

# ISDN mit Delphi

Version 2.0 / Juni 99

Copyright® 96 - 99 Stefan Graf

<http://www.isdn-tools.de>  
[info@isdn-tools.de](mailto:info@isdn-tools.de)

# Inhalt

<b>EINFÜHRUNG .....</b>	<b>7</b>
CAPI.....	7
DIE CAPI KOMPONENTE .....	7
TCapi20Handler.....	7
TCapi20Base.....	7
DIENSTE IM ISDN-NETZ.....	8
<b>INSTALLATION DER DATEIEN.....</b>	<b>10</b>
<b>EINBINDEN DER KOMPONENTE .....</b>	<b>10</b>
DEMO-VERSION .....	10
<b>EINSATZ DER KOMPONENTE .....</b>	<b>12</b>
INSTANZEN ERZEUGEN.....	12
AKTIVER VERBINDUNGSaufbau.....	12
PASSIVER VERBINDUNGSaufbau .....	13
ANRUF ENTGEGEN NEHMEN .....	14
B-KANAL VERBINDUNG aufbauen.....	15
<i>Daten-Verbindung.....</i>	<i>15</i>
<i>Voice-Verbindung .....</i>	<i>15</i>
<i>FAX G3 Verbindung (Nur ProVersion) .....</i>	<i>16</i>
DATEN SENDEN.....	17
DATEN EMPFANGEN.....	17
VERBINDUNG TRENNEN .....	18
PROZESSVERWALTUNG.....	19
<b>KOMPONENTE TCAPI20BASE .....</b>	<b>20</b>
PUBLISHED EIGENSCHAFTEN .....	20
CapiAvailable .....	20
ApplicationRegistered .....	20
AutoRegister .....	20
ApplicationID.....	20
Controller.....	20
BDataBlocks .....	21
BDataBlockSize .....	21
LogicalConnection.....	21
PUBLIC EIGENSCHAFTEN .....	22
CapiAvailable .....	22
DTMFAvailable .....	22
ApplicationRegistered .....	22
PUBLISHED EVENTS.....	23
<i>Messagebehandlung .....</i>	<i>23</i>
MessageReceived.....	23
MessageDump.....	24
<i>Fehlerbehandlung .....</i>	<i>25</i>
OnError.....	25
PROTECTED METHODEN .....	26
Release .....	26
ProcessCapiMessages.....	26
PUBLIC METHODEN.....	27
<i>An- und Abmeldung bei der CAPI .....</i>	<i>27</i>
RegisterApplication.....	27

ReleaseApplication.....	27
<i>Allgemeine Funktionen.....</i>	<i>28</i>
Synchronize.....	28
GetCommandStr.....	28
GetErrorStr.....	28
GetLastError.....	28
GetVersion.....	28
GetSerialNumber.....	29
GetHersteller.....	29
GetAnzahlController.....	29
GetOptionen.....	29
GetAnzahlBCannel.....	29
GetSupportedB1Protokoll.....	29
GetSupportedB2Protokoll.....	29
GetSupportedB3Protokoll.....	30
<i>Messageverwaltung.....</i>	<i>31</i>
DoCapiMessage.....	31
SendRequestMessage.....	31
SendResponsMessage.....	31
GetMessage.....	32
<b>KOMPONENTE CAPI20HANDLER.....</b>	<b>33</b>
EIGENSCHAFTEN.....	33
Buffers.....	33
BufferSize.....	33
AutoBConnect.....	33
B1Protokoll.....	33
TProtokolB1.....	33
B2Protokoll.....	34
TProtokolB2.....	34
B3Protokoll.....	34
TProtokolB3.....	34
Infos.....	34
TCapiInfo.....	34
OwnNumber.....	34
OwnSubAdr (Nur Pro Version).....	35
CapiDTMF.....	35
NebenStelle.....	35
AmtNr.....	35
ExternLen.....	35
UserData.....	35
PUBLIC EIGENSCHAFTEN.....	36
PLCI.....	36
NCCI.....	36
Verbindung.....	36
ConnectedService.....	36
TCapiService.....	36
ConnectStatus.....	36
TConnectStatus.....	36
ConnectedB1Protokol.....	37
ConnectedB2Protokol.....	37
ConnectedB3Protokol.....	37
PUBLISHED EVENTS.....	38
<i>Verbindungsauf- und abbau.....</i>	<i>38</i>
OnRing.....	38
TCallDaten.....	38
SubAdressierung.....	38
OnConnect.....	39

OnDisconnect.....	40
OnRequestBConnect .....	41
OnSetBProtokollEvent.....	41
OnGetBProtokollEvent .....	41
OnBConnect.....	41
OnBDisconnect .....	42
OnBReset.....	42
<i>Datenübertragung</i> .....	43
OnReceive.....	43
OnSend .....	43
<i>Informationen</i> .....	44
OnConfirm .....	44
OnIndication.....	44
OnFacility .....	44
OnInfo.....	44
OnReceiveDTMF (Nur Pro Version) .....	45
PUBLIC METHODEN .....	46
<i>Verbindungsauf und Abbau</i> .....	46
SetProtokol.....	46
EnableListen .....	46
AnswerCall .....	47
Call .....	47
OpenBConnect .....	48
CloseBConnect.....	48
ResetBConnect.....	48
DisconnectLine.....	49
GetDisconnectReasonStr .....	49
DATENÜBERTRAGUNG .....	50
Read.....	50
ReadEx .....	50
Write.....	50
WriteEx .....	51
Flush.....	51
GetSendBytes.....	51
GetReceiveBytes .....	51
GetConnectStatus.....	51
<b>KOMPONENTE TCAPIPROTOKOLL.....</b>	<b>52</b>
EIGENSCHAFTEN .....	52
Handler .....	52
<b>KOMPONENTE TT30PROTOKOLL .....</b>	<b>53</b>
TSFFHeader.....	53
TSFFPageHeader .....	53
PUBLISHED EIGENSCHAFTEN.....	54
FaxRecvInfo .....	54
TFaxResolution.....	54
TFaxFormat .....	54
TReceiverID.....	54
THeaderLine.....	54
TStationID .....	54
TFaxRecvInfo.....	54
FaxSendInfo .....	54
TFaxBaudRate .....	54
TFaxSendInfo.....	55
<b>KOMPONENTE TV110PROTOKOLL .....</b>	<b>56</b>
EIGENSCHAFTEN .....	56

Rate .....	56
BitsPerChar.....	56
TBitsPerChar .....	56
StopBits .....	56
TStopBits.....	56
Parity .....	56
TParity .....	56
<b>KOMPONENTE TV120PROTOKOLL .....</b>	<b>57</b>
EIGENSCHAFTEN .....	57
<b>KOMPONENTE TMODEMPROTOKOLL .....</b>	<b>58</b>
EIGENSCHAFTEN .....	58
Rate .....	58
BitsPerChar.....	58
StopBits .....	58
Parity .....	58
<b>KOMPONENTE TSFFFAXCONVERTER (NUR PRO VERSION).....</b>	<b>59</b>
EIGENSCHAFTEN .....	59
FileName .....	59
Pages.....	59
Resolution .....	59
TFaxResolution.....	59
MassEinheit .....	59
TMassEinheit.....	59
SeitenRandRechts .....	59
SeitenRandLinks .....	59
SeitenRandOben.....	60
SeitenRandUnten .....	60
PUBLIC METHODEN.....	61
OpenFax .....	61
CreateFax.....	61
CloseFax .....	61
GraphicToFax .....	61
FaxToGraphic .....	61
ConvertToFax .....	61
<b>SPRACHÜBERTRAGUNG IM ISDN .....</b>	<b>62</b>
<b>KOMPONENTE TWAVE.....</b>	<b>6201-63</b>
EIGENSCHAFTEN .....	6201-63
IsOpen.....	6201-63
WaveSize .....	6201-63
WaveSamples.....	6201-63
PUBLIC METHODEN.....	6201-63
Create .....	6201-63
Destroy.....	6201-63
SetFormat.....	6201-63
GetFormat.....	6201-63
Build.....	6201-64
Open .....	6201-64
Close.....	6201-64
Read.....	6201-64
Write.....	6201-64
<b>UNIT LAW.PAS .....</b>	<b>65</b>

LawToInt.....	65
IntToLaw.....	65
<b>HISTROY .....</b>	<b>66</b>

## Einführung

Über das ISDN-Netz können pro Kanal bis zu 8 KByte Daten pro Sekunde übertragen werden bei einer durchschnittlichen Fehlerrate von unter  $10^9$  Bytes.

Durch Einsatz von vordefinierten Protokollen wird jegliche Fehlerkorrektur bei der Datenübertragung zwischen zwei ISDN-Teilnehmern überflüssig.

Durch die Verwendung von Dienstkennungen und der Übertragung der Rufnummer des Anrufers ist das Handhaben von Verbindungen wesentlich einfacher als mit Modems auf der analogen Telefonleitung.

Im ISDN-Netz wird zwischen zwei Verbindungsarten und drei Verbindungsebenen unterschieden.

Jeder Anruf im ISDN beginnt mit einer D-Kanal Verbindung.

Aufbauen auf dieser Verbindung können nun mehrere B-Kanal-Verbindungen zum anderen Teilnehmer hergestellt werden, wobei jede dieser Verbindungen über drei Kommunikationsebenen verfügt.

Für die Datenübertragung auf diesen Ebenen wird jeweils ein eigenes Protokoll eingesetzt.

Alle Verbindungen im B-Kanal müssen sich aber die maximale Übertragungsrate von 64-kBits teilen.

## CAPI

Das Ansprechen des ISDN-Netz mit dem PC und der ISDN-Karte erfolgt über die standardisierte CAPI-Schnittstelle, die es in der Version 1.1 für das nationale ISDN (1TR6) und in der Version 2.0 für das internationale ISDN (DSS1) gibt.

Zum Einsatz kommt die CAPI-Version 2.0 in Form eines VxD-Treibers.

Diese Treiber gibt es von allen ISDN-Kartenherstellern für Windows 95/98/NT und einige wenige (AVM, TELES, ELSA) bieten ihn auch für Windows 3.1 an.

Für Windows NT gibt es nur von wenigen Herstellern, meist für aktive Karten, CAPI-Treiber.

Neben dem VxD-Treiber wird zum Ansprechen der CAPI eine DLL benutzt, die unter Windows 3.11 CAPI20.DLL. Für Windows 95 und Windows NT kommt die CAPI2032.DLL zum Einsatz.

Diese DLL's müssen im Windows-SYSTEM-Verzeichnis liegen. Dies ist im allgemeinen bei Windows 95/98 C:\Windows\System und bei Windows NT C:\WinNT\System32.

Gewisse Hersteller (TELSE) kopieren unter NT aber den Treiber in das Verzeichnis C:\WinNT\System, welches eigentlich für 16-Bit-Treiber vorgesehen ist.

Die CAPI-Komponente berücksichtigt dies aber und sucht unter NT auch in diesem Verzeichnis nach dem CAPI-Treiber.

Die neue CAPI 2.0 bringt zwar für die reine Datenübertragung wenig Vorteile, aber die Anrufentgegennahme ist komfortabler und im DSS1-Protokoll entfällt die Umsetzung der EAZ's aus dem 1TR6-Protokoll.

## Die CAPI Komponente

### TCapi20Handler

Die Komponente TCapi20Handler übernimmt die gesamte Kommunikation zwischen Anwendung und ISDN-Netz.

Sie kann Anrufe ausführen und entgegennehmen, Datenverbindungen aufbauen und darüber Zeichen empfangen und verschicken.

**!!! Eine Instanz vom TCapi20Handler kann nur eine physikalische Verbindung verwalten !!!**

### TCapi20Base

Die Komponente besteht aus zwei eigenständigen Teilen. Die Basisklasse TCapi20Base übernimmt die Kommunikation mit der CAPI. Davon abgeleitet ist die Klasse TCapi20Handler, die der Anwendung die nötigen Funktionen zur Verfügung stellt.

Sie enthält einen Handler der alle, vom der CAPI empfangenen, Nachrichten übernimmt und auswertet. Die Komponente wird ausschließlich durch Ereignisse gesteuert, d.h. die Anwendung muß nicht regelmäßig die Komponente abfragen, sondern sie wird über Events auf anstehenden Ereignissen - ein Anrufe, Daten empfangen usw.- aufmerksam gemacht.

Die Implementation erfolgt in den Units capi2bas.pas und capi2han.pas. Die Units capi2con.pas und capi2typ.pas enthalten die benötigten Konstanten bzw. die Typendefinitionen aller CAPI-Nachrichten.

Das CAPI 2.0 wird unter Windows 3.11 über die CAPI20.DLL und unter Windows 96 oder NT über die CAPI2032.DLL angesprochen.

Die Unterscheidung zwischen Windows 3.1 (16-Bit) und Windows 95/NT erfolgt zum Zeitpunkt der Übersetzung anhand des Defines WIN32.

Beim Starten der Anwendung prüft die Komponente, ob die passende DLL zur Verfügung steht und gibt gegebenenfalls eine Fehlermeldung aus.

Vor dem ersten Benutzen der CAPI wird geprüft, ob über ein CAPI-Treiber geladen wurde, und falls nicht wird eine Fehlermeldung ausgegeben.

Die Anwendung ist aber trotzdem bis auf die ISDN-Teile weiterhin ausführbar.

Es besteht theoretisch die Möglichkeit, in 32-Bit-Anwendungen mehrere Instanzen der Komponente zu erzeugen.

Bei 16-Bit-Anwendungen ist dies auf Grund der Interruptverarbeitung nicht möglich.

## Dienste im ISDN-Netz

Die Komponente unterstützt folgende Dienste direkt:

Dienst	Beschreibung
TELEPHONY	Allgemeine Sprachverbindung
ANALOG_TELEPHONY	Analoge Telefonteilnehmer, dies können auch G3-Faxgeräte sein.
DIGITAL_TELEPHONY	Digitale Telefonteilnehmer
FAX3	G3-Faxgerät; Dieser Service wird nur von ISDN-Tk-Anlagen unterstützt.
FAX4	G4-Faxgerät; sehr selten
DATA_64K	Digitale Datenübertragung mit 64 kBits pro Sekunde
DATA_56K	Digitale Datenübertragung mit 56 kBits pro Sekunde, z.B. CompuServe

Nicht alle Hersteller von ISDN-Karten und CAPI-Treibern unterstützen alle obengenannten Dienste, und die damit verbundenen Protokoll.

Die Komponente faßt alle unterstützten Dienste im den Datentype **TCapiService** zusammen.

```
TCapiService = (ALL, TELEPHONY, ANALOG_TELEPHONY, DIGITAL_TELEPHONY,
                FAX3, FAX4, DATA_64K, DATA_56K, UNKNOWN);
```

Der Type **ALL** bezeichnet alle möglichen Dienste und darf nur beim Aufruf von **EnableListen** benutzt werden.



Alle Dienste, die von der Komponente nicht unterstützt werden, erhalten den Type *UNKNOWN*.

## Installation der Dateien

Alle benötigten Dateien sollten in ein gemeinsames Verzeichnis kopiert werden. Dabei ist zu beachten, dass die dcr-Dateien - die Icons der Komponenten - sowohl in einer 16 als auch 32-Bit Version vorhanden sind.

Folgende Dateien werden benötigt:

<code>DLLUtil.pas</code>	Hilfsfunktion zum Laden von DLLs
<code>capi2con.pas</code>	Alle Konstanten
<code>capi2typ.pas</code>	Die Typendefinitionen der CAPI-Messages
<code>capi2bas.pas</code>	Die Basisfunktionen der CAPI
<code>capi2han.pas</code>	Der Messagehandler
<code>wave.pas</code>	Komponente für das Lesen und Schreiben von Wavedateien
<code>law.pas</code>	Konvertierfunktionen für das G.711 Audioformat
<code>capi2han.dcr</code>	Das Icon der Komponente <b>TCapi20Handler</b>

In der ProVersion werden zusätzlich folgende Dateien benötigt:

<code>DTMF_fil.pas</code>	Routinen für die DTMF-Erkennung
<code>SFFMisc.pas</code>	Typendefinition und Routinen für das SFF-Faxformat
<code>SFFConvert.pas</code>	Komponente zum Lesen und Schreiben von SFF-Dateien
<code>SFFConvert.dcr</code>	Das Icon der Komponente <b>TSFFFaxConverter</b>

## Einbinden der Komponente

Für den Einsatz der Komponente mit dem Objektmanager muß die Klasse **TCapi20Handler** in die VCL eingebunden werden.

Die Implementation der Komponente befindet sich in der Datei `capi2han.pas`. Diese setzt das Vorhandensein der Dateien `dllutil.pas`, `capi2bas.pas`, `capi2typ.pas` und `capi2con.pas` voraus.

