

TSP++ Version 2.2

Programming with the TSP++ Library

**Building Microsoft TAPI Service
Providers for Windows™ 3.1, Windows
95, and Windows NT 4.0**

TSP++

TAPI C++ Service Provider Library

Version 2.21a

Copyright © 1994-1999 JulMar Entertainment Technology, Inc.

All rights reserved

Updated 9/8/17 10:25 AM

Table of Contents

INTRODUCTION	1
HISTORY OF TAPI	1
TAPI SYSTEM COMPONENTS	1
<i>System Components</i>	1
<i>TSPI Components</i>	2
SYSTEM ARCHITECTURE OF TAPI	2
BASIC PROCESS FLOW	4
TAPI 1.X (16-BIT)	4
TAPI 2.X (32-BIT)	4
PURPOSE OF THE TSP++ LIBRARY	5
FILES IN THE TSP++ SDK.....	6
16-BIT INCLUDE FILES (INC16)	6
16-BIT SOURCE FILES (SRC\16BIT).....	6
32-BIT INCLUDE FILES (INC32)	7
32-BIT SOURCE FILES (SRC\32BIT).....	7
PRE-BUILT LIBRARIES (LIB)	7
SAMPLE SOURCE CODE (SAMPLES)	7
<i>ATSPV2</i>	8
<i>DSSP</i>	8
<i>ATSP32</i>	8
<i>DSSP32</i>	8
<i>BIN</i>	8
<i>INSTALL</i>	8
<i>1632THK</i>	8
CLASS LIBRARY ORGANIZATION.....	9
OBJECT LIFETIMES	9
TSP++ ORGANIZATION CHART	9
CLASS OVERRIDING	10
HANDLES USED TO COMMUNICATE WITH TAPI.....	11
TAPI NOTIFICATIONS	12
LINE_ADDRESSSTATE	12
LINE_CALLINFO	13
LINE_CALLSTATE	14
LINE_CLOSE	14
LINE_DEVSPECIFIC.....	14
LINE_DEVSPECIFICFEATURE.....	14
LINE_GATHERDIGITS.....	14
LINE_GENERATE.....	14
LINE_LINEDEVSTATE	15
LINE_MONITORDIGITS	16
LINE_MONITORMEDIA	16
LINE_MONITORTONE.....	16
LINE_REPLY	16
PHONE_BUTTON.....	16
PHONE_CLOSE.....	16
PHONE_DEVSPECIFIC.....	16
PHONE_REPLY	16
PHONE_STATE	17

LINE_CREATE	18
PHONE_CREATE	18
COMPANION APPLICATIONS	19
COMMANDS AND RESPONSES	19
DEFINING NEW COMMANDS AND RESPONSES.....	20
USING A 32-BIT COMPANION APPLICATION.....	20
USING AN INTERRUPT-DRIVEN APPROACH	21
<i>Interval Timer requirements</i>	21
USER-INTERFACE EVENTS.....	22
16-BIT SERVICE PROVIDERS	22
32-BIT SERVICE PROVIDERS	22
<i>Normal User Interface events</i>	22
<i>Spontaneous Dialogs</i>	23
DIALING INFORMATION	24
MEMORY ALLOCATIONS.....	26
SPECIAL CONSIDERATIONS FOR WINDOWS NT™.....	27
PORTABILITY TO OTHER PLATFORMS.....	27
UNICODE SUPPORT	27
THREADS.....	27
DEBUGGING	27
DEBUG FUNCTIONS	28
TECHNICAL SUPPORT	29
CSERVICEPROVIDER.....	30
SERVICE PROVIDER INITIALIZATION.....	30
<i>16-bit specific initialization</i>	30
<i>32-bit specific initialization</i>	30
PERSISTENT DATA STORAGE	30
REQUIRED AND TYPICAL OVERRIDES	31
CONSTRUCTION/DESTRUCTION - PUBLIC MEMBERS	31
INITIALIZATION - PUBLIC MEMBERS	31
INITIALIZATION - PROTECTED MEMBERS	31
OPERATIONS - PUBLIC MEMBERS.....	31
OVERRIDABLES - PUBLIC MEMBERS	32
OVERRIDABLES - PROTECTED MEMBERS	33
OVERRIDABLES - TAPI MEMBERS	33
CSERVICEPROVIDER::CANCELREQUEST	41
CSERVICEPROVIDER::CANHANDLEREQUEST	41
CSERVICEPROVIDER::CHECKCALLFEATURES	42
CSERVICEPROVIDER::CHECKDIALABLENUMBER	42
CSERVICEPROVIDER::CLOSEDEVICE	43
CSERVICEPROVIDER::CONVERTDIALABLETOCANONICAL	43
CSERVICEPROVIDER::CSERVICEPROVIDER	43
CSERVICEPROVIDER::DELETEPROFILE	44
CSERVICEPROVIDER::GETCONNINFOFROMLINEDEVICEID.....	45
CSERVICEPROVIDER::GETCONNINFOFROMPHONEDEVICEID	45
CSERVICEPROVIDER::GETCURRENTLOCATION	46
CSERVICEPROVIDER::GETDEVICE	46

CServiceProvider::GetDeviceByIndex	47
CServiceProvider::GetDeviceCount	47
CServiceProvider::GetDialableNumber	47
CServiceProvider::GetProviderInfo	48
CServiceProvider::GetProviderName	48
CServiceProvider::GetSupportedVersion	49
CServiceProvider::GetSystemVersion	49
CServiceProvider::MatchTones	49
CServiceProvider::OnNewRequest	50
CServiceProvider::OnTimer	50
CServiceProvider::OpenDevice	51
CServiceProvider::ProcessCallParameters	51
CServiceProvider::ProcessData	52
CServiceProvider::ReadProfileDWord	52
CServiceProvider::ReadProfileString	53
CServiceProvider::RenameProfile	53
CServiceProvider::SendData	54
CServiceProvider::SendThreadRequest	54
CServiceProvider::SetRuntimeObjects	55
CServiceProvider::SetTimerResolution	56
CServiceProvider::SetTimeout	56
CServiceProvider::StartNextCommand	56
CServiceProvider::UnknownDeviceNotify	57
CServiceProvider::WriteProfileDWord	58
CServiceProvider::WriteProfileString	58
CTSPIDevice	59
Device Initialization	59
Lines and Phones	59
Device Open/Close Events	59
Operations - Public Methods	60
Overridables - Public Members	60
Overridables - Protected Members	60
CTSPIDevice::CloseDevice	61
CTSPIDevice::CreateLine	61
CTSPIDevice::CreatePhone	61
CTSPIDevice::FindLineConnectionByDeviceID	62
CTSPIDevice::FindPhoneConnectionByDeviceID	62
CTSPIDevice::GenericDialogData	63
CTSPIDevice::GetLineCount	63
CTSPIDevice::GetLineConnectionInfo	64
CTSPIDevice::GetPermanentDeviceID	64
CTSPIDevice::GetPhoneCount	64
CTSPIDevice::GetPhoneConnectionInfo	65
CTSPIDevice::GetProviderID	65
CTSPIDevice::GetProviderHandle	66
CTSPIDevice::Init	66
CTSPIDevice::OnAsynchRequestComplete	67
CTSPIDevice::OnCancelRequest	67
CTSPIDevice::OnNewRequest	67
CTSPIDevice::OnTimer	68
CTSPIDevice::OpenDevice	68
CTSPIDevice::ReceivedData	68
CTSPIDevice::RemoveLine	69
CTSPIDevice::RemovePhone	69
CTSPIDevice::SendData	70

CTSPIREQUEST	71
ASYNCHRONOUS REQUESTS	71
REQUEST MANAGEMENT	71
PROCESSING A REQUEST	72
DESTRUCTOR - PUBLIC METHODS	73
OPERATIONS - PUBLIC METHODS	73
STATIC OPERATIONS - PUBLIC MEMBERS	74
OVERRIDABLES - PROTECTED MEMBERS	74
CTSPIREQUEST::COMPLETE	75
CTSPIREQUEST::GETASYNCHREQUESTID	75
CTSPIREQUEST::GETCALLINFO	75
CTSPIREQUEST::GETCOMMAND	76
CTSPIREQUEST::GETCONNECTIONINFO	76
CTSPIREQUEST::GETDATAPTR	76
CTSPIREQUEST::GETDATASIZE	77
CTSPIREQUEST::GETSTATE	77
CTSPIREQUEST::GETSTATEDATA	77
CTSPIREQUEST::HAVESENTRESPONSE	78
CTSPIREQUEST::INIT	78
CTSPIREQUEST::ISADDRESSOK	78
CTSPIREQUEST::SETCOMMAND	79
CTSPIREQUEST::SETSTATE	79
CTSPIREQUEST::SETSTATEDATA	79
CTSPIREQUEST::WAITFORCOMPLETION	80
CTSPICONNECTION	81
ASYNCHRONOUS REQUEST LIST	81
CONNECTION OPENS AND CLOSES	81
SYNCHRONOUS REQUESTS	82
OPERATIONS - PUBLIC MEMBERS	82
OVERRIDABLES - PUBLIC MEMBERS	83
OVERRIDABLES - PROTECTED MEMBERS	83
OPERATIONS - PROTECTED MEMBERS	83
CTSPICONNECTION::ADDASYNCHREQUEST	84
CTSPICONNECTION::ADDASYNCHREQUEST	84
CTSPICONNECTION::ADDDEVICECLASS	85
CTSPICONNECTION::CLOSEDEVICE	85
CTSPICONNECTION::COMPLETECURRENTREQUEST	86
CTSPICONNECTION::COMPLETEREQUEST	86
CTSPICONNECTION::FINDREQUEST	87
CTSPICONNECTION::GETCONNINFO	87
CTSPICONNECTION::GETCURRENTREQUEST	87
CTSPICONNECTION::GETDEVICECLASS	88
CTSPICONNECTION::GETDEVICEID	88
CTSPICONNECTION::GETDEVICEINFO	88
CTSPICONNECTION::GETNAME	89
CTSPICONNECTION::GETNEGOTIATEDVERSION	89
CTSPICONNECTION::GETREQUEST	89
CTSPICONNECTION::GETREQUESTCOUNT	90
CTSPICONNECTION::INIT	90
CTSPICONNECTION::ISLINEDEVICE	90
CTSPICONNECTION::ISPHONEDEVICE	90
CTSPICONNECTION::ONCANCELREQUEST	91
CTSPICONNECTION::ONNEWREQUEST	91
CTSPICONNECTION::ONREQUESTCOMPLETE	92

CTSPICONNECTION::ONTIMER	92
CTSPICONNECTION::OPENDEVICE	92
CTSPICONNECTION::RECEIVEDATA	93
CTSPICONNECTION::REMOVEDEVICECLASS	93
CTSPICONNECTION::REMOVEPENDINGREQUESTS	93
CTSPICONNECTION::REMOVEREQUEST	94
CTSPICONNECTION::SENDATA	94
CTSPICONNECTION::SENDSTRING	95
CTSPICONNECTION::SETCONNINFO	95
CTSPICONNECTION::SETDEVICEID	95
CTSPICONNECTION::SETNAME	96
CTSPICONNECTION::WAITFORALLREQUESTS	96
CTSPICONNECTION::WAITFORREQUEST	96
CTSPILINECONNECTION	97
LINE DEVICES	97
LINE CONNECTION INITIALIZATION	97
LINE OPEN/CLOSE EVENTS	97
CAPABILITIES	97
DEVICE SPECIFIC EXTENSIONS	98
STATUS INFORMATION	98
CHANNELS AND ADDRESSES	99
TERMINALS	99
REQUEST COMPLETIONS	100
OPERATIONS - PUBLIC MEMBERS	100
OVERRIDABLES - PUBLIC MEMBERS	101
OPERATIONS - PROTECTED MEMBERS	101
OVERRIDABLES - PROTECTED MEMBERS	101
OVERRIDABLES - TAPI MEMBERS	102
CTSPILINECONNECTION::ADDTERMINAL	104
CTSPILINECONNECTION::CANHANDLEREQUEST	104
CTSPILINECONNECTION::CANSUPPORTCALL	105
CTSPILINECONNECTION::CLOSE()	105
CTSPILINECONNECTION::CONDITIONALMEDIADETECTION	105
CTSPILINECONNECTION::CREATEADDRESS	106
CTSPILINECONNECTION::CREATEUIDIALOG	107
CTSPILINECONNECTION::CTSPILINECONNECTION	107
CTSPILINECONNECTION::~~CTSPILINECONNECTION	107
CTSPILINECONNECTION::DEVSPECIFICFEATURE	108
CTSPILINECONNECTION::FINDAVAILABLEADDRESS	108
CTSPILINECONNECTION::FINDCALLBYSTATE	109
CTSPILINECONNECTION::FINDCALLCOMPLETIONREQUEST	109
CTSPILINECONNECTION::FORCECLOSE	110
CTSPILINECONNECTION::FORWARD	110
CTSPILINECONNECTION::FREEDIALOGINSTANCE	111
CTSPILINECONNECTION::GATHERCAPABILITIES	111
CTSPILINECONNECTION::GATHERSTATUS	112
CTSPILINECONNECTION::GENERICDIALOGDATA	112
CTSPILINECONNECTION::GETADDRESS	113
CTSPILINECONNECTION::GETADDRESSCOUNT	113
CTSPILINECONNECTION::GETADDRESSID	113
CTSPILINECONNECTION::GETDEFAULTMEDIADETECTION	114
CTSPILINECONNECTION::GETDEVCONFIG	114
CTSPILINECONNECTION::GETICON	115
CTSPILINECONNECTION::GETID	115
CTSPILINECONNECTION::GETLINEDEVCAPS	115

CTSPILINECONNECTION::GETLINEDEVSTATUS	116
CTSPILINECONNECTION::GETLINEHANDLE	116
CTSPILINECONNECTION::GETTERMINALCOUNT	116
CTSPILINECONNECTION::GETTERMINALINFORMATION	117
CTSPILINECONNECTION::GETUIDIALOG	117
CTSPILINECONNECTION::INIT	117
CTSPILINECONNECTION::ISCONFERENCEAVAILABLE	118
CTSPILINECONNECTION::ISTRANSFERCONSULTAVAILABLE	118
CTSPILINECONNECTION::MAKECALL	119
CTSPILINECONNECTION::ONADDRESSFEATURESCHANGED	119
CTSPILINECONNECTION::ONCALLDELETED	120
CTSPILINECONNECTION::ONCALLFEATURESCHANGED	120
CTSPILINECONNECTION::ONCALLSTATECHANGE	121
CTSPILINECONNECTION::ONLINECAPABILITIESCHANGED	121
CTSPILINECONNECTION::ONLINEFEATURESCHANGED	121
CTSPILINECONNECTION::ONLINESTATUSCHANGE	122
CTSPILINECONNECTION::ONMEDIACONFIGCHANGED	122
CTSPILINECONNECTION::ONREQUESTCOMPLETE	122
CTSPILINECONNECTION::ONRINGDETECTED	123
CTSPILINECONNECTION::OPEN	123
CTSPILINECONNECTION::REMOVECALLCOMPLETIONREQUEST	124
CTSPILINECONNECTION::REMOVETERMINAL	124
CTSPILINECONNECTION::SENDDIALOGINSTANCEDATA	125
CTSPILINECONNECTION::SEND_TAPI_EVENT	125
CTSPILINECONNECTION::SETBATTERYLEVEL	126
CTSPILINECONNECTION::SETDEFAULTMEDIADETECTION	126
CTSPILINECONNECTION::SETDEVCONFIG	126
CTSPILINECONNECTION::SETDEVICESTATUSFLAGS	127
CTSPILINECONNECTION::SETLINEDEVSTATUS	127
CTSPILINECONNECTION::SETMEDIACONTROL	128
CTSPILINECONNECTION::SETRINGMODE	128
CTSPILINECONNECTION::SETROAMMODE	129
CTSPILINECONNECTION::SETSIGNALLEVEL	129
CTSPILINECONNECTION::SETSTATUSMESSAGES	129
CTSPILINECONNECTION::SETTERMINAL	130
CTSPILINECONNECTION::SETTERMINALMODES	130
CTSPILINECONNECTION::UNCOMPLETECALL	130
CTSPILINECONNECTION::VALIDATEMEDIACONTROLLIST	131
CTSPIPHONECONNECTION	132
PHONE INITIALIZATION	133
STATUS INFORMATION	133
DEVICE SPECIFIC EXTENSIONS	133
OPERATIONS - PUBLIC MEMBERS	133
OPERATIONS - PROTECTED MEMBERS	135
OVERRIDABLES - PROTECTED MEMBERS	135
OVERRIDABLES - TAPI FUNCTIONS	136
CTSPIPHONECONNECTION::ADDBUTTON	138
CTSPIPHONECONNECTION::ADDDISPLAYCHAR	138
CTSPIPHONECONNECTION::ADDDISPLAYSTRING	138
CTSPIPHONECONNECTION::ADDDOWNLOADBUFFER	139
CTSPIPHONECONNECTION::ADDDHOOKSWITCHDEVICE	139
CTSPIPHONECONNECTION::ADDUPLOADBUFFER	140
CTSPIPHONECONNECTION::CANHANDLEREQUEST	140
CTSPIPHONECONNECTION::CLEARDISPLAYLINE	141
CTSPIPHONECONNECTION::CLOSE()	141

CTSPHONECONNECTION::CTSPHONECONNECTION()	141
CTSPHONECONNECTION::~~CTSPHONECONNECTION()	141
CTSPHONECONNECTION::DEVSPECIFICFEATURE	142
CTSPHONECONNECTION::FORCECLOSE	142
CTSPHONECONNECTION::GATHERCAPABILITIES	143
CTSPHONECONNECTION::GATHERSTATUS	144
CTSPHONECONNECTION::GENERICDIALOGDATA	144
CTSPHONECONNECTION::GETBUTTONCOUNT	145
CTSPHONECONNECTION::GETBUTTONINFO	145
CTSPHONECONNECTION::GETBUTTONINFO	145
CTSPHONECONNECTION::GETCURSORPOS	146
CTSPHONECONNECTION::GETDATA	146
CTSPHONECONNECTION::GETDISPLAY	147
CTSPHONECONNECTION::GETDISPLAYBUFFER	147
CTSPHONECONNECTION::GETGAIN	148
CTSPHONECONNECTION::GETHOOKSWITCH	148
CTSPHONECONNECTION::GETICON	149
CTSPHONECONNECTION::GETID	149
CTSPHONECONNECTION::GETLAMP	150
CTSPHONECONNECTION::GETLAMPMODE	150
CTSPHONECONNECTION::GETPHONECAPS	151
CTSPHONECONNECTION::GETPHONEHANDLE	151
CTSPHONECONNECTION::GETPHONESTATUS	152
CTSPHONECONNECTION::GETRING	152
CTSPHONECONNECTION::GETVOLUME	153
CTSPHONECONNECTION::INIT	153
CTSPHONECONNECTION::ONBUTTONSTATECHANGE	154
CTSPHONECONNECTION::ONPHONECAPABILITIESCHANGED	154
CTSPHONECONNECTION::ONPHONEFEATURESCHANGED	154
CTSPHONECONNECTION::ONPHONESTATUSCHANGE	155
CTSPHONECONNECTION::ONREQUESTCOMPLETE	155
CTSPHONECONNECTION::OPEN	156
CTSPHONECONNECTION::RESETDISPLAY	156
CTSPHONECONNECTION::SEND_TAPI_EVENT	156
CTSPHONECONNECTION::SETBUTTONINFO	157
CTSPHONECONNECTION::SETBUTTONINFO	157
CTSPHONECONNECTION::SETBUTTONSTATE	158
CTSPHONECONNECTION::SETDATA	158
CTSPHONECONNECTION::SETDISPLAY	158
CTSPHONECONNECTION::SETDISPLAY	159
CTSPHONECONNECTION::SETDISPLAYCHAR	159
CTSPHONECONNECTION::SETDISPLAYCURSORPOS	160
CTSPHONECONNECTION::SETGAIN	160
CTSPHONECONNECTION::SETGAIN	160
CTSPHONECONNECTION::SETHOOKSWITCH	161
CTSPHONECONNECTION::SETHOOKSWITCH	161
CTSPHONECONNECTION::SETLAMP	162
CTSPHONECONNECTION::SETLAMPSTATE	162
CTSPHONECONNECTION::SETPHONEFEATURES	162
CTSPHONECONNECTION::SETRING	163
CTSPHONECONNECTION::SETRINGMODE	163
CTSPHONECONNECTION::SETRINGVOLUME	163
CTSPHONECONNECTION::SETSTATUSFLAGS	164
CTSPHONECONNECTION::SETSTATUSMESSAGES	165
CTSPHONECONNECTION::SETUPDISPLAY	165
CTSPHONECONNECTION::SETVOLUME	166

CTSPiPHONECONNECTION::SETVOLUME	166
CTSPiADDRESSINFO.....	167
ADDRESSES	167
ADDRESS SHARING	167
ADDRESS CONFIGURATION	167
ADDRESS INITIALIZATION.....	168
ADDRESS CAPABILITIES	168
DEVICE SPECIFIC EXTENSIONS	169
ADDRESS STATUS.....	169
REQUEST COMPLETIONS	169
OPERATIONS - PUBLIC MEMBERS	169
OVERRIDABLES - PUBLIC MEMBERS	171
OPERATIONS - PROTECTED MEMBERS.....	171
OVERRIDABLES - PROTECTED MEMBERS	171
OVERRIDABLES - TAPI MEMBERS	171
CTSPiADDRESSINFO::ADDASYNCHREQUEST	173
CTSPiADDRESSINFO::ADDCALLTREATMENT	173
CTSPiADDRESSINFO::ADDCOMPLETIONMESSAGE	173
CTSPiADDRESSINFO::ADDDEVICECLASS	174
CTSPiADDRESSINFO::ADDFORWARDENTRY	174
CTSPiADDRESSINFO::CANANSWERCALLS	175
CTSPiADDRESSINFO::CANFORWARD.....	175
CTSPiADDRESSINFO::CANHANDLEREQUEST	176
CTSPiADDRESSINFO::CANMAKECALLS	176
CTSPiADDRESSINFO::CANSUPPORTCALL.....	176
CTSPiADDRESSINFO::CANSUPPORTMEDIAMODES.....	177
CTSPiADDRESSINFO::COMPLETETRANSFER	177
CTSPiADDRESSINFO::CREATECALLAPPEARANCE	178
CTSPiADDRESSINFO::CREATECONFERENCECALL	178
CTSPiADDRESSINFO::CTSPiADDRESSINFO.....	179
CTSPiADDRESSINFO::~CTSPiADDRESSINFO	179
CTSPiADDRESSINFO::DELETEFORWARDINGINFO	179
CTSPiADDRESSINFO::FINDATTACHEDCALL.....	179
CTSPiADDRESSINFO::FINDCALLBYCALLID.....	180
CTSPiADDRESSINFO::FINDCALLBYHANDLE	180
CTSPiADDRESSINFO::FINDCALLBYSTATE	180
CTSPiADDRESSINFO::FORWARD.....	181
CTSPiADDRESSINFO::GATHERCAPABILITIES.....	181
CTSPiADDRESSINFO::GATHERSTATUSINFORMATION.....	182
CTSPiADDRESSINFO::GETADDRESSCAPS	183
CTSPiADDRESSINFO::GETADDRESSID	183
CTSPiADDRESSINFO::GETADDRESSSTATUS	183
CTSPiADDRESSINFO::GETAVAILABLEMEDIAMODES.....	184
CTSPiADDRESSINFO::GETBEARERMODE	184
CTSPiADDRESSINFO::GETCALLCOUNT	184
CTSPiADDRESSINFO::GETCALLINFO.....	185
CTSPiADDRESSINFO::GETCALLTREATMENTNAME	185
CTSPiADDRESSINFO::GETCOMPLETIONMESSAGE	185
CTSPiADDRESSINFO::GETCOMPLETIONMESSAGECOUNT	186
CTSPiADDRESSINFO::GETCURRENTRATE	186
CTSPiADDRESSINFO::GETDEVICECLASS.....	186
CTSPiADDRESSINFO::GETDIALABLEADDRESS	186
CTSPiADDRESSINFO::GETID	187
CTSPiADDRESSINFO::GETLINEOWNER	187
CTSPiADDRESSINFO::GETNAME	188

