere with other applications that do use MAPI, as we’ll explain a bit later on.

TAPI

No discussion about Windows 95 communications can go very far without mentioning the Telephony API (TAPI). In **Figure 1**, TAPI is in the middle of everything. Windows 95 uses TAPI to make and answer phone calls, period. It doesn’t matter what the application is—if it makes or answers phone calls, it uses TAPI. This is a very important point. Even if you write an application that calls MAPI or the Remote Network Access API (RNA), at a lower level you are using TAPI to make or answer phone calls.

Why is this important? Quite simply, because it marks a profound shift in the way telephone connections are established on the PC.

The Problem with AT Commands

Historically, the standard means for setting up and controlling phone calls with modems has been the Hayes® AT command set (standard TIA-602). (Fax calls also use AT commands, but we'll limit the discussion to modems.) An AT command is a string, prefixed by the letters AT, that has special meaning to a modem. A common AT command for dialing a phone number is ATDT. To dial 555-1234, you'd use the command

```
ATDT5551234
```

AT commands are written to the modem like data; they are interpreted as commands only when the modem is in command mode. Once the call is established, the modem switches out of command mode, so that data sequences that happen to correspond to AT commands do not cause undesirable behavior in the equipment. Communication applications use AT commands to make and answer phone calls, as well as to configure the communication hardware. Since different modem brands use different variants of AT commands, communication applications have to keep a database of which AT commands are used by which brands of modem. Often, applications simply query the user to enter the AT commands required to initialize their brand of modem. This makes using software more difficult than necessary.

In Windows 95, you can still open a serial port directly. If a modem is attached to the port, you can write AT commands directly through the port to configure the modem and set up phone calls. However, there is a better way.

The Windows 95 Control Panel contains an icon called Modems (see **Figure 2**). Microsoft has created a database with information on a large number of the modems on the market. By selecting a modem from the list, you tell Windows 95 which AT commands to use for configuring the modem and setting up calls. If your modem isn't on the supported list, most likely it is compatible with one that is (check your modem manual for compatibility). Select the brand of modem and close the applet. Now your applications don't need to know which AT commands work with your particular brand of modem, because Windows 95 will take care of it.

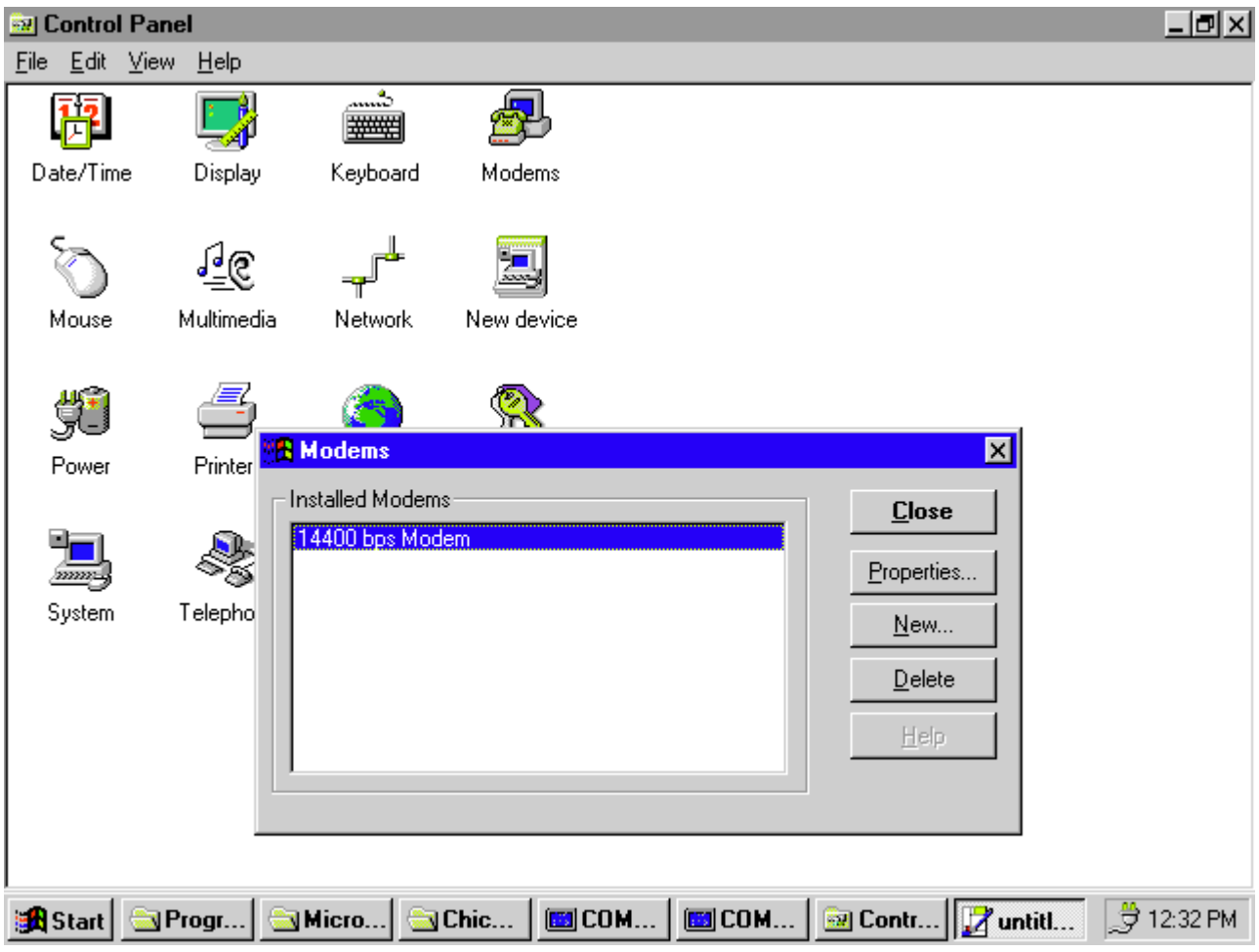

Figure 2 Windows 95 Modem Applet

So if AT commands are gone, what can an application use? TAPI. TAPI is "call central" in Windows 95. It provides hardware-independent functions for setting up and controlling phone calls and services for routing incoming calls to applications. An implementation layer directly below TAPI takes care of all the hardware-specific details for doing this. (For more information, see "Reach Out and Touch Someone's PC: The Windows Telephony API," by Charles Mirho and Andrew Raffman, MSJ, December 1993.) A hardware-specific Windows 95 layer called Unimodem handles the details of setting up calls using AT-style modems. It is Unimodem that accesses the Windows 95 modem database and determines which AT commands to use with the modem you selected from the Modems Control Panel applet.

The other advantage TAPI provides besides hardware abstraction is device sharing. Imagine the case where the applications in **Figure 1** are trying to use a single serial port with a single modem and phone line attached. Clearly all of the applications cannot use the port and modem at the same time, so TAPI manages the requests.

Say that two applications need to use the modem. Each must request the use of the phone line from TAPI. Application 1 requests permission to answer incoming calls. At the same time Application 2 requests permission to place an outgoing phone call. Using TAPI, the first application opens the phone line and requests that TAPI notify it of incoming calls from modems. Application 2 may still open the line because TAPI allows multiple applications to open the same line. Opening the line does not reserve the modem for exclusive use by the application, but merely expresses the application's interest in eventually using the modem. Even though Application 1 has the line open to answer calls, TAPI determines that the communications port associated with the phone line is not in use at this time and will allow Application 2 to place the call. After Application 2 finishes with the call the line is once again available to receive an incoming call. If an incoming call from a modem is then detected, TAPI will give this call to Application 1 to be answered.