Die in `capi2bas.pas` enthaltene Basisklasse **TCapi20Base** kann aber nicht in die VCL eingebunden werden, da sie für die Anwendungsentwicklung nicht direkt einsetzbar ist.

In der ProVersion kommt eine weitere Komponente namens `CapiProd.pas` zum Einsatz. Sie enthält alle erweiterten Protokollkomponenten z.B. **TV110Protokoll**.

Das Einbinden der Komponenten erfolgt in Delphi 1 über den Menüpunkt "Optionen|Komponente installieren", in Delphi 2 über den Menüpunkt "Komponente|Installieren" mit dem Button "Hinzufügen".

Unter Delphi 3 und 4 werden Komponenten in einem Dialog unter dem Menüpunkt "Komponente|Installieren" der VCL hinzugefügt.

Dabei kann man noch das gewünschte Package auswählen oder ein neues angelegt werden.

In allen Fällen kann über den Button "Durchsuchen" die gewünschte Datei gesucht werden.

## Demo-Version

Bei einer Demo-Version muß die \*.dcu-Datei ausgewählt werden, Registrierte Benutzer wählen die entsprechende \*.pas Datei.

Es sieht leider so aus, als hätte Delphi 4.x einen kleinen Fehler beim Installieren von Komponenten, für die keine Sourcen nicht vorhanden ist.

Es erscheint die Meldung "Mindestens eine Zeile war zu lang und mußte abgeschnitten werden" und die DCU-Datei wird in den Editor geladen

Dies Meldung kann/muss ignoriert werden, denn Delphi versuchte fehlerhafter weise, die dcu-Datei als Source auszuwerten.

Die Datei im Editor wird einfach **ohne die Änderungen zu übernehmen** geschlossen. Danach muss auch noch der Source des Benutzerkomponenten-Packets geschlossen werden.

Nun kann man das Packets im Packet-Fenster neu übersetzen und die Komponente wird ordnungsgemäss erkannt und in der Komponentenleiste aufgeführt.

## Einsatz der Komponente

Die Kommunikation mit der CAPI erfolgt über die Komponente **TCapi20Base**. Die eingehenden Messages werden an die Komponente **TCapi20Handler** weitergereicht.

Diese unterstützt den Anwender beim Aufbau von Verbindungen, beim Senden und Empfangen von Daten und liefert Verbindungsinformationen.

Ferner signalisierte sie eingehende Anrufe.

## Instanzen erzeugen

Die Komponente kann in der Entwurfsphase auf einem Formular abgelegt und mit dem Eigenschaften Objekt-Inspector verändert werden. Es ist aber auch möglich, eine Instanz der Komponente zur Laufzeit durch Aufrufen des Konstruktors **Create** zu erzeugen.

Mit der Basisfunktion Assign kann man einer neu erzeugten Komponente die Eigenschaften und Events einer bereits bestehenden Instanz zuweisen.

Die neue Komponente muss dann aber noch bei der CAPI mit **RegisterApplication** angemeldet werden.

```
for i:=1 to AnzahlLeitungen do begin
  Leitungen [i] := TLeitung.Create (Self);

  with Leitungen [i] do begin
    Index := i;

    Capi := TCapi20Handler.Create (Self);
    Capi.Assign (CapiHandler);

    Capi.UserData := Leitungen [i];

    AntwortOpen := false;
    AnrufIndex := -1;
  end;
end;
```

Um so erzeugten Komponenten individuell einzusetzen, besitzt jede Instanz die Eigenschaft **UserData**, die auf eine, vom Benutzer definierte, Datenstruktur zeigen kann. Damit kann man in den Eventroutinen schnell und sicher auf die zugehörigen Daten zugreifen.

```
procedure TForm1.CapiHandlerReceive(Sender:TObject; const NCCI:LongInt; const Count:Cardinal);
var
  readanz : Cardinal;
  buf : array [0..4096] of char;
begin
  with (Sender as TCapi20Handler) do begin
    readanz := Read (NCCI, buf,Count);

    with TLeitung (UserData) do begin
      if (AntwortOpen) then begin
        BlockWrite (AntwortFile,buf,readanz);

        if (IOResult <> 0) then;
          end;
        end;
      end;
    end;
  end;
```

## Aktiver Verbindungsaufbau

Die Verbindungen können von der Anwendung aktiv durch einen Anruf aufgebaut werden, oder passiv durch Anrufentgegennahme.

Ein aktiver Verbindungsaufbau wird durch den Anruf bei der Gegenstelle eingeleitet. Dies erfolgt mit der Funktion **Call**. Dabei muß die Anwendung neben der Telefonnummer auch den gewünschten Dienst vorgeben. Nur für diesen Dienst wird dann auch eine passende Verbindung aufgebaut.

```
procedure TForm1.DigitalClick(Sender: TObject);
var numstr : String [20];
begin
  numstr := InputBox ('Telefonnummer','Rufnummer eingeben','');
```

```
if (Length (numstr) > 0) then begin
  Capil.B1Protokol := HDLC_64k;
  Capil.B2Protokol := X75;
  Capil.B3Protokol := B3_TRANSPARENT;

  Capil.Call (numstr, DATA_64K);
end;
end;
```

## Passiver Verbindungsaufbau

Beim passiven Verbindungsaufbau wartet die Anwendung auf eingehende Anrufe. Dies wird durch die Funktion **EnableListen** eingeleitet, wobei festgelegt werden muß, welche Dienste angenommen werden soll.

Nach dem Erzeugen der Komponente ist diese Signalisierung deaktiviert, d.h. es werden keine eingehenden Anrufe gemeldet.

```
procedure TForm1.SetServiceClick(Sender: TObject);
var enableset : TServiceSet;
begin
  enableset := [];

  if (Telephony1.Checked) then
    enableset := enableset + [TELEPHONY, ANALOG_TELEPHONY, DIGITAL_TELEPHONY];

  if (Daten1.Checked) then enableset := enableset + [DATA_64k];
  if (FaxG31.Checked) then enableset := enableset + [FAX3];
  if (FaxG41.Checked) then enableset := enableset + [FAX4];

  Capil.EnableListen (enableset);
end;
```

## Anruf entgegen nehmen

Wenn ein Anruf für einen der vorgegebenen Dienste erfolgt, meldet dies die Komponente der Anwendung über den Event ***OnRing***.

Der Anruf wird nur dann angenommen, wenn der Rückgabewert der Funktion = ACCEPT\_CALL oder WAIT\_ACCEPT\_CALL ist. Ansonsten kommt keine Verbindung zustande.

Alle Daten der Verbindung werden in einer Struktur vom Typ *TCallDaten* übergeben.

```
function TForm1.CapilRing(Sender: TObject; const CallDaten : TCallDaten)
const
  ServiceStr : array [0..8] of String [20] = ('','TELEPHONY','ANALOG_TELEPHONY',
                                              'DIGITAL_TELEPHONY','FAX3','FAX4',
                                              'DATA_64K','DATA_56K','UNKNOWN');
begin
  with CallDaten do begin
    Write (LogFile, 'Call from ' + Copy (CallFrom,3,255) + ' to ' + Copy (CallTo,2,255);
    WriteLn(LogFile,' : Service : '+ServiceStr[Ord(Service)]+'('+IntToStr(Ord(Service))+')');

    if (Service = DATA_64K) then
      CapilRing := ACCEPT_CALL
    else if (Service in {TELEPHONY, ANALOG_TELEPHONY, DIGITAL_TELEPHONY}) then begin
      Capil.B1Protokol := TRANS_64K;
      Capil.B2Protokol := BITTRANS;
      Capil.B3Protokol := B3_TRANSPARENT;

      Timer1.Interval := 3000;
      Timer1.Enabled := true;

      CapilRing := WAIT_ACCEPT_CALL;
    end else CapilRing := IGNORE_CALL;
  end;
end;
```

Bei der Rufannahme mit WAIT\_ACCEPT\_CALL wird der Anruf für die Anwendung reserviert, aber noch keine Verbindung aufgebaut. Dies muß dann zu einem späteren Zeitpunkt mit ***AnswerCall*** geschehen. Dem Anrufenden wird dann während der Wartezeit das Klingelsignal übermittelt.

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  Timer1.Enabled := false;

  Capi1.AnswerCall (ACCEPT_CALL);
end;
```

Ein erfolgreicher Verbindungsaufbau meldet die Komponente über den Event ***OnConnect***.

Dies ist dann aber nur eine D-Kanal-Verbindung, über die noch keine Daten übertragen werden können.

## B-Kanal Verbindung aufbauen

Wenn das Property *AutoBConnect* auf *true* gesetzt ist, baut die Komponente nach dem erfolgreichen Verbindungsaufbau automatisch eine B-Kanal-Verbindung mit den vorgegebenen Protokollen *B1Protokoll*, *B2Protokoll* und *B3Protokoll* auf.

Der erfolgreiche Aufbau dieser Verbindung meldet die Komponente über den Event *OnBConnect*.

Wenn dies nicht der Fall ist, kann zu einem beliebigen, späteren Zeitpunkt eine B-Kanal-Verbindung mit der Funktion *OpenBConnect* aufgebaut werden.

Wenn die CAPI das gewünschte Protokoll nicht unterstützt meldet sie die Fehler \$3001 bis \$3007.

Ein bestehende B-Kanal-Verbindung kann jederzeit mit *CloseBConnect* getrennt werden, ohne daß dabei die physikalische D-Kanal-Verbindung aufgelöst wird.

So ist es möglich, z.B. zuerst eine FAX G3 (T.30) Verbindung, und wenn dies mißlingt, eine Sprach-Verbindung aufbauen. Dies wird aber erst von wenigen CAPI's unterstützt (AVM und Teles)

## Daten-Verbindung

Bei Datenverbindungen (Dienst = DATA\_64K) wird normalerweise das X.75-Protokoll eingesetzt.

Dazu müssen folgende Protokolle eingestellt werden:

*B1Protokoll* := HDLC\_64K

*B2Protokoll* := X75

*B3Protokoll* := B3\_TRANSPARENT

## Voice-Verbindung

Bei Sprache-Verbindungen (Dienst = TELEFONY) müssen alle Protokoll auf Transparent stehen.

*B1Protokoll* := TRANS\_64K

*B2Protokoll* := BITTRANS

*B3Protokoll* := B3\_TRANSPARENT

Bei einer Sprachverbindung werden permanent Daten über den B-Kanal verschickt.

## FAX G3 Verbindung (Nur ProVersion)

Für eine G3 kompatible Faxübertragung (Dienst = FAX3) zu analogen Endgeräten muß das T.30-Protokoll eingesetzt werden. Diese Protokoll wird aber nicht von allen CAPI-Treibern unterstützt. Prüfen kann man dies über die Funktion **GetSupportedB1Protokoll**. Wenn im Rückgabewert das Bit 4 gesetzt ist, wird das Protokoll unterstützt.

Die folgenden Hersteller unterstützen das T.30 Protokoll (Liste nicht vollständig)

- AVM B1
- AVM A1 Classic & Fritz!Card ab Version 2.01
- Eicon Diehl Diva Pro (Empfang erst ab Ver. 1.62)
- Teles ab Version 3.23

Die einzelne Eigenschaften für die Protokolle müssen wie folgt eingestellt werden:

**B1Protokoll** := T30\_B1

**B2Protokoll** := T30\_B2

**B3Protokoll** := T30\_B3 oder T30\_EXT

Des weiteren muß eine Instanz der Komponente **TT30Protokoll** auf den Formular vorliegen und mit dem eingesetzten CAPI-Handler verknüpft sein.

Diese Komponente enthält alle relevanten Daten des eigenen Anschlusses und überträgt sie beim Verbindungsaufbau zur Gegenstelle.

Nach Verbindungsabbau enthält sie auch die Informationen über die Gegenstelle.

Beim Aufbauen der B-Kanal-Verbindung muß die Anwendung angeben, welches FAX-Format übertragen werden soll.

Als Standard hat sich das **SFF-Format** etabliert. Die FAX-Daten sind bei diesem Format als Datei abgelegt und beim Übertragen wird der Inhalt byteweise zur Gegenstelle geschickt.

Theoretisch sind auch andere Format vorgesehen, doch wird dies von den Treibern meist nicht unterstützt.

Alle Hersteller von ISDN-Karten, liefern einen Druckertreiber mit, womit aus jeder beliebigen Anwendung heraus dieses FAX-Format erzeugt werden kann.

Die Komponente selber bietet keinen eigenen Druckertreiber.

Um aus einer Applikation heraus Faxdokumente zu generieren kann die Komponente **TSFFFaxConverter** eingesetzt werden. Sie ermöglicht das Erstellen von ein oder mehrseitigen Faxdokumente aus TMemo- und TRichEdit-Komponenten oder konvertiert TImage- oder TBitmap-Inhalte.

Das erweiterte T.30-Protokoll ermöglicht zusätzlich den Faxabruf und das Faxpolling. Dies wird erst von wenigen CAPI's unterstützt.

Das Versenden und Empfangen von G3-Faxen über das T.30 Protokoll wird nur in der Pro-Version der Komponente unterstützt.

Für das Setzen der obengenannten Dienste (TELEFONY, DATA\_64K und FAX3) stellt die Komponente die Funktion **SetProtokol** zur Verfügung.

Sie liefert als Rückgabewert *true*, wenn das Protokoll auf allen drei Ebenen unterstützt wird, und *false*, wenn das Protokoll vom CAPI-Treiber nicht unterstützt wird.



## Daten senden

Sobald eine B-Verbindung vorhanden ist, kann mit der Funktion **Write** Daten zur Gegenstelle geschickt werden.

```
procedure TForm1.SendTextfile(Sender: TObject);
var line      : String;
    filevar   : TextFile;
begin
  AssignFile (filevar, 'C:\autoexec.bat');
  Reset (filevar);
  if (IOResult = 0) then begin
    while not (Eof (filevar)) do begin
      ReadLn (filevar, line);
      Capi1.Write (DataKanal, line [1], Length (line));
    end;

    CloseFile (filevar);
  end;
end;
```

Sowohl die Komponente als auch die CAPI verfügen über interne Zwischenpuffer. Falls alle Puffer der Komponente belegt sind, kehrt die Funktion **Write** erst zurück, wenn wieder ein Puffer zur Verfügung steht, oder die Verbindung abgebrochen wurde.

Während dieser Wartezeit wird aber die Windows Messagequeue mit **Application.ProcessMessages** abgefragt.

## Daten empfangen

Den Empfang von Daten meldet die Komponente über den Event **OnReceive**. Die Anwendung holt dann die Daten mit der Funktion **Read** ab.

```
procedure TForm1.CapiReceive(Sender: TObject; const NCCI : LongInt; const Count: Cardinal);
var
  readanz : Cardinal;
  buf : array [0..4096] of char;
begin
  readanz := CapiHandler.Read (abKanal, buf, Count);

  if (AntwortOpen) then begin
    BlockWrite (AntwortFile, buf, readanz);
  end;
end;
```

## Verbindung trennen

Um eine bestehende Verbindung zu trennen muß die Funktion ***DisconnectLine*** aufgerufen werden. Als Parameter wird der Grund für die Verbindungsauflösung übergeben.

```
procedure TForm1.Disconnect1Click(Sender: TObject);
begin
  Capil.DisconnectLine (1);
end;
```

Wenn eine physikalische Verbindung getrennt wird, ruft die Komponente den Event ***OnDisconnect*** auf, beim Trennen einer logischen Verbindung wird ***OnBDDisconnect*** aufgerufen.

In beiden Fällen steht im Übergabeparameter der Grund / Ursache für die Verbindungsauflösung.

Eine logische Verbindung wird immer vor der physikalischen trennt.

```
procedure TForm1.CapilDisconnect(Sender: TObject; const Grund: Cardinal);
begin
  WriteLn (LogFile, 'Disconnect: ' + Capil.GetDisconnectReasonStr (Grund));

  if (recvflag) then
    CloseFile (recvfile);

  recvflag := false;
end;
```

Wenn die Anwendung von sich aus die Verbindung getrennt hat, wird als Grund immer 0 übergeben.

## Prozeßverwaltung

In der 32-Bit-Version der Komponente wird die CAPI-Messageverarbeitung in einen eigenen Thread, parallel zur Anwendung, durchgeführt.

Wenn innerhalb der Behandlungsroutinen Capi-Ereignis auf VCL Routinen zugegriffen wird, müssen daher alle Threads der Anwendung zuvor synchronisiert werden.