CTSPIADDRESSINFO::GETTERMINALINFORMATION	188
CTSPIADDRESSINFO::INIT.....	188
CTSPIADDRESSINFO::ONADDRESSCAPABILITIESCHANGED	189
CTSPIADDRESSINFO::ONADDRESSFEATURESCHANGED	189
CTSPIADDRESSINFO::ONADDRESSSTATECHANGE.....	190
CTSPIADDRESSINFO::ONCALLFEATURESCHANGED.....	190
CTSPIADDRESSINFO::ONCALLSTATECHANGED.....	191
CTSPIADDRESSINFO::ONCREATECALL	191
CTSPIADDRESSINFO::ONREQUESTCOMPLETE	191
CTSPIADDRESSINFO::ONTERMINALCOUNTCHANGED.....	192
CTSPIADDRESSINFO::ONTIMER.....	192
CTSPIADDRESSINFO::PICKUP	193
CTSPIADDRESSINFO::REMOVECALLAPPEARANCE	193
CTSPIADDRESSINFO::REMOVECALLTREATMENT.....	193
CTSPIADDRESSINFO::REMOVEDEVICECLASS.....	194
CTSPIADDRESSINFO::SETADDRESSFEATURES.....	194
CTSPIADDRESSINFO::SETCURRENTRATE.....	195
CTSPIADDRESSINFO::SETDIALABLEADDRESS.....	195
CTSPIADDRESSINFO::SETMEDIACONTROL	195
CTSPIADDRESSINFO::SETNAME	196
CTSPIADDRESSINFO::SETNUMRINGSNOANSWER.....	196
CTSPIADDRESSINFO::SETSTATUSMESSAGES	196
CTSPIADDRESSINFO::SETTERMINAL	197
CTSPIADDRESSINFO::SETTERMINALMODES	197
CTSPIADDRESSINFO::SETUPCONFERENCE.....	198
CTSPIADDRESSINFO::SETUPTRANSFER	198
CTSPIADDRESSINFO::UPARK.....	199
CTSPICALLAPPEARANCE.....	200
CALL APPEARANCES	200
CALL HANDLES	200
CALL STATES	200
BEARER AND MEDIA MODES.....	201
NEW CALL MEDIA MODE IDENTIFICATION	201
CALL APPEARANCE INITIALIZATION	202
CALL INFORMATION.....	203
CALL STATUS.....	203
DEVICE SPECIFIC EXTENSIONS	203
REQUEST COMPLETIONS	203
OPERATIONS - PUBLIC MEMBERS.....	204
OPERATIONS - STATIC MEMBERS.....	205
OPERATIONS - PROTECTED MEMBERS.....	205
OVERRIDABLES - PROTECTED MEMBERS	206
OVERRIDABLES - TAPI METHODS.....	206
CTSPICALLAPPEARANCE::ACCEPT	209
CTSPICALLAPPEARANCE::ADDASYNCHREQUEST	209
CTSPICALLAPPEARANCE::ADDDEVICECLASS.....	210
CTSPICALLAPPEARANCE::ANSWER.....	210
CTSPICALLAPPEARANCE::ATTACHCALL	211
CTSPICALLAPPEARANCE::BLINDTRANSFER.....	211
CTSPICALLAPPEARANCE::CANHANDLEREQUEST	212
CTSPICALLAPPEARANCE::CLOSE	212
CTSPICALLAPPEARANCE::COMPLETECALL.....	213
CTSPICALLAPPEARANCE::COMPLETEDIGITGATHER.....	213
CTSPICALLAPPEARANCE::CREATECONSULTATIONCALL	214
CTSPICALLAPPEARANCE::CTSPICALLAPPEARANCE.....	214

CTSPICALLAPPEARANCE::~CTSPICALLAPPEARANCE	214
CTSPICALLAPPEARANCE::DELETETONEMONITORLIST	215
CTSPICALLAPPEARANCE::DETACHCALL	215
CTSPICALLAPPEARANCE::DIAL	215
CTSPICALLAPPEARANCE::DROP.....	216
CTSPICALLAPPEARANCE::GATHERCALLINFORMATION.....	216
CTSPICALLAPPEARANCE::GATHERDIGITS	216
CTSPICALLAPPEARANCE::GATHERSTATUSINFORMATION	217
CTSPICALLAPPEARANCE::GENERATEDIGITS	217
CTSPICALLAPPEARANCE::GENERATE TONE	218
CTSPICALLAPPEARANCE::GETADDRESSINFO	218
CTSPICALLAPPEARANCE::GETATTACHEDCALL	219
CTSPICALLAPPEARANCE::GETCALLHANDLE	219
CTSPICALLAPPEARANCE::GETCALLINFO	219
CTSPICALLAPPEARANCE::GETCALLSTATE.....	219
CTSPICALLAPPEARANCE::GETCALLSTATUS.....	220
CTSPICALLAPPEARANCE::GETCALLTYPE.....	220
CTSPICALLAPPEARANCE::GETCONSULTATIONCALL	221
CTSPICALLAPPEARANCE::GETDEVICECLASS	221
CTSPICALLAPPEARANCE::GETID.....	222
CTSPICALLAPPEARANCE::GETLINECONNECTIONINFO	222
CTSPICALLAPPEARANCE::HOLD	222
CTSPICALLAPPEARANCE::INIT	223
CTSPICALLAPPEARANCE::ISACTIVECALLSTATE	223
CTSPICALLAPPEARANCE::ISCONNECTEDCALLSTATE.....	224
CTSPICALLAPPEARANCE::MAKECALL	224
CTSPICALLAPPEARANCE::MONITORDIGITS	225
CTSPICALLAPPEARANCE::MONITORMEDIA	225
CTSPICALLAPPEARANCE::MONITOR TONES	225
CTSPICALLAPPEARANCE::ONCALLINFOCHANGE	226
CTSPICALLAPPEARANCE:: ONCONSULTANTCALLIDLE.....	226
CTSPICALLAPPEARANCE::ONDETECTEDNEWMEDIAMODES.....	227
CTSPICALLAPPEARANCE::ONDIGIT	227
CTSPICALLAPPEARANCE::ONMEDIACONTROL.....	227
CTSPICALLAPPEARANCE::ONRECEIVEDUSERUSERINFO	228
CTSPICALLAPPEARANCE::ONRELATEDCALLSTATECHANGE.....	228
CTSPICALLAPPEARANCE::ONREQUESTCOMPLETE	228
CTSPICALLAPPEARANCE::ONTERMINALCOUNTCHANGED	229
CTSPICALLAPPEARANCE::ONTIMER	229
CTSPICALLAPPEARANCE::ONTONE.....	230
CTSPICALLAPPEARANCE::ONTONEMONITORDETECT	230
CTSPICALLAPPEARANCE::PARK.....	230
CTSPICALLAPPEARANCE::PICKUP	231
CTSPICALLAPPEARANCE::REDIRECT.....	231
CTSPICALLAPPEARANCE::RELEASEUSERUSERINFO	231
CTSPICALLAPPEARANCE::REMOVEDeviceCLASS	232
CTSPICALLAPPEARANCE::SECURE	232
CTSPICALLAPPEARANCE::SENDUSERUSERINFO	233
CTSPICALLAPPEARANCE::SETAPPSPECIFICDATA.....	233
CTSPICALLAPPEARANCE::SETBEARERMODE	233
CTSPICALLAPPEARANCE::SETCALLDATA	234
CTSPICALLAPPEARANCE::SETCALLDATA	235
CTSPICALLAPPEARANCE::SETCALLEDIDINFORMATION.....	235
CTSPICALLAPPEARANCE::SETCALLERIDINFORMATION	235
CTSPICALLAPPEARANCE::SETCALLFEATURES	236
CTSPICALLAPPEARANCE::SETCALLID.....	236

CTSPICALLAPPEARANCE::SETCALLORIGIN	236
CTSPICALLAPPEARANCE::SETCALLPARAMETERFLAGS.....	237
CTSPICALLAPPEARANCE::SETCALLPARAMS	237
CTSPICALLAPPEARANCE::SETCALLREASON	237
CTSPICALLAPPEARANCE::SETCALLSTATE	238
CTSPICALLAPPEARANCE::SETCALLTREATMENT	238
CTSPICALLAPPEARANCE::SETCALLTREATMENT	239
CTSPICALLAPPEARANCE::SETCALLTYPE	239
CTSPICALLAPPEARANCE::SETCONNECTEDIDINFORMATION	239
CTSPICALLAPPEARANCE::SETCONSULTATIONCALL.....	240
CTSPICALLAPPEARANCE::SETDATA RATE	240
CTSPICALLAPPEARANCE::SETDESTINATIONCOUNTRY	240
CTSPICALLAPPEARANCE::SETDIALPARAMETERS	241
CTSPICALLAPPEARANCE::SETDIGITMONITOR.....	241
CTSPICALLAPPEARANCE::SETMEDIACONTROL.....	241
CTSPICALLAPPEARANCE::SETMEDIAMODE	242
CTSPICALLAPPEARANCE::SETMEDIAMONITOR.....	242
CTSPICALLAPPEARANCE::SETQUALITYOFSERVICE.....	242
CTSPICALLAPPEARANCE::SETQUALITYOFSERVICE.....	243
CTSPICALLAPPEARANCE::SETREDIRECTINGIDINFORMATION	243
CTSPICALLAPPEARANCE::SETREDIRECTIONIDINFORMATION	244
CTSPICALLAPPEARANCE::SETRELATEDCALLID.....	244
CTSPICALLAPPEARANCE::SETTERMINAL.....	244
CTSPIADDRESSINFO::SETTERMINALMODES	245
CTSPIADDRESSINFO::SETTRUNKID	245
CTSPICALLAPPEARANCE::SWAPHOLD	246
CTSPICALLAPPEARANCE::UNHOLD.....	246
CTSPICALLAPPEARANCE::UNPARK.....	246
CTSPICONFERENCECALL	248
CONFERENCES.....	248
OTHER NOTES ABOUT CONFERENCING WITH TSP++	249
OPERATIONS - PUBLIC MEMBERS.....	249
OPERATIONS - PROTECTED MEMBERS.....	249
OVERRIDABLES - PROTECTED MEMBERS	249
OVERRIDABLES - TAPI MEMBERS	249
CTSPICONFERENCECALL::ADDTOCONFERENCE	251
CTSPICONFERENCECALL::ADDTOCONFERENCE	251
CTSPICONFERENCECALL::CANREMOVEFROMCONFERENCE.....	252
CTSPICONFERENCECALL:: CTSPICONFERENCECALL	252
CTSPICONFERENCECALL:: ~CTSPICONFERENCECALL.....	252
CTSPICONFERENCECALL:: GETCONFERENCECALL	252
CTSPICONFERENCECALL:: GETCONFERENCECOUNT	253
CTSPICONFERENCECALL:: ISCALLINCONFERENCE	253
CTSPICONFERENCECALL::ONREQUESTCOMPLETE	254
CTSPICONFERENCECALL::PREPAREADDTOCONFERENCE.....	254
CTSPICONFERENCECALL:: REMOVECONFERENCECALL.....	255
CTSPICONFERENCECALL::REMOVEFROMCONFERENCE.....	255
ASYNCHRONOUS REQUEST DATA STRUCTURES	256
DEVICEINFO	256
DIALINFO	256
<i>Member</i>	256
TSPICALLDATA	256
TSPICALLPARAMS	257
TSPICOMPLETECALL	257

TSPICONFERENCE.....	258
TSPIFORWARDINFO.....	258
TSPIGENERATE.....	259
TSPHOOKSWITCHPARAM.....	259
TSPILINEFORWARD.....	260
TSPILINEPARK.....	260
TSPILINEPICKUP.....	261
TSPILINESETTERMINAL.....	261
TSPIMAKECALL.....	261
TSPIPHONEDATA.....	262
TSPIPHONESETDISPLAY.....	262
TSPIQOS.....	263
TSPIRINGPATTERN.....	263
TSPISSETBUTTONINFO.....	263
TSPITRANSFER.....	264
CREATING A NEW SERVICE PROVIDER	265
TYPICAL OVERRIDDEN METHODS.....	265
GENERAL WORK FLOW	265
ASYNCHRONOUS REQUEST HANDLING	266
AN EXAMPLE OF AN ASYNCHRONOUS REQUEST.....	266
FUNCTIONS TO IMPLEMENT	268
ATSP V2 SAMPLE SERVICE PROVIDER.....	271
TSPI EXPORTS	271
FILES.....	271
BASIC CLASS STRUCTURE.....	272
<i>CATSPProvider</i>	272
<i>CATSPLine</i>	272
BASIC PROCESSING FLOW	273
INSTALLATION	273
CONFIGURATION	274
INI FILE KEYS.....	274
IMPLEMENTATION NOTES FOR ATSPV2	275
DIGITAL SWITCH EMULATOR.....	276
RUNNING THE EMULATOR.....	276
CONFIGURING THE EMULATOR.....	278
16-BIT DIGITAL SWITCH SAMPLE SERVICE PROVIDER (DSSP)	281
INSTALLATION	281
CONFIGURING DSSP	281
USING THE DSSP SAMPLE PROVIDER	281
FILES.....	281
BASIC CLASS STRUCTURE.....	282
<i>CDSProvider</i>	282
<i>CDSDevice</i>	282
<i>CDSLLine</i>	282
<i>CDSPhone</i>	283
BASIC PROCESSING FLOW	283
IMPLEMENTATION NOTES.....	283
32-BIT DIGITAL SWITCH SAMPLE SERVICE PROVIDER (DSSP32).....	284
INSTALLATION	284
CONFIGURING THE DSSP32 SAMPLE.....	285
USING THE DSSP32 SAMPLE PROVIDER.....	285

FILES.....	285
BASIC CLASS STRUCTURE.....	286
<i>CDSProvider</i>	286
<i>CDSDevice</i>	286
<i>CDSLine</i>	286
<i>CDSPhone</i>	286
BASIC PROCESSING FLOW	286
IMPLEMENTATION NOTES.....	287
ATSP32 SAMPLE SERVICE PROVIDER	288
INSTALLATION	288
CONFIGURING THE ATSP32 SAMPLE.....	288
REGISTRY INFORMATION.....	289
FILES.....	290
BASIC CLASS STRUCTURE.....	290
IMPLEMENTATION NOTES.....	290
MAIN PROCESSING LOGIC	291
TESTING AND DEBUGGING SERVICE PROVIDERS	293
SOME HINTS FOR DEBUGGING 16-BIT TSPs.....	293
SOME HINTS FOR DEBUGGING 32-BIT TSPs.....	294
DEBUG TAPI COMPONENTS	294
OTHER DEBUGGING NOTES	294
GENERAL NOTES ABOUT THE LIBRARY:	294
DEBUG STRINGS USED IN THE TSP++ LIBRARY	295
REFERENCES	301

Introduction

TAPI is an acronym that stands for **Telephony Applications Programming Interface**.

Telephony is the technology that integrates computers with a telephone network. With telephony, people can use their computers to take advantage of a wide range of sophisticated communication features and services over a telephone line.

An **Application Programming Interface (API)** is a set of library programming functions that assist programmers accessing internal and direct operations in systems such as Microsoft Windows.

TAPI is a solution provided by Microsoft for implementing telephony enabled applications in the Microsoft Windows environment. Microsoft TAPI is part of the **Windows Open Services Architecture (WOSA)**, which provides a single set of open-ended interfaces to which applications can be written to control a variety of back-end devices or services.

WOSA services such as Windows Telephony (TAPI) consist of two interfaces:

Application Programming Interface (API) - provides application-level support for enabling an application to communicate and control a telephony device.

Service Provider Interface (SPI) - controls the actual telephony network device. This is referred to as the *device-driver* in some models. TSP++ is designed to help implement and cut development time for this SPI layer.

History of TAPI

TAPI was originally released as an add-on product to Windows 3.1. The first released version of TAPI was 1.3. It was immediately followed with a second release (1.4) that was shipped with Windows 95. TAPI 1.4 still had a 16-bit architecture for the service providers, but supported 32-bit TAPI applications through a *thunk* layer that translated the parameters from 32-bit to 16-bit. Both of these versions were primarily developed for first-party telephony, i.e. the telephone device was connected directly to the PC where the service provider was installed, and all applications were run on that workstation.

With the release of Windows NT 4.0, Microsoft introduced the first fully 32-bit version of TAPI (2.0). It included changes in the user-interface calling-conventions to manage process-isolation, full multi-threading support, UNICODE, and a new service process to manage the lifetime of the service providers.

Recently, Microsoft has updated TAPI with version 2.1 to support third-party call control, where the application is running on a separate machine from the actual service provider and telephony hardware. They have also issued an update to TAPI for Windows 95 that brings full 32-bit support to that platform, allowing binary compatibility between Windows 95 and Windows NT. Microsoft has also announced the next major version of TAPI, version 3.0, which will presumably be released with Windows 98 and Windows NT 5.0.

TAPI System Components

The Windows TAPI subsystem consists of several components which work together to provide telephony support to the application. These components can be divided up into two main categories: the system components and the TSPI components.

System Components

The TAPI system components are supplied by Windows itself, and are used to control and direct telephony traffic on the workstation. The components work together to connect the application layer to the service provider layer and provide a device-independent interface for carrying out telephony tasks. This layer comprises what is officially called TAPI. The files which work together to provide this layer are:

TAPI.DLL - This is the library that 16-bit applications can link to in order to call telephony functions. It communicates with **TAPIEXE.EXE** to provide telephony support for the application. Under TAPI 2.x, this calls **TAPI32.DLL** to perform its tasks.

TAPI32.DLL - This is the library that 32-bit applications can link to in order to call telephony functions. Under TAPI 1.x, this library calls the 16-bit functions in **TAPI.DLL**, but under TAPI 2.x, this communicates with a server process called **TAPISRV.EXE** to provide telephony support for the application.

TAPIADDR.DLL - This is a supplementary library, which provides some support for address translation for the TAPI application. Specifically, this library can convert between *canonical* and *dialable* addresses (see the section on **DIALINFO** for more information on this topic).

TAPIEXE.EXE - This provides the 16-bit application context for the TAPI dynamic link library to carry out duties such as allocating memory or accessing data structures that have been swapped out to disk. This program is typically started when the first TAPI call is made by an application. In some cases, but not all, this process is used to call the service provider when a TAPI request is being processed. It is only present on TAPI 1.x systems.

TAPISRV.EXE - This is the 32-bit service manager under Windows NT and Windows 95 that provides the support for TAPI 2.x telephony. Since this is a separate process in the operating system, remote procedure calls (RPCs) are used to pass data back and forth between applications making telephony requests and this manager. This program will then verify the request and pass it onto the appropriate service provider to process. It is only present on TAPI 2.x systems.

TSPI Components

The TSPI components are the target for the TSP++ class library. The TSP component is typically provided by a third-party company or hardware manufacturer and is used to interact with the TAPI system components to complete the job of telephony control. This is the lowest layer in the telephony diagram.

Each TSP (TAPI Service Provider) is a dynamic link library (DLL) that carries out low-level device-specific actions needed to complete the telephony tasks. The TSP is always called by one of the TAPI system components, never by an application directly, and is generally connected to some piece of telephone network hardware such as a fax board, ISDN card, telephone, or modem.

To receive requests from TAPI, the service provider must implement the Telephony service provider interface (TSPI). This interface is a set of pre-defined functions that the service provider DLL provides and exports for usage.

A service provider can provide different levels of the service provider interface: *basic*, *supplementary*, or *extended*. For example, a simple service provider might provide basic telephony service, such as support for outgoing calls through a Hayes-compatible modem. A different service provider might provide a full range of incoming and outgoing call support through more advanced hardware such as an ISDN controller card.

System Architecture of TAPI

The following diagram shows the relationship between the Windows Telephony API and the Windows Telephony SPI. The gateway between these two layers is the **TAPI.DLL**.

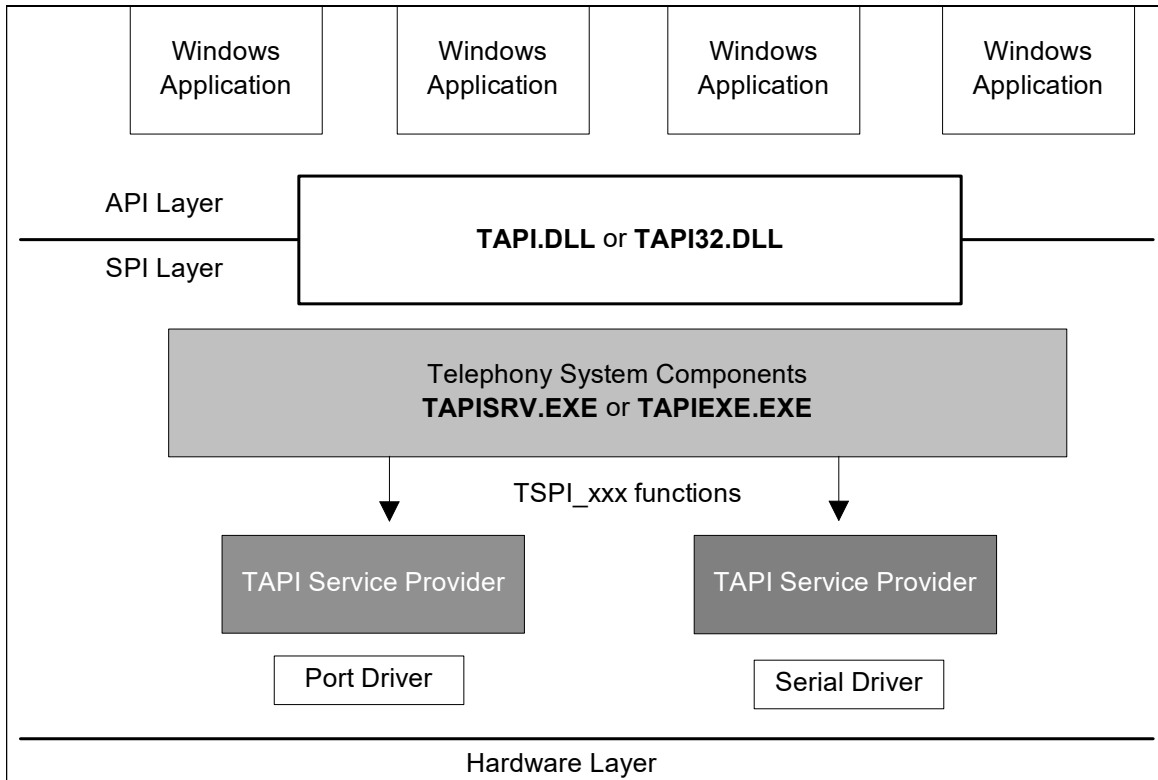


Figure 1: TAPI Architecture

As can be seen in the illustration, the application talks to the API layer of TAPI, which consists of either **TAPI.DLL** or **TAPI32.DLL**. This in turn will call the TAPI system component layer, either by a direct call (as in TAPI 1.x) or through a system RPC (TAPI 2.x). The system component will then determine which provider should be invoked and call the appropriate entry-point.