It really isn't very complicated. TAPI just decides if an application really needs to use the phone line right away or if the application is just waiting around for something to happen. Apps can still wait for calls but they don't tie up the line doing it. This is an improvement over previous versions of Windows in which an application waiting for incoming calls would open the serial port and tie up the line indefinitely. Modem-sharing here does not mean that more than one application can actually use the modem at a time to transfer data; it means that the power to tie up the line while waiting for the phone to ring has been taken out of the application's hands, and moved down deep into the system (Unimodem can take exclusive control of the serial port so that other system components can't use it).

TAPI will also determine the best application for handling incoming calls. Say there are two applications that want to answer the phone when a call arrives. One application is a bulletin board server that lets remote users log in and browse files on the computer, and the other is a fax answering machine that will receive faxes and store them on the hard disk. When an incoming call arrives, TAPI routes the call to the first application waiting for calls, in this case the fax application. The fax application will determine if the call is truly a fax, and if not, it will hand off the call to the bulletin board application. The BBS never gets fax calls because it wouldn't know what to do with them.

Here is where things can get ugly. If the MAPI driver is installed, Unimodem will route all incoming calls to the MAPI driver before they go anywhere else. The MAPI driver will respond to an incoming call by sending certain AT commands to the modem to generate fax tones. If it's a fax modem, then it will respond to these AT commands by generating that high-pitched tone you love so well. If a fax machine is on the other end of the call, it will respond with a similar tone, and a fax call is established. The modem then reports back to Unimodem that the call is a fax call, Unimodem tells the MAPI driver, and the MAPI driver takes control of the call. This means the fax answering machine application we just discussed is incompatible with MAPI. In other words, because the MAPI driver gets first dibs on fax calls, the fax answering machine won't ever receive a call when MAPI is running. Conversely, a fax answering machine that bypassed TAPI and opened the port directly would

shut out MAPI from ever receiving calls. The bottom line is to watch out for MAPI when writing apps that answer incoming calls.

Since most of today's comm programs for Windows do not use TAPI, instead opening the serial port directly and sending AT commands directly to the modem, these applications will have the just-discussed compatibility problems when run on Windows 95 without modification. Hopefully, the vendors of these programs will take the effort to modify their products to use TAPI and MAPI.

Now that we've discussed the overall framework of Windows 95 communications, the role of TAPI, and some of the programming gotchas, let's get back on track with a discussion of Win32 Comm.

The Win32 Comm API

The Win32 Communications API is not really one API but actually two. There is one set of functions and structures dedicated to configuring and managing communications devices (primarily modems). There's also a second "overloaded" (to twist a C++ term somewhat) set of functions and structures for communications I/O. These overloaded functions and structures are the same ones used for file I/O in Windows 95, and they are used in almost exactly the same way. Therefore, if you know how to use the 32-bit file I/O functions, you are already well on your way toward understanding how to write a communications program with Win32 Comm.

TTYCHIC, the sample program we've provided with this article, is a modified version of the Windows 3.1 TTY sample program (complete source code can be found on any MSJ bulletin board). TTYCHIC has menu items for dialing a remote computer and connecting via modem. It will also answer incoming modem calls. Once a connection has been established, TTYCHIC will allow you to type interactively in terminal mode or send an ASCII text file to the remote machine. It can also receive small text files (actually any size text file will work, but only the last 2000 bytes received are displayed on the screen). TTYCHIC allows you to configure the preferences of the screen, such as whether to use local echo when sending characters, whether to absorb line feeds, and what type of font to use when displaying characters.

Before running TTYCHIC on your Windows 95 beta, you must configure your communications hardware for Windows 95 with the Modems applet. In this applet (if your modem supports Plug and Play—recommended for any Windows 95-compatible modem) you simply select automatic detection of the modem type. If auto-detection doesn't work, you can manually select your modem. Choose the serial port to which the modem is connected, and configure the port with its default settings of baud rate, parity, and so on. Basic modem configuration consists of identifying the maximum baud rate and the speaker volume. Advanced configuration dialog boxes are provided that allow you to view and modify modem details, but this is not normally required. If everything goes smoothly you should see (for external modems) the LEDs flashing as the Modems applet tests the configuration. Then go to the Telephony icon and make sure that Unimodem is installed, as shown in **Figure 3** (you may have to remove and then reinstall it on certain May 1994 beta Windows 95 systems). You can also use the Telephony applet to configure the calling location and calling card information. A dialog box will prompt you for information about your current location, area code, your calling card number and any special dialing prefixes required.

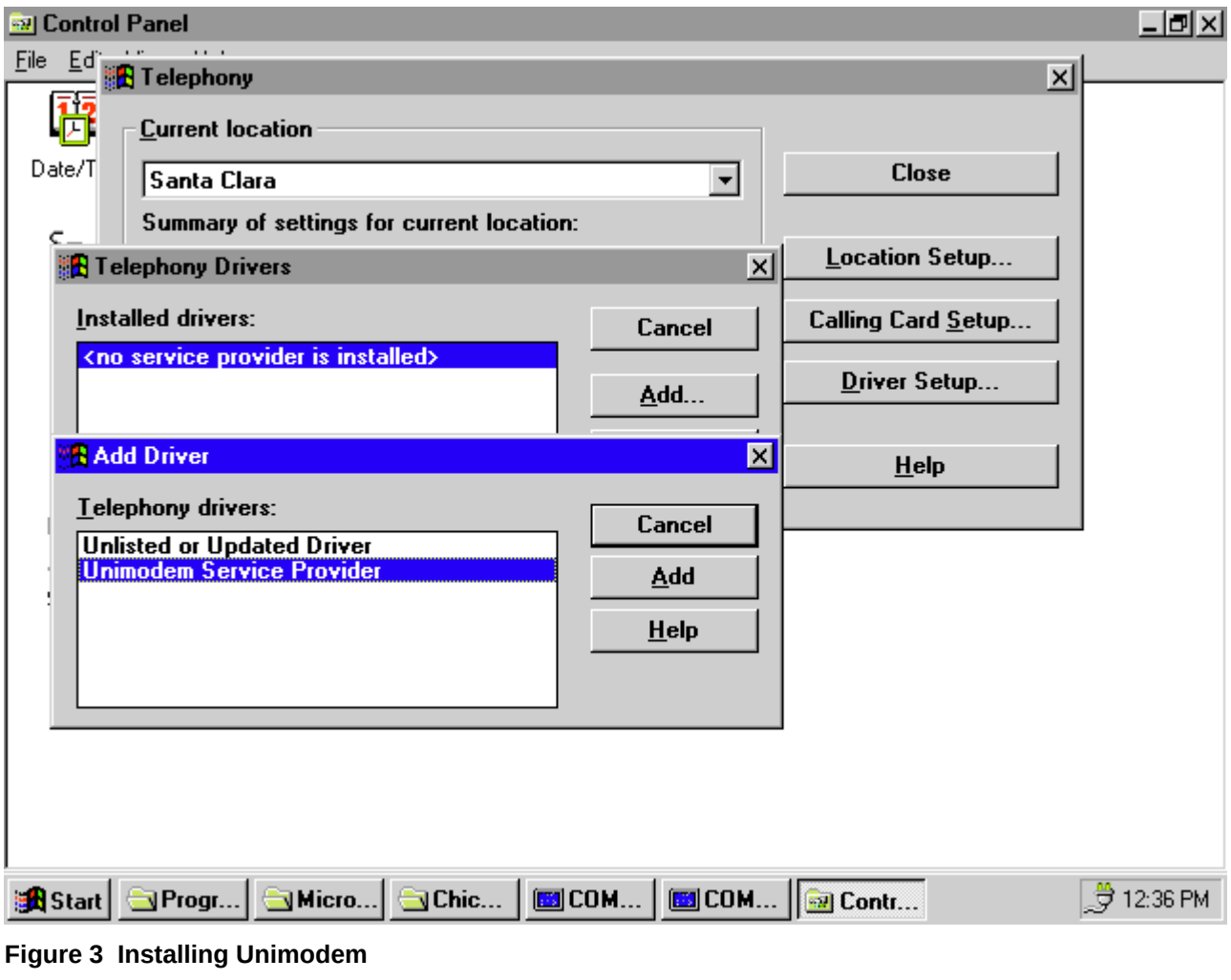

Figure 3 Installing Unimodem

The Modem Control Panel applet uses a special set of functions and data structures to configure the modem and port. Your application can call these same functions to do the configuration internally. See "Configuring a Modem Under Windows 95," page 80, for details of how to configure the modem and port from within an application.