```

procedure TForm1.CheckAnruf;
var
  line : String;
begin
  with LastCall do begin
    line := Copy (CallFrom,3,Length (CallFrom) - 2);
    line := line + ' für ' + Copy (CallTo,2,Length (CallTo) - 1);

    AnruferDisplay.Lines.Add (line);

    LastCallOk := (Service in [ANALOG_TELEPHONY, DIGITAL_TELEPHONY]);
  end;
end;

function TForm1.CapiHandlerRing(Sender: TObject; const CallDaten : TCallDaten): Cardinal;
begin
  LastCall := CallDaten;

  with Sender as TCapi20Handler do Synchronize (Form1.CheckAnruf);

  if not (LastCallOk) then
    CapiHandlerRing := IGNORE_CALL
  else begin
    CapiHandler.EnableListen ([]);

    CapiHandler.B1Protokol := TRANS_64K;
    CapiHandler.B2Protokol := BITTRANS;
    CapiHandler.B3Protokol := B3_TRANSPARENT;

    CapiHandlerRing := ACCEPT_CALL
  end;
end;

```

Um innerhalb eines Events auf Ereignisse der CAPI-Komponente zu warten, reicht es nicht aus, die Windows-Messagequeue mit ***Application.ProcessMessage*** abzufragen, da dadurch die CAPI-Messageverarbeitung nicht angestoßen wird. Die Komponente stellt dafür eine eigene Methode zur Verfügung. Mit ***ProcessCapiMessage*** wird sowohl die Windows-Messagequeue als auch die des CAPI-Treibers abgefragt.

Alternativ können in den einzelnen Events Messages an das jeweilige Hauptformular geschickt werden. Die zugehörigen Messagemethoden des Formulars kann dann alle weiteren Aktionen ausführen.

```

procedure Form1.Capi20BConnect(Sender: TObject; const NCCI: Longint);
begin
  DatenKanal := NCCI;
  PostMessage (TWinControl (Owner).Handle,wm_user + 1,0,0);
end;

procedure Form1.TransferStart (var Message : TMessage);
var i: Integer;
begin
  InfoText.Lines.Add('Übertragung wird gestartet !');

  TransferFile (SendFiles[i]);

  Capi20.DisconnectLine(1);

  InfoText.Lines.Add ('Übertragung beendet !');
End;

```

## Komponente TCAP120Base

Die Komponente TCAP120Base ist für die Kommunikation mit der CAPI zuständig. Sie schickt ihr übergeben Messages an die CAPI, nimmt Messages entgegen und leitet sie weiter. Eine Verarbeitung der empfangen Daten erfolgt nicht innerhalb der Komponente. Das Entgegennahmen und Weitergabe der Messages erfolgt asynchron über einen eigen Thread (Win32) oder über eine unsichtbares Window (Win16).

## Published Eigenschaften

### **CapiAvailable : Boolean;**

Wenn ein CAPI-Treiber geladen und betriebsbereit ist, liefert dieses Property *true*, ansonsten *false*. Nur wenn *true* zurückgegeben wird, kann die Komponente auf die CAPI zugreifen.

### **ApplicationRegistered : Boolean;**

Sobald die Anwendung beim CAPI mit **RegisterApplication** angemeldet wurde, steht dieses Property auf *true*. Die Eigenschaft **ApplicationID** liefert dann die ID, mit welcher die Anwendung bei der CAPI angemeldet ist.

### **AutoRegister : Boolean;**

Wenn diese Property auf *true* gesetzt ist, wird beim Erzeugen einer Instanz von TCapi20Handler automatisch die Anwendung beim CAPI angemeldet.

Wenn dies nicht gewünscht wird, muß das Property auf *false* gesetzt werden.

Dieses Verfahren funktioniert nur beim Benutzen des Objektinspectors und wenn die Komponente über den Designer ins Formular eingebunden wird.

Defaultmässig steht **AutoRegister** auf *true*.

### **ApplicationID : Boolean;**

Dies ist die ID, mit welcher die Anwendung bei der CAPI angemeldet ist. Jede Instanz der Komponente erhält eine eigene ID.

Sie ist erst gültig, wenn die Komponente bei der CAPI mit **RegisterApplication** bei der CAPI angemeldet wurde.

Sie kann beim Debuggen oder zur Laufzeit zum Identifizieren der einzelnen Instanzen der Komponenten herangezogen werden.

Wenn beim Erzeugen einer Instanz von TCapi20Handler das Property **AutoRegister** auf *true* steht werden automatisch die Zwischenpuffer für den CAPI-Treiber angelegt, und die Anwendung bei der CAPI angemeldet.

Daher sollten mindestens die Eigenschaften **LogicalConnection**, **BDataBlocks** und **BDataBlockSize** korrekt definiert sein.

### **Controller : Cardinal;**

Die CAPI 2.0 kann mehrere ISDN-Kontroller pro Rechner verwalten, die von 1 bis n durchnummeriert sind. Die Nummernvergabe erfolgt über den CAPI-Treiber.

Defaultmässig verwendet die Komponente den ersten Kontroller.

Eine Änderung dieses Wertes führt zum Verbindungsabbruch und Neuanmeldung beim CAPI-Treiber.

Mit der Funktion **GetAnzahlController** kann zur Laufzeit festgestellt werden, wie viele ISDN-Kontroller im System vorhanden sind.

**BDataBlocks: Cardinal;**

Maximale Anzahl der Datenblöcke, die das CAPI zwischenspeichern kann.

Eine Änderung dieses Wertes führt zum Abbruch aller bestehenden Verbindungen und initialisiert die CAPI neu.

Der Defaultwert ist 4.

**BDataBlockSize : Cardinal;**

Die Größe eines Zwischenpuffers in der CAPI.

Eine Änderung dieses Wertes führt zum Abbruch aller bestehenden Verbindungen und initialisiert die CAPI neu.

Dieser Wert legt gleichzeitig die maximale Framegröße der B-Protokolls fest.

Daher darf das Property *BufferSize* nicht größer als *BDataBlockSize* gewählt werden.

Der Defaultwert ist 2048.

**LogicalConnection : Cardinal;**

Maximale Anzahl der zugelassenen B-Kanal-Verbindungen für diese Anwendung.

Eine Änderung dieses Wertes führt zum Abbruch aller bestehenden Verbindungen und initialisiert die CAPI neu.

Bei passiven Karten, die nur für den S0-Bus geeignet sind, kommt es teilweise zu seltsamen Effekten, wenn versucht wird, mehr als 4 logische Verbindung zuzulassen.

Der Defaultwert ist 4.

## Puplic Eigenschaften

Die hier aufgeführten Eigenschaften stehen nur zur Laufzeit zur Verfügung und deren Werte können nur gelesen werden.

**CapiAvailable : Boolean;**

Mit dieser Eigenschaft kann man überprüfen, ob ein CAPI-Treiber vorhanden ist und einwandfrei funktioniert. Wenn ja, wird *true* zurückgegeben, ansonsten *false*.

**DTMFAvailable : Boolean;**

Wenn der CAPI-Treiber die Erkennung von DTMF-Tönen bei Sprachverbindungen unterstützt. Liefert diese Eigenschaft *true*, ansonsten *false*.

**ApplicationRegistered : Boolean;**

Sobald die Anwendung beim CAPI-Treiber angemeldet ist, liefert dieses Property *true*.

## Published Events

**In der Version für Win 95 / NT werden alle Events der Komponente aus innerhalb eines eigenen Thread aufgerufen.**

**Da diverse VCL-Routinen von Delphi nicht mehrfach aufgerufen werden dürfen, sollte die Event-Funktionen und -Prozeduren auf die Verwendung von VCL-Routinen verzichten.**

**Falls dies doch notwendig wird, kann die Anwendung z.B. per PostMessage über das CAPI-Ereignis informiert werden.**

**Die Event-Routinen sollten auf jeden Fall so kurz wie möglich gehalten werden, um die Laufzeit des CAPI-Threads nicht unnötig zu verlängern, da ansonsten die gesamte Anwendung ins Stokken gerät.**

## Messagebehandlung

### MessageReceived : TCapiMessageEvent

```
TCapiMessageEvent = procedure (Sender:TObject;
                               const CommandID, SubCommand, MsgID : Cardinal;
                               var Daten : ReceiveDatenType);
```

Sobald diesem Event eine Methode zugewiesen wird, schaltet sich der interne CAPI-Nachrichtenhandler der Komponente ab, und alle eingehenden Nachrichten der CAPI werden über diesen Event an das Anwenderprogramm weitergereicht.

Für das Entgegennehmen und Verarbeiten der Nachrichten ist dann die Anwendung selber zuständig.

Falls nur einige Kommandos speziell behandelt werden soll, aber die Funktionalität des Nachrichtenhandler erhalten bleiben soll, können die beiden Events **OnConfirm** und **OnIndication** aus der Komponente **TCapi20Handler** eingesetzt werden.

Eine Nachricht wird durch zwei Parameter, das Kommando (*CommandID*) und die Unterteilung (*SubCommand*) definiert.

Die Informationen werden in der Struktur *Daten* übergeben, die für jedes Kommando einen varianten Untertypen enthält.

Bei der Bearbeitung der eingehenden Nachrichten muß zwischen IND-Nachrichten und CONF-Nachrichten unterschieden werden.

Die IND-Nachrichten (SubCommand = IND) sind Nachrichten die das CAPI vom ISDN-Vermittlungsrechner erhält, und die Aktionen auf dem Netz anzeigen.

Sie müssen in jedem Fall mit der Methode **SendResponsMessage**, unter Benutzung des *MsgID*, quittiert werden.

Die CONF-Nachrichten (SubCommand = CONF) sind Quittierungen des CAPI-Treibers auf Kommandos der Anwendung. Sie enthalten wichtige Informationen und Fehlermeldungen.

Diese Nachrichten müssen nicht quittiert werden.

Beispiel:

```
procedure CapiMessage(Sender: TObject; const Cmd, SubCmd : Cardinal; var Daten:
IndicationDatenType);
var resp : RespondDatenType;
begin
  if (SubCmd = CONF) then begin
    case Cmd of
      CONNECT      : ...;
      .
      .
      DATA_B3     : ...;
    end;
  end if (SubCmd = IND) then begin
    case CommandID of
      CONNECT      : ...;
      DISCONNECT_B3 : ...;
      DISCONNECT   : ...;
```

```

        DATA_B3      : ...;
        INFO          : ...;

    end;

    TCapi(Sender).SendResponseMessage (Cmd, SubCmd, MsgID, resp);
end;
end;

```

Folgende Nachrichten kennt das CAPI:

Bezeichnung	Beschreibung
ALERT_REQ	Wartezustand für einen eingehenden Anruf
CONNECT_REQ	Einen Teilnehmer anwählen
CONNECT_IND	Eingehender Anruf
CONNECT_ACTIVE_IND	Bestätigung, wenn der Teilnehmer den Anruf angenommen hat.
DISCONNECT_REQ	Verbindung auflösen
DISCONNECT_IND	Die Gegenstelle hat die Verbindung aufgelöst
LISTEN_REQ	Auf eingehende Anrufe warten
INFO_REQ	Gewünschte Info-Elemente definieren
INFO_IND	Eine der gewünschten Info-Elemente steht an
FACILITY_REQ	Ansteuerung eines Handsets und Versenden von DTMF-Wahltonen
FACILITY_IND	Das Handset wurde bedient, oder DTMF-Wahltonen empfangen
SELECT_B_PROTOCOL_REQ	Definieren der Verbindungsprotokolle auf der Ebene 1 bis 3
CONNECT_B3_REQ	Eine B-Kanal Verbindung anfordern
CONNECT_B3_IND	Eine B-Kanal Verbindung wird von der Gegenstelle angefordert
CONNECT_B3_ACTIVE_IND	Eine B-Kanal Verbindung wurde erfolgreich aufgebaut
CONNECT_B3_T90_ACTIVE_IND	Eine B-Kanal Verbindung mit T70 oder T90 wurde erfolgreich aufgebaut
DISCONNECT_B3_REQ	Eine B-Kanal Verbindung abbauen
DISCONNECT_B3_IND	Eine B-Kanal Verbindung wurde von der Gegenstelle abgebaut
DATA_B3_REQ	Daten über den B-Kanal verschicken
DATA_B3_IND	Daten aus dem B-Kanal stehen an
RESET_B3_REQ	Eine B-Kanal Verbindung zurücksetzen, aber nicht abbauen
RESET_B3_IND	Das Zurücksetzen einer B-Kanal Verbindung wird angefordert

#### MessageDump : TCapiDumpEvent

```

TCapiDumpEvent = procedure (Sender:TObject;
    const CommandID, SubCommand : Cardinal;
    var Daten : ReceiveDataType);

```

Zu Debuggzzwecken wird bei jedem Empfangen einer CAPI-Nachricht dieser Event aufgerufen. Über die Methoden *GetCommandStr* kann die Bezeichnung des Kommandos bestimmt werden. In diesem Event sollte keinerlei Daten verarbeitet werden, da dies den Ablauf der Nachrichtenhandler stören könnte.



## Fehlerbehandlung

### OnError : TCapiErrorEvent;

TCapiErrorEvent = procedure (Sender:TObject; const CommandID, SubCommand, ErrorNumber : Cardinal);

Wenn beim Verarbeiten der CAPI-Meldungen ein Fehler aufgetreten ist, wird dieser Event aufgerufen.

Mit den Parametern *CommandID* und *SubCommand* kann über die Methode *GetCommandStr* die CAPI-Message, welche den Fehler verursacht hat, im Klartext bestimmt werden.

Falls es sich dabei um eine interne Fehlermeldung der Komponenten handelt, werden die Parameter *CommandID* und *SubCommand* auf 0 gesetzt.

Der Parameter *ErrorNumber* gibt die Fehlerursache an, deren Klartext über die Methode *GetErrorStr* bestimmt werden kann.

Beispiel:

```
procedure CapiError(Sender: TObject; const Cmd,SubCmd,Error: Cardinal);
var line : String;
begin
  line := '';

  if (Cmd <> 0) then
    line:= TCapi(Sender).GetCommandStr(Cmd,SubCmd) + ':';

  line := line + TCapi(Sender).GetErrorStr(Error);

  MessageDLG (line, mtWarning, [mbOK], 0);
end;
```

## Protected Methoden

### **Release**

Die Basisklasse TCapi20Base ruft beim Abmelden der Anwendung immer diese virtuelle Methode auf. Die Komponente TCapi20Handler überlädt diese Funktion und informiert damit die Anwendung über die Auflösung von noch bestehenden Verbindungen im B- und D-Kanal. Als Ursache für den Verbindungsabbruch wird \$8003 angegeben.

### **ProcessCapiMessages;**

Sobald innerhalb der Komponente auf ein Ereignis der CAPI gewartet wird, muß diese Funktion anstelle von Application.ProcessMessage aufgerufen werden, damit sowohl die Message- Queue von Windows als auch die der CAPI abgefragt werden.

## Public Methoden

### An- und Abmeldung bei der CAPI

#### **RegisterApplication;**

Hierüber wird die Anwendung beim CAPI angemeldet.

Jegliche Kommunikation ist erst nach dem Anmelden möglich.

Die Anwendung muß sich nur dann selber anmelden, wenn beim Erzeugen der Instanz die Eigenschaft **AutoRegister** nicht auf *true* gesetzt ist oder die Komponente erst zur Laufzeit erzeugt wurde.

Nach erfolgreicher Anmeldung steht das Property **ApplicationRegistered** auf *true*.

Die Eigenschaft **ApplicationID** liefert dann die ID, mit welcher die Anwendung bei der CAPI angemeldet ist.

#### **ReleaseApplication;**

Die Anwendung wird beim CAPI abgemeldet.

Beim Abmelden werden alle bestehenden Verbindungen im ISDN-Netz aufgehoben, und alle Datenbuffer freigegeben.

Beim Abmelden der Anwendung ruft die Basiskomponente die virtuelle Funktion **Release** auf, die in TCapi20Handler überladen wurde. Diese Funktion setzt alle Daten für die offenen Verbindungen zurück, und benachrichtigt die Anwendung für die Events **OnBDisconnect** und **OnDisconnect**.

Nach dem Abmelden steht das Property **ApplicationRegistered** auf *false*.

## Allgemeine Funktionen

### Synchronize (Method: TThreadMethod);

Da die CAPI-Komponente alle Events aus einem eigene Thread heraus aufruft, muß beim Einsatz von VCL-Routinen zuerst alle Threads synchronisiert werden. Dies gilt insbesondere beim Einsatz von Delphi 3.0

Die Funktion **Synchronize** arbeitet analog zur gleichnamigen Funktion in der Klasse TThread.

Beispiel

```
procedure TForm1.CheckAnruf;
var
  line : String;
begin
  with LastCall do begin
    line := Copy (CallFrom,3,Length (CallFrom) - 2);
    line := line + ' für ' + Copy (CallTo,2,Length (CallTo) - 1);

