

## Schnittstellen

# Telefonieren mit Konzept 16

von Michael Dudenhöffer

Die TAPI-Schnittstelle integriert Telefonie-Funktionen in Anwendungssoftware, die Umsetzung hängt von der Programmiersprache ab und ist im Entwicklungssystem aber nicht immer gleich schnell und einfach möglich. Die höhere Abstraktionsebene beim Datenbanksystem Konzept 16 hat diesbezüglich gegenüber der nativen Programmierung in C/C++ deutliche Vorteile.

Das Telefon spielt in vielen Geschäftsprozessen eine zentrale Rolle. Gleichzeitig liegen Informationen immer häufiger in digitaler Form vor, wie beispielsweise als Dokumente in einer Datenbank. Die Verknüpfung von Telefon und Datenbank kann daher entscheidend zu mehr Effizienz und schnelleren Abläufen beitragen. Ruft ein Kunde beispielsweise bei einem Versandhaus an, um den aktuellen Stand seiner Bestellung abzufragen, kann der zuständige Sachbearbeiter diese Anfrage zügiger bearbeiten, wenn die jeweilige Bestellung anhand der Rufnummer gleich automatisch aus der Datenbank geladen wird. Mit der TAPI-Schnittstelle (Telephony Application Programming Interface) werden unter Windows Telefonie-Funktionen in die Anwendungssoftware integriert.

Die erste, 1993 veröffentlichte TAPI-Spezifikation TAPI 1.3 war noch kein Bestandteil des Betriebssystems selbst, erst der Nachfolger TAPI 1.4 war dort integriert. Heute liegt das klassische TAPI in der Version 2.2 vor. Daneben gibt es ein TAPI 3.x, das auf COM aufbaut.

## TAPI mit C

Damit TAPI mit der Telefonie-Hardware zusammenarbeiten kann, muß diese TAPI-fähig sein. Dafür muß der Hersteller des Geräts einen sogenannten TSP (Telephony Service Provider) zur Verfügung stellen. Das ist ein in eine DLL gepackter Treiber, der genau spezifizierte Funktionen anbietet. Der Entwickler der TAPI-Anwendung ist dadurch unabhängig von der unterliegenden Hardware. Gleichgültig, ob es sich um eine ISDN-Karte oder eine über das Netzwerk verbundene Telefonanlage handelt – zum Wählen auf dem TAPI-fähigen Gerät braucht der Entwickler nur die Funktion *lineMakeCall* aufzurufen. Das ist wenigstens der Gedanke, der hinter der TAPI-Schnittstelle steckt.

Telefonie-Geräte verfügen über unterschiedliche Funktionen. Beispielsweise bieten sie oft im Gegensatz zu einfachen ISDN-Karten die Möglichkeit, Konferenzen einzuleiten. Bei einem eingehenden Anruf kann der TSP Informationen zum Anrufer wie Rufnummer oder Name des Anrufers übermitteln. Darüber hinaus unterstützt TAPI auch das Übertragen nicht sprachlicher Daten wie zum Beispiel beim Fax. Diese Funktionsvielfalt erklärt die hohe Komplexität der TAPI-Schnittstelle, die Einsteigern – selbst wenn es sich um im Windows-32-API erfahrene C/C++ Entwickler handelt – eine hohe Einarbeitungszeit abverlangt. Das nachfolgende C-Programm zeigt exemplarisch, welche Schritte auf API-Ebene notwendig sind, um einen Wahlvorgang auf einer Leitung eines TAPI-Geräts durchzuführen:

```
#define TAPI_CURRENT_VERSION TAPI_VERSION_2_0
#include <tapi.h>

LONG MakeCall(HINSTANCE aInstance, DWORD aDeviceID,
             LPCSTR aDialNumber) {

    memset(&tParams,0,sizeof tParams);

    tParams.dwTotalSize = sizeof tParams;
    tParams.dwNeededSize = sizeof tParams;
    tParams.dwUsedSize = sizeof tParams;
    tParams.dwOptions = LINEINITIALIZEEXOPTION_USEEVENT;

    tApiVersion = TAPI_CURRENT_VERSION;

    // Initialisierung der TAPI-Schnittstelle
    tError = lineInitializeEx(&tLineApp,
                            aInstance,
                            NULL,
                            (LPCSTR) "My TAPI Application",
                            (LPDWORD) &tapiDeviceTotalCount,
                            (LPDWORD) &tApiVersion,
                            &tParams);

    if (tError < 0) return(tError);

    // Aushandeln der TAPI-Version mit dem TSP
    memset(&tExtensionID, 0, sizeof tExtensionID);

    tError = lineNegotiateAPIVersion(tLineApp, aDeviceID,
                                    TAPI_VERSION_1_4, TAPI_VERSION_2_1,
                                    (DWORD*) &tApiVersion,&tExtensionID);

    if (tError < 0) return(tError);

    // Öffnen einer Leitung auf dem Device
    tError = lineOpen(tLineApp,aDevice,&tHdLine,
                    tApiVersion, 0, NULL,
                    LINECALLPRIVILEGE_NONE, 0, NULL);

    if (tError < 0) return(tError);

    // Übersetzen der Rufnummer in ein wählbares Format
    tError = lineTranslateAddress(tLineApp, aDeviceID,
                                tApiVersion,
                                (LPTSTR)aDialNumber,
                                0, 0,tOutput);

    if (tResult < 0) return(tError);
```

```

tDialNumber = (LPSTR) tOutput +
               tOutput->dwDialableStringOffset;

// Wählvorgang einleiten
tError = lineMakeCall(tHdlLine, tHdlCall, tDialNumber, 0,
                    NULL);

return(tError);
}