To run TTYCHIC, turn on your modem and then select the TTYCHIC icon. Run TTYCHIC or another communications application (such as the Windows 3.1 Terminal applet or the Windows 95 HyperTerminal accessory) on a second computer, then have one application dial the other (see **Figure 4**). TTYCHIC will answer incoming calls automatically and place outgoing calls using TAPI. For outgoing calls, you must, of course, enter the phone number to call. To exchange text interactively, simply type. Use the Configure menu option to configure TTYCHIC to accept line feeds and do local echo.

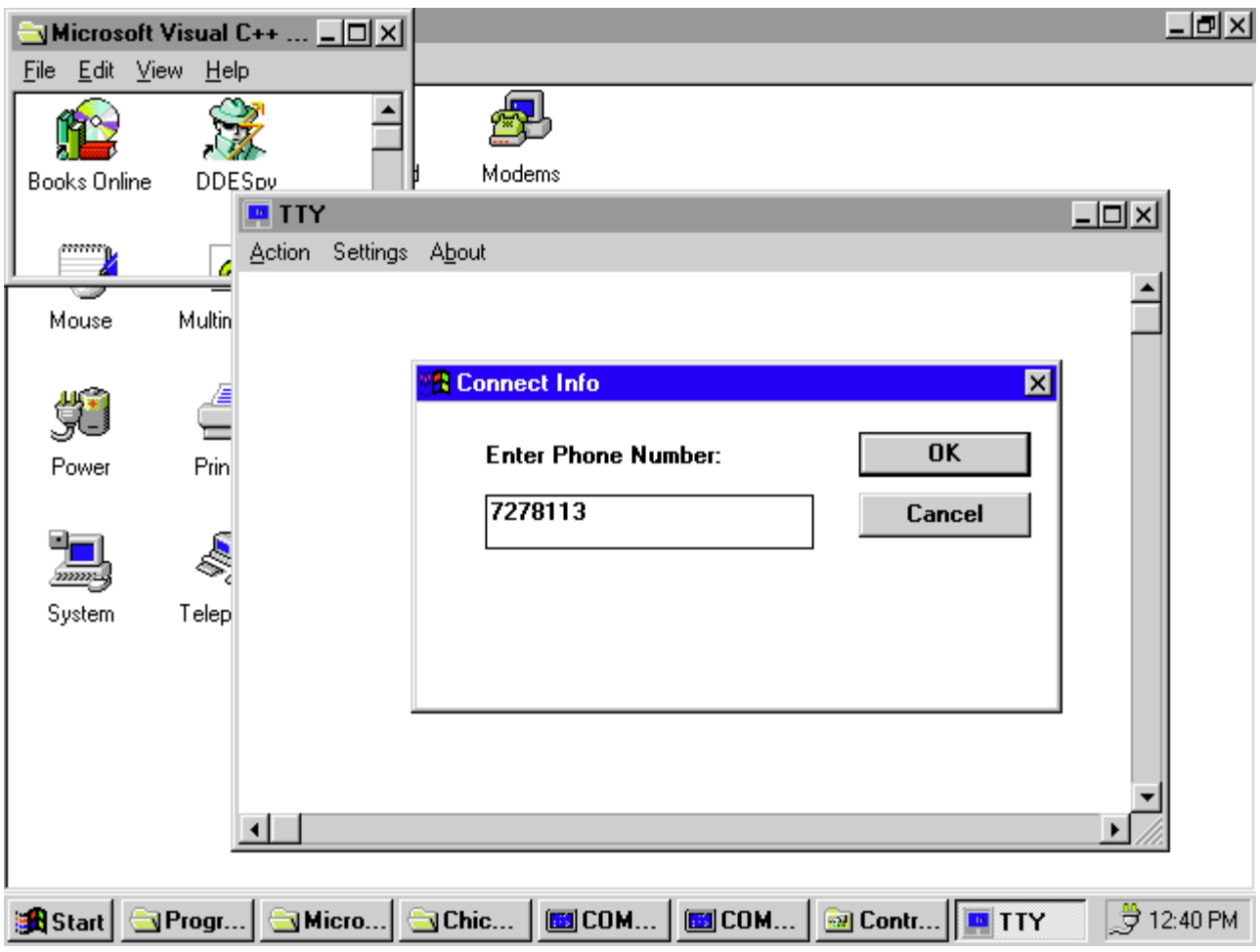

Figure 4 TTYCHIC

To send a file, select Send a File from the Settings menu. Enter the file name in the dialog box and click OK. The file will be sent, and if local echo is on, you will see the result onscreen. The file must be ASCII text and you must be connected to a remote computer to select this option. In TTYCHIC, you know that you are in the Connected state when the Disconnect menu selection under the Action menu is highlighted. Clicking on Disconnect when it is highlighted causes the call to hang up. Also, be sure your communications settings match on both machines by setting them using the Configure menu option.

One note: we mention TAPI often in the description of TTYCHIC. Don't be alarmed. While it is true that you need to understand TAPI to fully understand the sample code, we're focusing on the data transfer aspect of the application, which involves Win32 Comm, not TAPI. TAPI is only involved in placing, answering, and dropping calls, not data transfer or the configuration of modems and ports.

Here's how TTYCHIC works. First, it uses TAPI to establish a call. After TAPI has informed TTYCHIC that a connection has been established, TTYCHIC uses Win32 Comm to send or receive characters from the remote computer over the phone line. When the user is finished using the phone line TTYCHIC calls TAPI again to hang up the call. If the called party hangs up first, TAPI sends TTYCHIC a message notifying it that the call has been dropped.

Getting Down to Business with Win32 Comm

To read or write bytes from the modem, an application must obtain the handle that represents the modem. It gets this handle from TAPI, which acquires it from Unimodem. Unimodem obtains it by opening the serial port using the Win32 Comm CreateFile function. That's right, to use Win32 Comm to read and write data to a modem, you must call TAPI, which calls Unimodem, which turns around and uses Win32 Comm to create a handle to the modem! The reason for this apparently convoluted mechanism for obtaining a modem handle will become apparent shortly.

So how does the application actually obtain the handle from TAPI? The answer lies in the TAPI function lineGetID. Once a call has been established (either by dialing it from the application or answering an incoming call), use lineGetID to obtain a handle to the modem associated with the call. You must pass the TAPI handle for the call to lineGetID. TAPI then returns, through lineGetID, a handle for the modem that it received from Unimodem. It is this handle that the application uses in virtually all of the Win32 Comm functions.

```
LONG lineGetID(hLine, dwAddressID, hCall, dwSelect,
               lpDeviceID, lpszDeviceClass);
```

Remember, the call between the two computers must already exist before calling lineGetID. The first two parameters are not needed if the call already exists and should be set to NULL and 0, respectively. The third parameter is the TAPI handle to the already existing call. The fourth parameter, dwSelect, tells lineGetID which of the first three parameters is valid. Setting this parameter to LINECALLSELECT_CALL tells lineGetID to look at the third parameter only and to ignore the first two. The fourth parameter is a pointer to the structure in which the modem identifier will be returned. The last parameter tells lineGetID the format to use for the modem identifier it returns. More on this in a moment.