    AnruferDisplay.Lines.Add (line);

    if (Service in [ANALOG_TELEPHONY, DIGITAL_TELEPHONY]) then
      LastCallOk := true
    else LastCallOk := false
  end;
end;

function TForm1.CapiHandlerRing(Sender: TObject; const CallDaten : TCallDaten): Cardinal;
begin
  LastCall := CallDaten;

  with Sender as TCapi20Handler do Synchronize (Form1.CheckAnruf);

  if not (LastCallOk) then
    CapiHandlerRing := IGNORE_CALL
  else begin
    CapiHandler.EnableListen ([]);

    CapiHandler.B1Protokol := TRANS_64K;
    CapiHandler.B2Protokol := BITTRANS;
    CapiHandler.B3Protokol := B3_TRANSPARENT;

    CapiHandlerRing := ACCEPT_CALL
  end;
end;
```

### GetCommandStr (const Comand,SubCommand:Cardinal):String;

Diese Funktion liefert für das Kommando *Command* und die Unterteilung *SubCommand* den passenden Text.

### GetErrorStr (const Error:Cardinal):String;

Diese Funktion liefert für alle Fehler des CAPI's eine Klartextmeldung.

### GetLastError () : Integer;

Den letzten, von der CAPI gemeldeten Fehler auslesen.

Wenn seit dem letzten Abfragen kein Fehler aufgetreten ist, wird der Wert 0 zurückgegeben.

Der gespeicherte Fehlercode wird, analog zu IOResult, nach dem Auslesen auf 0 gesetzt.

### GetVersion : String;

Diese Funktion liefert die Version des CAPI-Treibers als String.

**GetSerialNumber : String;**

Diese Funktion liefert den Seriennummer des CAPI-Treibers als String.

**GetHersteller : String;**

Diese Funktion liefert die Herstellerbezeichnung als String.

**GetAnzahlController : Cardinal;**

Zurückgegeben wird die Anzahl der erkannten ISDN-Kontroller im System

**GetOptionen (const ControllerNr : Cardinal) : Cardinal;**

Auslesen aller unterstützten Erweiterungen des CAPI-Treibers.

Ausgelesen werden können Handset-Unterstützung (Bit 2) und die Unterstützung von DTMF-Wahltöne (Bit 3).

Über das Bit 0 kann zwischen internem (ISDN-Karte) und externem Kontroller unterschieden werden.

**GetAnzahlBCannel (const ControllerNr : Cardinal) : Cardinal;**

Die Anzahl der benutzbaren B-Kanäle.

Bei einem normalen S0-Anschluß sind dies immer 2.

**GetSupportedB1Protokoll (const ControllerNr : Cardinal) : LongInt;**

Zurückgegeben wird die Information über die unterstützten B1-Protokolle.

Die Kodierung erfolgt Bitweise.

Bit 0	64 kBit HDLC
Bit 1	64 kBit transparent
Bit 2	V.110 asynchron
Bit 3	V.110 synchron mit HDLC
Bit 4	T.30, Analog G3-Fax-Emulation

**GetSupportedB2Protokoll (const ControllerNr : Cardinal) : LongInt;**

Zurückgegeben wird die Information über die unterstützten B2-Protokolle.

Die Kodierung erfolgt Bitweise.

Bit 0	X.75
Bit 1	Transparent
Bit 2	SDLC
Bit 3	LAPD
Bit 4	T.30, Analog G3-Fax-Emulation

**GetSupportedB3Protokoll (const ControllerNr : Cardinal) : LongInt;**

Zurückgegeben wird die Information über die unterstützten B3-Protokolle.  
Die Kodierung erfolgt Bitweise.

Bit 0	Transparent
Bit 1	T.90NL
Bit 2	X.25 DTE-DTE auch ISO 8208
Bit 3	X.25 DCE
Bit 4	T.30, Analog G3-Fax-Emulation
Bit 5	T.30 extended, Analog G3-Fax-Emulation mit Faxabruf-Erweiterungen

## Messageverwaltung

### DoCapiMessage;

Mit dieser Methode wird das CAPI auf anstehende Nachrichten abgefragt, die dann durch den Messagehandler bearbeitet werden.

Diese Funktion sollte nicht direkt aufgerufen werden.

### SendRequestMessage (CommandID : Cardinal; var Daten) : Boolean;

Mit dieser Funktion werden Kommandos an das CAPI verschickt.

Der Parameter *CommandID* definiert nur das Kommando selber, die Unterteilung erfolgt in der Funktion automatisch.

Der Parameter Daten sollte einer der folgenden vordefinierten Typen sein:

Kommandotyp	Beschreibung
CONNECT	ConnectReqType
CONNECT_B3	ConnectB3ReqType
DISCONNECT	DisconnectReqType
DISCONNECT_B3	DisconnectB3ReqType
LISTEN	ListenReqType
SELECT_B_PROTOCOL	SelectBProtocolReqType
DATA	DataReqType
DATA_B3	DataB3ReqType
FACILITY	FacilityReqType
INFO	InfoReqType
RESET_B3	ResetB3ReqType

### SendResponseMessage (CommandID, MsgID : Cardinal; var Daten) : Boolean;

Mit dieser Funktion werden erhaltene IND-Nachrichten quittiert.

Der Parameter *CommandID* definiert nur das Kommando selber, die Unterteilung erfolgt in der Funktion automatisch.

Damit das CAPI auch die zu quittierende Nachricht eindeutig identifizieren kann, muß als *MsgID* der Wert, der in der IND-Nachricht übergeben wurde, eingesetzt werden.

Der Parameter Daten sollte einer der folgenden vordefinierten Typen sein:

Nachrichtentyp	Beschreibung
CONNECT	ConnectRespType
CONNECT_ACTIVE	ConnectActiveRespType
CONNECT_B3	ConnectB3RespType
DISCONNECT	DisconnectRespType
DISCONNECT_B3	DisconnectB3RespType
DATA	DataRespType
DATA_B3	DataB3RespType
INFO	InfoRespType
RESET_B3	ResetB3RespType

**GetMessage (var CommandID, SubCommand, MsgID : Cardinal;  
var Daten : IndicationDatenType):Cardinal;**

Diese Funktion holt eine Nachricht vom CAPI ab.

Falls keine Nachricht ansteht wird der Wert \$1002 zurückgegeben, ansonsten 0.

Die Parameter *CommandID* und *SubCommand* definieren die Nachricht, der Parameter *MsgID* wird für die Quittierung benötigt und in der Struktur *Daten* befinden sich die Informationen der Nachricht.

Nach Bearbeitung muß jede erhalten Nachricht mit der Funktion *SendResponssMessage* quittiert werden.



## Komponente CAPI20Handler

Die Komponente TCAPI20Handler übernimmt die gesamte Verarbeitung aller CAPI-Messages und stellt der Anwendung ein Interface zur Verfügung, über welches in einfachster Weise eine Verbindungen auf- und abgebaute werden kann.

Über diese Verbindung können mit einfachen Funktionen Daten zur Gegenstelle übertragen und vor ihr empfangen werden.

Eingehende Anrufe werden der Anwendung unmittelbar signalisiert. Diese kann den Anruf dann annehmen oder ablehnen.

Die gesamte Kommunikation wird über Ereignisse abgewickelt, so dass die Anwendung nie auf die CAPI warten muss.

Über Eigenschaften und Zugriffsfunktionen kann zu jedem Zeitpunkt geprüft werden, ob und welcher Art von Verbindung besteht.

## Eigenschaften

### **Buffers : Cardinal;**

Anzahl der Zwischenspeicher für Senden und Empfangen von Daten.

Der Wert muß vor dem Aufbauen von Verbindungen auf der B-Ebene definiert werden.

Beim Versenden von Daten werden diese zuerst in den Sendepuffer übernommen, damit die Anwendung ohne Wartezeit weiter arbeiten kann.

Beim Empfangen von Daten werden diese unmittelbar vom CAPI übernommen und in die Empfangspuffer abgelegt.

Die Anwendung kann sie dann zu einem beliebigen Zeitpunkt abholen.

Der Defaultwert ist 2.

### **BufferSize : Cardinal;**

Die Größe der Zwischenpuffer für Senden und Empfangen.

Die Größe muß kleiner gleich dem Wert **BDataBlockSize** für die Buffer des CAPI's sein. Falls ein größerer Wert angegeben wird, begrenzt die Komponente ihn automatisch.

Der Wert muß vor dem Aufbauen der B-Ebene definiert werden.

Der Defaultwert ist 2048.

### **AutoBConnect : Boolean;**

Wenn dieses Property auf *true* gesetzt ist, versucht der CAPI-Nachrichtenhandler der Komponente nach dem erfolgreichen Verbindungsaufbau automatisch eine B-Kanal-Verbindung mit den Protokollen **B1Protokol**, **B2Protokol** und **B3Protokol** aufzubauen.

Defaultwert ist *true*.

### **B1Protokol: TProtokolB1;**

TProtokolB1 = (HDLC\_64K, TRANS\_64K, V110\_ASYNCH, V110\_SYNCH, T30\_B1, INV\_HDLC\_64K, ADD\_56K);

Das zu benutzende Protokoll auf der B1-Ebene.

Das Protokoll muß vor dem Aufbau einer B1-Verbindung definiert sein.

Diese Protokollebene ist für die physikalische Übertragung der Daten zuständig.

Für die Datenübertragung wird im allgemeinen HDLC\_64K und für Sprachverbindungen

TRANS\_64 eingesetzt

Für das Setzen der obengenannten Dienste (TELEFONY, DATA\_64K und FAX3) stellt die Komponente die Funktion **SetProtokol** zur Verfügung.

#### **B2Protokol : TProtokolB2;**

TProtokolB2 = (X75,BITTRANS,SDLC,LAPD,T30\_B2,PPP,B2\_TRANSPARENT);

Das zu benutzende Protokoll auf der B2-Ebene.

Das Protokoll muß vor dem Aufbau einer B2-Verbindung definiert sein.

Auf dieser Protokollebene erfolgt die Datenpaketbildung und die Fehlersicherung.

Für die Datenübertragung wird normal X.75 eingesetzt, wobei die maximale Datenblockgröße durch das Property **BufferSize** bestimmt wird.

Für Sprachverbindungen kommt das Protokoll BITTRANS zum Einsatz.

#### **B3Protokol : TProtokolB3;**

TProtokolB3 = (B3\_TRANSPARENT,T90NL,ISO8208,X25,T30\_B3);

Das zu benutzende Protokoll auf der B3-Ebene.

Das Protokoll muß vor dem Aufbau einer B3-Verbindung definiert sein.

In dieser Ebene erfolgen protokollspezifische Umsetzungen der Daten.

Für Datenübertragung und Sprachverbindungen wird B3\_TRANSPARENT benutzt, d.h. die Daten werden direkt übernommen.

#### **Infos : TInfoSet;**

TCapInfo = (Gebuehren,DatumZeit,Display,UserToUser,Facility,DisconnectCause,RufStatus,CalledNumber);

Set mit allen freigebenden Informationen der Vermittlungsstelle.

Nur die Informationen die hier vorgegeben sind, werden der Anwendung mitgeteilt, wenn sie vom CAPI empfangen wurden.

Folgende Information können freigegeben werden:

Information	Beschreibung
Gebuehren	Die Angefallenen Gebühren für ein Gespräch. Normalerweise erscheint dieses Info erst am Ende des Gespräches.
DatumZeit	Das aktuelle Datum und die Uhrzeit zu Beginn eines Gespräches
Display	Die Gegenstelle hat Daten für das Display geschickt
UserToUser	Es wurden Userdaten auf dem D-Kanal empfangen
Facility	Informationen vom Handset oder DTMF-Töne
DisconnectCause	Ursache für die Trennung der Verbindung
RufStatus	Aktueller Status eines Anrufes

Sobald eine Verbindung aufgebaut wird, werden alle vorgegebenen Infos aktiviert

#### **OwnNumber : TCallNumber;**

TCallNumber = String [32];

In diesem Property wird die eigene, für den jeweiligen Anruf zu benutzte MSN-Nummer abgelegt.

Bei ITR6-Anschlüssen wird hier eine einstellige EAZ eingetragen.

Wenn die Nummer dem Angerufenen nicht bekannt gegeben werden soll, kann dieser Eintrag vor dem Aufruf von **Call** gelöscht werden.

Wenn eine ungültige Nummer eingesetzt wird, überträgt die TELEKOM-Vermittlung automatisch die erste zugeordnete MSN.

**OwnSubAdr : TSubAddress;**

TSubAddressr = String [32];

Dies ist die eigene SubAdresse, die immer beim Verbindungsaufbau neben der eigenen Rufnummer der Gegenstelle mitgeteilt wird.

Dies Adresse wird vermutlich auch ohne gebührenpflichtige Zusatzmerkmal zur Gegenstelle übertragen.

Zusammen mit der Rufnummer kann zum Beispiel der Anruf eindeutig identifiziert werden, oder es können mehrere Anwendungen auf den selben Dienst und die gleiche Rufnummer unterschiedlich reagieren.

**CapiDTMF : Boolean**

Über diese Eigenschaft kann festgelegt werden, ob die DTMF-Erkennung bei Sprachverbindungen über die CAPI erfolgen soll, oder durch die, in der Komponente integrierte, Funktion..

Zur Zeit bieten nur wenige Treiber - z.B. AVM - die diese Erweiterung bieten. Ob die Erkennung unterstützt wird, kann man über die Eigenschaft *DTMFAvailable* abfragen.

**NebenStelle : Boolean**

Mit diesem Property wird festgelegt ob die ISDN-Karte des PC direkt am NTBA oder über eine Tk-Anlage mit internem S0-Bus betrieben wird.

Wenn die Karte an einem internen S0-Bus angeschlossen ist, muß für jede externe Verbindung eine oder mehrere Ziffern für die Amtsholung vorweg gewählt werden. Diese kann im Property *AmtNr* vorgegeben werden.

Der Defaultwert ist false.

**AmtNr : TCallNumber**

Wenn die Karte am internen S0-Bus angeschlossen ist, muss für jede externe Verbindung eine oder mehrere Ziffern vor der eigentlichen Rufnummer gewählt werden. Mit diesem Property kann man die dafür notwendigen Ziffern vorgeben

Sie werden aber erst dann benutzt, wenn die gewünschte Rufnummer länger ist, als im Property *ExternLen* vorgegeben.

Der Defaultwert ist '0'.

**ExternLen : Cardinal**

Diese Eigenschaft legt die minimale Länge einer Telefonnummer fest, ab der automatisch die Ziffern für die Amtsholung vorweg gewählt werden. Dadurch ist sichergestellt, dass man auch eine Verbindung zu einem internen Anschluss aufbauen kann, da die zugehörigen Rufnummern im allgemeinen nur 2- oder 3-stellig sind.

Der Defaultwert ist 3.

**UserData : Pointer;**

Mit dieser Eigenschaft kann man einer Instanz der Komponente eine, von Benutzer definierte, Datensatz zuordnen, der zur Laufzeit einen einfachen Zugriff in den Eventroutinen erlaubt.

## Public Eigenschaften

Die folgenden Eigenschaften liefern aktuelle Zustände der Komponente. Sie sind alle read only und stehen nur zur Laufzeit zur Verfügung.

### PLCI : LongInt;

Das Handle für die aktuelle bestehende physikalische Verbindung.

Falls keine Verbindung besteht, steht **PLCI** auf -1.

Dieses Handle wird nur benötigt, wenn die CAPI direkt über die Funktionen **SendRequestMessage** und **SendResponseMessage** angesprochen wird.

### NCCI : BKanalType;

Das Handle für die aktuelle bestehende logische Verbindung. Die Eigenschaft ist erst gültig, wenn mit **OnBConnect** eine B-Kanal-Verbindung aufgebaut wurde.

Falls keine solche Verbindung besteht, steht **NCCI** auf -1.

Dieses Handle wird nur benötigt, wenn die CAPI direkt über die Funktionen **SendRequestMessage** und **SendResponseMessage** angesprochen wird.

### Verbindung : TVerbindung;

TVerbindung = (Keine, Active, Passive);

Anhand dieses Eigenschaften kann die Art des Verbindungsaufbaues unterschieden werden.

Wenn die Anwendung die Verbindung von sich aus mit der Funktion **Call** veranlaßt hat, ist **Verbindung** = *Active*, wenn sie den Anruf entgegengenommen ist **Verbindung** = *Passive*.

Falls zur Zeit keine Verbindung besteht liefert das Property den Werte *Keine*.

### ConnectedService : TCapiService;

TCapiService =  
(ALL, TELEPHONY, ANALOG\_TELEPHONY, DIGITAL\_TELEPHONY, FAX3, FAX4, DATA\_64K, DATA\_56K, PACKET\_MODE, UNKNOWN);

Sobald eine Verbindung aufgebaut wird, enthält dieses Property den zugehörigen Dienst.

Solange keine Verbindung aufgebaut ist, liefert diese Property den Wert UNKNOWN.

### ConnectStatus : TConnectStatus

TConnectStatus = (ConnectNone,  
ConnectDWait, ConnectD,  
ConnectBWait, ConnectB,  
DisconnectD, DisconnectB)

Der aktuelle Status einer Verbindung. Das selbe Ergebnis liefert auch die Funktion **GetConnectStatus**.

Zustand	Beschreibung
ConnectNone	Es besteht keine Verbindung
ConnectDWait	Nach dem Ausführen eines Anrufes mit <b>Call</b> oder nach der Annahme in <b>OnRing</b> , wird auf eine D-Kanal-Verbindung mit der Gegenstelle gewartet.
ConnectD	Es besteht eine Verbindung auf dem D-Kanal

Zustand	Beschreibung
ConnectNone	Es besteht keine Verbindung
ConnectBWait	Nach dem Aufrufen von <i>OpenBConnect</i> , wird auf eine B-Kanal-Verbindung mit der Gegenstelle gewartet.
ConnectB	Es besteht eine Verbindung auf dem B-Kanal
DisconnectB	Die aktuelle B-Kanal-Verbindung wird gerade abgebaut
DisconnectD	Die aktuelle D-Kanal-Verbindung wird gerade abgebaut

**ConnectedB1Protokol : TProtokolB1**

Das Ebene 1 Protokoll der aktuellen Verbindung.

Dies entspricht dem aktuellen Wert der Eigenschaft *B1Protokol* beim Aufbau der B-Kanal-Verbindung.

**ConnectedB2Protokol : TProtokolB2**

Das Ebene 3 Protokoll der aktuellen Verbindung.

Dies entspricht dem aktuellen Wert der Eigenschaft *B2Protokol* beim Aufbau der B-Kanal-Verbindung.

**ConnectedB3Protokol : TProtokolB3**

Das Ebene 3 Protokoll der aktuellen Verbindung.

Dies entspricht dem aktuellen Wert der Eigenschaft *B3Protokol* beim Aufbau der B-Kanal-Verbindung.

## Published Events

### Verbindungsauf- und abbau

#### D-Kanal-Verbindungen

##### OnRing : TCapiRingEvent;

TCapiRingEvent = function (Sender : TObject; const CallDaten : TCallDaten) : Cardinal;

Wenn die Anwendung die Rufannahme freigegeben hat und ein Anruf für die vorgegebenen Dienste eingeht, wird dieser Event aufgerufen.

Alle Daten über die Anruf werden in einer Struktur vom Typ *TCallDaten* übergeben.