The **TAPIADDR.DLL** library is not shown in this diagram, but it sits directly in the API layer along with a portion of **TAPI.DLL** and **TAPI32.DLL**.

Basic Process Flow

Any number of service providers can be installed onto a single computer. Each service provider is typically associated with a piece of hardware or a group of similar or identical devices connected to the PC. Since the telephony hardware capabilities can vary so drastically between devices, a service provider is custom-built for any device or set of devices that it controls. The association between the hardware and service provider is maintained by the service provider itself, by either asking the user for a location (such as a serial communications port), or some form of auto-detect plug and play. If Windows doesn't support the device directly (such as a custom piece of hardware), a device driver might need to be installed which will allow the service provider to communicate with the device. Once the hardware is connected, the service provider will notify TAPI as to the number of devices it represents, and TAPI will expose those devices to any interested applications.

When an application calls a TAPI function, it uses a unique sequential index to identify the device it wants to control or monitor. The TAPI system then correlates that device identifier to a specific service provider installed on the system and routes the request to that provider for actual processing. It is important to note that the application itself is generally not aware of multiple service providers. It deals with the exposed devices not with the providers themselves.

Applications use the TAPI functions through **TAPI.DLL** (or **TAPI32.DLL**) to determine which services are available on the given computer. The TAPI dynamic link library determines what service providers are available and provides information about their capabilities to the applications. In this way, any number of applications can request services from the same TSP by using the functions within the TAPI dynamic link library.

TAPI 1.x (16-bit)

In the 16-bit TAPI 1.x world, the service provider is always a 16-bit dynamic link library with defined exports. Each export is given a specific ordinal number defined by Microsoft in *tspi.h*. This allows the TAPI dynamic link library to load the service provider dynamically through a **LoadLibrary** call, and then use **GetProcAddress** to get a pointer to each function it needs to call. This also allows TAPI to dynamically determine what features the service provider can support based on which functions are actually exported.

When a request is made from an application, it calls either **TAPI.DLL** or **TAPI32.DLL** (which calls down to **TAPI.DLL** in most cases), which will verify the parameters and pass the request directly to the TSP. This implies that the current process thread might not always be consistent across calls to the TSP. This means that process objects such as file handles and local memory blocks will not be valid if allocated on a system call. This is the main reason the 16-bit architecture uses a companion application to operate effectively (see the section on *Companion Applications*).

TAPI 2.x (32-bit)

In TAPI 2.x, a new process executable was created to manage the TAPI system. This manager runs as a service executable. This means that under Windows NT, any explicit calls to normal Windows UI APIs generally result in a fault or failure. The service providers for this environment are 32-bit DLLs with named exports. The exports are not located by ordinal number. Instead, the **TAPISRV.EXE** module that loads the provider uses the direct names of the functions to invoke and determine the capabilities of the provider.

The management of the functions in the 32-bit library is identical to the 16-bit version of the library. The provider DLL is loaded only by **TAPISRV.EXE** unless a user-interface event is required (see *User Interface Events*), and any calls made to the TSP are guaranteed to be in the process space of the TAPI service manager.

When a request is made from an application, it is sent to **TAPISRV.EXE** through an RPC facility by **TAPI32.DLL**. The request is verified and then is passed through to the appropriate entry-point

in some TSP to manage the request. **TAPISRV.EXE** will then reply to the RPC, releasing the original calling thread in the application.

Purpose of the TSP++ Library

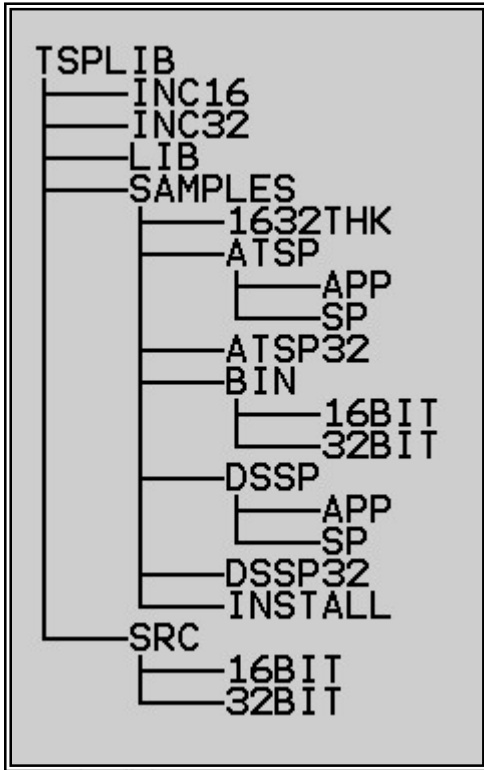
Writing a service provider for TAPI can be a challenging experience. In the Microsoft TAPI SDK, a sample service provider is provided as an example. The service provider is about as basic as possible and still requires over 1000 lines of code. The TSP++ product is designed to help implement a service provider in substantially less code on the developers part. It still assumes that the service provider writer understands the TAPI interface, and is familiar with its methods and functions. The Telephony SPI is beyond the scope of this manual. For more information about the TSPI and service providers in general, see the *Telephony Service Provider Programmer's Reference*.

The TSP++ library is a set of C++ classes that are designed to ease the task of writing a new TAPI service provider. The library uses the Microsoft Foundation Classes (MFC) and requires Visual C++ 1.5 for 16-bit development and Visual C++ 5.0 for 32-bit development.

The current levels of TAPI supported are version 1.3 under Windows 3.1, version 1.4 under Windows 95, and version 2.0 or 2.1 under Windows NT and Windows 95/98.

Files in the TSP++ SDK

Several directories are installed onto the hard disk when the TSP++ product is installed. Each directory contains a group of related files. The samples expect the directory structure installed



onto the disk.

16-bit Include files (INC16)

Filename	Description
<i>splib.h</i>	Main header file for the 16-bit TSP++ library.
<i>spuser.h</i>	Companion application header file used in the 16-bit library.

16-bit Source Files (SRC\16BIT)

Filename	Description
<i>address.cpp</i>	Implements the address level class (CTSPIAddressInfo).
<i>buttinfo.cpp</i>	Implements the CPhoneButtonInfo and CPhoneButtonArray .
<i>call.cpp</i>	Implements the call level class (CTSPICallAppearance).
<i>confcall.cpp</i>	Implements the derived conference call class (CTSPIConferenceCall).
<i>conn.cpp</i>	Implements the basic connection object (CTSPIConnection).
<i>device.cpp</i>	Implements the device level class (CTSPIDevice).
<i>display.cpp</i>	Implements the telephone display used internally (CPhoneDisplay).
<i>lineconn.cpp</i>	Implements the line level class (CTSPILineConnection).
<i>misc.cpp</i>	Miscellaneous class methods.
<i>phoneconn.cpp</i>	Implements the telephone level class (CTSPIPhoneConnection).
<i>request.cpp</i>	Implements the asynchronous request object (CTSPIRequest).
<i>sp.cpp</i>	Basic service provider class with all general methods (CServiceProvider).

<i>spdll.cpp</i>	Main TSPI_xxxx interceptor and handle validation layer.
<i>spline.cpp</i>	Service provider methods for TSPI_linexxxx functions.
<i>spphone.cpp</i>	Service provider methods for TSPI_phonexxxx functions.
<i>spthread.cpp</i>	Service provider methods for communicating with companion application.

32-bit Include files (INC32)

Filename	Description
<i>splib.h</i>	Main header file for the 32-bit TSP++ library.

32-bit Source Files (SRC\32BIT)

Filename	Description
<i>address.cpp</i>	Implements the address level class (CTSPIAddressInfo).
<i>buttinfo.cpp</i>	Implements the CPhoneButtonInfo and CPhoneButtonArray .
<i>call.cpp</i>	Implements the call level class (CTSPICallAppearance).
<i>confcalls.cpp</i>	Implements the derived conference call class (CTSPIConferenceCall).
<i>conn.cpp</i>	Implements the basic connection object (CTSPIConnection).
<i>device.cpp</i>	Implements the device level class (CTSPIDevice).
<i>display.cpp</i>	Implements the telephone display used internally (CPhoneDisplay).
<i>lineconn.cpp</i>	Implements the line level class (CTSPILineConnection).
<i>map_ds.cpp</i>	Implements a DWORD to string translation map class.
<i>misc.cpp</i>	Miscellaneous class methods.
<i>phoneconn.cpp</i>	Implements the telephone level class (CTSPIPhoneConnection).
<i>request.cpp</i>	Implements the asynchronous request object (CTSPIRequest).
<i>sp.cpp</i>	Basic service provider class with all general methods (CServiceProvider).
<i>spdll.cpp</i>	Main TSPI_xxxx interceptor and handle validation layer.
<i>spline.cpp</i>	Service provider methods for TSPI_linexxxx functions.
<i>spphone.cpp</i>	Service provider methods for TSPI_phonexxxx functions.
<i>spthread.cpp</i>	Service provider methods for communicating with thread context process.
<i>uisupp.cpp</i>	User interface support functions callable from the UI DLL context.

Pre-built Libraries (LIB)

The 16-bit library files were built using Visual C++ 1.52c, and the 32-bit libraries were built using Visual C++ 5.0. If you receive linker errors concerning MFC symbols, check to make sure that your compiler version is the same as the listed versions, or recompile the library source to match your level of compiler.

Filename	Description
<i>SPLIB.lib</i>	16-bit retail build
<i>SPLIBD.lib</i>	16-bit DEBUG build
<i>SPLIB32.lib</i>	32-bit retail build (single-byte)
<i>SPLIB32D.lib</i>	32-bit DEBUG build (single-byte)
<i>SPLIB32U.lib</i>	32-bit retail build (UNICODE)
<i>SPLIB32UD.lib</i>	32-bit DEBUG build (UNICODE)

Sample Source code (SAMPLES)

This directory contains all the sample code for the library. Several samples are presented there including:

ATSPV2

This directory contains all the source code for the POTS modem 16-bit service provider for use under Windows 3.x and Windows 95 with TAPI 1.4 installed. It will not work under Windows NT or TAPI 2.1. This sample provider duplicates the functionality of the sample shipped with the Microsoft TAPI SDK. See the section on ATSPV2 for installation information.

DSSP

This directory contains all the source code for the 16-bit digital emulated service provider. This TSP is a 16-bit service provider for use under Windows 3.x and Windows 95 with TAPI 1.4 installed. It will not work under Windows NT or TAPI 2.1. It supports the majority of the TAPI 1.4 specification through the usage of an emulator program that is installed in the **SAMPLES\BIN** directory. See the section on DSSP for more information and installation instructions.

ATSP32

This directory contains all the source code for the POTS modem 32-bit service provider for use under Windows NT and Windows 95 with TAPI 2.1 installed. It will not work under Windows 3.x or Windows 95 with TAPI 1.4. This sample duplicates the sample Microsoft TAPI provider, and adds support for asynchronous UI events, answering incoming calls, and supporting multiple devices. For more information on this sample and installation, see the section on ATSP32.

DSSP32

This directory contains all the source code for the 32-bit digital emulated service provider. This TSP is a 32-bit service provider for use under Windows NT and Windows 95 with TAPI 2.1 installed. It will not work under Windows 3.1 or Windows 95 without TAPI 2.1. It supports the majority of the TAPI 2.0 specification through an emulator program that is installed in the **SAMPLES\BIN** directory. Do not attempt to run the sample on pre-releases of NT 4.0, several problems in TAPI were corrected with the final build of NT 4.0. See the section on DSSP32 for more information and installation instructions.

BIN

This directory contains all the pre-built samples and a couple of tools for use with the library. The emulator program (**EMULATOR.EXE**) should be used with the DSSP and DSSP32 sample service providers. For information on this tool, see the section on DSSP or DSSP32.

The **INSTALL.EXE** program is used to install and remove the 16-bit samples under Windows 3.x and Windows 95. Do not run this program under Windows NT.

The **ANSWER.EXE** program is a sample 32-bit TAPI program that lists all the providers which support DATAMODEM and VOICE capabilities. It can then OPEN the device and detect incoming calls and CALLERID information. Detected calls may be answered or dropped.

INSTALL

This directory contains the source code for the install program that shows how to use the TAPI 1.4 APIs to install a service provider, or to manipulate the TELEPHON.INI file directly under Windows 3.x. It does not copy the providers into the Windows SYSTEM directory, it simply adds and removes the example providers from TAPI.

1632THK

This directory contains a sample 16 to 32 bit thunk layer for writing 32-bit companion applications for Windows 95. See the section on *Companion Applications* for more information on this subject.

Class Library Organization

The TSP++ library organization closely models the organization of the TAPI objects themselves. For each of the objects exposed through TAPI, a C++ object has been created to manage the information associated with the object.

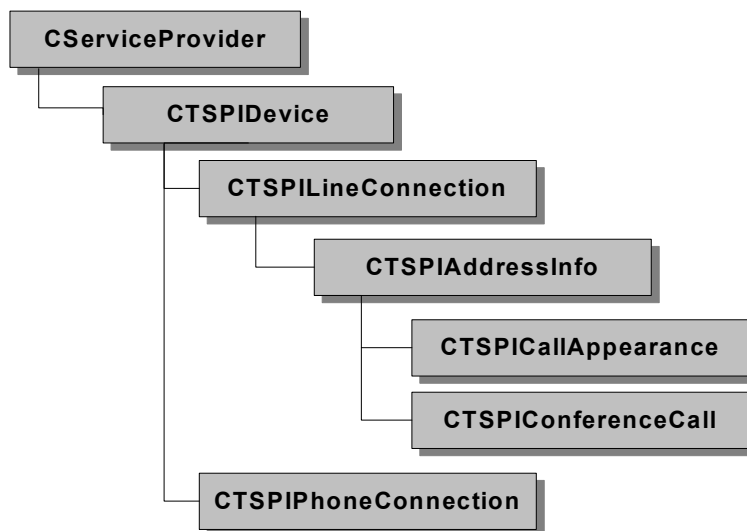
Object Lifetimes

Some of the objects, such as devices, have static lifetimes. They are created once and kept around until the provider shuts down. Others, such as the call appearances, are created and destroyed as needed while the provider runs. In most cases, the library will manage the lifetime of each of the objects. They are created and destroyed in the context of the class library and its public members.

All of the objects are derived from the MFC **CObject** class and have run-time creation support associated with them. This allows the developer to modify their behavior, but the library keeps control over creation and destruction of each object. For example, a **CTSPIDevice** object is created when the provider is loaded by TAPI in response to the **TSPI_providerInit** function. It is destroyed when the **TSPI_providerShutdown** function is called. The class itself may be overridden and modified, but nothing needs to change within the library to create/destroy the new class - through run-time creation support, the proper object type is created automatically. For more information on this topic, see the following section on *Class Overriding*.

TSP++ Organization Chart

The following chart shows the working relationship between each of the objects within the class library. This does **not** show a class derivation relationship, the objects are not related to each other in C++ terms; all the objects are instead derived from the MFC **CObject** class.



The class hierarchy starts with the **CServiceProvider** class. This is the owner and manager of all TSPi methods processed by the service provider. Each of the **TSPi_xxx** entry-points that are invoked by TAPI are checked by a thin validation layer and then passed to a method of the **CServiceProvider** class. This class locates the appropriate object to manage the function and passes the request down to the appropriate object method.

The **CTSPIDevice** object manages a single physical device for the service provider. Since multiple physical devices may be modeled in a TAPI service provider, this class contains all the lines and telephones associated with a device. In most cases, there will only be one of these objects. There would be more than one if the TSP is listed in the service provider registry more than one time (and therefore has **TSPI_providerInit** called multiple times).

The **CTSPILineConnection** and **CTSPIPhoneConnection** objects manage a single line and telephone object respectively. One is created for each line/phone reported to TAPI during initialization. Typically, these will remain during the lifetime of the provider, although the library has support for dynamic line and telephone creation using TAPI 2.x.

The **CTSPIAddressInfo** object maintains the information for an address on a line object. Since a single line can have multiple addresses (this is available on analog equipment in the form of telephone services such as distinctive ring), there can be multiple instantiations of this object.

The **CTSPICallAppearance** object is used to model a single call on an address. These objects come and go in the lifetime of the provider. There can also be more than one call on the address. Generally, one call will be in an active state, and the others will be either *idle* or in some waiting state (such as *onHold* or *pendTransfer*).

The **CTSPIConferenceCall** is a special version of the call object. It maintains a list of calls that are present in a conference call and its call state reflects the state of the conference itself.

Class Overriding

The entire set of dynamic objects (line, address, phone, call appearance, asynchronous requests, etc.) are allocated within the library source. This allows the library to perform most of the management and cleanup, making the service provider easier to develop. Unfortunately, this prohibits the derived service provider from enhancing or altering these basic objects through standard C++ derivation. The MFC library has defined a method of dynamically determining the runtime characteristics of a class and runtime class construction. This method is used by the service provider object to allow the basic dynamic objects to be overridden, but still managed completely by the library. Using the MFC **CObject** class, and storing the runtime information for the basic object types allows the library to create derivative objects which it is not completely aware of. Using the **CServiceProvider::SetRuntimeObjects** method, all the basic objects used internally in the library may be overridden and enhanced.

Since the main object **CServiceProvider** has all the **TSPI** functions routed through it, the entire provider could be written simply by overriding this object. Actually this was the original 1.0 implementation of the TSP++ library.

However, it is better designed and easier to manage if the overrides are placed at the layer which manages the specific piece of data. This generally implies an override of the **CServiceProvider** (required), **CTSPILineConnection** and **CTSPIPhoneConnection** objects, and possibly the **CTSPIDevice** object.

Handles used to communicate with TAPI

All of the TAPI functions pass *opaque handles* to the service provider which represent the different device objects recognized by the TAPI dynamic link library. TAPI has defined a few handle types that the library automatically manages:

Handle Name	Description
HTAPILINE	TAPI handle to a line device.
HTAPIPHONE	TAPI handle to a telephone device.
HTAPICALL	TAPI handle to a call appearance.
HDRVLINE	Service provider handle to a line device.
HDRVPHONE	Service provider handle to a telephone device.
HDRVCALL	Service provider handle to a call appearance.

Each of these handle types are defined as a 32-bit DWORD value which has no meaning except to the creator of the handle. During the opening of a line or telephone, or the creation of a call appearance, the TAPI dynamic link library will send **HTAPIxxx** handles where appropriate and expect the service provider to return **HDRVxxx** handles. This “swapping” of handles allows TAPI to inform the service provider about events occurring on the handle through the service providers’ representation of the device (line, telephone, or address). In the same fashion, all events the service provider sends to TAPI must use the appropriate TAPI handle so that TAPI will know which object is being represented.

In the TSP++ library, the service provider handles are always the address of the object in question. Therefore, when a **HDRVLINE** is returned to TAPI, it will be a pointer to the **CTSPILineConnection** that the line handle represents. This allows for quick lookup with no table manipulation. The TAPI dynamic link library never passes the handles outside of itself and the service provider, so the need for verification of the handle on return is minimal.

TAPI Notifications

When any event occurs in the service provider, typically some type of notification is sent to the TAPI subsystem within Windows so that it may forward the event onto any application which is interested.

TAPI uses the **TSPI_lineSetStatusMessages** and **TSPI_phoneSetStatusMessages** functions to restrict the output of the service provider to a subset of what *could* be generated. This cuts down on the amount of processing necessary by TAPI itself.

For the most part, the class library manages event notifications to TAPI itself. There are very few times when the service provider itself must generate a notification. The exceptions to this rule are when some device-specific extensions are supported by the provider (**xxx_DEVSPECIFIC**).

LINE_ADDRESSSTATE

This message is sent to notify TAPI about changes associated with the **LINEADDRESSCAPS** or **LINEADDRESSSTATUS** structures. The **LINE_ADDRESSSTATE** event is generated from within the **CTSPIAddressInfo** object.

Message	CTSPIAddressInfo methods
LINEADDRESSSTATE_OTHER	SetAddressFeatures
LINEADDRESSSTATE_DEVSPECIFIC	Not generated by the TSP library
LINEADDRESSSTATE_INUSEZER	OnCallStateChange
LINEADDRESSSTATE_INUSEONE	OnCallStateChange
LINEADDRESSSTATE_INUSEMANY	OnCallStateChange
LINE_ADDRESSSTATE_NUMCALLS	CreateConference CreateCallAppearance RemoveCallAppearance OnCallStateChange
LINEADDRESSSTATE_FORWARD	SetNumRingsNoAnswer OnRequestComplete
LINEADDRESSSTATE_TERMINALS	SetTerminalModes OnTerminalCountChanged
LINEADDRESSSTATE_CAPSCHANGE	OnAddressCapabilitiesChanged

LINE_CALLINFO

This message is sent to notify TAPI about changes associated with the **LINECALLINFO** or **LINECALLSTATUS** structure within the call appearance. The **LINE_CALLINFO** event is generated from the **CTSPICallAppearance** object.

Message	CTSPICallAppearance methods
LINECALLINFOSTATE_OTHER	SetCallParameterFlags SetDestinationCountry
LINECALLINFOSTATE_DEVSPECIFIC	Not generated by the TSP library
LINECALLINFOSTATE_BEARERMODE	SetBearerMode
LINECALLINFOSTATE_RATE	SetDataRate
LINECALLINFOSTATE_MEDIAMODE	SetMediaMode
LINECALLINFOSTATE_APPSPECIFIC	SetAppSpecific
LINECALLINFOSTATE_CALLID	SetCallID
LINECALLINFOSTATE_RELATEDCALLID	SetRelatedCallID
LINECALLINFOSTATE_ORIGIN	SetCallOrigin
LINECALLINFOSTATE_REASON	SetCallReason
LINECALLINFOSTATE_COMPLETIONID	SetCompletionID
LINECALLINFOSTATE_TRUNK	SetTrunk
LINECALLINFOSTATE_CALLERID	SetCallerIDInformation
LINECALLINFOSTATE_CALLEDID	SetCalledIDInformation
LINECALLINFOSTATE_CONNECTEDID	SetConnectedIDInformation
LINECALLINFOSTATE_REDIRECTIONID	SetRedirectionIDInformation
LINECALLINFOSTATE_REDIRECTINGID	SetRedirectingIDInformation
LINECALLINFOSTATE_USERUSERINFO	SetUserUserInfo
LINECALLINFOSTATE_TERMINAL	SetTerminalModes OnTerminalCountChanged
LINECALLINFOSTATE_DIALPARAMS	SetDialParams
LINECALLINFOSTATE_MONITORMODES	SetMonitorModes
LINECALLINFOSTATE_TREATMENT	SetCallTreatment
LINECALLINFOSTATE_QOS	SetQualityOfService
LINECALLINFOSTATE_CALLDATA	SetCallData

LINE_CALLSTATE

This method notifies TAPI when a call changes *state*, in other words, the call transitions on the hardware from one known state into another. An example of this would be when a telephone device is picked up, the first state is **Dialtone**, once the user begins to dial, the state would become **Dialing**. TAPI applications work off the call state notifications to determine what is going on with the call in question.

The TSP library maintains call information in the **CTSPICallAppearance** object. The function **CTSPICallAppearance::SetCallState** is used to adjust the state of the call which the object represents. This function will in turn notify TAPI about the current call state.