Using lineGetID to identify a modem is not unique to Win32 Comm. In general, any Windows 95 data stream API will call lineGetID when it needs an identifier for a modem. (A data stream API is any API whose primary purpose is moving data around. The WAVE API for real-time sound is another example.) The modem identifier returned by lineGetID is API-specific; it has to be because different data stream APIs use different formats for device identification. If instead of sending and receiving characters, which requires a handle compatible with Win32 Comm, you wanted to play a WAVE file over the phone line, you would request a modem identifier in the format of the WAVE API (be aware, however, that Unimodem does not support the streaming of real-time sound data over the phone line, so you would need to replace or supplement Unimodem to obtain this functionality). The WAVE API uses a simple numeric identifier (such as 0, 1, ... n) to identify different wave devices, not a handle as Win32 Comm does. When you call lineGetID, the last parameter is a string that identifies the API format you want the modem identifier returned in. For the WAVE API, this parameter is set to "wave," and for Win32 Comm it is set to "comm/datamodem".

The fifth parameter to lineGetID is a structure of type VARSTRING. A VARSTRING structure does not contain any useful data itself; rather, it is used as a flexible container for other structures.

```
typedef struct varstring_tag {
    DWORD    dwTotalSize;
    DWORD    dwNeededSize;
    DWORD    dwUsedSize;
    DWORD    dwStringFormat;
    DWORD    dwStringSize;
    DWORD    dwStringOffset;
} VARSTRING, FAR *LPVARSTRING;
```

The dwTotalSize field contains the total size of the container, which will typically be sizeof(VARSTRING) + x, where x is the number of extra bytes needed by lineGetID to contain all of the desired information. You can figure out what x should be by calling lineGetID with dwTotalSize set to sizeof(VARSTRING). lineGetID will return with the dwNeededSize field set to the actual number of bytes needed to hold the desired information. The value of x is the difference between size of(VARSTRING) and dwNeededSize. Don't bother with the dwUsedSize field; it's

supposed to contain the number of bytes in the container actually holding useful information, but we've never discovered any use for this information once dwNeededSize is known.

The dwStringFormat field indicates the format of the extended information returned in the container (the x bytes). For Win32 Comm this will be set to the constant STRINGFORMAT_BINARY to indicate that the extended information in the container is binary data (as opposed to ASCII strings). Finally, the dwStringSize and dwStringOffset are the size and the offset of the extended information in the container. The offset is always relative to the first byte in the VARSTRING container structure.

All of this information should become clearer in the following example. To obtain the Win32 Comm modem handle and modem name from lineGetID, use the code shown in **Figure 5**.

Figure 5 Obtaining the Modem Handle and Name

```
/* define structure returned by UniModem which contains modem handle and modem ?* name */
typedef struct tagCommID {
    HANDLE hComm;
    char szDeviceName[1];
} CommID;

HANDLE GetCommHandle (HCALL htCall, char *szDeviceName)
{
CommID FAR *cid;
VARSTRING  *vs;
LONG lrc;
DWORD dwSize;

    vs = (VARSTRING *) malloc (sizeof(VARSTRING));
    vs->dwTotalSize = sizeof(VARSTRING);
    vs->dwStringFormat = STRINGFORMAT_BINARY;

    do {
        /* get modem handle associated with the call
         -  the call handle
        came from TAPI after making or answering a call */
        if ( ((lrc = lineGetID(NULL, 0L, htCall,
             LINECALLSELECT_CALL, vs, "comm/datamodem")) = =
             LINEERR_STRUCTURETOOSMALL)
                 || (vs->dwTotalSize < vs->dwNeededSize)) {
            dwSize = (lrc  = = LINEERR_STRUCTURETOOSMALL) ? (1000) :
                      (vs-> dwNeededSize);
            free (vs);
            vs = (VARSTRING *) malloc(dwSize);
            vs->dwTotalSize = dwSize;
        } /* end if (need more space) */
        else if (lrc < 0) {
            /* handle error */
        } /* end if (error getting comm device handle) */
        else
            break; /* success  */
    } while (TRUE);

    cid = (CommID FAR *) ((LPSTR)vs + vs->dwStringOffset);
    lstrcpy (szDeviceName, &cid->szDeviceName[0]);

    return cid->hComm;

} /* end function (GetCommHandle) */
```

In **Figure 5**, hComm contains a modem handle that can be used with any of the Win32 Comm functions. szDeviceName points to a null-terminated ASCII string that contains the name of the modem. If hComm is returned with a NULL value, it could mean several things. For one, the dwSelect parameter used in lineGetID was not set to LINECALLSELECT_CALL. Another reason could be that Unimodem did not have a serial port already opened when lineGetID was called. Neither of these cases should occur if you are careful to call lineGetID only after a call has been established. If hComm is NULL, the application can still recover by calling the Win32 Comm CreateFile function along with the string returned by lineGetID in szDeviceName to obtain the modem handle. This string will be the name of the modem you selected from the modem list in the Modems Control Panel applet. You should write your application to handle either case, since Unimodem may not always be present if a third-party hardware vendor has replaced it with its own telephone driver. Third-party telephone drivers may return a NULL hComm. Unimodem will almost always be present, but since there is a small chance it won't, let us explain the CreateFile function so that your applications can call it as a backup procedure.

CreateFile is a complex function. One reason is that is used for so many different but related operations. CreateFile can be used to open existing files, create new files, and to open devices that aren't files at all. Let's use it to open a modem.

```
CreateFile (szDevice, fdwAccess, fdwShareMode, lpsa,
            fdwCreate, fdwAttrsAndFlags, hTemplateFile);
```

The first parameter is the logical name of the modem device. Use the szDeviceName in the VARSTRING structure returned by lineGetID. The second parameter, fdwAccess, is a bitflag specifying the type of access to the modem. Just as files can, the modem can be opened for reading or writing or both. Setting this field to GENERIC_READ opens for reading, setting it to GENERIC_WRITE opens the modem for writing, and ORing the two flags opens for read/write access. Virtually all modem communication is bidirectional, so the most common setting is:

```
fdwAccess = GENERIC_READ | GENERIC_WRITE;
```

The third parameter, fdwShareMode, specifies the sharing attributes of the modem. This flag is used for files that may be opened by multiple processes simultaneously. A modem cannot be opened in shared mode by applications, so set this parameter to zero. If the modem is being used by another process when CreateFile is called, the function returns an error. However, multiple threads of the same process may open the modem multiple times, and the modem handle returned by CreateFile may be inherited by child processes of the process that opened the modem just as file handles may be inherited.

The fourth parameter is interesting. lpsa references a security attributes structure that defines attributes such as how the modem handle returned by CreateFile may be inherited by children of the creating process. Setting this parameter to NULL will cause Windows 95 to assign the default security attributes, which means the modem handle cannot be inherited.

The fifth parameter of CreateFile is another bitflag. These flags indicate what to do if the file or device already exists. For regular files, these flags can specify that the function fail if the file already exists or overwrite existing files. For devices like modems, this flag must always be set to OPEN_EXISTING, which tells Windows 95 to proceed without an error when the modem already exists, which of course it does. (If you invent a function call that can create modems, please call us. We would like to invest in your company.)