```
TCallDaten = record
  Service      : TCapiService;
  AddService : Cardinal;
  CallFrom     : TCallNumber;
  SubFrom      : TSubAddress;
  CallTo       : TCallNumber;
  SubTo        : TSubAddress;
end;
```

Der Wert *Service* gibt dem, vom Anrufer gewünschten Dienst an, in *ServiceAdd* werden allfällige Zusatzinformationen über den Dienst angeben.

Zur Zeit wird das Feld *ServiceAdd* von den Vermittlungsstellen nicht benutzt. Zu Testzwecken übergibt die Komponente in diesem Eintrag die Servicenummer, die von der Vermittlungsstelle übertragen wurde.

In *CallFrom* wird die Nummer des Anrufers übergeben.

Die erste beiden Zeichen der Nummer enthält den Rufnummerentyp und erst ab dem dritten Zeichen folgt die eigentliche Rufnummer.

In *CallTo* wird die angerufene Nummer abgelegt. Hier beginnt die eigentliche Nummer ab dem zweiten Zeichen. Bei 1TR6-Anschlüssen wird hier nur die einstellige EAZ angegeben

Über die Werte *SubFrom* und *SubTo* kann eine erweiterte Adressierung - SubAdressierung - durchgeführt werden.

Hierüber besteht die Möglichkeit, eingehende Anrufe genauer zu selektieren, oder Zugangsberechtigungen zu überprüfen.

Die SubAdressierung ist ein erweiterter Dienst der Telekom und muß extra beantragt werden.

Versuche habe aber gezeigt, daß die eigen SubAdresse beim Anruf zur Gegenstelle übertragen wird.

Diese Erweiterung steht nur in der Pro Version der Komponente zur Verfügung.

An Hand der Rufnummer und dem gewünschten Dienst können nun die Zwischenpuffer und die Übertragungsprotokolle definiert werden.

Für eine Rufannahme muß eine 0 zurückgegeben werden, bei Ablehnung ein Wert  $\neq 0$ , der gleichzeitig den Grund für die Ablehnung angibt.

Für den Rückgabe werde sind folgende Konstanten in capi2con.pas definiert:

Konstante	Beschreibung
ACCEPT_CALL	Der Anruf wird angenommen
WAIT_ACCEPT_CALL	Der Anruf wird für die Anwendung reserviert, ohne aber eine Verbindung aufzubauen. Dies kann zu einem späteren Zeitpunkt mit <i>AnswerCall</i> geschehen. Der Anrufer hört derweil das Klingelzeichen.
IGNORE_CALL	Der Anruf wird ignoriert, es können aber andere Netzteilnehmer den Ruf entgegennehmen.
CLEARING_REJECT_CALL	Der Anruf wird nicht entgegengenommen
BUSY_REJECT_CALL	Der Angerufene ist schon besetzt
REQ_NOT_AVAILABLE_REJECT_CALL	Der gewünschte Service wird nicht unterstützt
FACILITY_REJECT_CALL	???
CHANNEL_REJECT_CALL	Auf dem gewünschten Kanal ist die Rufannahme nicht möglich
INCOMPATIBLE_REJECT_CALL	Die gewünschte Rufkonfiguration kann nicht unterstützt werden
OUT_OF_ORDER_REJECT_CALL	Der Angerufene ist zur Zeit nicht funktionsbereit

Für die bevorstehende Verbindung werden die, im Property *Infos* vordefinierten, Info-Elemente zugelassen.

Beispiel:

```
function Ring (Sender : TObject; const CallDaten : TCallDaten) : Cardinal;

var line,number : String;

begin
  with CallDaten do begin
    number := Copy (Caller,2,Length (Caller) - 1);
    line := 'Call from ' + number;

    WriteLn (LogFile,line);

    if (Service=DATA_64K) and (number='743278') then
      CapilRing := ACCEPT_CALL
    else CapilRing := IGNORE_CALL;
  end;
end;
```

### OnConnect : TCapiConnectEvent;

TCapiConnectEvent = procedure (Sender:TObject);

Wenn eine D-Kanal-Verbindung zustande kommt, wird dies über diesen Event mitgeteilt.

Diese zeigt an, daß ein Anruf oder eine Rufannahme erfolgreich war.

Wenn das Property *AutoBConnect* auf *true* gesetzt ist, versucht der komponenteninterne Nachrichtenhandler nun eine B-Kanal-Verbindung aufzubauen.

Wenn dieser automatische Verbindungsaufbau ausgeschaltet ist, muß eine B-Kanal-Verbindung über die Funktion *OpenBConnect* initiiert werden.

**OnDisconnect : TCapiDisconnectEvent;**

TCapiDisconnectEvent = procedure (Sender:TObject; const Grund : Cardinal);

Dieser Event wird immer dann aufgerufen, wenn die Gegenstelle von sich aus die Verbindung gelöst hat.

Der Parameter *Grund* gibt die Ursache für das Lösen der Verbindung an.

Wenn die Anwendung von sich aus die Verbindung gelöst hat, wird in *Grund* 0 zurückgegeben.

Folgende Ursachen sind möglich:

Code	Beschreibung
\$3301	Fehler beim Aufbauen der Ebene 1
\$3302	Fehler beim Aufbauen der Ebene 2
\$3303	Fehler beim Aufbauen der Ebene 3
\$3304	Eine andere Anwendung hat den Anruf entgegen genommen
\$3305	Gegenstelle baute Ebene 1 ab
\$3306	Gegenstelle baute Ebene 2 ab
\$3307	Gegenstelle baute Ebene 3 ab
\$3435	Kein Anschluß unter dieser Nummer
\$3438	Rufnummer hat sich geändert
\$343A	Gegenstelle nimmt nicht ab
\$343B	Gegenstelle ist besetzt
\$343E	Gegenstelle weist den Anruf ab
\$8003	Die Anwendung wurde bei der CAPI abgemeldet

Wenn als Ursache für den Verbindungsabbau der Wert \$8003 angegeben wird, bedeute dies, daß die Verbindung wegen der Abmeldung der Anwendung aufgehoben wurde.

In diesem Falle können dem CAPI keine weiteren Kommandos übertragen werden.



## B-Kanal-Verbindungen

### OnRequestBConnect : TCapiRequestBConnectEvent;

```
TCapiRequestBConnectEvent = function (Sender : TObject;
                                     const DestNCPI : CapiParameterType;
                                     var NCPI : CapiParameterType) : Boolean;
```

Wenn die Gegenstelle eine B-Kanal-Verbindung aufbauen will, informiert die Komponente die Anwendung mit diesen Event.

Der Parameter *DestNCPI* enthält die zusätzlichen Protokollinformation, die die Gegenstelle geliefert hat.

Falls die Verbindung zustande kommen soll, muß *true* zurückgegeben werden, bei *false*, wird der Aufbau der Verbindung verhindert.

Wenn eine B-Kanal-Verbindung aufgebaut werden soll, kann der CAPI über den Rückgabeparameter *NCPI* Zusatzinformation übergeben werden.

Wenn diesem Event keine Methode zugeordnet wird, erfolgt der Verbindungsaufbau in jedem Fall.

### OnSetBProtokollEvent : TSetBProtokollEvent;

```
TSetBProtokollEvent = procedure (Sender:TObject; var ProtokollParam : CapiBProtocolType);
```

Dieser Event wird immer vor dem Setzen der Protokoll-Parameter aufgerufen.

Falls die Anwendung für das gewünschte B-Kanal-Protokoll besondere, vom Standart abweichende Parameter setzen will, können diese im Parameter *ProtokollParam* eingetragen werden, Der Datenaufbau muß dem Format der jeweiligen CAPI-Protokolldefinition entsprechen.

Dieser Event wird nur in der Pro-Version der Komponente unterstützt.

### OnGetBProtokollEvent : TGetBProtokollEvent;

```
TGetBProtokollEvent = procedure (Sender:TObject; const NCPI : CapiParameterType);
```

Jedes Mal, wenn die Komponente eine B-Kanal-Verbindung aufgebaut hat, wird dieser Event aufgerufen.

Im Parameter *NCPI* wird dabei die Protokollinformationen , in der Form, wie sie die CAPI liefert übergeben

Dieser Event wird nur in der ProVersion der Komponente unterstützt.

### OnBConnect : TCapiBConnectEvent;

```
TCapiBConnectEvent = procedure (Sender:TObject; const NCCI : BKanalType);
```

Sobald eine B-Kanalverbindung erfolgreich aufgebaute wurde, wird dieser Event aufgerufen.

Der Parameter *NCCI* ist das Handle für die neu aufgebaute Verbindung und wird bei jeden Aufrufen von *Read*, *Write*, *Flush* usw. benötigt.

Beispiel:

```
procedure TForm1.CapilDataConnect(Sender: TObject; const Kanal : BKanalType);
begin
  rcvflag := False;
```

```
DataKanal := Kanal;  
end;
```

**OnBDisconnect: TCapiBDisconnectEvent;**

TCapiBDisconnectEvent = procedure (Sender:TObject; const NCCI : BKanalType; const Grund : Cardinal);

Wenn eine B-Kanal-Verbindung abgebaut worden ist, wird dieser Event aufgerufen.  
Der Parameter *NCCI* definiert die abgebaute Verbindung.

Wenn als Ursache für den Verbindungsabbau der Wert \$8003 angegeben wird, bedeute dies, daß die Verbindung wegen der Abmeldung der Anwendung aufgehoben wurde.  
In diesem Falle können dem CAPI keine weiteren Kommandos übertragen werden.

**OnBReset : TCapiBResetEvent;**

TCapiBResetEvent = procedure (Sender:TObject; const NCCI : BKanalType);

Immer wenn die Gegenstelle eine Zurücksetzen der B-Kanal-Verbindung initiiert wird dieser Event ausgelöst.

Er kann dazu benutzt werden um Parameter eines Übertragungsprotokolles neu zu initialisieren. Und Datenpuffer zu leeren.

Die Anwendung kann über die Funktion ***ResetBConnect*** von sich aus die Verbindung zurücksetzen.

## Datenübertragung

### OnReceive : TCapiReceiveEvent;

TCapiReceiveEvent = procedure (Sender:TObject; const NCCI : BKanalType; const Count : Cardinal);

Sobald das CAPI einen neuen Datenblock übergibt, wird dieser Event aufgerufen.

Die betreffende Verbindung wird über den Parameter *NCCI* definiert, der bei der Erzeugen festgelegt und der Anwendung über den Event **OnBConnect** mitgeteilt wurde.

Der Parameter *Count* gibt an, wie viele Bytes zur Übernahme anstehen.

Die eigentlichen Daten können dann mit der Methode **Read** abholen.

Beispiel:

```
procedure CapilReceive(Sender: TObject; const NCCI : BKanalType; const Count: Cardinal);
var
  readanz : Cardinal;
  buffer : array [0..2048] of byte;
begin
  if (Count > 0) then begin
    readanz := TCapi(Sender).Read (NCCI,buffer,Count);
    if (readanz > 0) then
      BlockWrite (filevar,buffer,readanz);
  end;
end;
```

Die Daten können aber auch erst zu einem späteren Zeitpunkt abgeholt werden.

### OnSend : TCapiSendEvent;

TCapiSendEvent = procedure (Sender:TObject; const NCCI : BKanalType);

Dieser Event wird immer dann aufgerufen, wenn ein Zwischenbuffer der Verbindung *NCCI* frei geworden ist.

Dies bedeutet, daß das CAPI die Daten übernommen hat, nicht aber daß sie bereits bei der Gegenstelle angekommen sind. Der Einfachheit halber kann man aber davon ausgehen, da ansonsten die Verbindung durch die Vermittlungsstelle automatisch getrennt würde.

## Informationen

### OnConfirm : TConfirmEvent

```
TConfirmEvent = procedure (Sender:TObject;
                           const CommandID : Cardinal;
                           const Daten : ConfirmDatenType;
                           var ErrorCode : Cardinal;
                           var Done : Boolean)
```

Jede REQ-Message die an die CAPI empfängt, quittierte sie mit einer CONF-Message. Deren Inhalt gibt unter anderem darüber Auskunft, ob alle Daten korrekt waren, und ob die Message korrekt ausgeführt werden konnte. Wenn alles in Ordnung war, enthält *ErrorCode* den Wert 0. Das quittierte Kommando wird in *CommandID* und die zugehörigen Daten im Parameter *Daten* übergeben.

Normalerweise erfolgt die Verarbeitung dieser Quittierungen im Nachrichtenhandler der Komponente, der Event selber sollte der Applikation nur Informationen liefern. Daher wird der Parameter *Done* beim Aufrufen des Events mit *false* initialisiert.

Wenn aus irgendwelchen Gründen die Verarbeitung der Quittierungen nicht im Nachrichtenhandler statt finde sollte, muss der Parameter *Done* auf *true* gesetzt werden.

### OnIndication : TIndicationEvent

```
TIndicationEvent = procedure (Sender:TObject;
                              const CommandID : Cardinal;
                              const MsgID : Cardinal;
                              const Daten : IndicationDatenType;
                              var Done : Boolean)
```

Dieser Event wird immer dann ausgelöst, wenn von der CAPI-Schnittstelle eine IND-Message empfangen wurde.

Im Parameteren *CommandID* wird das CAPI-Kommando übergeben, *MsgID* enthält die CAPI-Messagenummer, welche zum quittieren mit einer CONF-Message benötigt wird, und in *Daten* sind die Informationen der Message enthalten.

Wenn das Flag *Done* beim verlassen der Eventbehandlungsroutine auf *true* gesetzt ist, wird die IND-Message vom aufrufenden Handler nicht mehr weiter bearbeitet.

Über diesen Event können einzelne CAPI-Messages speziell behandelt werden.

### OnFacility : TCapiFacilityEvent