LINE_CLOSE

The **LINE_CLOSE** event tells TAPI to close the line in question, regardless of whether the application(s) is/are finished with it or not. The **CTSPILineConnection::ForceClose** method may be used to send this event.

LINE_DEVSPECIFIC

This notification is used in supporting device-specific extensions which are not part of TAPI. The class library doesn't provide any direct support for the generation of this message. Instead, the **CTSPILineConnection::Send_TAPI_Event** function may be used to directly send the event to TAPI.

LINE_DEVSPECIFICFEATURE

This notification is used in supporting device-specific extensions which are not part of TAPI. The class library doesn't provide any direct support for the generation of this message. Instead, the **CTSPILineConnection::Send_TAPI_Event** function may be used to directly send the event to TAPI.

LINE_GATHERDIGITS

This event is used during the digit gathering process started by **TSPI_lineGatherDigits** (**CTSPICallAppearance::GatherDigits**). It is sent to TAPI by the class library when the digit gathering is completed by the **CTSPICallAppearance::CompleteDigitGather** function. This feature is an automatic part of the library and should never need to be directly sent by the provider.

LINE_GENERATE

This event is used during the digit and tone generation processes started by **TSPI_lineGenerateDigits** and **TSPI_lineGenerateTone** (**CTSPICallAppearance::GenerateDigits** and **CTSPICallAppearance::GenerateTone**). It is sent to TAPI by the class library when either of these events are completed by the **CTSPICallAppearance::OnRequestComplete** function.

LINE_LINEDEVSTATE

This notification is sent when information within the **LINEDEVCAPS** or **LINEDEVSTATUS** data structures change on a line object. The line object is modeled by the **CTSPILineConnection** C++ object within the class library. As such, all **LINE_LINEDEVSTATE** are sent from this object.

Message	CTSPILineConnection methods
LINEDEVSTATE_OTHER	SetLineFeatures SetRingMode
LINEDEVSTATE_RINGING	OnRingDetected
LINEDEVSTATE_CONNECTED	SetDeviceStatusFlags
LINEDEVSTATE_DISCONNECTED	SetDeviceStatusFlags
LINEDEVSTATE_MSGWAITON	SetDeviceStatusFlags
LINEDEVSTATE_MSGWAITOFF	SetDeviceStatusFlags
LINEDEVSTATE_INSERTSERVICE	SetDeviceStatusFlags
LINEDEVSTATE_OUTOFSERVICE	SetDeviceStatusFlags
LINEDEVSTATE_MAINTENANCE	Not generated by the TSP library
LINEDEVSTATE_OPEN	Open
LINEDEVSTATE_CLOSE	Close
LINEDEVSTATE_NUMCALLS	OnCallStateChange
LINEDEVSTATE_NUMCOMPLETIONS	OnRequestComplete RemoveCallCompletionRequest
LINEDEVSTATE_TERMINALS	AddTerminal RemoveTerminal SetTerminalModes
LINEDEVSTATE_ROAMMODE	SetRoamMode
LINEDEVSTATE_BATTERY	SetBatteryLevel
LINEDEVSTATE_SIGNAL	SetSignalLevel
LINEDEVSTATE_DEVSPECIFIC	Not generated by the TSP library
LINEDEVSTATE_REINIT	Not generated by the TSP library
LINEDEVSTATE_LOCK	SetDeviceStatusFlags
LINEDEVSTATE_CAPSCHANGE	OnLineCapabilitiesChanged
LINEDEVSTATE_CONFIGCHANGE	OnMediaConfigChanged
LINEDEVSTATE_COMPLCANCEL	RemoveCallCompletionRequest

LINE_MONITORDIGITS

This event is used to notify TAPI that digits have been detected on a call. It is generated by **CTSPICallAppearance::OnDigit** method which must be called by the service provider when any digit is seen on the line.

LINE_MONITORMEDIA

This event is used to notify TAPI that media modes have changed on a call and the pattern matches something TAPI requested. It is generated by the **CTSPICallAppearance::OnDetectedNewMediaModes** method which must be called by the service provider to support media detection.

LINE_MONITORTONE

This event is used to notify TAPI about detected tones on the device. It is generated by the **CTSPICallAppearance::OnTone** method which must be called by the service provider when any tone is seen on the line.

LINE_REPLY

This event tells TAPI when any asynchronous request has completed. It is sent by the **CTSPIRequest::Complete** method when a request is finally completed by a line or phone device. For more information on the processing of asynchronous requests, see the section on *Asynchronous requests*.

PHONE_BUTTON

This event is used to notify TAPI when a button event occurs on the phone device. It is generated by the **CTSPIPhoneConnection::OnButtonStateChange** function which must be called by the service provider to support button state changes.

PHONE_CLOSE

The **PHONE_CLOSE** event tells TAPI to close the phone in question, regardless of whether the application(s) is/are finished with it or not. The **CTSPIPhoneConnection::ForceClose** method may be used to send this event.

PHONE_DEVSPECIFIC

This event is used to support device-specific extensions on the phone device. It is not directly generated by the class library. If the service provider needs to generate this event, call the **CTSPIPhoneConnection::Send_TAPI_Event** function.

PHONE_REPLY

This event tells TAPI when any asynchronous request has completed. It is sent by the **CTSPIRequest::Complete** method when a request is finally completed by a line or phone device. For more information on the processing of asynchronous requests, see the section on *Asynchronous requests*.

PHONE_STATE

This notification is sent when information within the **PHONECAPS** or **PHONESTATUS** data structures change on a phone object. The phone object is modeled by the **CTSPiPhoneConnection** C++ object within the class library. As such, all **PHONE_STATE** are sent from this object.

Message	CTSPiPhoneConnection methods
PHONESTATE_OTHER	SetPhoneFeatures
PHONESTATE_CONNECTED	SetStatusFlags
PHONESTATE_DISCONNECTED	SetStatusFlags
PHONESTATE_DISPLAY	SetDisplay
PHONESTATE_LAMP	SetLampState
PHONESTATE_RINGMODE	SetRingMode
PHONESTATE_RINGVOLUME	SetRingVolume
PHONESTATE_HANDSETHOOKSWITCH	SetHookSwitch
PHONESTATE_HANDSETVOLUME	SetVolume
PHONESTATE_HANDSETGAIN	SetGain
PHONESTATE_SPEAKERHOOKSWITCH	SetHookSwitch
PHONESTATE_SPEAKERVOLUME	SetVolume
PHONESTATE_SPEAKERGAIN	SetGain
PHONESTATE_HEADSETHOOKSWITCH	SetHookSwitch
PHONESTATE_HEADSETVOLUME	SetVolume
PHONESTATE_HEADSETGAIN	SetGain
PHONESTATE_SUSPEND	SetStatusFlags
PHONESTATE_RESUME	SetStatusFlags
PHONESTATE_DEVSPECIFIC	This is not generated by the TSP library
PHONESTATE_REINIT	This is not generated by the TSP library
PHONESTATE_CAPSCHANGE	OnPhoneCapabilitiesChanged

LINE_CREATE

This notification event is used to create a new line device while the provider is running and loaded by TAPI. It dynamically allocates a new device and is used for Plug & Play support within TAPI. The **CTSPIDevice** object generates this event when a new line is dynamically created using **CTSPIDevice::CreateLine**.

PHONE_CREATE

This notification event is used to create a new phone device while the provider is running and loaded by TAPI. It dynamically allocates a new device and is used for Plug & Play support within TAPI. The **CTSPIDevice** object generates this event when a new phone is dynamically created using **CTSPIDevice::CreatePhone**.

Companion Applications

Important Note:

This section does not apply to the 32-bit library. It uses threads owned by the **TAPISRV.EXE** application to perform its work and therefore does not need or support the companion application concept.

In the 16-bit versions of Windows, each application running has at least one thread of execution (in Windows 3.1, each application can have only one thread, in Windows 95, 32-bit applications can create additional threads). These threads are then scheduled by Windows to provide the image of multitasking. Each application running in the 16-bit system is called a *task*. Any object created, such as a window, or a block of allocated memory, is attached to the task, and deleted or destroyed when the task exits. DLLs are not considered tasks, and if they perform any object allocations (such as a **CreateWindow** or **LocalAlloc**), then the object is attached to the task which called the DLL. (To be technically accurate, global memory can be a special exception to this). Since a DLL is not a task, it is not scheduled directly by Windows. It cannot get control of the processor to run code within the DLL unless an application calls it.

Since TAPI is a 16-bit subsystem that runs under Windows 3.1 and Windows 95, the service provider is also required to be 16-bit. In addition, many of the TSPI functions are required to be *asynchronous*. This indicates that the requesting application wants the service provider to initiate the operation, return to the application, and finish the request in the background. Once the request has completed (with either an error or success), the service provider notifies TAPI through a callback and tells the application that the operation has completed. Essentially the service provider needs some way to get control to perform tasks without a TAPI application calling it.

The TSP++ library handles this by starting an optional *companion application*. This companion application is an executable program which is periodically scheduled by Windows (since it is a task), and calls back into the service provider through an exported function in order to process responses from the device, or initiate requests. The companion application gives the service provider an independent executable thread to perform asynchronous work. Allowing the allocation of resources within Windows such as file handles, devices, and memory, etc.

Commands and Responses

The companion application interface is defined in the *spuser.h* include file. A simple command set is defined for opening a device connection, closing a device connection, sending a block of data to a device, and receiving a block of data from a device. The basic design of the companion application is that all device I/O is sent through the companion application to go to the device. This is not a requirement of the library, if another design is required, then the appropriate interfaces may be overridden in the appropriate objects, and this can be changed. The basic defined commands are:

Command	Description
COMMAND_OPENCONN	Open a connection to a device. A unique 32-bit value is used to identify the provider and line or telephone device. This same identifier will be used for any further request for this device.
COMMAND_CLOSECONN	Close an open connection to a device. The device is identified by the 32-bit value passed when it was opened.
COMMAND_SENDDATA	Send a block of data to the device.
COMMAND_WAITINGREQ	This causes the companion application to call into the service provider stopping the current thread, and activating the provider in the context of the companion application.
COMMAND_END	This is defined as a starting point for implementing new commands in the companion application for provider-specific

extensions.

Once the companion application has processed a request and received a response from the hardware device it is working with, it will send a response back to the service provider through a public exported function called **DeviceNotify**. This function will then call an object within the class library to process the response. The defined responses from a companion application are:

Command	Description
RESULT_INIT	Initialization request from the companion application. This is the first request passed through the provider and calls the CServiceProvider::InitializeRequestThread function.
RESULT_RCVDATA	This is a notification of received data from the companion application. The connection id which was used during the COMMAND_OPENCONN is passed back through to direct the response to a particular CTSPIDevice object.
RESULT_INTERVALTIMER	An Interval timer interrupt from the companion application. This should be periodically generated in the companion application if necessary. It calls the CTSPIDevice::OnTimer function.
RESULT_CONTEXTSWITCH	This is the response to a COMMAND_WAITINGREQ command from the service provider. It calls the CTSPIDevice object with a NULL data packet.

Defining new commands and responses

The companion application interface is defined in the *spuser.h* header file. If new commands or responses are created for the derived service provider, they should fall after the **COMMAND_END** and **RESULT_END** values. To send a command to the companion application, the method **CServiceProvider::SendThreadRequest** may be used. If new responses from the companion application are required, they may be filtered by overriding the method **CServiceProvider::UnknownDeviceNotify**.

Using a 32-bit companion application

The Microsoft UNIMODEM service provider supports a new **lineGetID** type "*comm/datamodem*" which returns a 32-bit COMM handle that can be used by any of the 32-bit communications functions in Windows 95.

To support this kind of functionality, the companion application must also be 32-bit so that it obtains handles to 32-bit devices. The TSP++ library can support 32-bit companion applications but there are some rules that must be followed. Some of this is done automatically by the class library but the developer must make sure to manage the data properly.

- All data sent to the companion application from the service provider is automatically *thunked* using the Win32 **WM_COPYDATA** message. To properly use this support, any pointer sent as the **lParam** of the **CServiceProvider::SendThreadRequest** method must be identified as a pointer by setting the **flParamIsPointer** to **TRUE**. Otherwise, the data will *not* be translated across the process boundary.
- Embedded pointers within passed structures are not supported. The companion application receives all data in the **WM_COPYDATA** message handler. If the service provider is to communicate with a 32-bit companion application, it **must** pass all pointers using this method.
- When data is sent to the companion application, the data cannot be altered by the receiving companion application. The thunk mechanism implemented by the **WM_COPYDATA** message *copies* the data into the process space of the target window. This means that changes made to the data by the receiver will not be reflected in the data buffer of the owner since it is a separate memory buffer.

- To callback into the service provider from the 32-bit companion application, a thunk DLL needs to be written. This DLL is called where the **DeviceNotify** function was normally invoked. A sample of this DLL is in the **SAMPLES\1632THK** directory. The sample defines a new function call **DeviceNotify32** which calls the 16-bit **DeviceNotify**. The **1632THK** directory contains all the technical information on this procedure and how to implement it in a service provider.

Using an interrupt-driven approach

If the device that the service provider works with can be interrupt driven, it may not be necessary to have a companion application. For instance, if the device generates an interrupt when it has completed a task (vs. sending a Windows message), interrupt code can be written and simply call the **DeviceNotify** method of the **CServiceProvider** class. Additionally, the **ProcessRequest** method can simply take each request and issue the appropriate commands *on the application execution context*, and then let the interrupt service routine notify it when the command completes. An example of this approach would be to use a VxD to control the device and perform the callbacks into the TSP.

Interval Timer requirements

If media control, digit gathering, or tone monitoring is available, then a periodic timer *must* cycle through the library within the minimum duration allowed for tone/digit monitoring. In the default configuration, this is achieved through the companion application and an event timer. Without a companion application present, either a substitute timer must be supplied (potentially through the hardware device and another interrupt routine), or these features must be disabled by not exporting them from the TSP.

User-Interface Events

16-bit service providers

The 16-bit implementation of TAPI does not support remote service providers or client-server style access like Novell's **TSAPI** standard. Generally, unless programmed by the developer, it can be assumed that the TSP is running on the same machine as the client requesting service. Therefore, user-interface events such as windows or dialogs may be performed directly within the service provider code.

When TAPI requests a user-interface event through the standard functions **TSPI_lineConfigDialog**, **TSPI_lineConfigDialogEdit**, **TSPI_phoneConfigDialog**, **TSPI_providerConfig**, **TSPI_providerInstall**, and **TSPI_providerRemove**, the provider may run a message loop and should return from the function when the user-interface object has been destroyed.

When the service provider is displaying a user-interface object without TAPI specifically requesting it (a *modem talk/drop dialog* for example), then care should be taken to insure that the thread which runs the message loop is part of the companion application. This is due to the hybrid 16/32 bit architecture of Windows 95. If a message for a 32-bit application is dispatched by a 16-bit application, then the high bits of the **wParam** field are truncated, and could cause a fault in an application other than the service provider.

32-bit service providers

In the 32-bit implementation of TAPI, service providers execute in the process space of the **TAPISRV.EXE** application, a Windows NT service. Applications calling TAPI functions invoke functions in the **TAPI32.DLL** module which routes the request using an RPC to the **TAPISRV** application, which in turn calls the appropriate **TSPI_XXX** function in the service provider. This allows for not only process-separation of the provider, but also client-server abilities, allowing the service provider to execute on a completely separate machine, but be controlled from other client machines. Therefore, the provider can no longer display user-interface dialogs directly within the provider code, since they are not within the same process, or even the same machine. The user-interface dialog must be invoked within the application context on the machine where the application is executing.

To facilitate this, Microsoft has defined a new telephony service provider UI DLL interface. The **TAPI32.DLL** module uses this interface to find, load, and run a user-interface dialog within the application process space. TAPI uses a user-interface DLL (**UIDLL**) which is targeted by the service provider, and has specific interface entry-points to run the user-interface logic for the provider. For information on the architecture and inner workings of this **UIDLL** mechanism, see the Win32 documentation on the *Telephony Service Provider UI DLL Interface*.

Normal User Interface events

The **TSP++** library manages this new **UIDLL** architecture automatically by naming the TSP module as user-interface module, which causes **TAPI32** to load the TSP into the application space. The UI functions are then wrapped into **CServiceProvider** methods that are invoked within the application process space. These include: **TSPI_lineConfigDialog**, **TSPI_lineConfigDialogEdit**, **TSPI_phoneConfigDialog**, **TSPI_providerConfig**, **TSPI_providerInstall**, and **TSPI_providerRemove**.

The **providerXXX** functions in the **CServiceProvider** class should be overridden to supply user-interface dialogs when necessary. These methods will be called only within the application context when a UI event is necessary. They will not be called within the context of **TAPISRV**, so the line/phone/device information will *not* be available and should not be assumed.

Spontaneous Dialogs

The line object has a new method **CreateUIDialog** that will cause TAPI to invoke the UIDLL in the appropriate application space. The service provider may interact with the dialog through the **SendDialogInstanceData** method. Events sent from the dialog are received by the **providerGenericDialogData** method of the **CServiceProvider** class.

TAPI WARNING

Due to the thread state architecture of MFC, and the method that TAPI 2.0 uses to generate spontaneous UI events, the thread may not be setup correctly for MFC to use it as a UI generating thread. TAPI loads the DLL on one thread and then uses a second thread to actually invoke the dialog. Therefore, it is advised that the asynchronous dialogs generated through **providerGenericDialog** do not use MFC for the dialog. This problem was present in the original TAPI 2.0 release under Windows NT, and may be corrected in future releases.

Dialing Information

Most of the addresses received from TAPI are in *dialable address format*.

This format is composed of the following sections:

Dialable Number | Subaddress ^ Name CRLF ...

Section	Description
<i>DialableNumber</i>	Contains digits and modifiers 0-9 A-D * # , ! W w P p T t @ \$? ; delimited by ^ CRLF or the end of the dialable address string.
<i>Subaddress</i>	The plus sign (+) is a valid character in dialable strings. It indicates that the phone number is a fully qualified international number. A variably sized string containing a subaddress. The string is delimited by the next + ^ CRLF or the end of the address string. When dialing, subaddress information is passed to the remote party. It can be for an ISDN subaddress, an email address, and so on.
<i>Name</i>	A variably sized string treated as name information. Name is delimited by CRLF or the end of the dialable address string. When dialing, name information is passed to the remote party.
CRLF	ASCII Hex (0D) followed by ASCII Hex (0A). If present, this optional character indicates that another dialable number is following this one. It is used to separate multiple dialable addresses as part of a single address string (for inverse multiplexing).

Within the dialable number portion, the following characters are allowed:

Character	Description
0-9, A-D, *, #, +	ASCII characters corresponding to the DTMF and/or pulse digits.
!	ASCII Hex (21). Indicates that a hookflash (one-half second on hook, followed by one-half second off-hook before continuing) is to be inserted in the dial string.
P, p	ASCII Hex (50) or Hex (70). Indicates that pulse dialing is to be used for the digits following it.
T, t	ASCII Hex (54) or Hex (74). Indicates that tone (DTMF) dialing is to be used for the digits following it.
,	ASCII Hex (27). Indicates that dialing is to be paused. The duration of a pause is device specific and can be retrieved from the line's device capabilities. Multiple commas can be used to provide longer pauses.
W, w	ASCII Hex (57) or Hex (77). An uppercase or lowercase W indicates that dialing should proceed only after a dial tone has been detected. This will only be processed if LINEDEVCAPS has the LINEDEVCAPFLAGS_DIALDIALTONE flag set in the dwDevCapFlags field.
@	ASCII Hex (57) or Hex (77). An uppercase or lowercase W indicates that dialing should proceed only after a dial tone has been detected. This will only be processed if LINEDEVCAPS has the LINEDEVCAPFLAGS_DIALQUIET flag set in the dwDevCapFlags field.
\$	ASCII Hex (24). It indicates that dialing the billing information is to wait for a "billing signal" (such as a credit card prompt tone). This will only be processed if LINEDEVCAPS has the LINEDEVCAPFLAGS_DIALBILLING flag set in the dwDevCapFlags field.

- ? ASCII Hex (3F). It indicates that the user is to be prompted before continuing with dialing. The provider does not actually do the prompting, but the presence of the "?" forces the provider to reject the string as invalid, alerting the application to the need to break it into pieces and prompt the user in-between.
- ; ASCII Hex (3B). If placed at the end of a partially specified dialable address string, it indicates that the dialable number information is incomplete and more address information will be provided later.

The **CServiceProvider::CheckDialableAddress** method is provided to break the information out of the dialable address format from TAPI. This function is used throughout the class library to break a dialable address into its component parts. These parts are stored into an internal data structure called a **DIALINFO** structure. For more information on **DIALINFO**, see the section on *Structures: DIALINFO*.

Memory Allocations

All memory allocated within the 16-bit library is done with the Windows API **GlobalAlloc** function so that the memory is valid no matter which process it was allocated within.

In the 32-bit library, the **LocalAlloc** function is used since it is always on the same process.

Generally, there should never be too many outstanding asynchronous requests pending, so the memory usage is relatively minimal with most TSP implementations.

The following global functions are used internally and are available in the library to manage memory:

Command	Description
AllocMem	Allocate a block of memory
FreeMem	Free a block of memory
CopyBuffer	Copy a block of memory from one location to another.
CopyVarString	Copies a buffer into a VARSTRING pointer.
FillBuffer	Initialize a buffer with a known value.
DwordAlignPtr	DWORD aligns an address (32-bit only).
AddDataBlock	Adds a new VARSTRING block to a buffer.

Special Considerations for Windows NT™

Portability to other platforms

Since the Windows NT system is portable to other hardware platforms, this implies that the service providers must also be portable to other platforms. To facilitate this, the 32-bit library **DWORD** aligns all data structures using the function **DwordAlignPtr** that is located in *spdll.cpp*. It is recommended that any data structures that are passed to applications using the device-specific TSP APIs also be **DWORD** aligned so that it is easily portable to other processor architectures.

UNICODE support

Windows NT works internally on UNICODE - where each character is represented by 16-bits vs. the normal 8-bit ASCII approach used in previous versions of Windows. **TAPISRV.EXE**, the 32-bit TSP manager requires that the TSP fully support UNICODE in the sense that all strings passed to and from the TSP will always be in UNICODE form. To allow for porting of older service providers which were written using previous versions of TSP++, the 32-bit library is shipped in two forms. The first assumes that all string manipulations are done using UNICODE functions. The second sets a translation layer between **TAPISRV** and the TSP driver code which converts all incoming strings to SBCS, and all outgoing strings in all structures to UNICODE. This allows the TSP to be written and compiled without UNICODE support, but still conforms to the requirements of **TAPISRV.EXE**

Threads

The TSP++ library is fully re-entrant and expects that the derived provider will be as well. The interval timer is called by a thread created in the library and is subject to Win32 task-switching rules.

There is a special class in the 32-bit library called **CEnterCode** that takes any of the TSP++ library objects as its first parameter and locks the object for updates. Each of the TSPI objects have a critical section associated with them so that any other thread that needs to update/access volatile information will be suspended until the lock is released. The destructor of the **CEnterCode** class will automatically release the lock. You *must* synchronize any access to data structures which could be modified through some fashion or turn off the interval timer through the **CServiceProvider::SetTimeout** method and tell TAPI that the provider is not re-entrant through a **CServiceProvider::providerInit** override.