The sixth parameter contains still more flags describing various attributes of the thing being opened. For files, many attributes are possible, but for modems the only attribute of interest is FILE_FLAG_OVERLAPPED. The overlapped flag indicates whether I/O to the modem should proceed in the background. We'll get to background I/O later on.

The last parameter is a handle to a template to use with the open device. For files, this parameter can be used to specify additional

parameters. For modems, this parameter must be set to NULL.

You may never need to use CreateFile for your comm programs. Only use CreateFile when lineGetID returns a NULL modem device handle. When using CreateFile to open a modem, remember that the following conventions should be followed:

- The sharing mode flags must be set to zero for exclusive access to the modem.
- The creating flags must be set to OPEN_EXISTING.
- The handle to the template must be set to NULL.

Fortunately, closing a modem is a lot simpler than opening one. Simply call the general-purpose CloseHandle function with the handle returned by lineGetID or CreateFile.

```
CloseHandle (hComm):
```

You will need to close the handle even if Unimodem opened the modem for you. If you forget to close the handle, the modem stays “open” and Unimodem will not be able to access it the next time any app wants to answer or place an outbound call.

The Polling Method

Once you have obtained a modem handle you can begin to read and write data to and from the modem using the Win32 Comm I/O functions. These are the same ones used for file I/O. The first method we’ll examine for reading and writing bytes involves polling.

When polling, a thread continually checks the status of the modem for the arrival or departure of bytes. This takes up a great deal of the CPU’s time. The advantage is that it is relatively straightforward to implement; you dedicate a single thread to polling the modem.

Figure 6 shows how to read bytes from the modem using polling. The program polls the modem to determine if bytes are waiting in the receive queue. If there are bytes waiting, all of the waiting bytes are read up to the maximum number. The read is blocking; that is, the read function does not return until all of the bytes are read.

Figure 6 Reading Modem Data via Polling

```
   // From WinMain...
   while (TRUE) {
        /* if there is a message ... */
        if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) {
                // Acquire and dispatch messages until
                // a WM_QUIT message is received.
                if (msg.message = = WM_QUIT)
                    break;
                else if (!TranslateAccelerator (msg.hwnd,
                            ghAccel, &msg)) {
                    TranslateMessage(&msg);// Translates virtual key codes
                     DispatchMessage(&msg); // Dispatches message to window
                 }
         }
         else {
              /* read bytes if session connected */
              if (SESSIONCONNECTED()) {
              BYTE abyData[MAXREADSIZE];
              DWORD nBytes;
              int i;
                  if (ReadBytesFromModem(&abyData[0], &nBytes) = = 0)    {
                    //ProcessTTYCharacter( hTTYWnd, abyData[i]));
                    if (nBytes)
                        WriteTTYBlock(hTTYWnd, &abyData[0], nBytes ) ;
                } /* end if (no error reading bytes) */
              }  /* end if (connected inbound call) */
       } /* end if (no message pending) */
    } /* end while (TRUE) */
o
o
o
/* Reading With Polling */
DWORD ReadBytesFromModem(BYTE *abyBuffer, DWORD *pnBytes)
{
DWORD dwErrorMask;        /* mask of error bits */
COMSTAT comstat;          /* status structure */
DWORD nToRead;            /* number of bytes to read */
    *pnBytes = 0;        /* zero read size */
    /* get status of read and write queues */
    ClearCommError (SESSIONDEVICEHANDLE(), &dwErrorMask, &comstat);
    /* check error flags */
    if (dwErrorMask) {    /* handle error */
        return dwErrorMask;
    } /* end if (error) */

    /* if too many bytes to read, just read the maximum */
    if (comstat.cbInQue > MAXREADSIZE)
        nToRead = MAXREADSIZE;
    else
        nToRead = comstat.cbInQue; /* otherwise read them all */

    /* if no bytes to read, return */
    if (nToRead = = 0)
        return 0L;

    /* read the bytes */
    if (!ReadFile(SESSIONDEVICEHANDLE(), abyBuffer,
            nToRead, pnBytes, NULL)) {
        return GetLastError ();
    } /* end if (error reading bytes) */

    return 0L;
} /* end function (ReadBytesFromModem) */
```

The two important functions in the ReadBytesFromModem function in **Figure 6** are ClearCommError and ReadFile. ClearCommError returns the current status of the communications device, and indicates whether an error has occurred. Like almost all functions involving communications (there is one exception), it takes as a parameter the modem handle returned from Unimodem by lineGetID or CreateFile. It also takes a reference to a COMSTAT structure, which has fields for returning the current status of the communication device.

```
typedef struct _COMSTAT {
  DWORD fCtsHold : 1;   /* Tx waiting for CTS signal */
  DWORD fDsrHold : 1;   /* Tx waiting for DSR signal */
  DWORD fRlsHold : 1;   /* Tx waiting for RLSD signal */
  DWORD fXoffHold : 1;  /* Tx waiting, XOFF char
                           received */
  DWORD fXoffSent : 1;  /* Tx waiting, XOFF char sent */
  DWORD fEof : 1;       /* EOF character sent */
  DWORD fTxim : 1;      /* character waiting for Tx */
  DWORD fReserved : 25; /* reserved */
  DWORD cbInQue;        /* bytes in input buffer */
  DWORD cbOutQue;       /* bytes in output buffer */
} COMSTAT, *LPCOMSTAT;
```

A close look at this structure reveals just how specific ClearCommError is to serial communications and RS-232 in particular. The first three fields (actually bits in a bitfield) are flags that indicate the status of the signals Clear-To-Send, Data-Set-Ready, and Carrier Detect. A value of 1 indicates that these signal lines are set. The next two bitfields indicate whether the data transfer is suspended due to XOFF being sent or received. The two fields after that indicate whether an EOF character has been sent and whether there are characters waiting in the transmit buffer.

The next two fields are the most useful to us in this example. The cbInQue and cbOutQue fields specify the number of bytes waiting in the I/O buffers to be read or transmitted. TTYCHIC checks the value of cbInQue; if the number of bytes received is less than the maximum, it reads all the bytes, otherwise it ends at the maximum.

Reading the bytes involves a call to ReadFile, which like CreateFile is a function that can be used with either files or communication devices. The modem handle returned by CreateFile is passed to this function, along with a buffer in which to put the received characters. The function returns the actual number of bytes read from the receive queue. Notice the last parameter, set to NULL. We pass a special structure here to make ReadFile execute in the background, a topic we'll discuss later. For now just understand that the NULL causes the function not to return until all of the requested bytes are read (there is yet another way to control when the function returns, using timeouts, which is covered next).

The code fragment in **Figure 6** has some big problems. What if the requested number of bytes never arrives? What if the connection drops in the middle of reading bytes, or if the remote party simply stops transmitting? You then have a thread that's blocked indefinitely. Fortunately you can specify how long the read function should wait before returning even if the requested number of bytes never arrive. Just use timeouts. Use the SetCommTimeouts function to specify how long the read should wait before giving up.

```
SetCommTimeouts (hComm, &timeouts);
```

The second parameter references a COMMTIMEOUTS structure with the following fields.

```
typedef struct _COMMTIMEOUTS {
    DWORD ReadIntervalTimeout;
    DWORD ReadTotalTimeoutMultiplier;
    DWORD ReadTotalTimeoutConstant;
    DWORD WriteTotalTimeoutMultiplier;
    DWORD WriteTotalTimeoutConstant;
} COMMTIMEOUTS;
```

Windows 95 recognizes two types of communication timeouts. The interval or intercharacter timeout applies to read operations only. The interval timeout occurs when too much time elapses between the receipt of two characters. A timer is started when the first character is received and this timer is reset each time a new character comes in. If the timer exceeds a specified threshold, the timeout occurs. The second type of timeout applies to both read and write operations. This is the total or accumulated timeout, which occurs when the total time required to read or write many (more than one) bytes exceeds a threshold. Read operations may have both interval and total timeouts set simultaneously.