```

In diesem Beispiel wird zunächst mit der Compilerdirektive `#define` die gewünschte TAPI-Version festgelegt; das entscheidet über die Funktionen und Features. Die Option `LINEINITIALIZEEXOPTION_USEEVENT`, die über Kommunikations-Ereignisse benachrichtigt, wird beispielsweise erst ab der TAPI-Version 2.0 unterstützt. Das Beispiel initialisiert die Schnittstelle anschließend mit dem Aufruf von `lineInitializeEx`. Falls das erfolgreich war, liefert die Funktion einen Handle in `tLineApp`, der an nachfolgende Funktionsaufrufe übergeben wird und die TAPI-Anwendung somit eindeutig kennzeichnet.

Nachdem die gewünschte TAPI-Version festgelegt ist, muß noch mit dem TSP eine Version ausgehandelt werden, denn es ist durchaus erlaubt, daß auf einem System mehrere TAPI-Treiber für unterschiedliche Geräte in verschiedenen Versionen installiert sind:

```

tError = lineNegotiateAPIVersion(tLineApp, aDeviceID,
                                TAPI_VERSION_1_4, TAPI_VERSION_2_1,
                                (DWORD*) &tApiVersion, &tExtensionID);

```

Der TAPI-Anwendung wird hier der Versionsbereich übergeben, den der TSP erwartet. Im Beispiel muß der Treiber mindestens TAPI 1.4 und maximal TAPI 2.1 unterstützen. In der Variable `tApiVersion` wird die tatsächlich ausgehandelte Version zurückgeliefert.

Danach öffnet das Programm eine Leitung auf dem durch `aDeviceID` gekennzeichneten Gerät:

```

tError = lineOpen(tLineApp, aDevice, &tHdlLine, tApiVersion,
                 0, NULL, LINECALLPRIVILEGE_NONE, 0, NULL);

```

In `tApiVersion` steht die zuvor ausgehandelte TAPI-Version. Wenn die Funktion erfolgreich zurückkehrt, befindet sich in `tHdlLine` ein neues Handle, dieses Mal für die Leitung. Bevor im nächsten Schritt mit `lineMakeCall` der Wählvorgang initiiert werden kann, muß jetzt noch die zu wählende Rufnummer mit `lineTranslateAddress` in das passende Format umgewandelt werden:

## Schnittstellen

```
tError = lineTranslateAddress(tLineApp, aDeviceID,
                             tApiVersion,
                             (LPTSTR)aDialNumber, 0, 0,
                             tOutput);
```

Sie wandelt die in *aDialNumber* enthaltene Rufnummer in eine für den TAPI-Treiber verständliche Rufnummer. Wenn auch dieser Vorgang erfolgreich durchgeführt wurde, kann der Programmierer die so ermittelte Rufnummer (in *tOutput*) an *lineMakeCall* übergeben, die den Anruf einleitet. Dieses ausführliche Beispiel zeigt, daß die Entwicklung TAPI-fähiger Anwendungen mit dem reinen C-API sehr zeitaufwendig und fehleranfällig ist. Die Schnittstelle stellt nur grundlegende Funktionen bereit, die geradezu nach einer objektorientierte Kapselung verlangen. Dies war vielleicht auch einer der Gründe für die Entwicklung des COM-basierten TAPI 3.0.