Debugging

Since the 32-bit TSP drivers under Windows NT run under an NT service (**TAPISRV.EXE**), any output generated through the tracing facility cannot be seen unless a debugger is attached to the **TAPISRV** process, or a system-wide debugger such as NuMega's SoftIce/NT is installed. This makes debugging the TSP a bit more difficult than in previous versions. There is a new task manager in Windows NT, which allows the debugger to be attached to any process. This new feature will allow the **TAPISRV.EXE** process to be attached to the debugger once TAPI has been started. The TRACE mechanism in the library uses **OutputDebugString** so any attached debugger will see the output.

There is no global DBWIN (as in 16-bit development) available in the Win32 SDK, although there are several shareware versions available for Windows 95 and Windows NT.

Debug functions

In addition to the standard MFC debug macros, the following global functions are provided in the DEBUG version of the library to help with debugging. These functions output through the standard debug TRACE facility.

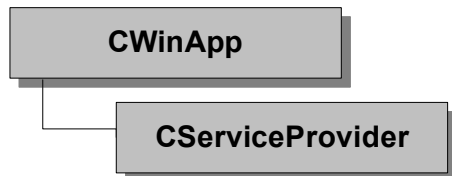
Command	Description
DumpMem	Dump a block of memory in HEX/ASCII out to the debug terminal. The results can be captured using DBWIN.EXE or any Microsoft-compatible debugger.
DumpVarString	Dump a VARSTRING pointer in HEX/ASCII out to the debug terminal. The results can be captured using DBWIN.EXE or any Microsoft-compatible debugger.

Technical Support

Please contact **JulMar Entertainment Technology, Inc.** by sending E-mail to staff@julmar.com for technical support concerning the TSP++ library:

In additional, upgrades to the library will be released through the company web site at www.julmar.com.

CServiceProvider



In the TSP++ class library, the basic class that will always be present is a class derived from **CServiceProvider**. This class is derived from the MFC class **CWinApp** so it doubles in usage as the application object for the dynamic link library. The **CServiceProvider** object is the main manager of all the functions going on in the service provider. As each "C" function is called, the *spdll.cpp* **TSPI_** function layer will call the **CServiceProvider** object to perform the work. All the functions are named identically to the **TSPI_** exports that TAPI calls, without the **TSPI_** prefix. Therefore, if TAPI calls the **TSPI_lineOpen** function, *spdll.cpp* would call the **lineOpen** method of the service provider object.

The *spdll.cpp* layer and **CServiceProvider** class do minimal validation on the request and then pass the request to the object most suited to handle it. Each **TSPI_**xxx handler is declared virtual so it may be overridden at the service provider level if that is the most convenient.

All buffers which are received by the service provider object and passed down to the lower levels of the library are automatically copied into local memory buffers to insure that the pointers cannot become invalid when the request goes asynchronous.

Service Provider Initialization

All line and telephone device objects are created during initialization of the service provider. This occurs when TAPI calls the **TSPI_providerInit** function.

Once the **CServiceProvider::providerInit** function completes, all the device objects should be initialized and ready for use. The derived class could then configure the lines/phones in the appropriate fashion, overriding the default values specified by the library if necessary.

16-bit specific initialization

The companion application is started during the **providerInit** function. It is specified by passing an executable name on the constructor to the **CServiceProvider**. If this executable name is NULL, then no application is started. Otherwise, it is assumed that it identifies an executable program that will be started during the initialization of the **CServiceProvider** class. A common use of this executable is to perform the actual low-level I/O to the device, and supply periodic interval timers which are used for event timing and error checking in the library. For more information on this executable, see the above section titled *Context Thread Application*.

32-bit specific initialization

The interval timer thread is started during the **providerInit** function, the actual interval value can be adjusted through the **SetTimerResolution** method. Since the 32-bit version supports threads, no companion application is started, it is expected that some form of thread will be started to support input from the device.

Persistent data storage

Any persistent data that needs to be stored by the service provider should use the following functions in the library, which have been created for this purpose:

- ReadProfileString
- ReadProfileDWord
- WriteProfileString,
- WriteProfileDWord

These functions store the information into an area dedicated to the provider. For the 16-bit version of the library, this will be a section of the **TELEPHON.INI** file. For the 32-bit library, this area will be a section of the registry under

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Telephony\Providers.

Required and Typical Overrides

The two functions that generally must be overridden in this object are the constructor, which is called during the initialization of the driver, and **providerEnumDevices** which is used to determine how many line/phone devices are connected to the service provider.

The constructor is where any replacement of object types needs to take place (using **SetRuntimeObjects**), for more information on this topic, see the section titled *Class Overriding*.

The **providerEnumDevices** function is a required override for TAPI 2.x, and an optional override for TAPI 1.x. It should fill in the count of lines and phones supported by the provider as this information is used by TAPI to call the **providerInit** function after **providerEnumDevices** returns.

Important note: If a 16-bit provider is written for Windows 3.1, or the **providerEnumDevices** function is not overridden, it must fill out two fields in the **TELEPHON.INI** which define the number of phones and lines for the device. These two entries are **NumLines** and **NumPhones** and are placed in the provider section (*[ProviderX]*) of the INI file. The provider will not load without these entries.

Construction/Destruction - Public Members

CServiceProvider	Constructs a CServiceProvider object. This takes the TSP name, companion application to start, highest version of TAPI to negotiate to, and provider information.
-------------------------	--

Initialization - Public Members

SetTimerResolution	Sets the internal timer which is run by an internal thread and replaces the interval timer supplied by the companion application (32-bit only).
---------------------------	---

Initialization - Protected Members

SetRuntimeObjects	Override the default object types for each of the basic objects (line, phone, device, request, class, and address).
SetTimeout	This sets the default time-out used for waiting on a request to finish before reporting an error.

Operations - Public Members

DeleteProfile	Delete a profile from the registry or Telephon.INI
GetConnInfoFromLineDeviceID	Return a pointer to a CTSPILineConnection based on a line device id.
GetConnInfoFromPhoneDeviceID	Return a pointer to a CTSPIPhoneConnection based on a phone device id.
GetCurrentLocation	Returns the current location (valid under TAPI 1.4 only).

GetDevice	Return a pointer to a CTSPIDevice based on a device id.
GetDeviceCount	Returns a count of existing devices (32-bit only).
GetDeviceByIndex	Returns a pointer to a CTSPIDevice based on an index (32-bit only).
GetProviderInfo	Returns provider specific information which was supplied by the constructor.
GetProviderName	Returns the provider name which was supplied by the constructor. Not applicable for the 32-bit library.
GetSupportedVersion	Returns the version of TAPI the service provider negotiates at. This would be the highest level of TAPI supported by the service provider. The library supports all versions up to 1.4 (2.0 for 32-bit).
GetSystemVersion	Returns the version of TAPI.DLL that started the service provider.
ReadProfileDWord	Reads a DWORD from the registry or <i>Telephon.INI</i> .
ReadProfileString	Reads a string from the registry or <i>Telephon.INI</i> .
RenameProfile	Rename a profile in the registry or <i>Telephon.INI</i> .
SendThreadRequest	This is called to send a command packet to the companion application. The derived class should not call it unless new commands are defined in the companion application.
WriteProfileDWord	Writes a DWORD to the registry or <i>Telephon.INI</i> .
WriteProfileString	Writes a string to the registry or <i>Telephon.INI</i> .

Overridables - Public Members

CheckDialableNumber	Determine if an address is valid. Splits the dialable address into component DIALINFO objects which contain address, name, and sub-address information. This is called for any dialable address passed into the service provider.
CloseDevice	Close an open device through the companion application. Default 16-bit implementation sends a COMMAND_CLOSECONN request to the companion application. This is called from CTSPIDevice::Close .
ConvertDialableToCaonical	Converts a dialable number to the best canonical equivalent that can be determined. This is called to format caller id information.
GetDialableNumber	Strips out any non-specified characters from a dialable string. Always leaves any numbers in place.
MatchTones	Match a set of frequencies for tone monitoring. The default implementation compares each frequency for an exact match.
OpenDevice	Open a device through the companion application. Default 16-bit implementation sends a COMMAND_OPENCONN request to the companion application. This is called from CTSPIDevice::Open .
OnTimer	Perform some work on a periodic basis. This is called by the CTSPIDevice::OnTimer .
ProcessCallParameters	Check a set of call parameters for validity. This should be overridden if special requirements are needed for LINECALLPARAM structures passed from TAPI.
SendData	Send a block of data to a device through the companion application. Default 16-bit implementation sends a COMMAND_SENDDATA command to the companion application. This is called from the CTSPIDevice::SendData method.
UnknownDeviceNotify	This is called when an unknown response is sent from the companion application. Override this if new responses are

defined. (16-bit only)

Overridables - Protected Members

CancelRequest	Cancel a request which has already begun on the device. This is called when calls/lines are closed with pending running requests. The default implementation does nothing.
OnNewRequest	This is called by the connection object each time a new request packet is added to the request list. It is called BEFORE the request has actually been added, and allows the derived provider to manipulate the request list or cancel this request by returning FALSE. TRUE is returned by default.
CanHandleRequest	Return whether a request can be handled by this service provider. Default implementation checks to see if the TSPI_xxxx function is exported from the DLL.
CheckCallFeatures	This method is called when the call features for an active call change. It may be overridden to adjust the features available to the line, address, or call as the call changes states.
ProcessData	Process a data result from a device. This is the main worker function of the service provider. It should process each request for the device. This must be overridden if it is not handled at the device or connection level.
StartNextCommand	Start the next request in the request list. The default implementation calls the ProcessData method with a NULL data value in the context of the companion application.

Overridables - TAPI Members

providerConfig	This is called in response to the TSPI_providerConfig command. The default implementation returns FALSE.
providerFreeDialogInstance	This needs to be overridden to supply a user-interface dialog. This is called in 32-bit TAPI when a user-interface dialog is being terminated (32-bit only).
providerGenericDialog	This is called to provide a generic user-interface dialog. (32-bit only).
providerGenericDialogData	This is called to provide information to an existing generic user-interface dialog. (32-bit only).
providerUIIdentify	Identifies the user-interface DLL which is loaded into the application process space for UI events. (32-bit only).
providerInit	This is called in response to the TSPI_providerInit command. The default implementation initializes the device objects and starts the companion application.
providerInstall	This is called in response to the TSPI_providerInstall command. The default implementation returns FALSE. This should be overridden if special installation steps need to be taken (such as determining configuration).
providerRemove	This is called in response to the TSPI_providerRemove command. The default implementation returns FALSE. This should be overridden if special removal steps need to occur (such as removing additional files).
providerShutdown	This is called in response to the TSPI_providerShutdown command. The default implementation shuts down the companion application.
providerEnumDevices	This is called in response to the TSPI_providerEnumDevices command. The default implementation returns the count of lines and phones listed in the telephon.ini for the supplied provider id.

providerCreateLineDevice	This is called in response to the TSPI_providerCreateLineDevice command. The default implementation assigns the correct line device ID to the line object.
providerCreatePhoneDevice	This is called in response to the TSPI_providerCreatePhoneDevice command. The default implementation assigns the correct line device ID to the phone object.
lineAccept	This is called in response to the TSPI_lineAccept command. The default implementation validates and copies parameters and calls the CTSPICallAppearance::Accept method.
lineAddToConference	This is called in response to the TSPI_lineAddToConference command. The default implementation validates and copies parameters and calls the CTSPIConferenceCall::AddToConference method.
lineAnswer	This is called in response to the TSPI_lineAnswer command. The default implementation validates and copies parameters and calls the CTSPICallAppearance::Answer method.
lineBlindTransfer	This is called in response to the TSPI_lineBlindTransfer command. The default implementation validates and copies parameters and calls the CTSPICallAppearance::BlindTransfer method.
lineClose	This is called in response to the TSPI_lineClose command. The default implementation calls the CTSPILineConnection::Close method.
lineCloseCall	This is called in response to the TSPI_lineCloseCall command. The default implementation calls the CTSPICallAppearance::Close method.
lineCompleteCall	This is called in response to the TSPI_lineCompleteCall command. The default implementation validates and copies parameters and calls the CTSPICallAppearance::CompleteCall method.
lineCompleteTransfer	This is called in response to the TSPI_lineCompleteTransfer command. The default implementation validates and copies parameters and calls the CTSPIAddressInfo::CompleteTransfer method.
lineConditionalMediaDetection	This is called in response to the TSPI_lineConditionalMediaDetection command. The default implementation calls the CTSPILineConnection::ConditionalMediaDetection method.
lineConfigDialog	This is called in response to the TSPI_lineConfigDialog command. The default implementation calls the CTSPILineConnection::ConfigDialog method.
lineDevSpecific	This is called in response to the TSPI_lineDevSpecific command. The default implementation calls the CTSPILineConnection::DevSpecificFeature method.
lineDevSpecificFeature	This is called in response to the TSPI_lineDevSpecificFeature command. The default implementation calls the CTSPILineConnection::DevSpecificFeature method.
lineDial	This is called in response to the TSPI_lineDial command. The default implementation validates and copies parameters and calls the CTSPICallAppearance::Dial method.
lineDrop	This is called in response to the TSPI_lineDrop command. The default implementation validates and copies parameters

lineForward	and calls the CTSPICallAppearance::Drop method. This is called in response to the TSPI_lineForward command. The default implementation validates and copies parameters and calls the CTSPIAddressInfo::Forward method.
lineGatherDigits	This is called in response to the TSPI_lineGatherDigits command. The default implementation validates and copies parameters and calls the CTSPICallAppearance::GatherDigits method.
lineGenerateDigits	This is called in response to the TSPI_lineGenerateDigits command. The default implementation validates and copies parameters and calls the CTSPICallAppearance::GenerateDigits method.
lineGenerateTone	This is called in response to the TSPI_lineGenerateTone command. The default implementation validates and copies parameters and calls the CTSPICallAppearance::GenerateTone method.
lineGetAddressCaps	This is called in response to the TSPI_lineGetAddressCaps command. The default implementation validates and copies parameters and calls the CTSPIAddressInfo::GatherCapabilities method.
lineGetAddressID	This is called in response to the TSPI_lineGetAddressID command. The default implementation validates and copies parameters and calls the CTSPILineConnection::GetAddressID method.
lineGetAddressStatus	This is called in response to the TSPI_lineGetAddressStatus command. The default implementation validates and copies parameters and calls the CTSPIAddressInfo::GatherStatusInformation method.
lineGetCallAddressID	This is called in response to the TSPI_lineGetcallAddressID command. The default implementation manages this completely.
lineGetCallInfo	This is called in response to the TSPI_lineGetCallInfo command. The default implementation validates and copies parameters and calls the CTSPICallAppearance::GatherCallInformation method.
lineGetCallStatus	This is called in response to the TSPI_lineGetCallStatus command. The default implementation validates and copies parameters and calls the CTSPICallAppearance::GatherStatusInformation method.
lineGetDevCaps	This is called in response to the TSPI_lineGetDevCaps command. The default implementation validates and copies parameters and calls the CTSPILineConnection::GatherCapabilities method.
lineGetDevConfig	This is called in response to the TSPI_lineGetDevConfig command. The default implementation validates and copies parameters and calls the CTSPILineConnection::GetDevConfig method.
lineGetExtensionID	This is called in response to the TSPI_lineGetExtensionID command. The default implementation fills in the fields with zeros and returns FALSE.
lineGetIcon	This is called in response to the TSPI_lineGetIcon command. The default implementation calls the CTSPILineConnection::GetIcon method.
lineGetID	This is called in response to the TSPI_lineGetID command.

lineGetLineDevStatus	The default implementation passes this onto the CTSPICallAppearance , CTSPIAddressInfo , and CTSPILineConnection object(s) depending on what parameters (line/address/call) are passed into the function. This is called in response to the TSPI_lineGetLineDevStatus command. The default implementation validates and copies parameters and calls the CTSPILineConnection::GatherStatus method.
lineGetNumAddressIds	This is called in response to the TSPI_lineGetNumAddressIds command. The default implementation uses the address count of the line to handle this request.
lineHold	This is called in response to the TSPI_lineHold command. The default implementation validates and copies parameters and calls the CTSPICallAppearance::Hold method.
lineMakecall	This is called in response to the TSPI_lineMakecall command. The default implementation validates and copies parameters and calls the CTSPILineConnection::MakeCall method.
lineMonitorDigits	This is called in response to the TSPI_lineMonitorDigits command. The default implementation validates and copies parameters and calls the CTSPICallAppearance::MonitorDigits method.
lineMonitorMedia	This is called in response to the TSPI_lineMonitorMedia command. The default implementation validates and copies parameters and calls the CTSPICallAppearance::MonitorMedia method.
lineMonitorTones	This is called in response to the TSPI_lineMonitorTones command. The default implementation validates and copies parameters and calls the CTSPICallAppearance::MonitorTones method.
lineNegotiateExtVersion	This is called in response to the TSPI_lineNegotiateExtVersion command. The default implementation sets the negotiated version to zero and returns FALSE if the other parameters are valid.
lineNegotiateTSPIVersion	This is called in response to the TSPI_lineNegotiateTSPIVersion command. The default implementation performs a negotiation which takes into account the highest version the provider indicated, and down to TAPI 1.3.
lineOpen	This is called in response to the TSPI_lineOpen command. The default implementation validates and copies parameters and calls the CTSPILineConnection::Open method.
linePark	This is called in response to the TSPI_linePark command. The default implementation validates and copies parameters and calls the CTSPICallAppearance::Park method.
linePickup	This is called in response to the TSPI_linePickup command. The default implementation validates and copies parameters and calls the CTSPIAddressInfo::Pickup method.
linePrepareAddToConference	This is called in response to the TSPI_linePrepareAddToConference command. The default implementation validates and copies parameters and calls the CTSPIConferenceCall::PrepareAddToConference method.
lineRedirect	This is called in response to the TSPI_lineRedirect command. The default implementation validates and copies

	parameters and calls the CTSPICallAppearance::Redirect method.
lineRemoveFromConference	This is called in response to the TSPI_lineRemoveFromConference command. The default implementation validates and copies parameters and calls the CTSPICallAppearance::RemoveFromConference method.
lineSecureCall	This is called in response to the TSPI_lineSecureCall command. The default implementation validates and copies parameters and calls the CTSPICallAppearance::Secure method.
lineSelectExtVersion	This is called in response to the TSPI_lineSelectExtVersion command. The default implementation returns LINEERR_OPERATIONUNAVAIL .
lineSendUserUserInfo	This is called in response to the TSPI_lineSendUserUserInfo command. The default implementation validates and copies parameters and calls the CTSPICallAppearance::SendUserUserInfo method.
lineSetAppSpecific	This is called in response to the TSPI_lineSetAppSpecific command. The default implementation validates and copies parameters and calls the CTSPICallAppearance::SetAppSpecific method.
lineSetCallParams	This is called in response to the TSPI_lineSetCallParams command. The default implementation validates and copies parameters and calls the CTSPICallAppearance::SetCallParams method.
lineSetDefaultMediaDetection	This is called in response to the TSPI_lineSetDefaultMediaDetection command. The default implementation calls the CTSPILineConnection::SetDefaultMediaDetection method.
lineSetDevConfig	This is called in response to the TSPI_lineSetDevConfig command. The default implementation calls the CTSPILineConnection::SetDevConfig method.
lineSetMediaControl	This is called in response to the TSPI_lineSetMediaControl command. The default implementation validates and copies parameters and calls either the CTSPICallAppearance , CTSPIAddressInfo , or CTSPILineConnection::SetMediaControl method depending on the parameters passed.
lineSetMediaMode	This is called in response to the TSPI_lineSetMediaMode command. The default implementation validates and copies parameters and calls the CTSPICallAppearance::SetMediaMode method.
lineSetStatusMessages	This is called in response to the TSPI_lineSetStatusMessages command. The default implementation validates and copies parameters and calls the CTSPILineConnection::SetStatusMessages method.
lineSetTerminal	This is called in response to the TSPI_lineSetTerminal command. The default implementation validates and copies parameters and calls either the CTSPICallAppearance , CTSPIAddressInfo , or the CTSPILineConnection::SetTerminal method depending on the parameters passed.
lineSetupConference	This is called in response to the TSPI_lineSetupConference command. The default implementation validates and copies parameters and calls the

lineSetupTransfer	CTSPIAddressInfo::SetupConference method. This is called in response to the TSPI_lineSetupTransfer command. The default implementation validates and copies parameters and calls the
lineSwapHold	CTSPIAddressInfo::SetupTransfer method. This is called in response to the TSPI_lineSwapHold command. The default implementation validates and copies parameters and calls the
lineUncompleteCall	CTSPICallAppearance::SwapHold method. This is called in response to the TSPI_lineUncompleteCall command. The default implementation validates and copies parameters and calls the
lineUnhold	CTSPILineConnection::UncompleteCall method. This is called in response to the TSPI_lineUnhold command. The default implementation validates and copies parameters and calls the CTSPICallAppearance::Unhold method.
lineUnpark	This is called in response to the TSPI_lineUnpark command. The default implementation validates and copies parameters and calls the CTSPIAddressInfo::Unpark method.
lineDropOnClose	This is called in response to the TSPI_lineDropOnClose command. The default calls the
lineDropNoOwner	CTSPICallAppearance::DropOnClose method. This is called in response to the TSPI_lineDropNoOwner command. The default implementation calls the
lineSetCurrentLocation	CTSPICallAppearance::DropNoOwner method. This is called in response to the TSPI_lineSetCurrentLocation command. The default implementation saves the current location off in the CServiceProvider object and returns FALSE.
lineConfigDialogEdit	This is called in response to the TSPI_lineConfigDialogEdit command. The default implementation calls the
lineReleaseUserUserInfo	CTSPILineConnection::ConfigDialogEdit method. This is called in response to the TSPI_lineReleaseUserUserInfo command. The default implementation validates and copies parameters and calls the
phoneClose	CTSPICallAppearance::ReleaseUserUserInfo method. This is called in response to the TSPI_phoneClose command. The default implementation calls the
phoneConfigDialog	CTSPIPhoneConnection::Close method. This is called in response to the TSPI_phoneConfigDialog command. The default implementation calls the
phoneDevSpecific	CTSPIPhoneConnection::ConfigDialog method. This is called in response to the TSPI_phoneDevSpecific command. The default implementation calls the
phoneGetButtonInfo	CTSPIPhoneConnection::DevSpecific method. This is called in response to the TSPI_phoneGetButtonInfo command. The default implementation copies the parameters and calls the
phoneGetData	CTSPIPhoneConnection::GetButtonInfo method. This is called in response to the TSPI_phoneGetData command. The default implementation calls the
phoneGetDevCaps	CTSPIPhoneConnection::GetData method. This is called in response to the TSPI_phoneGetDevCaps command. The default implementation calls the
phoneGetDisplay	CTSPIPhoneConnection::GatherCapabilities method. This is called in response to the TSPI_phoneGetDisplay

phoneGetExtensionID	command. The default implementation calls the CTSPiPhoneConnection::GetDisplay method. This is called in response to the TSPI_phoneGetExtensionID command. The default implementation fills all the fields in with zeros and returns FALSE.
phoneGetGain	This is called in response to the TSPI_phoneGetGain command. The default implementation calls the CTSPiPhoneConnection::GetGain method.
phoneGetHookswitch	This is called in response to the TSPI_phoneGetHookswitch command. The default implementation calls the CTSPiPhoneConnection::GetHookswitch method.
phoneGetIcon	This is called in response to the TSPI_phoneGetIcon command. The default implementation calls the CTSPiPhoneConnection::GetIcon method.
phoneGetID	This is called in response to the TSPI_phoneGetID command. The default implementation calls the CTSPiPhoneConnection::GetID method.
phoneGetLamp	This is called in response to the TSPI_phoneGetLamp command. The default implementation calls the CTSPiPhoneConnection::GetLamp method.
phoneGetRing	This is called in response to the TSPI_phoneGetRing command. The default implementation calls the CTSPiPhoneConnection::GetRing method.
phoneGetStatus	This is called in response to the TSPI_phoneGetStatus command. The default implementation calls the CTSPiPhoneConnection::GetData method.
phoneGetVolume	This is called in response to the TSPI_phoneGetVolume command. The default implementation calls the CTSPiPhoneConnection::GetVolume method.
phoneNegotiateExtVersion	This is called in response to the TSPI_phoneNegotiateExtVersion command. The default implementation returns PHONEERR_OPERATIONUNAVAIL .
phoneNegotiateTSPIVersion	This is called in response to the TSPI_phoneNegotiateTSPIVersion command. The default implementation negotiates a valid TSP version based on what the provider says it can support and TAPI 1.3.
phoneOpen	This is called in response to the TSPI_phoneOpen command. The default implementation calls the CTSPiPhoneConnection::Open method.
phoneSelectExtVersion	This is called in response to the TSPI_phoneSelectExtVersion command. The default implementation returns PHONEERR_OPERATIONUNAVAIL .
phoneSetButtonInfo	This is called in response to the TSPI_phoneSetButtonInfo command. The default implementation copies the parameters and calls the CTSPiPhoneConnection::SetButtonInfo method.
phoneSetData	This is called in response to the TSPI_phoneSetData command. The default implementation copies the parameters and calls the CTSPiPhoneConnection::SetData method.
phoneSetDisplay	This is called in response to the TSPI_phoneSetDisplay command. The default implementation copies the parameters and calls the CTSPiPhoneConnection::SetDisplay method.
phoneSetGain	This is called in response to the TSPI_phoneSetGain command. The default implementation copies the parameters

phoneSetHookSwitch

and calls the **CTSPiPhoneConnection::SetGain** method. This is called in response to the TSPI_phoneSetHookSwitch command. The default implementation copies the parameters and calls the **CTSPiPhoneConnection::SetHookswitch** method.

phoneSetLamp

This is called in response to the TSPI_phoneSetLamp command. The default implementation copies the parameters and calls the **CTSPiPhoneConnection::SetLamp** method.

phoneSetRing

This is called in response to the TSPI_phoneSetRing command. The default implementation copies the parameters and calls the **CTSPiPhoneConnection::SetRing** method.

phoneSetStatusMessages

This is called in response to the TSPI_phoneSetStatusMessages command. The default implementation calls the **CTSPiPhoneConnection::SetStatusMessages** method.

phoneSetVolume

This is called in response to the TSPI_phoneSetVolume command. The default implementation copies the parameters and calls the **CTSPiPhoneConnection::SetVolume** method.