```
TCapiFacilityEvent = procedure (Sender      : TObject;
                               const NCCI   : BKanalType;
                               const Selector : Cardinal;
                               const Parameter: FacilityIndParam);
```

Für jede empfangene Facilityinformation wird dieser Event aufgerufen.

Der Parameter *NCCI* gibt die zugehörige B-Kanalverbindung an, in *Selector* steht der Facility-Typ und in *Parameter* die zugehörigen Daten.

### OnInfo : TCapiInfoEvent;

```
TCapiInfoEvent = procedure (Sender:TObject; const Info : Cardinal ; var InfoDaten : TInfoDaten);
```

Wenn ein vorher freigegebene Information übermittelt wurde, wird dieser Event aufgerufen.

Der Parameter *Info* enthält die Bezeichnung des Information, in *InfoDaten* wird der Inhalt der Information als String zurückgegeben.

Die einzelnen Informationen können über die Funktion **EnableListen** freigegeben bzw. gesperrt werden.

Für die Unterscheidung der einzelnen Möglichkeiten wird *Info* benötigt.

Folgende Infos sind möglich:

Kennung	Bedeutung
\$0008	Hiermit wird der Grund für den Verbindungsabbruch mitgeteilt
\$4000	Gebühren Die bisher angefallenen Gebühren in Einheiten. Die Einheiten selber stehen als LongInt im Parameter <i>InfoDaten</i> .
\$0029	Aktuelle Uhrzeit und das Datum Die Daten werden im Format tt.mo.jj-hh:mm ss angegeben, jeweils in einem Byte Je nach Vermittlungsstelle werden die Sekunden angegeben oder nicht.

Den genauen Inhalt der einzelnen Informationen ist dem Autor zur Zeit leider noch nicht bis ins letzte Detail bekannt, da diese nicht vollständig dokumentiert sind.

#### OnReceiveDTMF : TReceiveDTMFEvent

TReceiveDTMFEvent = procedure (Sender:TObject; const Taste : char)

Bei Sprachverbindungen spielen die Mehrfrequenzwahlöne - DTMF - eine wichtige Rolle.

Sie werden für die Fernabfrage von Anrufbeantwortern, für die Steuerung von Vermittlungsautomaten usw. benutzt.

Mit Mehrfrequenzwahlöne können 16 verschiedene Zeichen dargestellt werden. Neben den normalen Wählfziffern 0 .. 9 können die Zeichen \* und # und die Buchstaben A .. D übermittelt werden.

Bei jede Verbindung, die mit dem B2-Protokoll BITRANS aufgebaut wurde, untersucht die Komponente den empfangenen Datenstrom auf das Vorhandensein von Mehrfrequenzwahlöne.

Sobald ein gültiges Wählton erkannt wurde, löst die Komponente den Event **OnReceiveDTMF** aus und übergibt als Parameter das entsprechende Zeichen.

Beispiel:

```
procedure TForm1.CapiHandlerReceiveDTMF(Sender: TObject;
const Taste: Char);
begin
  if (CodeStatus = 0) then begin
    if (CodeStr [CodeIdx] <> taste) then
      CodeIdx := 1
    else begin
      Inc (CodeIdx);
      if (CodeIdx > Length (CodeStr)) then begin
        CodeStatus := 1;
        PostMessage (Handle,WM_USER, 0, 0);
      end;
    end;
  end else if (CodeStatus = 1) then begin
    if (taste = '*') then
      SkipPlay := -1
    else if (taste = '#') then
      SkipPlay := 1;
  end;

  CAPIDisplay.Lines.Add ('DTMF : ' + taste + '(' + IntToStr (CodeIdx) + ')');
end;
```

Dieser Event wird nur in der Pro-Version der Komponente unterstützt.

## Public Methoden

### Verbindungsauf und Abbau

**SetProtokol (const Service : TCapiService) : Boolean;**

Diese Funktion setzt für die Dienste TELEFONY, DATA\_64k und FAX3 die Standard-Protokolle auf allen drei B-Ebenen.

Sie liefert als Rückgabewert *true*, wenn das Protokoll auf allen drei Ebenen unterstützt wird, und *false*, wenn das Protokoll vom CAPI-Treiber nicht unterstützt wird.

Für den Dienst DATA\_64k werden folgende Protokolle gesetzt:

```
B1Protokol := HDLC_64k;
B2Protokol := X.75;
B3Protokol := B3_TRANSPARENT;
```

Wenn für eine Verbindung nicht die Standardprotokolle eingesetzt werden sollen, kann man über die Eigenschaften *B1Protokol*, *B2Protokol* und *B3Protokol* die gewünschten Protokolle einstellen.

**EnableListen (const Service : TServiceSet) : Boolean;**

```
TCapiService = ( ALL,
  TELEPHONY,
  ANALOG_TELEPHONY,
  DIGITAL_TELEPHONY,
  FAX3,
  FAX4,
  DATA_64K,
  DATA_56K,
  PACKET_MODE,
  UNKNOWN);
```

```
TServiceSet = set of TCapiService;
```

Mit dieser Funktion wird der Komponente mitgeteilt, bei welche Anrufe der Event **OnRing** ausgelöst werden soll.

Über das Set *Service* wird festgelegt, auf welche Dienste reagiert werden soll.

Wenn leeres Set (= [ ] ) übergeben wird, reagiert die Komponente auf keine weiteren Anrufe.

Nach der Entgegennahme eines Anrufes sollte die Anwendung die weitere Rufannahme mit **EnableListen ([ ])** unterbinden, da die Komponente nur ein aktive Verbindung zulässt.

Es besteht aber auch die Möglichkeit alle weitere Anruf mit BUSY\_REJECT\_CALL abzuweisen. Der Anrufer erhält dann die Besetztmeldung.

Über diese Funktion werden auch die Info-Elemente aktiviert oder deaktiviert.

Sobald das Property **Info** verändert, ruft die Komponente automatisch die Funktion mit dem zuletzt benutzten Serviceset auf.

**AnswerCall (const AcceptCode : Cardinal) : Boolean;**

Entgegennahme eines anstehenden Anrufes.

Sobald eine Anruf erkannt wird, ruft die Komponente das Property **OnRing** auf.  
Die damit verbundene Funktion liefert als Rückgabewert die Information über das weitere Vorgehen.  
Wenn mit WAIT\_ACCEPT\_CALL geantwortet wird, wird dies dem Anrufer über die ALERT-Message mitgeteilt. und der Anruf kann zu einem späteren Zeitpunkt angenommen werden.

Wenn der Anruf erfolgreich entgegengenommen wurde, liefert die Funktion *true* zurück, ansonsten *false*. Gleichzeitig meldet die Komponente einen Fehler, welcher über den Event **OnError** erfaßt werden kann.

**Call (const Number : TCallNumber;  
const SubAdr : TSubAddress;  
const Service : TCapiService) : Boolean;**

Diese Funktion ruft einen Teilnehmer unter der Nummer *Number* mit dem gewünschten Dienst *Service*.

Nach erfolgreichem Absetzen der Rufnummer wird der Zustand der Eigenschaft **ConnectStatus** auf *ConnectWait* gesetzt.

Falls der Teilnehmer erreichbar ist, den Dienst unterstützt und den Ruf annimmt, wird das CAPI mit der Nachricht CONNECT antworten und die Komponente ruft den Event **OnConnect** auf.  
Der Zustand der Eigenschaft **ConnectStatus** wird dann auf *ConnectD* gesetzt.

Falls die Gegenstelle den Anruf nicht entgegen nehmen will, oder den gewünschten Dienst nicht unterstützt, meldet die CAPI einen Verbindungsabbau welcher über den Event **OnDisconnect** signalisiert wird.

Über die Parameter *SubAdr* kann eine erweiterte Adressierung - Subadressierung - der Gegenstelle durchgeführt werden.

Die Subadressierung ist ein erweiterter Dienst der Telekom und muß extra beantragt werden.  
Versuche habe aber gezeigt, daß die eigen Subadresse beim Anruf zur Gegenstelle übertragen wird.

Diese Erweiterung steht nur in der Pro-Version der Komponente zur Verfügung.

Beim Einsatz des automatischen Verbindungsaufbaus über das Property **AutoBConnect**, versucht die Komponente selbständig eine B-Kanal-Verbindung mit den Übertragungsprotokollen **B1Protokol**, **B2Protokol** und **B3Protokol** aufzubauen  
Das Zustandekommen dieser Verbindung wird dann über den Event **OnBConnect** mitgeteilt.  
Der Zustand der Eigenschaft **ConnectStatus** wird dann auf *ConnectB* gesetzt.

Wenn der Anruf erfolgreich ausgeführt wurde konnte, d.h. die Vermittlungsstelle hat den Ruf entgegen genommen, liefert die Funktion *true* zurück, ansonsten *false*.  
In diesem Fall meldet die Komponente gleichzeitig einen Fehler, welcher über den Event **OnError** erfaßt werden kann.

**OpenBConnect (const NCPI : CapiParameterType) : Boolean;**

Erzeugen einer neuen B-Kanal-Verbindung, mit den Protokollparametern aus den Eigenschaften **B1Protokol**, **B2Protokol** und **B3Protokol**.

Die Funktion liefert immer dann *true* zurück, wenn es theoretisch möglich ist, eine Verbindung aufzubauen. Ob die Verbindung auch wirklich zustande kommt, ist dadurch noch nicht gesichert.

Mit dem Parameter *NCPI* können Zusatzinformationen an die CAPI weitergegeben werden. Deren Aufbau und Inhalt hängt vom dem verwendeten Protokoll ab. Im Normalfall kann ein Leerstring übergeben werden.

Wenn die Verbindung wirklich zustande gekommen ist, wird dies über den Event **OnBConnect** mitgeteilt und der Zustand der Eigenschaft **ConnectStatus** wird dann auf *ConnectB* gesetzt.

Der dabei übergebene Verbindungshandel muß dann bei allen Kommunikationen mit diese Verbindung abgegeben werden.

Es können pro D-Kanal Verbindung maximal eine B-Kanal-Verbindungen geöffnet werden.

Wenn die Verbindung erfolgreich aufgebaut wurde, liefert die Funktion *true* zurück, ansonsten *false*. Gleichzeitig meldet die Komponente einen Fehler, welcher über den Event **OnError** erfaßt werden kann.

**CloseBConnect (const NCCI : BKanalType; const NCPI : CapiParameterType) : Boolean;**

Schließen einer B-Kanal-Verbindung. Dabei ist es unerheblich, ob die Anwendung diese Verbindung auch erzeugt hat.

Der Parameter *NCCI* definiert die Verbindung, die zu schließen ist.

Mit dem Parameter *NCPI* können Zusatzinformationen an die CAPI weitergegeben werden. Deren Aufbau und Inhalt hängt vom dem verwendeten Protokoll ab. Im Normalfall kann ein Leerstring übergeben werden.

Nach dem Aufruf der Funktion ist der Zustand der Eigenschaft **ConnectStatus** auf *DisconnectB* gesetzt.

Wenn die Verbindung vom CAPI aufgehoben wurde, wird dies über den Event **OnBDisconnect** mitgeteilt und der Zustand der Eigenschaft **ConnectStatus** wird dann auf *ConnectD* gesetzt.

Falls noch die CAPI noch Daten zu übertragen hat, wartet die Funktion, bis die CAPI den Verbindungsabbau quittiert.

Dies ist bei Verbindungen zu analogen Gegenstellen, z.B. G3-Faxgeräte, wichtig, da sonst Daten verloren gehen.

Wenn die B-Kanal Verbindung erfolgreich aufgebaut wurde, liefert die Funktion *true* zurück, ansonsten *false*. Gleichzeitig meldet die Komponente einen Fehler, welcher über den Event **OnError** erfaßt werden kann.

**ResetBConnect (const NCCI : BKanalType; const NCPI : CapiParameterType) : Boolean;**

Die Anwendung kann über diese Funktion von sich aus die B-Kanal-Verbindung *NCCI* zurücksetzen.



Mit dem Parameter *NCPI* können Zusatzinformationen an die CAPI weitergegeben werden. Deren Aufbau und Inhalt hängt vom dem verwendeten Protokoll ab. Im Normalfall kann ein Leerstring übergeben werden.

#### **DisconnectLine (const Grund : Cardinal) : Boolean;**

Mit dieser Funktion kann eine bestehende Verbindung aufgelöst werden.

Im Parameter *Grund* kann angegeben werden warum sie gelöst wurde. Im Normalfall wird der Wert 0 = "Normaler Verbindungsabbau" verwendet.

Wenn zum Zeitpunkt des Funktionsaufrufes noch aktive B-Kanal Verbindungen bestehen, werden dies zuerst über die Funktion **CloseBConnect** abgebaut.

Nach dem Aufruf der Funktion ist der Zustand der Eigenschaft **ConnectStatus** auf *DisconnectD* gesetzt.

Durch Aufruf der Funktion können auch Anwahlvorgänge vorzeitig beendet werden. Die Komponente löst auch in diesen Fällen dann den Event **OnDisconnect** aus.

Wenn die Verbindung erfolgreich abgebaut wurde, liefert die Funktion *true* zurück, ansonsten *false*. Gleichzeitig meldet die Komponente einen Fehler, welcher über den Event **OnError** erfaßt werden kann.

#### **GetDisconnectReasonStr (const Grund : Cardinal) : String;**

Für jeden Ursache, übergeben im Parameter *Grund*, wird eine Klartextmeldung zurück geliefert.

## Datenübertragung

Mit dem folgenden Funktionen wird die Datenübertragung über den B-Kanal realisiert. Alle Funktionen erwarten eine gültiges Verbindungshandle, welches eine bestehende Verbindung definiert. Dieses Handle wird mit den Event ***OnBConnect*** übergeben

**Read (const NCCI : BKanalType; var Daten; const Count : Cardinal) : Integer;**

Mit dieser Funktion werden Daten aus dem Empfangspuffer der Verbindung *NCCI* abgeholt. Es werden maximal *Count* Bytes nach Daten kopiert.

Die Funktion wartet nicht bis die gewünschte Anzahl Zeichen empfangen wurden, sondern kehrt sofort wieder zurück.

Zurückgegeben wird die Anzahl der gelesenen Bytes.

**ReadEx (const NCCI : BKanalType; var Daten; const Count : Cardinal; var Flags : Cardinal) : Integer;**

Mit dieser Funktion kann zusätzlich der Zustand das Flag im empfangenen Datenblock ausgelesen werden.

Ansonsten entsprechen die Parameter denen der zuvor beschriebenen Funktion ***Read***.

Gewisse Protokolle wie zum Beispiel ISO8208 (EFT-Protokoll) liefern je nach Zustand unterschiedliche Flags mit.

Im Normalfall ist das Flag aber auf 0 gesetzt.

**Write (const NCCI : BKanalType; var Daten; const Count : Cardinal; var Flags : Cardinal) : Integer;**

Mit dieser Funktion werden Daten über die B-Kanal-Verbindung mit dem Handle *NCCI* übertragen. Das Verbindungshandle wird beim Erzeugen der Verbindung zugewiesen und mit dem Event ***OnBConnect*** an die Anwendung weitergegeben.

Die Funktion kehrt erst zurück, wenn die Daten übergeben wurden.

Das heißt aber nicht, daß sie dann schon beim Empfänger angekommen sind.

Der Parameter *Daten* enthält die zu versendenden Daten und in *Count* steht die Größe der Daten in Bytes. Zurückgegeben wird die Anzahl der übertragenen Bytes.

Wenn die angegebene B-Kanal Verbindung nicht, oder nicht mehr aufgebaut ist, wird der Wert 0 zurückgegeben.

Falls zum Aufrufzeitpunkt keine Sendepuffer frei ist, wird solange gewartet, bis einer frei geworden ist.

Wenn das CAPI die Daten übernommen hat, wird der Event ***OnSend*** aufgerufen.

**WriteEx (const NCCI : BKanalType; var Daten; const Count : Cardinal; const Flags : Cardinal) : Integer;**

Mit dieser Funktion kann beim Übertragen von Datenblöcken mit dem zusätzlichen Parameter *Flags* der Zustand das Flag in der CAPI-Struktur gesetzt werden. Ansonsten entsprechen die Parameter denen der zuvor beschriebenen Funktion *Write*.

Dies wird für gewisse Protokolle wie zum Beispiel ISO8208 (EFT-Protokoll) benötigt. Im Normalfall kann das Flag auf 0 gesetzt werden.

**Flush (const NCCI : BKanalType) : Boolean;**

Diese Funktion verschickt alle noch anstehenden Daten im Sendepuffer der Verbindung *NCCI*.

Die Rückkehr erfolgt erst nach dem alle Puffer leer sind, oder die betreffende B-Kanal Verbindung abgebaut wurde.

**GetSendBytes (const NCCI : BKanalType) : Cardinal;**

Hierüber wird angegeben, wieviele Bytes in der Verbindung *NCCI* noch zum Versenden anstehen. Wenn noch Daten zu versenden ist, sollte normalerweise keine Verbindung aufgelöst werden, da sonst die Daten im Sendepuffer den Empfänger nicht erreichen.

**GetReceiveBytes (const NCCI : BKanalType): Cardinal;**

Hierüber wird angegeben, wie viele Bytes noch im Empfangspuffer der Verbindung *NCCI* zu Abholen anstehen.

**GetConnectStatus : TConnectStatus;**

TConnectStatus = (ConnectNone, ConnectWait, ConnectD, ConnectB)

Hierüber wird festgelegt welche Art von Verbindung im ISDN-Netz besteht.

Der Zustand ist nur dann korrekt, wenn der komponenteninterne automatische Verbindungsaufbau zum Einsatz kommt.

Zustand	Beschreibung
ConnectNone	Es besteht keine Verbindung
ConnectWait	Nach dem Ausführen eines Anrufes , wird auf die Rufannahme der Gegenstelle gewartet.
ConnectD	Es besteht eine Verbindung auf dem D-Kanal
ConnectB	Es besteht eine Verbindung auf dem B-Kanal

## Komponente TCapiProtokoll

Diese Komponente ist die Basis für alle Protokoll-Komponenten. Sie verwaltet für einen CAPI-Handler Komponenten alle Protokollinformationen.

Beim Verbindungsaufbau wird an Hand der Informationen in ***B1Protokol***, ***B2Protokol*** und ***B3Protokol*** die zugehörige Protokollkomponente aktiviert, welche die passenden Daten in die CAPI-Struktur einfügt.

Sie überlädt dazu zur Laufzeit die Events ***OnSetBProtokollEvent*** und ***OnGetBProtokollEvent*** des eingetragenen Handlers.

## Eigenschaften

### Handler : TCapi20Handler;

Dies ist die Instanz der zugehörigen CAPI-Komponente.

Dieses kann sowohl mit dem Objekt-Inspektor als auch programmgesteuert zur Laufzeit gesetzt werden.

Dieses Property erbt jede Komponente, die von TCapiProtokoll abgeleitet wird.

## Komponente TT30Protokoll

Diese Protokollkomponente wird immer dann aktiviert, wenn die Protokolle

**B1Protokoll** := T30\_B1

**B2Protokoll** := T30\_B2

**B3Protokoll** := T30\_B3

oder

**B3Protokoll** := T30\_EXT

benutzt werden soll.

Die Komponente überträgt beim Verbindungsaufbau automatisch die Daten aus **FaxSendInfo** in die CAPI-Struktur und identifiziert sich damit bei der Gegenstelle.

Das Standard Format für eine T.30-Fax-Übertragung ist SFF. Dies wird in einer Datei abgelegt, die aus einem Dokumentenkopf und ein oder mehreren Dokumentenseiten besteht. Jede dieser Seite enthält wiederum einen Kopf und den Seiteninhalt. Der Inhalt wird in Form eines komprimierten Images abgelegt. Die Komprimierung entspricht der internationalen Norm für G3-Fax.

Der Dokumentenkopf wird als Struktur mit fester Länge abgelegt.