The interval timeout for read operations is specified, in milliseconds, in the field ReadIntervalTimeout. You can also specify how long the function should wait before giving up. As you may have noticed from examining the structure, these total timeouts cannot be specified directly. Rather, the total timeouts are specified in two parts, using a constant and a multiplier. The total timeout period is computed from these values:

```
ReadTotalTimeout =
          (ReadTotalTimeoutMultiplier * bytes_to_read) +
           ReadTotalTimeoutConstant;
WriteTotalTimeout =
        (WriteTotalTimeoutMultiplier * bytes_to_write) +
         WriteTotalTimeoutConstant;
```

Using these equations instead of a fixed value for the total timeout allows the timeout value to vary with the number of bytes to read or write. As the number of bytes to read or write increases, the total timeout values also increase. As the number of bytes decreases, the total timeout values decrease until they are equal to a multiplier + constant for just one byte. Thus the constant sets a lower range beyond which the total timeout value cannot fall. Either the multiplier or the constant can be zero. If the multiplier is zero, then the total timeout is equal to the constant value in milliseconds. In this case the total timeout will not increase with larger and larger block sizes, so that you run the risk of timing out before large blocks are read or written. If the constant is zero, then the total timeout increases with larger packet sizes, but the lowest value of the total timeout shrinks to 1 5 multiplier, for one-byte reads and writes. The constant and multiplier may both be zero, in which case Windows 95 does not attempt to apply a total timeout period to reads and writes.

Windows 95 allows all timeout values to be set to zero. This is in fact the default setting for devices that have never before had their timeouts configured. However, it is very dangerous to set these values to zero if you are performing any reads or writes. If timeouts are not used, a read operation will not complete until all the requested bytes have been read; if the connection is bad, this could take a very long time, like forever. The same is true for write operations when timeouts are not set.

Suppose that you want to poll the device for incoming characters; if no characters are waiting, you want to return immediately to your program to continue execution. Furthermore, you don't want to worry about timeouts or querying the level of the receive queue. You just want to read whatever characters are waiting in the receive buffer, if any, and then either process the characters or continue on. Windows 95 provides a way to poll like this.

```
COMMTIMEOUTS cto;
o
o
o
cto.ReadIntervalTimeout = MAXDWORD;
cto.ReadTotalTimeoutMultiplier = cto.ReadTotalTimeoutConstant = 0;
SetCommTimeouts(hComm, &cto);
o
o
o
ReadFile (hComm, &inbuf, nBytesToRead,
          &nBytesActuallyRead, NULL);
```

Notice that the ReadIntervalTimeout value is set to the magic value MAXDWORD. The read timeout multiplier and constant are both set to 0. Those familiar with the 16-bit Windows COM port API will be pleased to know that this configuration duplicates the functionality of the ReadComm function in that API.

To retrieve the current timeout settings, call GetCommTimeouts. The current timeout settings will be read into a COMMTIMEOUTS structure that you pass to the function. You can then change only the timeout settings that interest you and then write the modified settings to the

modem using SetCommTimeouts.

```
GetCommTimeouts (hComm, &timeouts);
```

What about writing bytes? Writing is similar to reading, except that you're concerned with keeping the output buffer from going empty rather than keeping the input buffer from overflowing. The next function, WriteBytesToModem, writes out a block of data of a given size (see **Figure 7**). It tries to keep the output buffer level above a minimum threshold. The function does not return until the entire block of data is written to the transmit queue.

Figure 7 Writing a Block of Data

```
//from WM_CHAR processing and SendFile processing...
/* echo to local window if echo turned on */
dwCount = 0;
do {
    dwWriteSize = min (MAXTXSIZE, dwFileSize);
    WriteTTYBlock(hWnd, (LPBYTE)&pBuffer[dwCount], dwWriteSize);

    if (SESSIONCONNECTED()) {
        if (WriteBytesToModem((LPBYTE)&pBuffer[dwCount],
                    wWriteSize) != 0)    {
            /* handle error */
        } /* end if (error writing bytes) */
    }  /* end if (connected) */

    dwCount += dwWriteSize;
    dwFileSize -= dwWriteSize;

} while (dwFileSize);
o
o
o
/* write a potentially large block of data to the modem */
int WriteBytesToModem(BYTE *abyByteBuffer, DWORD nToWrite)
{
DWORD dwErrorMask;        /* mask of error bits */
COMSTAT comstat;           /* status structure */
DWORD nActualWrite;        /* actual num bytes written */

    do {
        /* get status TX queue */
        ClearCommError (SESSIONDEVICEHANDLE(), &dwErrorMask, &comstat);
        if(dwErrorMask) return -1;
        /* if we are under the minimum threshold of the TX queue */
        if (comstat.cbOutQue < LOWWATER) {
            /* if there is room, write 'em all */
            if (LOWWATER+ nToWrite <= MAXTXSIZE) {
                WriteFile (SESSIONDEVICEHANDLE(),
                           abyByteBuffer, nToWrite,
                           &nActualWrite, NULL);
                nToWrite -= nActualWrite;
                (BYTE *)abyByteBuffer += nActualWrite;
            } /* end if (room for all the bytes in TX buf) */
            /* if too many bytes to write at once,
                     just fill TX buf and loop */
            if (LOWWATER + nToWrite > MAXTXSIZE){
                WriteFile (SESSIONDEVICEHANDLE(), abyByteBuffer,
                    MAXTXSIZE-comstat.cbOutQue,
                    &nActualWrite, NULL);
                nToWrite -= nActualWrite;
                (BYTE *)abyByteBuffer += nActualWrite;
            } /* end if (can't fit all the bytes in the TX queue) */
        } /* end if (bytes in output queue below minimum level) */
    } while (nToWrite);

    return 0;

} /* end function (WriteBytesToModem) *
```

The function immediately falls into a loop that checks the current level of the transmit queue. If the number of bytes in the queue has not fallen below the minimum threshold (LOWWATER), the function continues to poll the queue until the minimum level is breached. Once the minimum level is breached, the function checks if there is room in the queue for all the bytes to write, and if there is, all the bytes are written. The byte count decrements to zero, and the loop terminates. If there isn't room for all the bytes, the function writes as many as it can and then loops for the next time. Eventually all of the bytes are written and the loop terminates.

Why wait for the transmit queue to fall below some minimum threshold? Why not just stuff as many bytes as will fit into the transmit queue with each iteration of the loop until there are no more bytes? Actually, either method would work just fine, but I suspect there might be critical code somewhere inside WriteFile and prefer to call it less frequently.

Events

Let's look at a different way to read and write bytes. Events occur when something significant happens. It could be a change in the state of a hardware line like CTS or DSR. It could be the detection of a ring on the line. It could be an error, the receipt of a special character, or any character. An event could also occur when the transmit buffer becomes empty. Using events is a less CPU-intensive method than polling. In Windows 3.1, events from the modem result in a WM_COMMNOTIFY message being sent to the application's window function. In Windows 95, the event will cause a thread to wake up and service the event.

This last case is the one that interests us. If we wait for an event to tell us when it is time to read or write bytes, then the thread does not waste time polling the status of the device.