CServiceProvider::CancelRequest

Protected

```
virtual void CancelRequest (CTSPIRequest* pReq);
```

pReq Request packet which is being cancelled.

Remarks

This method is called by the device object when a request is canceled and it has already been started on the device. It gives the derived provider an opportunity to examine the request and do some post-processing before it is deleted.

CServiceProvider::CanHandleRequest

Protected

```
virtual BOOL CanHandleRequest (CTSPIConnection* pConn,  
                               CTSPIAddressInfo* pAddr, CTSPICallAppearance* pCall,  
                               WORD wRequest, DWORD dwData = 0);
```

pConn Connection which the request will be started on.

pAddr Address object the request will be running on (may be NULL).

pCall Call appearance the request will be running on (may be NULL).

wRequest Request type to be started

dwData Data associated with request.

Remarks

This method is used to determine if an asynchronous request packet should be generated for each available event type (**wRequest**). By default, the request will be generated if the function is exported by the DLL. This is invoked by each of the TAPI handlers in all the objects to determine at the most basic level if the function is available.

Return Value

TRUE if the service provider supports the request type

FALSE if the request cannot be processed and should return an error.

CServiceProvider::CheckCallFeatures

Protected

```
virtual DWORD CheckCallFeatures(CTSPICallAppearance* pCall,
                                DWORD dwFeatures);
```

pCall Call which has changed.

dwFeatures Address object the request will be running on (may be NULL).

Remarks

This method is called by the call appearance object each time the call features are changed. If the hardware being modeled has restrictions as to the feature list that are not standard TAPI specifications, then this may be overridden to adjust the features of the call as it changes state.

By default the library simply returns the passed call features.

Return Value

Returning call features.

CServiceProvider::CheckDialableNumber

```
virtual LONG CheckDialableNumber(CTSPILineConnection* pLine,
                                  CTSPIDialableAddress* pAddr, LPCTSTR lpszDigits, COBJDATA* parrEntries,
                                  DWORD dwCountry, LPCTSTR pszValidChars = NULL);
```

pLine Line object the dial string is associated with

pAddr Address object the dial string is associated with

lpszDigits Digits to dial

parrEntries Returning array of **DIALINFO** structures

dwCountry Country code to use with dialing code.

pszValidChars Valid list of characters in a dial string. NULL indicates default TAPI restrictions.

Remarks

This method checks the passed dialable number to determine if we support the dialable address. The final form address is returned in the as a series of **DIALINFO** structures (at least one) for each valid address found in the dialable string.

The valid characters for a dial string are: **0123456789ABCD*#IWPT@\$+,.**

If the hardware has additional valid characters, then the final parameter needs to include the above list along with the additional characters to pass through the dial string.

Any character found in the dial string which is not in the valid list is stripped from the resultant **DIALINFO** structure.

For more information on dial strings, see the section on *Dialing Information* or the section on *Structures: DIALINFO*.

Return Value

TAPI result code or FALSE for success.

CServiceProvider::CloseDevice

virtual BOOL CloseDevice (CTSPICConnection* pConn);

pConn Line or phone device which invoked the final close.

Remarks

This method is called by the **CTSPIDevice** object when the device is closed. In the 16-bit library, this will generate a **COMMAND_CLOSECONN** request to the companion application. In the 32-bit library, this function does nothing.

Return Value

TRUE if the device was closed successfully.

FALSE if the close failed.

CServiceProvider::ConvertDialableToCanonical

**virtual CString ConvertDialableToCanonical (LPCTSTR pszNumber,
DWORD dwCountry=0)**

<i>pszNumber</i>	Dialable number.
------------------	------------------

dwCountry Country code to use.

Remarks

This method converts a dialable number to a standardized canonical format. The standard canonical format is: **+1 (800) 555-1212**. This is the format all phone numbers are reported back to TAPI in (for things such as caller id).

For more information on dial strings, see the section on *Dialing Information* or the section on *Structures: DIALINFO*.

Return Value

String with number formatted in standard canonical format.

CServiceProvider::CServiceProvider

16-bit

```
CServiceProvider (LPCSTR pszAppName, LPCSTR pszExeName,  
                  DWORD dwTapiVer = TAPIVER_13, LPCSTR pszProviderInfo = NULL);
```

32-bit

```
CServiceProvider(LPCTSTR pszAppName, LPCTSTR pszProviderInfo = NULL,  
                  DWORD dwTapiVer = TAPIVER_20);
```

pszAppName A null-terminated string that contains the name of the user-interface DLL that TAPI should load for all user-interface events. This is generally the name of the TSP itself including the ".TSP" extension.

pszExeName A null-terminated string that contains the name of the companion application to start. If this is NULL, then no companion application will be started.

pszProviderInfo A null-terminated string with the copyright string for the provider which is reported as part of the provider information and capabilities structure.

dwTapiVer This is the minimum negotiable version of TAPI to allow the provider to run under. This will allow TAPI to determine if the provider is too new to run on the machine. Under TAPI 2.x, this value must be at least **TAPIVER_20**.

Remarks

Constructs a **CServiceProvider** object. Only one object should be created within a service provider, and it should be a global variable within the provider source code.

CServiceProvider::DeleteProfile

```
BOOL DeleteProfile (DWORD dwPPid);
```

dwPPid Provider ID assigned by TAPI to the device storing data.

Remarks

This function destroys the persistent data associated with the service provider and device id. In a 16-bit service provider, this section will be inside the TELEPHON.INI file. In a 32-bit provider, it will be inside the registry. The provider ID is used to distinguish between multiple devices within the provider. Since TAPI guarantees that they will be unique system-wide, they are used as part of the key to store the data.

This function is automatically called when the provider is de-installed using **TSPI_providerRemove**.

Return Value

TRUE if the section was removed.

CServiceProvider::GetConnInfoFromLineDeviceID

```
CTSPILineConnection* GetConnInfoFromLineDeviceID(DWORD dwDevId);
```

dwDevId TAPI Device ID to search for.

Remarks

This function searches the line connection array for a **CTSPILineConnection** object which matches the passed device identifier. If no object is found with the matching ID, NULL is returned.

Return Value

Line object which is associated to the passed TAPI line device id. If no object is found this function returns NULL.

CServiceProvider::GetConnInfoFromPhoneDeviceID

```
CTSPIPhoneConnection* GetConnInfoFromPhoneDeviceID(DWORD dwDevId);
```

dwDevId TAPI Device ID to search for.

Remarks

This function searches the phone connection array for a **CTSPIPhoneConnection** object which matches the passed device identifier. If no object is found with the matching ID, NULL is returned.

Return Value

Phone object which is associated to the passed TAPI phone device id. If no object is found this function returns NULL.

CServiceProvider::GetCurrentLocation

DWORD GetCurrentLocation() const;

Remarks

This function simply returns the current location information which is settable by TAPI.

Return Value

The current TAPI location identifier.

CServiceProvider::GetDevice

CTSPIDevice* GetDevice(DWORD dwProviderID) const;

dwProviderID Provider ID to locate. This is a sequential index from within TAPI which might not start with zero.

Remarks

This function searches the device array for a **CTSPIDevice** object which matches the passed provider identifier. If no object is found with the matching ID, NULL is returned.

TAPI assigns provider identifiers in a sequential fashion, starting at zero and moving through each service provider installed in the system. Each device in the library will be assigned a unique identifier. Since devices can be removed from within TAPI, the provider identifiers might not be complete (i.e. they can have holes in the numeric sequence). For this reason, do not assume any order or sequence to the identifiers.

If you want to locate a device based on a zero-based index within the provider, use the function **CServiceProvider::GetDeviceByIndex**.

Return Value

The device object which is associated to the passed TAPI provider id. If no object is found this function returns NULL.

CServiceProvider::GetDeviceByIndex

CTSPIDevice* GetDeviceByIndex(int iDevice) const;

iDevice Zero-based index to locate. This is a sequential index from within the service provider.

Remarks

This function returns the device at the specified position within the device array. If there is no device at that position, NULL is returned.

If you want to locate a device based on a provider id from TAPI, use the function **CServiceProvider::GetDevice**.

Return Value

Device object which is associated to the passed index. If no object is found this function returns NULL.

CServiceProvider::GetDeviceCount

DWORD GetDeviceCount() const;

Remarks

This function returns the count of devices present in the service provider. There should always be at least once device as long as the provider has been completely initialized.

Return Value

Count of devices installed in TAPI which are being run out of this service provider.

CServiceProvider::GetDialableNumber

**virtual CString GetDialableNumber (LPCTSTR pszNumber,
LPCTSTR pszAllowChar = NULL) const;**

pszNumber Any format phone number.

pszAllowChar List of allowable characters.

Remarks

This method strips information out of a phone number which is not required to dial the number. This includes things such as area-code braces, pause codes, wait-for-dialtone requests, etc.

The allowable character set allows the caller to determine what is left when the number is returned.

Return Value

String with number formatted in standard dialable format.

CServiceProvider::GetProviderInfo

LPCTSTR GetProviderInfo() const;

Remarks

This function returns the information about the provider, which was supplied, on the constructor.

Return Value

NULL terminated pointer to the constant string which was passed to the constructor of the provider.

CServiceProvider::GetProviderName

LPCTSTR GetProviderName() const;

Remarks

This function returns the information about the provider name, which was supplied, on the constructor.

Return Value

NULL terminated pointer to the constant string which was passed to the constructor of the provider.

CServiceProvider::GetSupportedVersion

DWORD GetSupportedVersion() const;

Remarks

This function returns the version of TAPI which the service provider is willing to negotiate to. This will be the highest level of TAPI which is supported by this provider.

Return Value

The TAPI version which was supplied to the constructor of the object

CServiceProvider::GetSystemVersion

DWORD GetSystemVersion() const;

Remarks

This function returns the level of TAPI which is installed on the PC.

Return Value

Version of TAPI found on this PC.

CServiceProvider::MatchTones

virtual **BOOL MatchTones** (DWORD dwSFreq1, DWORD dwSFreq2,
 DWORD dwSFreq3, DWORD dwTFreq1,
 DWORD dwTFreq2, DWORD dwTFreq3);

<i>dwSFreq1</i>	Search Frequency tone number 1
<i>dwSFreq2</i>	Search Frequency tone number 2
<i>dwSFreq3</i>	Search Frequency tone number 3
<i>dwTFreq1</i>	Target Frequency tone number 1
<i>dwTFreq2</i>	Target Frequency tone number 2
<i>dwTFreq3</i>	Target Frequency tone number 3

Remarks

This method compares a set of tones against each other. The search frequency is what the provider is looking for, the target being the frequency that the hardware detected. The default implementation simply compares the three together. This most likely will not be adequate for

most providers and therefore this method should be overridden to support full tone frequency matching.

This method is called when tone monitoring is turned on for specific tones through **lineMonitorTones**.

Return Value

TRUE if the frequencies match, or FALSE if they do not.

CServiceProvider::OnNewRequest

Protected

```
virtual BOOL OnNewRequest (CTSPICConnection* pConn, CTSPIDRequest* pReq,  
                           int* piPos);
```

<i>pConn</i>	Connection which the request is starting on.
<i>pReq</i>	Request object which is about to be added to the list.
<i>piPos</i>	Position which the request is about to be inserted at.

Remarks

This method is called by the device object each time a new request packet is added to the request list. It is called before the request is officially added to the list and allows the derived provider to manipulate the list before insertion.

If the **piPos** value is modified, it may be set to zero for the head of the queue, or (-1) for the end of the queue, or any number from zero to the total number of queued requests on the given connection.

Return Value

TRUE if the connection object should continue to add the request.
FALSE if the request is to be canceled.

CServiceProvider::OnTimer

```
virtual void CServiceProvider::OnTimer(CTSPICConnection* pConn);
```

<i>pConn</i>	Line/Phone device to process interval timer on.
--------------	---

Remarks

This method is called by the device object to process an interval timer. The timer is generated by the companion application in the 16-bit library and by a dedicated thread in the 32-bit library.

This method by default does nothing. It should be overridden if the provider requires interval timer support.

CServiceProvider::OpenDevice

virtual BOOL OpenDevice (CTSPConnection* pConn);

pConn Connection which is being opened.

Remarks

This method is called by the **CTSPIDevice** object when the device is opened. In the 16-bit library, this will generate a **COMMAND_OPENCONN** request to the companion application. In the 32-bit library, this function does nothing.

Return Value

TRUE if the device was opened successfully. FALSE if the open failed.

CServiceProvider::ProcessCallParameters

virtual LONG ProcessCallParameters(CTSPILineConnection* pLine,
LPLINECALLPARAMS lpCallParams);

pLine Line object which owns the call.

lpCallParams **LINECALLPARAMS** structure from TAPI.

Remarks

This method is called whenever a **LINECALLPARAMS** structure is passed through the provider. It validates the structure against the line to ensure that the values for media, data rate, and dialing information are all within allowed values.

It may be overridden by derived providers to provide specialized handling of the **LINECALLPARAMS** structure.

Return Value

TAPI result code or FALSE if the structure was valid for the provider.

CServiceProvider::ProcessData

Protected

```
virtual BOOL ProcessData(CTSPICConnection* pConn, DWORD dwData = 0,  
    const LPVOID lpBuff = NULL, DWORD dwSize = 0);
```

<i>pConn</i>	Connection which is associated with data (NULL for none)
<i>dwData</i>	DWORD data passed from companion application or input thread.
<i>lpBuff</i>	Pointer to data block from companion application or input thread.(NULL for none)
<i>dwSize</i>	Size of the data block.

Remarks

The line and phone connection objects invoke this method when data is received from either the companion application or some input thread in the 32-bit library. It is responsible for processing the data and running all pending asynchronous requests.

This method may be overridden to see all data for all lines and phones. Another option is to override the version in the line and phone objects and process the line-specific requests and phone-specific requests there.

The **CTSPIDevice** object enumerates through all line and phone objects when data is received and calls the **CTSPICConnection::ReceiveData** function to process the input. The object will stop enumerating lines and phones if the **ReceiveData** function returns **TRUE**. Otherwise, the device will locate the next line or phone and pass it through to that until there are no more lines and phones.

Return Value

TRUE if the request was processed and the device should stop looking for an owner.

FALSE if the request was not processed or the device object should continue routine the request to other connection objects (line and phone).

CServiceProvider::ReadProfileDWord

```
DWORD ReadProfileDWord (DWORD dwPPid, LPCTSTR pszEntry,  
    DWORD dwDefault = 0);
```

<i>dwPPid</i>	Provider ID assigned by TAPI to the device storing data.
<i>pszEntry</i>	Key to read from.
<i>dwDefault</i>	Default number to return if not found.

Remarks

This function reads a numeric value from the storage section devoted to the service provider. In a 16-bit service provider, this section will be inside the TELEPHON.INI file. In a 32-bit provider, it will be inside the registry. The provider ID is used to distinguish between multiple devices within

the provider. Since TAPI guarantees that they will be unique system-wide, they are used as part of the key to store the data.

Any data stored using this API will be automatically removed when the provider is de-installed using **TSPI_providerRemove**.

Return Value

DWORD read from persistent storage or default value if not found.

CServiceProvider::ReadProfileString

```
CString ReadProfileString (DWORD dwPPid, LPCTSTR pszEntry,  
                           LPCTSTR pszDefault = "");
```

<i>dwPPid</i>	Provider ID assigned by TAPI to the device storing data.
<i>pszEntry</i>	Key to read from.
<i>pszDefault</i>	Default string to return if not found.

Remarks

This function reads a string from the storage section devoted to the service provider. In a 16-bit service provider, this section will be inside the TELEPHON.INI file. In a 32-bit provider, it will be inside the registry. The provider ID is used to distinguish between multiple devices within the provider. Since TAPI guarantees that they will be unique system-wide, they are used as part of the key to store the data.

Any data stored using this API will be automatically removed when the provider is de-installed using **TSPI_providerRemove**.

Return Value

String read from persistent storage or default value if not found.

CServiceProvider::RenameProfile

```
BOOL ReadProfileString (DWORD dwPPid, DWORD dwNewPPid);
```

<i>dwPPid</i>	Provider ID assigned by TAPI to the device storing data.
<i>dwNewPPid</i>	New Provider ID to move previous information into.

Remarks

This function copies all existing profile information stored under the listed provider id into a new section based on the new id. In a 16-bit service provider, this section will be inside the TELEPHON.INI file. In a 32-bit provider, it will be inside the registry. The provider ID is used to distinguish between multiple devices within the provider. Since TAPI guarantees that they will be unique system-wide, they are used as part of the key to store the data.

This function should be used if the provider supports dynamic creation of devices and therefore needs to move profile information around.

Any data stored using this API will be automatically removed when the provider is de-installed using **TSPI_providerRemove**.

Return Value

TRUE if the profile was renamed successfully. FALSE if the rename failed.

CServiceProvider::SendData

```
virtual BOOL SendData(CTSPICConnection* pConn, LPCVOID lpBuff,  
                     DWORD dwSize);
```

pConn Line or phone object this was generated by.

lpBuff Buffer to send to the device.

dwSize Size of the above buffer.

Remarks

This method is called by the **CTSPIDevice** object when data is sent from a line or phone object. In the 16-bit library it generates a **COMMAND_SENDDATA** request to the companion application and thunks the buffer.

In the 32-bit library, this function does nothing.

Return Value

TRUE if the data was sent to the companion application. FALSE if the **SendThreadRequest** function failed.

CServiceProvider::SendThreadRequest

16-bit only

```
BOOL SendThreadRequest (WORD wCommand, LPARAM lParam,  
                       BOOL flParamsIsPointer = FALSE, DWORD dwSize = 0);
```

wCommand Command to pass through to the companion application

lParam Optional parameter to pass to the companion application.

flParamsIsPointer TRUE if **lParam** is a pointer to a block of memory.

dwSize Size of the memory block if **lParam** is a pointer.

Remarks

This is the function 16-bit service providers use to communicate with the companion application. The **wCommand** parameter comes from the *spuser.h* header file. For more information on this command, see the section on *Companion Applications*.

Return Value

TRUE if the function was successful in notifying the companion application or FALSE if it failed.

CServiceProvider::SetRuntimeObjects

Protected

```
void SetRuntimeObjects(CRuntimeClass* pDevObject,  
                      CRuntimeClass* pReqObject = NULL,  
                      CRuntimeClass* pLineObject = NULL,  
                      CRuntimeClass* pAddrObject = NULL,  
                      CRuntimeClass* pCallObject = NULL,  
                      CRuntimeClass* pConfObject = NULL,  
                      CRuntimeClass* pPhoneObject = NULL);
```

pDevObject This is the class object which should be used in place of the standard device object. It must be derived from **CTSPIDevice**.

pReqObject This is the class object which should be used in place of the standard asynchronous request object. It must be derived from **CTSPIRequest**.

pLineObject This is the class object which should be used in place of the standard line device object. It must be derived from **CTSPILineConnection**.

pAddrObject This is the class object which should be used in place of the standard address object. It must be derived from **CTSPIAddressInfo**.

pCallObject This is the class object which should be used in place of the standard call appearance object. It must be derived from **CTSPICallAppearance**.

pConfObject This is the class object which should be used in place of the standard conference call object. It must be derived from **CTSPIConferenceCall**.

pPhoneObject This is the class object which should be used in place of the standard phone device object. It must be derived from **CTSPIPhoneConnection**.

Remarks

This function allows the derived provider to replace or supplement the existing functionality within each object type with derived objects. This function should be called within the constructor of the **CServiceProvider** object to override each desired class. If the function passed NULL in for any parameter, the default object type is used.

Note that this function should only be called from within the constructor of the service provider object. If you attempt to call the function at any other time, unpredictable results may occur.

CServiceProvider::SetTimerResolution

32-bit only

```
void SetTimerResolution(LONG IRes);
```

IRes Resolution of the interval timer in milliseconds. If zero is supplied, the interval timer is disabled.

Remarks

This function adjusts the resolution of the interval timer thread which periodically runs through each device in the provider and calls the **CTSPIDevice::ProcessIntervalTimer** method.

Note that the timer resolution should only be adjusted in the constructor of the service provider object. If you attempt to call the function at any other time, unpredictable results may occur. This is especially true if you attempt to disable the timer after it has been started.

CServiceProvider::SetTimeout

Protected

```
void SetTimeout(LONG ITimeout);
```

ITimeout Resolution of the wait timer used for asynchronous requests to complete.

Remarks

This function adjusts the resolution of the wait timer that is used to wait on pending requests in the asynchronous request queue. It generally does not need to be changed, and is only documented for providers that need specialized control over the processing of the asynchronous request queue.

CServiceProvider::StartNextCommand

```
virtual void StartNextCommand(CTSPICConnection* pConn);
```

pConn Line or phone object this was generated by.

Remarks

This method begins the processing of the next asynchronous request block on our list. It gets called when a request finishes, or when a new request is inserted and no pending requests are being processed.