```
TSFFHeader = packed record
  ID : LongInt;
  Version      : Byte;
  Reserve      : Byte;
  UserInfo     : Word;
  PageCount    : Word;
  FirstPage    : Word;
  LastPage     : LongInt;
  DataEnd      : LongInt;
end;
```

Der Kopf einer Faxseite beinhaltet Informationen über die Auflösung und die Gesamtlänge. Die Auflösung verändert sich innerhalb eines Dokumentes aber nie.

```
TSFFPageHeader = packed record
  ID              : Byte;
  HeaderLen       : Byte;
  VerticalResolution : Byte;
  HorizontalResolution : Byte;
  Coding          : Byte;
  Reserve         : Byte;
  MaxLineLen      : Word;
  PageLen         : Word;
  PrevPage        : LongInt;
  NextPage        : LongInt;
end;
```

Um eine SFF-Datei zu erstellen, benötigt man z.B. einen passenden Druckertreiber. Die meisten Hersteller liefern eine solchen in der Softwareunterstützung der ISDN-Karten mit. Z.B kann man beim Fritz!32 im Anwahl-Dialog in eine Datei drucken lassen.

Alternative besteht die Möglichkeit über die Komponente **SFFConvert** den Inhalt diverser Delphi-Komponenten in eine SFF-Datei auszugeben.

## Published Eigenschaften

### FaxRecvInfo : TFaxRecvInfo

```
TFaxResolution = (Standard, High);
TFaxFormat      = (SFF, PlainFax, OCX, DCX, TIFF, ASCII, ExtANSI, Bin);
TReceiverID     = String [128];
THeaderLine     = String [128];
TStationID      = String [128];
```

```
TFaxRecvInfo = record
    Baudrate      : Word;
    Resolution    : TFaxResolution;
    Format        : TFaxFormat;
    Pages         : Word;
    ReceiverID    : TReceiverID;
end;
```

Wenn ein G3-Fax über das T.30-Protokoll empfangen wurde, steht in diesem Property die Angaben über den Versender.

Die Information steht erst nach Abbau der B-Kanal-Verbindung zur Verfügung.

In *ReceiverID* steht die ID der Gegenstelle. Im allgemeinen wird da die gültigen Telefonnummer (z.B. ++49-xxx-xxxxxx) des Faxgerätes angegeben.

Der Wert *Baudrate* gibt an, mit welcher Geschwindigkeit das Fax übertragen wurde, in *Resolution* wird die Auflösung des Faxes angegeben und der Wert *Pages*, aus wieviel Seiten das Fax besteht.

An Hand von *Format* kann man feststellen, in welchem Format das Fax abgespeichert wurde.

### FaxSendInfo : TFaxSendInfo

```
TFaxBaudRate = (Navigation, b2400, b4800, b7200, b9600, b12000, b14400)
```

Navigation : Die zu benutzende Baudrate handeln die beiden Teilnehmer beim Verbindungsaufbaus untereinander aus.

```
TFaxSendInfo = class
    property Baudrate      : TFaxBaudRate;
    property Resolution    : TFaxResolution;
    property Format        : TFaxFormat;
    property StationID     : TStationID;
    property HeaderLine    : THeaderLine;
end;
```

Über diese Property werden die Daten für das Versenden eines G3-Faxes mit dem T.30 Protokoll eingestellt.

Mit *Baudrate* wird die maximal zulässige Verbindungsrate festgelegt. Beim Wert Navigation wird diese zwischen den beiden Verbindungspartnern ausgehandelt.

*Format* gibt das Format der zu übertragenden Faxdaten vor. Im allgemeinen wird SFF (standard fax format) verwendet.

All weiteren Formate werden in den seltensten Fällen von den CAPI-Treibern unterstützt.

Das Property *Resolution* legt fest, mit welcher Auflösung das Fax übertragen wird. Wenn die Faxdaten in Form einer SFF-Datei vorliegen, hat die Einstellung von *Resolution* keinen Einfluß auf die Übertragung, da die gültige Auflösung in der SFF-Datei abgelegt ist.

Über die Eigenschaft *StationID* wird die ID des eigenen Faxgerätes festgelegt. Diese Information wird beim Verbindungsaufbau zur Gegenstelle übertragen. Im allgemeinen besteht diese ID aus den international gültigen Telefonnummer (z.B. ++49-xxx-xxxxxx)

Der Text in *HeaderLine* wird auf jeder Seite des zu übertragenden Faxes als Kopfzeile eingefügt.

In jedem Fall müssen das Property *Format* korrekt eingestellt werden. Als Defaultwert enthalten es den Wert SFF.

Die SFF-Dateiwn erstellen alle FAX-Druckertreiber der div. ISDN-Kartenhersteller. Alternativ kann in der ProVersion die Komponente *TSFFFaxConverter* zum Erzeugen von SFF-Dateien benutzt werden.

Die Komponente selber stellt keinen passenden Druckertreiber zur Verfügung

## Komponente TV110Protokoll

Das V.110-Protokoll kann immer da eingesetzt werden, wo Daten zwischen einer analogen Modem und einer ISDN-Leitung ausgetauscht werden. Durch einsetzen von Füllbytes wird die effektive Datenrate auf den gewünschten Wert reduziert.

Viel ISDN-Karten unterstützen nur das V.110-Asynchron-Protokoll.

Das Protokoll wird z.B. in Deutschland für Datenübertragung im GSM-Netz eingesetzt.

Diese Protokollkomponente wird immer dann aktiviert, wenn die Protokolle

**B1Protokoll** := V110\_ASYNC oder V110\_SYNC

**B2Protokoll** := BITTRANS

**B3Protokoll** := B3\_TRANSPARENT

benutzt werden soll.

## Eigenschaften

**Rate** : Cardinal

Die zu benutzende Baudrate für die Modemverbindung  
default 38400;

**BitsPerChar** : TBitsPerChar

TBitsPerChar = 5..8;

Die Anzahl der Bits pro Zeichen.

Die Eigenschaft hat bei *B1Protokoll* = *V110\_SYNC* keine Bedeutung.  
default 8;

**StopBits** : TStopBits

TStopBits = 1..2;

Die Anzahl Stoppbits pro Zeichens

Die Eigenschaft hat bei *B1Protokoll* = *V110\_SYNC* keine Bedeutung.  
default 1;

**Parity** : TParity

TParity = (NoParity, OddParity, EvenParity);

Der einzusetzende Parity-Typ für die Generierung beim Senden und Überprüfung beim Empfangen.

Die Eigenschaft hat bei *B1Protokoll* = *V110\_SYNC* keine Bedeutung.  
default NoParity;



## Komponente TV120Protokoll

Das V.120-Protokoll kann - genau so wie V.110 - immer da eingesetzt werden, wo Daten zwischen einer analogen Modem und einer ISDN-Leitung ausgetauscht werden. Durch einsetzen von Füllbytes wird die effektive Datenrate auf den gewünschten Wert reduziert. Viel ISDN-Karten unterstützen nur das V.120-Asynchron-Protokoll.

Das Protokoll wird z.B. in der Schweiz für Datenübertragung im GSM-Netz eingesetzt.

Diese Protokollkomponente wird immer dann aktiviert, wenn die Protokolle

***B1Protokoll*** := V120\_ASYNCH oder V120\_TRANSPARENT

***B2Protokoll*** := -

***B3Protokoll*** := -

benutzt werden soll.

## Eigenschaften

## Komponente TModemProtokoll

Das TModemProtokoll kann mit Unterstützung der CAPI eine Verbindung zu einem analogen Modem aufbauen.

Zur Zeit wird dieses Modemprotokoll nur von der neuesten Teles-CAPI (Version 3.29) korrekt unterstützt.

Diese Protokollkomponente wird immer dann aktiviert, wenn die Protokolle

***B1Protokoll*** := -  
***B2Protokoll*** := B2\_MODEM  
***B3Protokoll*** := -

benutzt werden soll.

## Eigenschaften

**Rate** : Cardinal

Die zu benutzende Baudrate für die Modemverbindung  
default 38400;

**BitsPerChar** : TBitsPerChar

Die Anzahl der Bits pro Zeichen  
default 8;

**StopBits** : TStopBits

Die Anzahl Stoppbits pro Zeichens  
default 1;

**Parity** : TParity read

Der einzusetzende Parity-Typ für die Generierung beim Senden und Überprüfung beim Empfangen.  
default NoParity;

## Komponente TSFFFaxConverter (Nur Pro Version)

Mit der Komponente *TSFFFaxConverter* können Text und Graphikkomponente in SFF-kodierte Faxdokumente konvertiert werden.

Diese Dokumente können direkt über die CAPI zu einem analogen Faxgerät oder Faxmodem übertragen werden.

### Eigenschaften

#### **FileName** : TFileName

Über diese Eigenschaft kann man den Namen des aktuell geöffneten Faxdokumentes abfragen.

Falls zur Zeit kein Dokument geöffnet ist, liefert die Eigenschaft eine Leerstring.

Diese Eigenschaft kann nur gelesen werden.

#### **Pages** : Integer

Die Anzahl der Seiten im aktuell geöffneten Dokument.

Wenn das Dokument mit *CreateFax* erzeugt wurde, liefert diese Eigenschaft die Anzahl der eingefügten Seiten zurück.

Diese Eigenschaft kann nur gelesen werden.

#### **Resolution** : TFaxResolution

type TFaxResolution = (Std\_Resolution, High\_Resolution);

Dieses Property legt fest, ob das Faxdokument mit einer vertikalen Auflösung 96 - Std\_Resolution - oder 198 - High\_Resolution - dpi (Dots per Inch) erzeugt wird.

Normalerweise wird *Std\_Resolution* als 96 dpi benutzt.

Diese Eigenschaft kann gelesen und vor dem Erzeugen eines neuen Faxdokumentes mit *CreateFax* geschrieben werden.

#### **MassEinheit** : TMassEinheit

type TMassEinheit = (muMillimeter, muInche);

Mit dieser Eigenschaft wird die Masseinheit für die Angaben der Seitenränder festgelegt.

Wenn die Einheit gewechselt wird, rechnet die Komponente automatisch die jeweiligen Angaben in die neue Masseinheit um.

#### **SeitenRandRechts** : Double

Die Breite des rechten Seitenrandes bei der Konvertierung von ganzen Dokumenten mit der Funktion *ConvertToFax*.

Die Massangaben erfolgt in Millimeter oder Inch, abhängig vom Zustand der Eigenschaft **Masseinheit**.

#### **SeitenRandLinks** : Double

Die Breite des linken Seitenrandes bei der Konvertierung von ganzen Dokumenten mit der Funktion *ConvertToFax*.

Die Massangaben erfolgt in Millimeter oder Inch, abhängig vom Zustand der Eigenschaft **Masseinheit**.

**SeitenRandOben** : Double

Die Höhe des oberen Seitenrandes bei der Konvertierung von ganzen Dokumenten mit der Funktion **ConvertToFax**.

Die Massangaben erfolgt in Millimeter oder Inch, abhängig vom Zustand der Eigenschaft **Masseinheit**.

**SeitenRandUnten** : Double

Die Höhe des unteren Seitenrandes bei der Konvertierung von ganzen Dokumenten mit der Funktion **ConvertToFax**.

Die Massangaben erfolgt in Millimeter oder Inch, abhängig vom Zustand der Eigenschaft **Masseinheit**.

## Public Methoden

**OpenFax** (FName : TFileName) : Integer;

Das Faxdokument mit dem Dateinamen **FName** wird geöffnet.

Nach dem Öffnen können mit der Funktion *FaxToGraphic* die einzelnen Seiten in grafische Objekte übernommen werden.

**CreateFax** (FName : TFileName) : Integer;

Mit dieser Funktion wird ein neues Faxdokument in der, mit dem Property *Resolution* vordefinierten Auflösung mit dem Dateinamen **FName** erzeugt.

Nach dem Erzeugen des Dokumentes kann die gewählte Fauxauflösung nicht mehr verändert werden.

Die einzelnen Seiten können über die Funktion *GraphicToFax* in das Dokument aufgenommen werden.

Bevor das Dokument übertragen oder angezeigt werden kann, muss es mit der Funktion *CloseFax* geschlossen werden.

**CloseFax** : Integer;

Schliessen eines mit *CreateFax* oder *OpenFax* geöffneten Faxdokumentes.

Falls das Dokument mit *CreateFax* neu erzeugt wurde, wird vor dem Schliessen der Datei noch die Kopfdaten geschrieben. Erst dann ist das Dokument vollständig und kann übertragen oder angezeigt werden.

**GraphicToFax** (Source : TPersistent) : Integer;

Das grafische Objekt **Source** wird in ein geöffnetes Faxdokument ausgegeben.

**Source** kann zur Zeit nur vom Typen *TBitmap*, *TImage* und *TMetafile* sein.

Falls kein Faxdokument offen ist, wird -1 zurückgegeben.

**FaxToGraphic** (Ziel : TPersistent; const PageNr : Cardinal) : Integer;

Die Seite **PageNr** des, mit *OpenFax* geöffneten Faxdokumentes wird in das übergebene Grafikobjekt **Ziel** übertragen.

Zur Zeit werden die Objekttypen *TBitmap* und *TImage* unterstützt.

**ConvertToFax** (Source : TComponent; FName : TFileName) : Integer;

Es wird ein neues Faxdokument mit dem Namen **FName** erzeugt und darin das Objekt **Source** ausgegeben.

Als **Source** können zur Zeit nur Objekte vom Typ *TRichEdit* benutzt werden.

Mit den Eigenschaften **SeitenRandRechts**, **SeitenRandLinks**, **SeitenRandOben** und **SeitenRandUnten** kann man den Ausgabebereich einschränken.

Nach der vollständigen Konvertierung wird das Faxdokument geschlossen.

## Sprachübertragung im ISDN

Bei Sprachverbindungen im ISDN wird das zu übermittelnde Signal beim Sender digitalisiert und auf der Empfangsseite wieder zurückgewandelt.

Bei der Digitalisierung wird das Sprachsignal mit einer Samplerate von 8 kHz in 12 Bit Werte abgetastet.

Die so gewonnenen Signalwerte werden mit einer verlustbehafteten Komprimierung auf 8 Bit verkleinert und zum Empfänger geschickt.

Das eingesetzte Verfahren beruht auf einer logarithmischen Lautstärkenkomprimierung bei der davon ausgegangen wird, daß das menschliche Ohr bei lauten Tönen einen Pegelunterschied erst bei größeren Differenzen als bei leisen erkennt.

Die Komprimierung ist weltweit genormt, und unter den Bezeichnungen Law, uLaw und G.711 bekannt.

Für die Sprachübertragung werden folgende Protokolle benutzt:

```
B1Protokoll := TRANS_64K;  
B2Protokoll := BITTRANS;  
B3Protokoll := B3_TRANSPARENT;
```

Bedingt durch die Protokolleinstellung erfolgt keine Fehlerkorrektur.

Unmittelbar nach dem Verbindungsaufbau beginnt die Datenübertragung. In beide Richtungen werden jeweils 8000 Bytes pro Sekunde verschickt.

Dies bedeutet, daß bei einer Blockgröße von 2048 Bytes alle 250 ms der Event **OnReceive** aufgerufen wird.

Aus diesem Grund sollten die Daten möglichst umgehend abgeholt und gespeichert werden. Innerhalb der Receive-Routine sollte dann aus Zeitgründen keine Ausgaben o.ä. getätigt werden.

Solange die Applikation nicht selber Daten verschickt, überträgt die CAPI automatisch Datenpacket die einen Nullpegel darstellen.

Das verschicken von Samples erfolgt über die Funktion **Write**.

Aus Zeitgründen sollte auch da die Blockgröße von 2048 Bytes beibehalten werden.

Die Unit Law.pas enthält alle nötigen Routinen für die Komprimierung und Dekomprimierung der Signaldaten.

In der Unit Wave.pas befindet sich eine Komponente die das Benutzen von Wav-Dateien erleichtert.

## Komponente TWave

Die Komponente **TWave** stellt alle notwendigen Funktionen zur Verfügung, um Wave-Dateien zu lesen und zu schreiben.

### Eigenschaften

property IsOpen : Boolean;

Diese Property liefert immer dann true, wenn eine Wavedatei geöffnet ist.

property WaveSize : LongInt;

Die Summe aller Samples in Bytes.

property WaveSamples : LongInt;

Die Gesamtanzahl der Samples in der Wavedatei.

### Public Methoden

constructor Create;

Der Konstruktor der Komponente. Hier werden alle Daten initialisiert.

destructor Destroy;

Der Destruktor der Komponente. Falls die Wavedatei noch geöffnet ist, wird sie automatisch geschlossen. Wenn es sich dabei um eine neu erzeugte Datei handelt, wird auch der Kopfsatz korrekt geschrieben.

procedure SetFormat (const ChannelsAnz : Cardinal;  
                      const Rate : LongInt;  
                      const Size : TWaveSize);

Hiermit kann man das Format für eine neu zu erzeugende Wavedatei definieren. Die Funktion muss vor dem Erzeugen durch die Funktion *BuildWave* aufgerufen werden.

*ChannelsAnz* : Anzahl der aufzuzeichnenden Audiokanälen; Zulässig ist 1 oder 2

*Rate* : Die Abtastrate der Aufzeichnung.  
Gültig sind 8000, 11025, 12000, 16000, 22050, 24000, 44100 und 48000  
Zwischenwerte sind zwar zulässig aber nicht standardisiert.

*Size* : Die Grösse eines Samplewertes; Zulässig sind nur die Werte Bit\_8 und Bit\_16;

procedure GetFormat (Var ChannelsAnz : Cardinal;  
                      Var Rate : LongInt;  
                      Var Size : TWaveSize);

Hiermit kann das Format der aktuell geöffneten Wavedatei ausgelesen werden.  
Die Parameter haben die selbe Bedeutung wie bei *SetFormat*.

function Build (wavname : String) : Integer;

Mit dem Aufruf dieser Funktion wird eine neue Wavedatei erzeugt. Die Parameter müssen zuvor mit der Funktion *SetFormat* definiert werden.

Nach dem Erzeugen können die Samples mit der Funktion *Write* geschrieben werden.

Beim Schliessen der Datei mit *Close*, wird der Kopfsatz aktualisiert.

function Open (wavname : String) : Integer;

Öffnen einer Wavedatei ausschliesslich zum Lesen der Samples.

Das Format der Date kann nach dem Öffnen mit *GetFormat* abgefragt werden.

Die einzelnen Samples werden mit der Funktion *Read* ausgelesen.

Die Gesamtanzahl der Samples kann über das Property *WaveSamples* und die Filegrösse mit *WaveSize* bestimmt werden.

function Close : Integer;

Schliessen der Wavedatei.

Wenn es sich dabei um eine, mit der Funktion *Build*, neu erzeugte Datei handelt, wird auch der Kopfsatz korrekt geschrieben.

function Read (var Buffer; const maxanz : Cardinal) : Integer;

Lesen von *<maxanz>* Samples in den Puffer *<Buffer>*. Der Buffer muss ausreichend demissioniert sein. Ein Sample kann je nach Format aus 1 (8 Bit/mono), 2 (16 Bit/mono oder 8 Bit/stereo) oder 4 Bytes (16 Bit/stereo) bestehen.

Es kann nur gelesen werden, wenn die Datei mit der Funktion *Open* geöffnet wurde

function Write (var Buffer; const anz : Cardinal) : Integer;

Schreiben der Samples im Puffer *<Buffer>*. Der Parameter *<anz>* gibt die Anzahl der zu speichernden Samples vor.

Ein Sample kann je nach Format aus 1 (8 Bit/mono), 2 (16 Bit/mono oder 8 Bit/stereo) oder 4 Bytes (16 Bit/stereo) bestehen.

Es kann nur geschrieben werden, wenn die Datei mit der Funktion *Build* erzeugt wurde



## Unit Law.pas

Die Unit dient der Konvertierung von komprimierten Voicedaten. Die Komprimierung und Dekomprimierung erfolgt nach dem G.711 Format.

function **LawToInt** (lawwert : Byte) : SmallInt;

Hiermit werden die aus dem ISDN empfangenen komprimierten 12 Bit Samples in ein 16 Bit Sempel umgewandelt. Damit erfolgt eine automatische Amplitudenanpassung von 12 auf 16 Bit.

function **IntToLaw** (IntWert : SmallInt) : Byte;

Mit dieser Function wird 16 Bit Sample in einen komprimierten 12 Bit Samples umgewandelt. Auch hier erfolgt automatische Amplitudenanpassung von 16 auf 12 Bit.

## History

### 1.1 Erste veröffentlichte Version

### 1.2

- Die Funktionen RegisterApplication und ReleaseApplication hinzugefügt
- Änderung in der Thread-Verwaltung bei der Win 95/NT-Version

### 1.3

- Das Property ConnectedService hinzugefügt.
- Der CAPI-Thread bleibt bei Programmende nicht mehr hängen.

### 1.4

- Die Eigenschaften CapiAvailable und ApplicationRegistered eingeführt.

### 1.5

- Anpassung an Delphi 3
- Einführung der, für Delphi 3 notwendigen, Funktion Synchronize.

### 1.6

- Faxübertragung mit T.30 (nur Pro Version)
- SubAdressierung (nur Pro Version)
- DTMF-Erkennung (nur Pro Version)
- Weitere Änderung in der Thread-Verwaltung bei der Win 95/NT-Version

### 2.0 (Nur noch als Pro Version)

- Die Protokolle V.110 V.120 und analog Modem-Simulation werden unterstützt.
- Das erweiterte T.30-Protokoll wird unterstützt.
- Alle erweiterten Protokolle werden über eine eigene Komponente verwaltet.
- Automatische Unterstützung bei Betrieb an einer Nebenstelle
- Konvertierungskomponente für kodierten Faxdokumente
- Komponente für das Anzeigen von kodierten Faxdokumenten
- Unterstützung bei der Erzeugung zur Laufzeit

## - Index -

---

### I

ITR6 32; 36

---

### A

ACCEPT\_CALL 12; 37  
 AmtNr 33  
 AnswerCall 12; 37; 45  
 ApplicationID 18; 25  
 ApplicationRegistered 18; 20; 25  
 AutoBConnect 13; 31; 37  
 AutoConnect 45  
 AutoRegister 18; 25

---

### B

B1Protokol 27; 31; 35; 44; 45; 46; 50  
 B1Protokoll 13; 14; 51; 54; 55; 56; 60  
 B2\_MODEM 56  
 B2Protokol 27; 31; 32; 35; 44; 45; 46; 50  
 B2Protokoll 13; 14; 51; 54; 55; 56; 60  
 B3\_TRANSPARENT 32; 44; 60  
 B3Protokol 28; 31; 32; 35; 44; 45; 46; 50  
 B3Protokoll 13; 14; 51; 54; 55; 56; 60  
 BDataBlocks 18; 19  
 BDataBlockSize 18; 19  
 BITTRANS 43  
 BitsPerChar 54; 56  
 BITTRANS 32; 60  
 Buffers 31  
 BufferSize 19; 31; 32

BUSY\_REJECT\_CALL 44

---

## C

Call 10; 32; 34; 45  
CAPI20.DLL 6; 7  
CAPI2032.DLL 6; 7  
capi2bas.pas 7  
capi2con.pas 7; 37  
capi2han.pas 7  
capi2typ.pas 7  
CapiAvailable 18; 20  
CapiDTMF 33  
CapiProd.pas 8  
CloseBConnect 13; 46; 47  
CloseFax 59  
ConnectB 34; 49  
ConnectD 34; 49  
ConnectedB1Protokol 35  
ConnectedB2Protokol 35  
ConnectedB3Protokol 35  
ConnectedService 34  
ConnectNone 34; 49  
ConnectStatus 34; 45  
ConnectWait 34; 49  
Controller 18  
ConvertToFax 59  
Create 10  
CreateFax 59

---

## D

DATA\_64K 13; 44  
Datenübertragung 48  
Datum/Uhrzeit 43  
Dienst 7; 36; 45  
DisconnectB 34  
DisconnectD 34  
DisconnectLine 16; 47  
DoCapiMessage 29  
DSS1 6  
DTMFAvailable 20; 33

---

## E

EAZ 6  
EnableListen 11; 43; 44  
ExternLen 33

---

## F

FAX3 14; 44  
FaxRecvInfo 52  
FaxSendInfo 51; 52

FaxToGraphic 59  
FileName 57  
Flush 39; 49

---

## G

G.711 60  
Gebühren 43  
GetAnzahlBCannel 27  
GetAnzahlController 18; 27  
GetCommandStr 22; 23; 26  
GetConnectStatus 34; 49  
GetDisconnectReasonStr 47  
GetErrorStr 23; 26  
GetHersteller 27  
GetLastError 26  
GetMessage 30  
GetOptionen 27  
GetReceiveBytes 49  
GetSendBytes 49  
GetSerialNumber 26  
GetSupportedB1Protokoll 14; 27  
GetSupportedB2Protokoll 27  
GetSupportedB3Protokoll 28  
GetVersion 26  
GraphicToFax 59

---

## H

HDLC 27  
HDLC\_64K 31; 44

---

## I

IGNORE\_CALL 37  
Info 44  
Infos 32  
IntToLaw 63

---

## L

Law 60  
Law.pas 60; 63  
LawToInt 63  
LogicalConnection 18; 19

---

## M

MassEinheit 57  
MessageDump 22  
MessageReceived 21  
MSN 32

---

*N*

NCCI 34  
 NCPI 39; 46  
 NebenStelle 33

---

*O*

OnBConnect 13; 34; 39; 41; 45; 46; 47; 48  
 OnBDisconnect 16; 25; 40; 46  
 OnBReset 40  
 OnConfirm 21; 42  
 OnConnect 12; 37; 45  
 OnDisconnect 16; 25; 38; 45; 47  
 OnError 23; 45; 46; 47  
 OnFacility 42  
 OnGetBProtokollEvent 39; 50  
 OnIndication 21; 42  
 OnInfo 42  
 OnReceive 15; 41; 60  
 OnReceiveDTMF 43  
 OnRequestBConnect 39  
 OnRing 12; 34; 36; 44; 45  
 OnSend 41; 48  
 OnSetBProtokollEvent 39; 50  
 OpenBConnect 13; 34; 37; 46  
 OpenFax 59  
 OwnNumber 32  
 OwnSubAdr 32

---

*P*

Pages 57  
 Parity 54  
 PLCI 34  
 ProcessCapiMessage 17  
 ProcessCapiMessages 24  
 Pro-Version 39; 45  
 ProVersion 8; 39

---

*R*

Rate 54; 56  
 Read 15; 39; 41; 48  
 ReadEx 48  
 RegisterApplication 10; 18; 25  
 Release 24; 25  
 ReleaseApplication 25  
 ResetBConnect 40; 46  
 Resolution 57

---

*S*

SeitenRandLinks 57  
 SeitenRandOben 58  
 SeitenRandRechts 57  
 SeitenRandUnten 58  
 SendRequestMessage 29; 34  
 SendResponseMessage 21; 29; 34  
 SetProtokol 14; 31; 44  
 SFF 53  
 SFFConvert 51  
 SFF-Format 14  
 StopBits 54; 56  
 Synchronize 26

---

*T*

T.30 14; 27; 52; 53  
 T30\_B1 14; 51  
 T30\_B2 14; 51  
 T30\_B3 14; 51  
 T30\_EXT 14; 51  
 TBitsPerChar 54; 56  
 TCallDaten 12; 36  
 TCallNumber 32; 33; 45  
 TCapi20Base 6; 8; 10  
 TCapi20Handler 6; 8; 10; 18; 21; 24; 25; 50  
 TCapiBConnectEvent 39  
 TCapiBDisconnectEvent 40  
 TCapiBResetEvent 40  
 TCapiConnectEvent 37  
 TCapiDisconnectEvent 38  
 TCapiDumpEvent 22  
 TCapiErrorEvent 23  
 TCapiFacilityEvent 42  
 TCapiInfoEvent 42  
 TCapiMessageEvent 21  
 TCapiProtokoll 50  
 TCapiProtokoll.Handler 50  
 TCapiReceiveEvent 41  
 TCapiRequestBConnectEvent 39  
 TCapiRingEvent 36  
 TCapiSendEvent 41  
 TCapiService 7; 34; 44; 45  
 TConfirmEvent 42  
 TConnectStatus 49  
 TELEFONY 13; 44  
 TFaxRecvInfo 52  
 TFaxResolution 57  
 TFaxSendInfo 52  
 TGetBProtokollEvent 39  
 TIndicationEvent 42  
 TInfoSet 32  
 TModemProtokoll 56  
 TParity 54  
 TProtokolB1 31  
 TProtokolB2 32

TProtokolB3 32  
TRANS\_64 31  
TRANS\_64K 60  
TReceiveDTMFEvent 43  
TServiceSet 44  
TSetBProtokollEvent 39  
TSFFFaxConverter 14; 53; 57  
TSFFHeader 51  
TSFFPageHeader 51  
TStopBits 54; 56  
TSubAddress 32; 45  
TT30Protokoll 14; 51  
TV110Protokoll 8; 54  
TV120Protokoll 55  
TVerbindung 34  
TWave 6201-61  
TWave.Build 6201-62  
TWave.Close 6201-62  
TWave.Create 6201-61  
TWave.Destroy 6201-61  
TWave.GetFormat 6201-61  
TWave.IsOpen 6201-61  
TWave.Open 6201-62  
TWave.Read 6201-62  
TWave.SetFormat 6201-61  
TWave.WaveSamples 6201-61  
TWave.WaveSize 6201-61  
TWave.Write 6201-62

---

## *U*

uLaw 60  
UserData 10; 33

---

## *V*

V.110 27  
V110\_ASYNCH 54  
V110\_SYNCH 54  
Verbindung 34  
Verbindungsaufbau 44

---

## *W*

WAIT\_ACCEPT\_CALL 12; 45  
Wave.pas 60  
WIN32 7  
Write 15; 39; 48  
WriteEx 49

---

## *X*

X.75 13; 27; 32; 44