To specify which events to wait for, use SetCommMask.

```
SetCommMask (hComm, fdwEventMask);
```

fdwEventMask specifies the events to monitor for. To tell Windows 95 to generate an event whenever a character is received, use

```
fdwEventMask = EV_RXCHAR;
```

To have Windows 95 generate an event whenever the last byte in the transmit buffer is sent, use

```
fdwEventMask = EV_TXEMPTY;
```

You can OR these flags to receive events for both conditions.

```
fdwEventMask = EV_RXCHAR | EV_TXEMPTY;
```

You should call the function first to retrieve the current settings of the event mask. This function returns the current event mask. Then you can OR your own flags into the event mask and call SetCommMask to add the new events. This way, calling SetCommMask does not have the unexpected side effect of clearing events that were already in effect.

```
//get current event mask
GetCommMask (hComm, &fdwEventMask);
 //add new events
fdwEventMask |= EV_RXCHAR | EV_TXEMPTY;
  //set new events
SetCommMask (hComm, fdwEventMask);
```

Windows 95 can also generate an event when the number of bytes in the receive buffer exceeds a certain threshold (commonly called a high-water mark), or falls below a certain threshold (low-water mark). This is done by specifying hardware flow control and trapping the events for a changed CTS state.

Once you have called SetCommMask to set the event mask, you can wait for one of the events to occur by calling WaitCommEvent.

```
WaitCommEvent (hComm, &fdwEventMask, NULL);
```

The thread is now waiting efficiently (consuming very few CPU cycles) for one of the events in the event mask to occur. When one or more events occur, the function returns. Read the fdwEventMask parameter to determine which of the events that you were waiting for occurred.

The third parameter, which we have set to NULL, can be used to make WaitCommEvent execute in the background.

The next example shows the polling method in TTYCHIC modified to the event-driven method for both reading and writing bytes (see **Figure 8**).

Figure 8 Reading and Writing with Events

```
BOOL SetupEvents (HANDLE hComm)
{ DWORD fdwEventMask;
    if (!GetCommMask (hComm, &fdwEventMask)) //get current event mask
        return FALSE;
    fdwEventMask |= EV_RXCHAR | EV_TXEMPTY;        //add new events
    if (!SetCommMask (hComm, fdwEventMask))    //set new events
        return FALSE;
    return TRUE;
} /* end function (SetupEvents) */

LPTHREAD_START_ROUTINE ReadThread (LPDWORD lpdwParam1)
{  BYTE abyData[MAXREADSIZE];
   DWORD nBytes;

    while (dwThreadControl & READ_THREAD) {
        if (ReadEventBytesFromModem(&abyData[0], &nBytes) = = 0) {
            if (nBytes)
                WriteTTYBlock(hTTYWnd, &abyData[0], nBytes ) ;  }
        else { /* handle error */  }
    } /* end while (read thread active) */
    PurgeComm(SESSIONDEVICEHANDLE(), PURGE_RXCLEAR);
    ExitThread (0);
} /* end function (ReadThread) */

/* read bytes when a character is recieved */
int ReadEventBytesFromModem(BYTE *abyByteBuffer, DWORD *pnNumBytes)
{  DWORD dwErrorMask;    /* mask of error bits */
   COMSTAT comstat;      /* status structure */
   DWORD nToRead;        /* number of bytes to read */
   DWORD fdwEventMask;   /* event mask returned by WaitCommEvent */

    *pnNumBytes = 0;     /* zero read size */

    if (!SESSIONDEVICEHANDLE())
        return 0;
    if (!WaitCommEvent(SESSIONDEVICEHANDLE(), &fdwEventMask, NULL)){
        return -1;   }   /* end if (error waiting for event) */

    if (fdwEventMask & EV_RXCHAR) {
        ClearCommError (SESSIONDEVICEHANDLE(), &dwErrorMask, &comstat);
        /* check error flags */
        if (dwErrorMask) {
            /* handle error */
            return -1; } /* end if (error) */

        /* if too many bytes to read, just read the maximum */
        if (comstat.cbInQue > MAXREADSIZE)
            nToRead = MAXREADSIZE;
        else
            nToRead = comstat.cbInQue; /* otherwise read them all */

        /* read the bytes */
        ReadFile(SESSIONDEVICEHANDLE(), abyByteBuffer, nToRead, pnNumBytes, NULL);
    } /* end if (received data event) */

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


/* write bytes when the queue goes empty */
int WriteEventBytesToModem(BYTE *abyByteBuffer, DWORD nToWrite)
{  DWORD nActualWrite;    /* actual num bytes written */
   DWORD nOut;            /* bytes to write this iteration */
   DWORD fdwEventMask;    /* event mask returned by WaitCommEvent */

   do { if (!WaitCommEvent(SESSIONDEVICEHANDLE(), &fdwEventMask, NULL)){
            return -1;
        } /* end if (error waiting for event) */

        /* if write queue empty */
```

```
        if (fdwEventMask & EV_TXEMPTY) {
            if (nToWrite > MAXTXSIZE)
                nOut = MAXTXSIZE;
             else
                nOut = nToWrite;
            WriteFile (SESSIONDEVICEHANDLE(), abyByteBuffer,
                       nOut, &nActualWrite, NULL);
            nToWrite -= nActualWrite;
            abyByteBuffer += nActualWrite;  }   /* end if (TX buf empty) */
    } while (nToWrite);

    return 0;
} /* end function (WriteEventBytesToModem) */
```

Separate threads should be dedicated to reading and writing when using events. Otherwise, the calling thread is blocked by WaitCommEvent and cannot receive any window messages. The threads should be created in WinMain with the CREATE_SUSPENDED flag set so that they don't execute right away. If the threads ran immediately upon creation they would have to poll to determine when a connection was established, since the modem handle is not valid until then. Once created in the suspended state, the threads should be started using ResumeThread when the CONNECTED call state is received from TAPI. They should be suspended again using SuspendThread when the IDLE or DISCONNECTED call state is received from TAPI, indicating that the connection is no longer valid.

Now let's look at reading and writing bytes in the background, otherwise known as asynchronous I/O. Two functions, ReadFileEx and WriteFileEx, can be used to initiate reads and writes in the background and then return immediately to the calling thread. You'd do this anytime the calling thread needs to perform time-critical functions while waiting for the I/O to complete. As an example, suppose your thread was responsible for monitoring several modems at once, waiting for data to be received on any one of them. This might be the case in a data-logging application with the computer connected via serial port to several equipment sensor panels. The application could initiate several overlapped read operations, one per port, and then wait for data to arrive on any one of them. When the data arrives a special function called a completion routine is run, which can be used to process the bytes. Each modem could be assigned its own completion routine. In **Figure 9**, TTYCHIC has been modified yet again, this time to use overlapped I/O. The new read loop for a modem might look like **Figure 9**.