The default behavior is to switch to the companion application context in the 16-bit library (thereby entering **ProcessData**), or simply calling **ProcessData** in the 32-bit library.

CServiceProvider::UnknownDeviceNotify

16-bit only

```
virtual void UnknownDeviceNotify (WORD wCommand, DWORD dwConnId,  
                                  DWORD dwResult, LPVOID lpvData, DWORD dwSize);
```

<i>wCommand</i>	Command from companion application (RESULT_xxx)
<i>dwConnId</i>	Connection ID which identifies line/phone this request is for.
<i>dwResult</i>	Data response from companion application
<i>lpvData</i>	Pointer to any data from companion application (may be NULL).
<i>dwSize</i>	Size of data pointed to by lpvData .

Remarks

This method is invoked by the **DeviceNotify** handler when it doesn't recognize the command passed from the companion application. When the command isn't one of the defined responses in *spuser.h*, it is considered an *Unknown* packet and is routed to this method for processing.

For more information on this, see the section on *Companion Applications*.

The default class implementation does nothing.

CServiceProvider::WriteProfileDWord

BOOL WriteProfileDWord (DWORD dwPPid, LPCSTR pszEntry, DWORD dwValue);

<i>dwPPid</i>	Provider ID assigned by TAPI to the device storing data.
<i>pszEntry</i>	Key to write to.
<i>dwValue</i>	Numeric value to write to the key.

Remarks

This function writes a numeric value into the storage section devoted to the service provider. In a 16-bit service provider, this section will be inside the TELEPHON.INI file. In a 32-bit provider, it will be inside the registry. The provider ID is used to distinguish between multiple devices within the provider. Since TAPI guarantees that they will be unique system-wide, they are used as part of the key to store the data.

Any data stored using this API will be automatically removed when the provider is de-installed using **TSPI_providerRemove**.

Return Value

TRUE if the value was stored successfully.

CServiceProvider::WriteProfileString

BOOL WriteProfileString (DWORD dwPPid, LPCSTR pszEntry, LPCSTR pszValue);

<i>dwPPid</i>	Provider ID assigned by TAPI to the device storing data.
<i>pszEntry</i>	Key to write to.
<i>pszValue</i>	String value to write to the entry.

Remarks

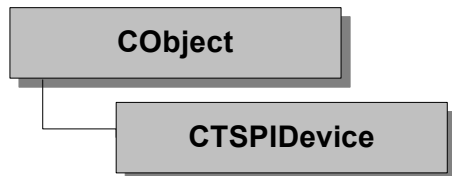
This function writes a string into the storage section devoted to the service provider. In a 16-bit service provider, this section will be inside the TELEPHON.INI file. In a 32-bit provider, it will be inside the registry. The provider ID is used to distinguish between multiple devices within the provider. Since TAPI guarantees that they will be unique system-wide, they are used as part of the key to store the data.

Any data stored using this API will be automatically removed when the provider is de-installed using **TSPI_providerRemove**.

Return Value

TRUE if the value was stored successfully.

CTSPIDevice



The **CTSPIDevice** object maintains the information specific to a particular TAPI device. Each device installed into TAPI will be assigned a unique 32-bit identifier called a *Provider ID*. This identifies the device *and* provider to TAPI. Each provider device may then manage connections to the physical telephony network. The device is represented in the TSP++ library as a **CTSPIDevice** object. In general there will only be one **CTSPIDevice** in a service provider, but this is not a rule, multiple devices can be supported through a single provider in TAPI.

16-bit note

If multiple physical devices are to be supported by the provider simultaneously, then the companion application needs to be able to route requests by using the device connection id that gets passed for each request or command.

Device Initialization

When the device object is first constructed by the **CServiceProvider::providerInit** function, it creates line and phone device objects to control the lines and phones that are present on the device. The number of objects created is passed to the device constructor and is supplied by TAPI using the **NumLines** and **NumPhones** fields in the **TELEPHON.INI** (16-bit), or the **CServiceProvider::providerEnumDevices** method (32-bit).

Lines and Phones

The device object is responsible for maintaining the list of line and phone devices that are present on this physical connection to the network. The device object creates each line and phone object during the initialization process. The number of lines and phones present is determined either by the service provider or by TAPI during initialization and is generally static during the life of the provider. If lines and phones are dynamic, then the provider must support the Plug & Play features to notify about line and phone additions. This support is provided through the **CreateLine** and **CreatePhone** methods of the device object. Removal of a line or phone is performed through the **RemoveLine** and **RemovePhone** methods that are available in the 32-bit library. Line and phone removal in 16-bit TAPI is not dynamic and requires that TAPI be reinitialized.

Device Open/Close events

The device object may be used to control access to the physical telephony device if desired. When any line or phone connection is opened by TAPI, the **OpenDevice** method will be invoked on the device. This may occur more than once while the TSP is open, so if only one physical connection is established, reference counting must be performed. When a line or phone is closed, the **CloseDevice** method is called. By overriding these two methods, all open/close events for all lines and phones for a single device may be handled in one place. If the functions are not overridden, then they are forwarded automatically to the **CServiceProvider** object (see **CServiceProvider::OpenDevice** and **CServiceProvider::CloseDevice**).

Operations - Public Methods

CreateLine	Dynamically add a new line device to TAPI. This requires that TAPI v1.4 or above is running on the system.
CreatePhone	Dynamically add a new phone device to TAPI. This requires that TAPI v1.4 or above is running on the system.
FindLineConnectionByDeviceID	Return the line object based on TAPI line device id.
FindPhoneConnectionByDeviceID	Return the phone object based on TAPI phone device id.
GetLineCount	Return the total number of line devices.
GetLineConnectionInfo	Return the line object associated with an index.
GetPermanentDeviceID	Returns the device ID assigned by the class library to represent this device. This represents the position within the class library device list.
GetPhoneCount	Return the total number of phone devices.
GetPhoneConnectionInfo	Return the phone object associated with an index.
GetProviderID	Return the TAPI assigned provider id for this device.
GetProviderHandle	Returns the provider handle assigned by TAPI (32-bit only).
RemoveLine	Remove an existing line from the device (32-bit only).
RemovePhone	Remove an existing phone from the device (32-bit only).

Overridables - Public Members

CloseDevice	Called by the CTSPICConnection::CloseDevice method.
GenericDialogData	Processes dialog user-interface events.
OpenDevice	Called by the CTSPICConnection::OpenDevice method. Default implementation passes control to CServiceProvider::OpenDevice .
ReceiveData	This is called by DeviceNotify when data is received from the companion application. Default implementation calls the appropriate connection ReceiveData method if a specific connection/device id is specified, or goes through <i>all</i> lines and phones if only a provider id is given from the companion application.
SendData	Called by the CTSPICConnection::SendData method.

Overridables - Protected Members

Init	Initialization method called after the constructor.
OnAsynchRequestComplete	Called when any asynchronous request completes. Default implementation notifies TAPI through the asynchronous callback.
OnCancelRequest	Called by the CTSPICConnection object when a request is being canceled.
OnNewRequest	Called by the CTSPICConnection object when a new request is created and being added to the device list.
OnTimer	Called by the CServiceProvider::DistributeIntervalTimer method. Default implementation gives a periodic timer to all connections on the device.

CTSPIDevice::CloseDevice

```
virtual BOOL CloseDevice (CTSPIConnection* pConn);
```

pConn Line or phone connection which is requesting close.

Remarks

This method is called from the **CTSPILineConnection** and **CTSPIPhoneConnection** objects to close the physical device.

If the function is not overridden, it will forward the request onto **CServiceProvider::CloseDevice**.

Return Value

TRUE/FALSE success indicator.

CTSPIDevice::CreateLine

```
int CreateLine();
```

Remarks

This method dynamically adds a line to the device array. It is only available if the TSP is running under TAPI 1.4 or better and the provider exports the **TSPI_providerEnumDevices** method to allow the class library to get a pointer to the line creation callback.

As soon as this function returns, the line is available from within the service provider. TAPI will notify all running applications that a new line has been created, and eventually will assign the new line a permanent line identifier. TAPI will notify the provider about the new line identifier through a callback which indicates that the line is fully ready for use by applications. This notification is managed within the library automatically.

Return Value

The return value is the index in the line device array where the newly allocated line was placed. This position will not change during the life of the provider. If the line could not be added, the function returns (-1).

CTSPIDevice::CreatePhone

```
int CreatePhone();
```

Remarks

This method dynamically adds a telephone to the device array. It is only available if the TSP is running under TAPI 1.4 or better and the provider exports the **TSPI_providerEnumDevices** method to allow the class library to get a pointer to the phone creation callback.

As soon as this function returns, the phone is available from within the service provider. TAPI will notify all running applications that a new phone has been created, and eventually will assign the new phone a permanent phone identifier. TAPI will notify the provider about the new phone identifier through a callback which indicates that the phone is fully ready for use by applications. This notification is managed within the library automatically.

Return Value

The return value is the index in the phone device array where the newly allocated phone was placed. This position will not change during the life of the provider. If the phone could not be added, the function returns (-1).

CTSPIDevice::FindLineConnectionByDeviceID

```
CTSPILineConnection* FindLineConnectionByDeviceID(  
    DWORD dwDeviceID) const;
```

dwDeviceID TAPI-assigned device identifier for line device.

Remarks

This method runs through the line device array and searches for the **CTSPILineConnection** which is matched to the specified TAPI device id.

Each line and phone device is assigned a unique index within the TAPI system, based on the order in which it was added to TAPI. The number is not guaranteed to be sequential within the provider (although generally will be).

Return Value

The line connection object which was found to match the device id or NULL if it could not be found.

CTSPIDevice::FindPhoneConnectionByDeviceID

```
CTSPIPhoneConnection* FindPhoneConnectionByDeviceID(  
    DWORD dwDeviceID) const;
```

dwDeviceID TAPI-assigned device identifier for phone device.

Remarks

This method runs through the phone device array and searches for the **CTSPIPhoneConnection** which is matched to the specified TAPI device id.

Each line and phone device is assigned a unique index within the TAPI system, based on the order in which it was added to TAPI. The number is not guaranteed to be sequential within the provider (although generally will be).

Return Value

The phone connection object which was found to match the device id or NULL if it could not be found.

CTSPIDevice::GenericDialogData

32-bit only

virtual LONG GenericDialogData (LPVOID lpParam, DWORD dwSize);

<i>lpParam</i>	Parameter from the UI dialog.
<i>dwSize</i>	Size of the passed parameter block.

Remarks

This method is called when the user-interface component of the TSP is sending data back to the service provider, and the object type specified in the **TSPi_providerGenericDialogData** function was set to **TUISPIDLL_OBJECT_PROVIDERID**.

Return Value

Standard TAPI return code.
FALSE indicates success.

CTSPIDevice::GetLineCount

int GetLineCount() const;

Remarks

This method returns the number of lines which are in the internal provider line array. This is indicative of the number of lines initially assigned to the provider, along with any dynamically added lines while the provider has been running.

Return Value

Count of lines present in the provider. This will always be one more than the highest retrievable index using **GetLineConnectionInfo**.

Returns zero if no phones are available.

CTSPIDevice::GetLineConnectionInfo

CTSPIDevice::GetLineConnectionInfo(int nIndex) const;

nIndex Zero-based index of the line to retrieve. This should not exceed the value returned by **GetLineCount**.

Remarks

This method returns **CTSPIDevice** object that is at the specified array position.

Return Value

Line object which is at the specified array position or NULL if no line is available at that position.

CTSPIDevice::GetPermanentDeviceID

DWORD GetPermanentDeviceID() const;

Remarks

This method returns a unique device identifier within this provider which is used to create unique line/phone indexes to pass through to the companion application.

This identifier is created using the *permanent provider identifier* which is assigned by TAPI to the device during **TSPI_providerInstall**. The permanent provider id is converted to a 16-bit value and placed in the high-order word of the returning device id. The low-order word is assigned by each of the line or phone devices to further identify an object owned by this device.

The return value from this function can be used to mask out lines and phone devices when multiple **CTSPIDevice** objects are supported within a single provider.

Return Value

Permanent provider identifier for this device.

CTSPIDevice::GetPhoneCount

int GetPhoneCount() const;

Remarks

This method returns the number of phones which are in the internal provider phone array. This is indicative of the number of phones initially assigned to the provider, along with any dynamically added phones while the provider has been running.

Return Value

Count of phones present in the provider. This will always be one more than the highest retrievable index using **GetPhoneConnectionInfo**.

Returns zero if no phones are available.

CTSPIDevice::GetPhoneConnectionInfo

CTSPIDevice::GetPhoneConnectionInfo(int nIndex) const;

nIndex Zero-based index of the phone to retrieve. This should not exceed the value returned by **GetPhoneCount**.

Remarks

This method returns **CTSPIDevice::GetPhoneConnectionInfo** object that is at the specified array position.

Return Value

Phone object which is at the specified array position or NULL if no phone is available at that position.

CTSPIDevice::GetProviderID

DWORD GetProviderID() const;

Remarks

This method returns the unique device identifier which was assigned to this provider/device combination. This corresponds to the *permanent provider identifier* which is used in the **ReadProfileXX** and **WriteProfileXX** functions in the **CServiceProvider** object.

This identifier is assigned by TAPI to the device during **TSPI_providerInstall**, and will *always* be the same - until the provider is de-installed.

Since the TSP++ library supports multiple devices within a single provider shell, the provider id is somewhat of a misnomer. It really represents a combination of provider and device within the provider to the library.

Return Value

Permanent provider identifier for this device.

CTSPIDevice::GetProviderHandle*32bit only*

```
HRESULT GetProviderHandle() const;
```

Remarks

This method returns the provider handle assigned by TAPI to this provider/device.

Return Value

Provider handle assigned to the device during **TSPI_providerInit**.

CTSPIDevice::Init

Protected

16-bit version

```
virtual void Init(DWORD dwProviderId, DWORD dwBaseLine, DWORD dwBasePhone,
                 DWORD dwLines, DWORD dwPhones,
                 ASYNC_COMPLETION lpfnCompletion);
```

32-bit version

```
virtual void Init(DWORD dwProviderId, DWORD dwBaseLine, DWORD dwBasePhone,
                 DWORD dwLines, DWORD dwPhones, HPROVIDER hProvider,
                 ASYNC_COMPLETION lpfnCompletion);
```

<i>dwProviderId</i>	Permanent provider identifier assigned by TAPI.
<i>dwBaseLine</i>	Base index of first line to create
<i>dwBasePhone</i>	Base index of first phone to create
<i>dwLines</i>	Number of lines to initialize.
<i>dwPhones</i>	Number of phones to initialize
<i>hProvider</i>	Provider handle assigned by TAPI (32-bit only)
<i>lpfnCompletion</i>	Asynchronous completion callback pointer.

Remarks

This method is called directly after the constructor to initialize the provider/device. It is called during the **TSPI_providerInit** function by the **CServiceProvider** object. It is responsible for setting up the device and creating all the line and phone objects.

If this function is overridden, you *must* call the base class implementation.

CTSPIDevice::OnAsynchRequestComplete

Protected

```
virtual void OnAsynchRequestComplete(LONG IResult = 0L,  
                                     CTSPIDevice* pReq = NULL);
```

<i>IResult</i>	Final return result of the completed request.
<i>pReq</i>	Request object which has completed.

Remarks

This method is called by the **CTSPIDevice** object when any request on the device completes. It gives the device an opportunity to do any post-request cleanup or simply monitor the completions as they occur.

If you override this function, you *must* call the base class implementation.

CTSPIDevice::OnCancelRequest

Protected

```
virtual void OnCancelRequest (CTSPIDevice* pReq);
```

<i>pReq</i>	Request packet which is being cancelled.
-------------	--

Remarks

This method is called by the phone or line object when a request is canceled and it has already been started on the device. It gives the device object an opportunity to examine the request and do some post-processing before it is deleted.

The default behavior is to pass control to the **CServiceProvider::CancelRequest** method.

CTSPIDevice::OnNewRequest

Protected

```
virtual BOOL OnNewRequest (CTSPIDevice* pConn, CTSPIDevice* pReq);
```

<i>pConn</i>	Connection which the request is starting on.
<i>pReq</i>	Request object which is about to be added to the list.

Remarks

This method is called by the connection object each time a new request packet is added to the request list. It is called before the request is officially added to the list and allows the device object to manipulate the list before insertion.

The default behavior is to pass control to the **CServiceProvider::OnNewRequest** method.

Return Value

TRUE if the connection object should continue to add the request.
 FALSE if the request is to be canceled.

CTSPIDevice::OnTimer

Protected

virtual VOID ProcessIntervalTimer();

Remarks

This method is called from either the companion application using the **RESULT_INTERVALTIMER** response, or from the interval timer thread in the 32-bit library.

If the function is not overridden, it will run through all the lines and phones in the object array and pass the timer to each of them.

CTSPIDevice::OpenDevice

virtual BOOL OpenDevice (CTSPIConnection* pConn);

pConn Line or phone connection which is requesting open.

Remarks

This method is called from the **CTSPILineConnection** and **CTSPIPhoneConnection** objects to open the physical device.

If the function is not overridden, it will forward the request onto **CServiceProvider::OpenDevice**.

Return Value

TRUE if the device was opened successfully.

FALSE if the device failed to open, the **TSPI_lineOpen** or **TSPI_phoneOpen** which generated the request will fail and return an error.

CTSPIDevice::ReceiveData

virtual void ReceiveData (DWORD dwConnID, DWORD dwData,
 const LPVOID lpBuff, DWORD dwSize);

<i>dwConnID</i>	Connection ID which data is being routed to. This is a combination of the return value from GetPermanentDeviceID and the line/phone index in the internal array.
<i>dwData</i>	32-bit numeric data response.
<i>lpBuff</i>	Pointer to buffer with received data.
<i>dwSize</i>	Size of the above buffer

Remarks

This method is called from the companion application using a **RESULT_RCVDATA** response. The device object will determine the appropriate line or phone device using the **dwConnID** parameter. The LOWORD of the number is the array index of the line or phone. The method will then invoke the **ReceiveData** method of the object it located.

If the connection id is zero, then the device will route the data request through each line in turn, and then through each phone until one of them responds with a **TRUE** result (indicating that it processed the result) or the device runs out of lines and phones.

CTSPIDevice::RemoveLine

32bit only

```
void RemoveLine(CTSPILineConnection* pLine);
```

pLine Line device object to remove from the system.

Remarks

This method dynamically removes a line from the device array. It is only available if the TSP is running under TAPI 2.0 or better and the provider exports the **TSPI_providerEnumDevices** method to allow the class library to get a pointer to the line creation/removal callback.

The line is not de-allocated or physically removed from the line array until the provider shuts down. This is to keep the same array indexes throughout the life of the library. You should no longer process requests for the line or access this object once this completes.

CTSPIDevice::RemovePhone

32bit only

```
void RemovePhone(CTSPIPhoneConnection* pPhone);
```

pPhone *Phone* device object to remove from the system.

Remarks

This method dynamically removes a phone from the device array. It is only available if the TSP is running under TAPI 2.0 or better and the provider exports the **TSPI_providerEnumDevices** method to allow the class library to get a pointer to the phone creation/removal callback.

The phone is not de-allocated or physically removed from the line array until the provider shuts down. This is to keep the same array indexes throughout the life of the library. You should no longer process requests for the phone or access this object once this completes.

CTSPIDevice::SendData

```
virtual BOOL SendData (CTSPIDevice* pConn, LPCVOID lpBuff,  
                      DWORD dwSize);
```

<i>pConn</i>	Line or phone connection to send data to.
<i>lpBuff</i>	Buffer to send
<i>dwSize</i>	Data size of above buffer.

Remarks

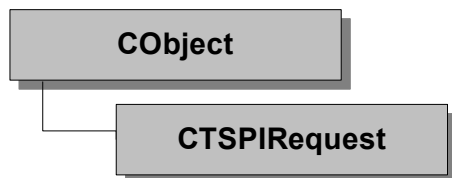
This method is called from the **CTSPIDevice** or **CTSPIDevice** object to send data out to the physical device. It is generally not called by a derived provider, but is documented to allow the overriding of the function to catch all sending of data particular to a device.

If the function is not overridden, it will forward the request onto **CServiceProvider::SendData**.

Return Value

TRUE/FALSE success indicator.

CTSPIRequest



The **CTSPIRequest** object maintains the status of a pending asynchronous TAPI operation.

Asynchronous Requests

The interactive nature of telephony requires that the TSP operate as if it were in a real-time operating environment. Many of the service provider methods are required to complete quickly and return their results to the TAPI dynamic link library synchronously. Other functions (such as dialing or answering) may not be able to complete as quickly and therefore operate asynchronously.

When an operation completes synchronously it performs all of its processing in the function call made by the application - In 16-bit Windows, *this means that the service provider could be executing on the application execution context*. If the function is successful, it will return a zero return code. Otherwise, if there is an error, one of the errors from *tapi.h* will be returned.

When an operation completes asynchronously it performs part of its processing in the function call made by the application and the remainder of it in an independent execution thread after the application has returned from the function call. This thread is our companion application started by the **CServiceProvider** initialization process. A request packet is created and placed into the device queue. Finally, either a negative error result from *tapi.h* or a positive request ID defined by the TAPI dynamic link library is returned.

Once the asynchronous request completes, the service provider calls the TAPI dynamic link library through a callback function and reports that "*Event X has completed*". Passed to this callback is the request ID and a final result code for the operation indicating whether it was successful or not. Again, this will either be zero (for success) or one of the errors from *tapi.h*.

Request Management

Each asynchronous request initiated by the TAPI dynamic link library will cause a request object to be created by the line or phone device that is being targeted, and inserted into the asynchronous request list of the connection object. If there are no pending requests active, then the newly inserted request will be started by calling the **CServiceProvider** method **StartNextCommand**. Otherwise, the request will simply be queued until the provider has completed all previously pending requests. Since the asynchronous list is maintained at the connection level, there generally will not be more than one pending request at a time. Requests to drop a call are considered special requests by the connection object and are always inserted at the front of the queue and started immediately.

As each request completes, the service provider should call the **CTSPIConnection::CompleteRequest** or **CTSPIConnection::CompleteCurrentRequest** methods to inform TAPI that the request is finished and to delete the request from the list. If a request is canceled, or the provider wishes to remove all pending requests based on some criteria, the **RemoveRequest** and **RemovePendingRequests** methods may be used. Specific requests may be located through the **FindRequest** method.

Processing a request

Some requests may require multiple operations to be performed with the telephone network to complete. For instance, when a dial request is received for a modem device, it must first take the modem off-hook, wait for a response, and then send a dial string. To handle this multiple-step processing, each request object has a *request-state* associated with it. Initially, the state starts out as **STATE_INITIAL**, and from that point forward, it is up to the service provider to define and change the state of each request.

For example, in a simple modem driver such as **ATSP**, on **STATE_INITIAL**, it can take the modem device off-hook. The state in the **CTSPiRequest** would then be changed using the **SetState** method, to be state number 2 (defined by the service provider). Control would then exit, and when the modem sends the response, the request packet would again get processed, and the service provider would determine that we left off at state number 2. At this point, the result would be checked for an error, and if it was OK, the service provider would send a dial string and set the state to state number 3. Each request would be handled in this fashion. Once the service provider determines that the modem really did dial, it would complete the request, at which point it would get deleted and removed from the queue, and the service provider would get the *next* request off the list.