### TAPI mit Konzept 16

Aus den vorangegangenen Erläuterungen wird ersichtlich, daß zur Verbesserung der TAPI-Anbindung die Abstraktion der Schnittstelle notwendig ist. Bei Konzept 16 ist ein solches API fester Bestandteil der Entwicklungsumgebung. Das folgende Beispiel zeigt, wie die Funktionalität des geschilderten C-Beispiels in Konzept 16 umgesetzt wird:

```
tDeviceList # TapiOpen();
...
if (tDeviceList > 0) {
    tDevice # tDeviceList->CteRead(_CteFirst):
    tDevice->TapiDial(DialNumber, _TapiAsyncDial);
}
...
if (tDeviceList > 0)
    tDeviceList->TapiClose();
```

Zunächst wird die Schnittstelle mit *TapiOpen* geöffnet. In *tDeviceList* liefert Konzept 16 eine Liste der auf dem System vorhandenen TAPI-Geräte. Jedes Listenelement enthält ein Gerät, auf das mit den *Cte*-Befehlen von Konzept 16 zugegriffen wird. Im Beispiel ermittelt *CteRead* das erste Gerät aus der Liste. Anschließend leitet *TapiDial* den Wählvorgang ein. Die Option *\_TapiAsyncDial* gibt dabei an, daß Konzept 16 auf das Zustandekommen der Verbindung warten soll, der Befehl kehrt unmittelbar nach dem Einleiten des Wählvorgangs zurück. *TapiClose* schließt die TAPI-Schnittstelle wieder. Alternativ kann man auch auf das Herstellen der Verbindung warten. Dazu übergibt man im zweiten Parameter des *TapiDial*-Befehls den Wert Null oder läßt ihn einfach weg:

```
tResult # tDeviceList->TapiDial(DialNumber);
```

Der Rückgabewert ist *\_ErrOK*, wenn die Verbindung hergestellt wurde, oder ein Fehlerwert, wenn die Verbindung nicht zustande kam.

Ist zum Beispiel die Leitung belegt, meldet Konzept 16 den Wert *\_ErrTapiBusy*. Um die Details braucht man sich also nicht zu kümmern, Konzept 16 nimmt die Versionsprüfung, die Adreßumsetzung und das Warten auf das Herstellen der Verbindung ab.

Jedes TAPI-Gerät verfügt über die Eigenschaften Name und Version, die mit den folgenden Zeilen ausgelesen werden:

```
tName # tDevice->spName;
tVersion # tDevice->spVersion;
```

Danach steht in der Variable *tName* der Name des TAPI-Geräts und in *tVersion* die durch *TapiOpen* mit dem TSP ausgehandelte Versionsnummer (also zum Beispiel 1.4 für einen TSP, der TAPI-Version 1.4 unterstützt). Damit ist es recht einfach, eine Liste mit verfügbaren Geräten zur Auswahl anzubieten.

### TAPI-Geräte

Neben der Verbindungswahl *TapiDial* gibt es noch weitere Befehle, die sich auf ein TAPI-Gerät anwenden lassen. Mit *TapiListen* wird die Überwachung von Ereignissen, die auf einem TAPI-Gerät auftreten, aktiviert:

```
tDialog # WinOpen('CallList', _WinOpenDialog);
if (tDialog > 0) {
    tDevice->TapiListen(TRUE, tDialog);
    tDialog->WinDialogRun(_WinDialogCenterScreen);
    tDialog->WinClose();
}
```

In diesem Beispiel öffnet *WinOpen* den Dialog *CallList* aus der Datenbank.

Danach wird mit *TapiListen* die Anruf-Überwachung eingeschaltet und der Dialog mit *WinDialogRun* zentriert auf dem Bildschirm dargestellt. Interessant ist der zweite Parameter des Befehls *TapiListen*. Er kennzeichnet den Dialog, der die Ereignisse erhält.

Beim Dialog-Objekt kann dafür eine Ereignisfunktion hinterlegt werden:

```
sub EvtTapi (
    aEvt      : event; // Ereignis
    aTapiDevice : handle; // TAPI Device
    aCallID    : int; // Call-ID
    aCallState : int; // Anrufstatus
    aCallTime  : caltime; // Datum und Uhrzeit
    aCallerID  : alpha; // Rufnummer
    aCalledID  : alpha // Angerufener Teilnehmer
) : logic;
```

*TapiListen* kann mehrere TAPI-Geräte gleichzeitig überwachen. Deshalb übergibt die Ereignisfunktion im Argument *aTapiDevice* das Handle des Geräts. Jeder Anruf hat eine eindeutige Kennung, die sogenannte Call-ID. *aCallState* ist der aktuelle Zustand des Anrufs, *aCallTime* enthält Datum und Uhrzeit, zu dem das Ereignis auftrat. *aCallerID* und *aCalledID* sind die Nummern des Anrufers und des Angerufenen.