Figure 9 Using Overlapped I/O

```
OVERLAPPED overlapped;
int nActuallyRead;
LPOVERLAPPED_COMPLETION_ROUTINE ReadCompletionFcn (DWORD dwError,
                                    DWORD dwActRead, LPOVERLAPPED lpOverlapped);

LPTHREAD_START_ROUTINE ReadThread (LPDWORD lpdwParam1)
{ BYTE abyData[MAXREADSIZE];
  DWORD dwErrorMask;
  int nToRead;
  COMSTAT comstat;                /* status structure */

  overlapped.hEvent = CreateEvent (NULL,TRUE,FALSE,NULL);
  overlapped.Offset = overlapped.OffsetHigh = 0;

  while (dwThreadControl & READ_THREAD) {
     ClearCommError (SESSIONDEVICEHANDLE(), &dwErrorMask, &comstat);
     /* if too many bytes to read, just read the maximum */
     if (comstat.cbInQue > MAXREADSIZE)
         nToRead = MAXREADSIZE;
     else
         nToRead = comstat.cbInQue; /* otherwise read them all */
     if (!nToRead) continue;

     if (!ReadFileEx(SESSIONDEVICEHANDLE(), &abyData[0], nToRead, &overlapped,
         (LPOVERLAPPED_COMPLETION_ROUTINE)ReadCompletionFcn)) {
         /* handle error - WARNING: Beta Chicago may return 120
         extended error code, indicating ReadFileEx not implemented.
         This should be fixed in later releases. */
         dwErrorMask = GetLastError(); } /* end if (error reading asynchr.) */
     else { /* do other time critical stuff...then wait for I/O to complete */
        if (WaitForSingleObjectEx (overlapped.hEvent, 1000 /* timeout
            in milliseconds */, TRUE) = = -1) {
            dwErrorMask = GetLastError ();
            if (dwErrorMask = = WAIT_TIMEOUT) {
                if (GetOverlappedResult(SESSIONDEVICEHANDLE(), &overlapped,
                    &nActuallyRead, FALSE))  {
                    } /* end if (got result of overlapped read) */
                } /* end if (timed out during read) */
            } /* end if (error waiting for async to complete) */

          ResetEvent (overlapped.hEvent);
          if (nActuallyRead)
              WriteTTYBlock(hTTYWnd, &abyData[0], nActuallyRead ) ;
          } /* end if (initiated async read ok) */
      } /* end while (read thread active) */
   CloseHandle (overlapped.hEvent);
   PurgeComm(SESSIONDEVICEHANDLE(), PURGE_RXCLEAR);
   ExitThread(0);
} /* end function (ReadThread) */

LPOVERLAPPED_COMPLETION_ROUTINE ReadCompletionFcn (DWORD dwError,
                                    DWORD dwActRead, LPOVERLAPPED lpOverlapped)
{   if (dwError)  ;
        else nActuallyRead = (int)dwActRead;
} /* end function (ReadCompletionFcn) */
```

Notice that the ReadFileEx function is used instead of ReadFile. ReadFileEx will initiate the read process and then return immediately to the calling thread. ReadFileEx is passed a reference to an overlapped structure (explained in a moment) as well as a pointer to a function. This function is called the I/O completion routine, and it will be called when ReadFileEx completes the read process. The calling thread does other time-critical stuff and then calls the function WaitForSingleObject. This function places the thread into a sleep state, consuming very few CPU cycles, until the read completes or until the timeout interval elapses. The timeout interval is specified in the second parameter to the WaitForSingleObject function, in milliseconds (a one-second timeout in the example). The ReadFileEx function, unlike ReadFile, does not return the number of bytes actually read. This is because the function does not know the number, since it only initiates the read before returning immediately. The actual number of bytes read is returned as a parameter to the I/O completion procedure.

To initiate overlapped (asynchronous) I/O, you need to specify a structure of type OVERLAPPED in the call to ReadFileEx. This structure contains a handle to an auto-reset event object that is signaled by the WaitForSingleObject function when the read completes. Windows 95 resets the event to nonsignaled before returning from WaitForSingleObject. If an error is returned, check that it was not a timeout. A timeout

error is not necessarily fatal, since it could just indicate an interruption in the receiving of bytes from which the code will resume momentarily.

This is only one method for reading and writing bytes in the background. The other methods will not be presented but in general they are analogous to the methods used for file I/O.

Handling Errors

Communication, especially over long distances, is unpredictable. Phone lines can drop suddenly, atmospheric phenomena can cause sudden bursts of static, and equipment can fail. It is very important that a communications application be prepared to handle these situations. The simplest way to deal with these random failures is to implement timeouts. If the remote computer or equipment with which you are communicating fails to respond to reads or writes within a certain time interval, the application should assume that something bad has happened to the connection. At this time the application may attempt to reestablish communications or simply report the error. Errors can also occur because of buffer overflows or parity or framing errors. These errors are not always fatal and can sometimes be handled gracefully by the application.

When any of the communication functions return an error, the application should always call ClearCommError and check the error flags. Calling this function clears an internal error flag; Windows 95 will not allow any communications I/O to continue until the error flag is cleared. Some serious low-level errors are framing errors, parity errors, and overrun errors. The error mask will have flags set if any of these errors occurred (see **Figure 10**).

Figure 10 Communication Error Mask Flags

Value	Meaning
CE_BREAK	The hardware detected a break condition.
CE_FRAME	The hardware detected a framing error.
CE_IOE	An I/O error occurred.
CE_MODE	The requested mode of communication is not supported, or an invalid device handle was passed.
CE_OVERRUN	A character buffer overrun has occurred. The character following the overrun was lost.
CE_RXOVER	An input buffer overflow has occurred. Either there is no room in the input buffer,or a character was received after the EOF character was received.
CE_RXPARITY	The hardware detected a parity error.
CE_TXFULL	The output buffer was full when the application tried to transmit another character.

How an error is dealt with is process-specific. If the fAbortOnError flag in the DCB structure is set, then any pending reads or writes are aborted when an error occurs. The process must then call ClearCommError and may then request, using a high-level protocol such as the one used in XMODEM, that the lost or damaged data be retransmitted. It may request, when a damaged character is received, that the damaged character be replaced with a predefined character. This predefined character is called the error character, and is specified in the DCB structure. The ErrorChar field defines the character to use, and the fErrorChar bit is set to enable the replacement of characters received in error with this character. Often, the process that receives a communication error may flush the read and write buffers and terminate any pending I/O by calling the PurgeComm function.

```
PurgeComm (hComDev, fdwAction);
```

PurgeComm may prematurely terminate pending reads and writes, depending on the settings of the bitfield fdwAction. If the PURGE_TXABORT bit is set, all outstanding write operations are terminated immediately. Likewise, PURGE_RXABORT will terminate all outstanding read operations. PURGE_TXCLEAR and PURGE_RXCLEAR will flush the output and input buffers of the device in addition to terminating writes and reads immediately. PurgeComm returns TRUE if it executed successfully or FALSE otherwise. Extended error information is available through GetLastError (see **Figure 11**).

Figure 11 GetLastError

PURGE_TXABORT	Terminate pending writes
PURGE_RXABORT	Terminate pending reads
PURGE_TXCLEAR	Flush write buffer
PURGE_RXCLEAR	Flush read buffer

Conclusion

With Windows 95, Microsoft has taken a step in the right direction toward expanding communications on the PC. There were still a few bugs in our beta. For instance, it's sometimes necessary to remove Unimodem and then reinstall it before things work correctly. Undoubtedly this stuff will be ironed out by the time the retail Windows 95 ships.

More serious are the obstacles to porting communications programs from earlier versions of Windows to Windows 95. To take advantage of modem sharing and call routing, you'll have to learn TAPI and the new Win32 Comm methods for moving data and configuring modems, and that's just for interactive communications. You'll also need to learn MAPI and the WAVE APIs to cover messaging and voice calls. This is a lot to ask from programmers smothering under mountains of APIs. More than a few ISVs may simply ship legacy apps for a year or two until the need to port becomes compelling, and this will lead to all sorts of compatibility problems for those ISVs who do decide to port early on. Of course, those who move fast will reap the rewards of being first and best. Microsoft could help mitigate the plight of developers by providing better documentation and more code examples in the SDK.

Notwithstanding this criticism, Windows 95 is a truly cosmopolitan communications platform. The breadth and integration of the APIs will eventually lead to some truly killer apps.