Ein Anruf besitzt für die Dauer seiner Existenz eine eindeutige Call-ID und zu jedem Zeitpunkt einen Anrufstatus. In Konzept 16 gibt es hierfür entsprechende Konstanten:

```
_TapiCallStateOffer // Signalisiert einen
// eingehenden Anruf
_TapiCallStateProceeding // Wählvorgang abgeschlossen
_TapiCallStateRingback // Anrufer erreicht, Leitung frei
_TapiCallStateBusy // Anrufer erreicht, Leitung
// besetzt
_TapiCallStateConnected // Verbindung hergestellt
_TapiCallStateDisconnected // Verbindung getrennt
_TapiCallStateIdle // Finaler Status (kein Anruf
// aktiv)
```

Die Ereignisfunktion übergibt für einen neuen eingehenden Anruf zunächst seine Call-ID und den Status *\_TapiCallStateOffer*. Nimmt der Angerufene ab, ändert sich der Status in *\_TapiCallStateConnected*. Beendet er das Gespräch, wechselt der Status zunächst nach *\_TapiCallStateDisconnected*. Der letzte Zustand eines Anrufs ist *\_TapiCallStateIdle*. Danach ist die Call-ID nicht mehr gültig.

### Anruf-Status

Bei einem ausgehenden Anruf beispielsweise durch Wählen mit dem Befehl *TapiDial* erhält dieser zunächst den Status *\_TapiCallStateProceeding*. Dieser Zustand tritt ein, nachdem der Wählvorgang abgeschlossen ist, und wechselt in den Status *\_TapiCallStateRingback* oder *\_TapiCallStateBusy*, wenn der Angerufene erreicht wird beziehungsweise wenn besetzt ist. Kommt die Verbindung anschließend zustande, wechselt der Zustand nach *\_TapiCallStateConnected*. Konzept 16 bietet auch die Möglichkeit, auf Anrufe zu reagieren. Zuständig ist der Befehl *TapiCall(CallID, Operation)*. Soll die Möglichkeit geboten sein, einen Anruf auf Knopfdruck entgegenzunehmen, wenn dieser durch *\_TapiCallStateOffer* signalisiert wird, erreicht er das durch den Aufruf von *TapiCall(CallID, \_TapiCallOpAnswer)*. Darüber hinaus gibt es noch weitere Operationen wie das Trennen (*\_TapiCallOpDrop*) oder das Halten (*\_TapiCallOpHold*) einer Verbindung. Mit diesem Befehl kann außerdem der aktuelle Anrufstatus ermittelt werden. Dafür lautet die Syntax:

```
tStatus # TapiCall(CallID, _TapiCallOpState);
```

Nach der Rückkehr des Befehls befindet sich in *tStatus* der Anrufstatus (zum Beispiel *\_TapiCallStateConnected*, wenn der durch *CallID* gekennzeichnete Anruf verbunden ist). Für Konferenzen sind erweiterte Funktionen verfügbar. Basis ist der Aufruf

```
tConsultCall # TapiConference(CallID);
```

Wenn der Anwender einen eingehenden Anruf annimmt, so daß eine Verbindung zwischen zwei Gesprächspartnern hergestellt ist, wird im Ereignis *EvtTapi* der Befehl *TapiConference* aufgerufen, um eine Konferenz zu initiieren. Im Beispiel ist *CallID* die durch das Ereignis übergebene Anruf-ID. In *tConsultCall* wird dann die ID eines sogenannten Consultcalls zurückgeliefert. Sie wird benötigt, um im nächsten Schritt den dritten Gesprächspartner zur Konferenz einzuladen:

```
TapiConferenceDial(tConsultCall, '123');
```

Dafür werden die durch *TapiConference* ermittelte Consult-Call-ID (*tConsultCall*) und die Telefonnummer des dritten Konferenzpartners (hier im Beispiel 123) an *TapiConferenceDial* übergeben. Daraufhin legt der Befehl den ursprünglichen Anrufer (*CallID*) auf Halten und leitet analog zu *TapiDial* den Wählvorgang ein. Wenn die Verbindung zustandekommt, kehrt *TapiConferenceDial* mit *\_ErrOK* zurück. Der Status von *tConsultCall* ist dann *\_TapiCallStateConnected*. Um die Konferenz zu schalten, wird jetzt lediglich noch ein Aufruf zur Bestätigung der Konferenz benötigt:

```
TapiConferenceCommit(CallID, tConsultCallID);
```

Durch den Aufruf werden die drei Gesprächspartner zu einer Konferenz zusammengeschaltet. Bei *CallID* handelt es sich um die an *TapiConference* übergebene Anruf-ID. Der Aufruf von *TapiConferenceDial* im obigen Beispiel ist synchron, das heißt, der Befehl kehrt erst zurück, wenn die Verbindung zustandekam oder ein Fehler auftrat. Wie bei *TapiDial* kann hier aber auch das Argument *\_TapiAsyncDial* angegeben werden, dann kehrt der Befehl nach dem Einleiten des Wählvorgangs zurück. Im Programm kann anschließend über die Anrufüberwachung und das *EvtTapi-Ereignis* der weitere Verlauf des Anrufs verfolgt werden. ■