An asynchronous request is generated for each of the asynchronous commands which include:

CServiceProvider Method	Request type generated	GetDataPtr parameter	GetData Size parameter
lineAccept	REQUEST_ACCEPT	User information	Size
lineAddToConference	REQUEST_ADDCONF	TSPICONFERENCE	N/A
lineAnswer	REQUEST_ANSWER	User information	Size
lineBlindTransfer	REQUEST_BLINDXFER	Array of DIALINFO objects	Country Code
lineCompleteCall	REQUEST_COMPLETECALL	TSPICOMplete CALL	N/A
lineCompleteTransfer	REQUEST_COMPLETEXFER	TSPITRANSFER	N/A
lineDevSpecific	N/A	N/A	N/A
lineDevSpecificFeature	N/A	N/A	N/A
lineDial	REQUEST_DIAL	Array of DIALINFO objects	Country Code
lineDrop	REQUEST_DROP	User information	Size
lineForward	REQUEST_FORWARD	TSPILINE FORWARD	N/A
lineHold	REQUEST_HOLD	N/A	N/A
lineMakeCall	REQUEST_MAKECALL	TSPIMAKECALL	N/A
linePark	REQUEST_PARK	TSPILINEPARK	N/A
linePickup	REQUEST_PICKUP	TSPILINEPICKUP	N/A
linePrepareAddToConference	REQUEST_PREPAREADDCONF	TSPICONFERENCE	N/A
lineRedirect	REQUEST_REDIRECT	Array of DIALINFO objects	Country Code
lineRemoveFromConference	REQUEST_REMOVEFROMCONF	TSPICONFERENCE	N/A
lineSecureCall	REQUEST_SECURECALL	N/A	N/A
lineSendUserInfo	REQUEST_SENDUSERINFO	User information	Size
lineSetCallData	REQUEST_SETCALLDATA	TSPICALLDATA	N/A
lineSetCallParams	REQUEST_SETCALLPARAMS	TSPICALL PARAMS	N/A
lineSetCallTreatment	REQUEST_SETCALLTREATMENT	N/A	Call Treatment
lineSetDevStatus	REQUEST_SETDEVSTATUS	Status to change (DWORD)	Set flag
lineSetTerminal	REQUEST_SETTERMINAL	TSPILINESET TERMINAL	N/A
lineSetQualityOfService	REQUEST_SETQOS	TSPIQOS	N/A
lineSetupConference	REQUEST_SETUPCONF	TSPICONFERENCE	N/A
lineSetupTransfer	REQUEST_SETUPXFER	TSPITRANSFER	N/A
lineSwapHold	REQUEST_SWAPHOLD	CTSPICallAppearance of held call.	N/A
lineUncompleteCall	REQUEST_UNCOMPLETECALL	CTSPiRequest of original lineCompleteCall	N/A
lineUnhold	REQUEST_UNHOLD	N/A	N/A
lineUnpark	REQUEST_UNPARK	Array of DIALINFO objects.	N/A
phoneDevSpecific	N/A	N/A	N/A
phoneSetButtonInfo	REQUEST_SETBUTTONINFO	TSPISetBUTTONINFO	N/A
phoneSetData	REQUEST_SETPHONEDATA	TSPIPHONEDATA	N/A
phoneSetDisplay	REQUEST_SETDISPLAY	TSPIPHONEDISPLAY	N/A
phoneSetGain	REQUEST_SETHOOKSWITCHGAIN	TSPiHOOKSWITCHPARAM	N/A
phoneSetHookSwitch	REQUEST_SETHOOKSWITCH	TSPiHOOKSWITCHPARAM	N/A
phoneSetLamp	REQUEST_SETLAMP	TSPISetBUTTONINFO	N/A
phoneSetRing	REQUEST_SETRING	TSPiRINGPATTERN	N/A

CServiceProvider Method	Request type generated	GetDataPtr parameter	GetData Size parameter
phoneSetVolume	REQUEST_SETHOOKSWITCHVOL	TSPHOOKSWITCHPARAM	N/A

In addition the following commands, which return a SUCCESS result code, will generate an asynchronous request with a request ID of zero. These requests are generated to perform the H/W work for each command and the service provider should call back to the appropriate objects and set the appropriate state information based on the request being processed.

CServiceProvider Method	Request type generated	GetDataPtr parameter	GetData Size parameter
lineSetMediaControl	REQUEST_MEDIACONTROL	N/A	Media Event
lineGenerateDigits	REQUEST_GENERATEDIGITS	TSPGENERATE	N/A
lineGenerateTone	REQUEST_GENERATETONE	TSPGENERATE	N/A
phoneGetData	REQUEST_GETPHONEDATA	TSPPHONEDATA	N/A

Additionally, request codes are defined for the following in order to test the provider through the **CServiceProvider::CanHandleRequest** method to see if the function is supported. These requests will never be inserted into the asynchronous request list, they are simply tested for validity for status and feature bit setting.

CServiceProvider Method	Request type generated
lineMonitorDigits	REQUEST_MONITORDIGITS
lineMonitorMedia	REQUEST_MONITORMEDIA
lineMonitorTones	REQUEST_MONITORTONES
lineReleaseUserUserInfo	REQUEST_RELEASEUSERINFO
lineGatherDigits	REQUEST_GATHERDIGITS

Each request when completed will automatically delete any data associated with the request. Each of the TSPI parameter blocks allocated for each request will automatically delete any embedded data. All pointers except return buffers that are passed to **CServiceProvider** methods are automatically copied into local storage and are guaranteed to be valid if a value is given (and not NULL). For more information on the structures passed to each request, see the section titled *Asynchronous Request Data Structures*.

Destructor - Public Methods

~CTSPIRequest Destructor which automatically frees the data pointer for known request types. If any additional request types are added which require data to be allocated and stored, then the request object *must* be overridden to insure that the data is deleted in all cases.

Operations - Public Methods

GetAsynchRequestId Returns the TAPI generated asynchronous request ID.

GetCallInfo Returns the **CTSPICallAppearance** (if any) associated with this request.

GetCommand Returns the **REQUEST_XXX** value for this command

GetConnectionInfo Returns the **CTSPILineConnection** or **CTSPIPhoneConnection** associated with this request.

GetDataPtr Returns the 32-bit pointer associated with this request (see above list for passed pointers)

GetDataSize Returns the 32-bit size value associated with this request (see above list for passed values)

GetState Returns the current state of this request (service provider defined).

GetStateData Returns the currently associated state data with this request (service provider defined).

HaveSentResponse Return TRUE/FALSE indicating whether TAPI has been notified about this request finishing.

SetCommand	Set the current command type WARNING: DO NOT USE THIS ON THE PRE-DEFINED COMMAND!
SetState	Changes the current state of this request (service provider defined)
SetStateData	Changes the current 32-bit data value associated with the current state of the request.
WaitForCompletion	Wait for the request to finish.

Static Operations - Public Members

IsAddressOk	Return whether the specified address is valid.
--------------------	--

Overridables - Protected Members

Init	Called directly after the constructor by the CTSPIDevice owner object
Complete	Completes the request and potentially notifies TAPI.

CTSPIRequest::Complete

Protected

virtual void Complete (LONG IResult = 0L, BOOL fSendTN = TRUE);

<i>IResult</i>	Final result code for this request.
<i>fSendTN</i>	Whether to notify TAPI that the request is complete.

Remarks

This method is called by the device object when the request is completed. It optionally notifies TAPI that the request is complete, and runs through the line/phone owner, address owner, and associated call and notifies each object that the request is complete.

Any waiting thread on the request is released. A thread can wait on a request using the **WaitForCompletion** method.

Generally, you should call the **CTSPIConnection** object to complete requests so that the request is removed from the asynchronous request list.

CTSPIRequest::GetAsynchRequestId

DRV_REQUESTID GetAsynchRequestId() const;

Remarks

This method returns the TAPI asynchronous request identifier which is associated to this running request.

Return Value

The 32-bit asynchronous request identifier assigned by TAPI when it started the request.

CTSPIRequest::GetCallInfo

CTSPICallAppearance* GetCallInfo() const;

Remarks

This method returns the associated **CTSPICallAppearance** that this request is working with.

Return Value

The call object the request is associated to. NULL if there is no call.

CTSPIRequest::GetCommand

WORD GetCommand() const;

Remarks

This method returns the REQUEST_XXX command that this request object represents.

Return Value

The defined request type (see the above section on *Processing a Request* for more information).

CTSPIRequest::GetConnectionInfo

CTSPIConnectionInfo* GetConnectionInfo() const;

Remarks

This method returns the associated CTSPILineConnection or CTSPISPhoneConnection that this request is working with.

Return Value

The line/phone connection object the request is associated to or NULL if there is no object.

CTSPIRequest::GetDataPtr

void* GetDataPtr() const;

Remarks

This method returns the associated 32-bit data element that is associated with this request. The data element depends on the type of request this object represents. For each of the related data elements, see the above section on *Processing a Request*.

Note that if this is a memory block, and you replace it, you *must* allocate the block with the library allocator since the memory is freed automatically for all library-defined requests. If you define your own requests, you may use any allocator you wish.

Return Value

The 32-bit data element assigned to the request.

CTSPIRequest::GetDataSize

DWORD GetDataSize() const;

Remarks

This method returns the size of the associated 32-bit data element that is associated with this request. This is assigned when the request is created and is dependent on the request type.

Note that the class library itself doesn't use this field, it is supplied for the derived provider only.

Return Value

The size of the 32-bit data element assigned to the request.

CTSPIRequest::GetState

int GetState() const;

Remarks

This method returns the current assigned state of the request packet. The only library-defined state is **STATE_INITIAL** which is the first state the request is placed into. This is what drives the state-machine mechanics of the provider model.

Return Value

The request state in effect for this object.

CTSPIRequest::GetStateData

DWORD GetStateData() const;

Remarks

This method returns the request objects state item-data. This is a field which is initialized to zero, but may be changed using **SetStateData**, and then used during the lifetime of the request to track whatever the derived provider wishes. If you place a pointer here, you are responsible for freeing the data associated with the pointer.

Note that the class library itself doesn't use this field, it is supplied for the derived provider only.

Return Value

The requests item-data.

CTSPIRequest::HaveSentResponse

BOOL HaveSentResponse() const;

Remarks

This method returns whether or not this request has replied to TAPI as to the final result code. There are times when TAPI must be told the request is completed before the request is actually done processing. In these cases, the **CompleteRequest** method of the **CTSPILineConnection** or **CTSPIPhoneConnection** can be told to *not* delete the request, but reply to TAPI anyway.

Return Value

TRUE if TAPI has been told the request finished.

FALSE if TAPI has not yet been notified about the request completion.

CTSPIRequest::Init**Protected**

```
virtual void Init(CTSPIConnection* pConn, CTSPIAddressInfo* pAddr,
                 CTSPICallAppearance* pCall, WORD wRequest,
                 DRV_REQUESTID dwRequestId, LPVOID lpBuff, DWORD dwSize);
```

<i>pConn</i>	Phone or line connection object which is associated with request.
<i>pAddr</i>	Address object which is associated with request
<i>pCall</i>	Call object which is associated with request
<i>wRequest</i>	Command type (REQUEST_XXX)
<i>dwRequestId</i>	TAPI asynchronous request id which identifies this request.
<i>lpBuff</i>	Optional buffer to store with request (depends on request type)
<i>dwSize</i>	Size of above buffer (also sometimes used as DWORD value).

Remarks

This method is called right after the constructor to initialize the request object. This is called *before* the request is inserted into the device request list. In general, you should never need to override this function.

CTSPIRequest::IsAddressOk**Static**

```
BOOL IsAddressOk(LPVOID lpBuff, DWORD dwSize);
```

<i>lpBuff</i>	Pointer to check.
<i>dwSize</i>	Size of above buffer.

Remarks

This method verifies that the pointer is valid and can be read from for the size given. It relies on the **TOOLHELP** library (under Windows 3.1) and the built in Win32 functions under TAPI 2.x.

Return Value

TRUE if the passed pointer is readable for the size given.

FALSE if the pointer is invalid or is not the specified size.

CTSPIRequest::SetCommand

```
void SetCommand(WORD wCommand);
```

wCommand Command to set into the request packet.

Remarks

This method changes the request type of the object dynamically. You must be very careful when doing this to ensure that the data pointer gets cleaned up properly. Either the request must have the same type of data, or you must free and re-allocate the proper data type so the library doesn't get confused.

CTSPIRequest::SetState

```
void SetState(int iState) const;
```

iState State to transition the request into.

Remarks

This method sets the current assigned state of the request packet. The only library-defined state is **STATE_INITIAL** which is the first state the request is placed into. This is what drives the state-machine mechanics of the provider model.

CTSPIRequest::SetStateData

```
void SetStateData(DWORD dwStateData) const;
```

dwStateData Item-data for this request.

Remarks

This method sets the request objects state item-data. This is a field which is initialized to zero, but can be used during the lifetime of the request to track whatever the derived provider wishes. If you place a pointer here, you are responsible for freeing the data associated with the pointer.

Note that the class library itself doesn't use this field, it is supplied for the derived provider only.

CTSPIRequest::WaitForCompletion

virtual LONG WaitForCompletion (DWORD dwMSecs = INFINITE);

dwMSecs Number of milliseconds to wait for the request to complete.

Remarks

This method causes the current thread to be put to sleep waiting for the request object to be completed. The thread will be woken either when the request is complete, or when the elapsed wait time is over.

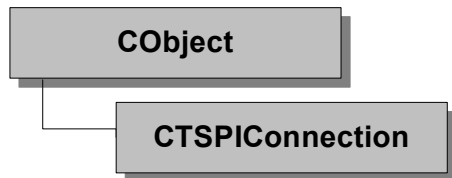
Warning: Do not put the processing thread to sleep unless you are certain that the request will complete!

Return Value

Final TAPI result code of the request.

(-1) if the request timed-out without completing within the given **dwMSecs** time period.

CTSPICConnection



The **CTSPICConnection** is a base class defined in the library to represent a generic connection to a physical medium. It is the base class for the line connection (**CTSPILineConnection**), and phone connection (**CTSPIPhoneConnection**) objects. It cannot be used on its own (it is a virtual base class), and is only documented for the functions which are usable from the line and phone devices.

Asynchronous Request List

The connection object is responsible for maintenance of all the requests pending on that line or phone device. Most requests which are generated by the TAPI dynamic link library are assigned an asynchronous request identifier which tags the request, so that when it is completed, the service provider can inform TAPI as to the result of the operation (for information on this process, see the section on **CTSPIRequest**). The service provider is required to initiate the operation and then return the asynchronous identifier to TAPI which will allow the application to proceed with other tasks until the operation completes.

Since there may be multiple pending requests for a line or phone device, each request from the TAPI dynamic link library which is asynchronous, will generate an asynchronous request object and be stored into a list maintained by the connection. This request list is a FIFO queue, with the topmost request being the one currently being executed. As each request completes, the connection object will remove the request object from the list and inform TAPI about the final result of the operation.

If the service provider has asynchronous requests which stay in the queue for long periods of time, it may be necessary to override the **AddAsynchRequest** function in order to allow other requests to start when the current request has not yet finished. This function is called internally to add each request to the asynchronous list and returns whether or not to immediately activate the request.

Connection Opens and Closes

Each time a line or phone device receives an open request from TAPI through the **lineOpen** or **phoneOpen** API, it calls the **OpenDevice** method of this base connection object. When the line or phone is closed, the **CloseDevice** method will be invoked. The default behavior of these two methods is to call the device object **Open/Close** methods which will eventually send a request to the companion application using the **CServiceProvider::OpenDevice** and **CServiceProvider::CloseDevice** methods.

16-bit special notes

The companion application would receive **COMMAND_OPENCONN** and **COMMAND_CLOSECONN** requests from the service provider by default if none of the above entry-points were overridden. The connection id passed to the companion application is a combination of the provider id, and the index for the line or phone device in the **CTSPIDevice** array. This can be used to uniquely identify the connection and the device which manages it. If

the connection is a phone device, the HI-bit of the index will be set. If multiple connections share the same device (for example, a multiple-line modem, with only one COMM port driving it, or a phone/line tied to the same physical connection), then either the **CServiceProvider** (or **CTSPILineConnection/CTSPIPhoneConnection**) method needs to be overridden to arbitrate each open to a reference count, or the companion application should be written to identify the relationship by the device/connection id passed on all requests. The connection ids are assigned as each line/phone connection is created.

32-bit special notes

One of the open/close methods must be overridden by 32-bit providers to actually provide support for the opening and closing of the device. Generally, it is recommended that this is handled in the Device Object (see **CTSPIDevice**).

Synchronous requests

In some cases, it is advantageous or even necessary to wait for a request to finish. The virtual method **WaitForRequest** will wait a specified amount of time for the request to finish. If no time-out is supplied, then the default library time-out specified in the **CServiceProvider** class will be used (see the **SetTimeout** method in **CServiceProvider**). Any request actually passed by TAPI, unless noted, should not be waited for. This functionality should be used for internal commands only. In most cases, the synchronous requests from the TAPI dynamic link library are handled without any device interaction by the base library.

Operations - Public Members

AddAsynchRequest	Create a new asynchronous request and add it to our list. Start the request if no active request is pending.
AddDeviceClass	Adds a new device class informational structure to the line/phone DEVCAPS structure (32-bit only).
CompleteCurrentRequest	Complete the current head request. This can optionally remove and delete the request.
CompleteRequest	Complete a specific request and notify TAPI. This can optionally remove and delete the request.
FindRequest	Find a request based on some criteria.
GetConnInfo	Return the connection (switch or phone) information for this connection. This will be reported in the LINEDEVCAPS and PHONEDEVCAPS automatically.
GetDeviceClass	Returns a device class informational structure based on the device class key string. (32-bit only)
GetDeviceID	Returns the TAPI-assigned device ID for this line or phone.
GetDeviceInfo	Returns a pointer to the CTSPIDevice owner of this connection object.
GetName	Return the textual name assigned to this phone or line device.
GetNegotiatedVersion	Returns the negotiated TSPI version for this phone or line device. This will be the highest version used for reporting line or phone events while the device is open.
GetRequest	Return a request based on position.
GetRequestCount	Return the count of requests pending.
IsLineDevice	Returns TRUE/FALSE based on whether the connection is a line device. Mutually exclusive from the IsPhoneDevice method.
IsPhoneDevice	Returns TRUE/FALSE based on whether the connection is a phone device. Mutually exclusive from the IsLineDevice method.
RemoveDeviceClass	Removes an associated device class structure from the provider list. (32-bit only)

RemovePendingRequests	This removes all pending requests based on a criteria.
RemoveRequest	Remove a request from our list. It can optionally be deleted.
SendString	Send a string to the line/phone device. This calls the SendData method.
SetConnInfo	Set the connection textual information for this connection. This will be returned in the PHONECAPS or LINEDEVCAPS based on whether this is a line or phone device.
SetName	This member function allows the name of the device to be set. Since the object is constructed in the library, the first time the derived service provider can change the information is in CServiceProvider::providerInit , and the device has already been setup.
WaitForAllRequests	Waits for all pending requests of the type specified on this line or phone device.

Overridables - Public Members

CloseDevice	Called when the last line/phone closes the device. The default implementation calls the CServiceProvider::CloseDevice method.
GetCurrentRequest	Return the current head of the request list.
OpenDevice	Called when the first line/phone opens the device. The default implementation calls the CServiceProvider::OpenDevice method.
ReceiveData	Called by the CTSPIDevice::ReceiveData method when data is received from the companion application for this specific connection (or for any line/phone on the device). Default implementation calls the CServiceProvider::ProcessData method.
SendData	Called to send a block of data to the device. The default implementation calls the CServiceProvider::SendData method.
WaitForRequest	Wait for a request to finish with time-out.

Overridables - Protected Members

AddAsynchRequest	Adds the request to the internal request list and returns whether or not to immediately start the request.
Init	Called directly after the constructor by the CServiceProvider owner object
OnCancelRequest	This is called when a request is canceled on the connection.
OnNewRequest	Processes a new request for the connection
OnRequestComplete	This is called when a request completes on this connection.
OnTimer	This method is called in response to the periodic timer generated by the CTSPIDevice object. It may be overridden to process events.

Operations - Protected Members

SetDeviceID	This method is called by the CServiceProvider class when a new line/phone is created and added to the system. It changes the temporary device id assigned by CTSPIDevice::CreateLine and CTSPIDevice::CreatePhone to a device id defined by TAPI.
--------------------	--

CTSPICConnection::AddAsynchRequest

```
CTSPIRequest* AddAsynchRequest(CTSPIAddressInfo* pAddr,
                                CTSPICallAppearance* pCall, WORD wReqId,
                                DRV_REQUESTID dwReqId = 0, LPCVOID lpBuff = NULL,
                                DWORD dwSize = 0);

CTSPIRequest* AddAsynchRequest(CTSPICallAppearance* pCall,
                                WORD wReqId, DRV_REQUESTID dwReqId=0, LPCVOID lpBuff = NULL,
                                DWORD dwSize = 0);
```

<i>pAddr</i>	Address object this request belongs to (for CTSPILineConnection insertions only).
<i>pCall</i>	<i>Call object this request belongs to (may be NULL).</i>
<i>wReqId</i> ,	REQUEST_ <i>xxx</i> value which will be assigned to the request.
<i>dwReqId</i> ,	TAPI asynchronous request identifier
<i>lpBuff</i>	Optional 32-bit pointer to buffer for request.
<i>dwSize</i>	Size of above buffer.

Remarks

This method inserts a new request into the pending request list. The **CTSPIRequest** object is created, initialized, and inserted into the list. If there are no running requests in the list, then this request is actually started and given the opportunity to run the **STATE_INITIAL** processing before this function returns.

Return Value

Request object that was created or NULL if function failed.

CTSPICConnection::AddAsynchRequest**Protected**

```
virtual BOOL AddAsynchRequest (CTSPIRequest* pReq);
```

<i>pReq</i>	Object to insert into the connection list.
-------------	--

Remarks

This method inserts a new request into the pending request list. If there are no running requests in the list, then this request is actually started and given the opportunity to run the **STATE_INITIAL** processing before this function returns.

This is the worker function called by the public versions of **AddAsynchRequest**. It is documented specifically so that it may be overridden to re-order the insertion of particular requests (for prioritizing).

By default, this function inserts the request at the end of the array.

Return Value

TRUE if the request was inserted into the list.

FALSE if the request was not inserted and will not be run.

CTSPIDevice::AddDeviceClass

32bit-only

```
int AddDeviceClass (LPCTSTR pszClass, DWORD dwData);
int AddDeviceClass (LPCTSTR pszClass, HANDLE hHandle, LPCTSTR lpszBuff);
int AddDeviceClass (LPCTSTR pszClass, HANDLE hHandle, LPVOID lpBuff,
    DWORD dwSize);
int AddDeviceClass (LPCTSTR pszClass, DWORD dwFormat, LPVOID lpBuff,
    DWORD dwSize, HANDLE hHandle = INVALID_HANDLE_VALUE);
int AddDeviceClass (LPCTSTR pszClass, LPCTSTR pszBuff, DWORD dwType = -1L);
```

<i>pszClass</i>	Device class to add/update for (e.g. "tapi/line", etc.)
<i>dwData</i>	DWORD data value to associate with device class.
<i>hHandle</i>	Win32 handle to associate with device class.
<i>lpszBuff</i>	Null-terminated string to associate with device class.
<i>lpBuff</i>	Binary data block to associate with device class
<i>dwSize</i>	Size of the binary data block
<i>dwType</i>	STRINFORMAT of the lpszBuff parameter.

Remarks

This method adds an entry to the device class list, associating it with the specified class name. If you associate class information with an object, it is automatically reported back through the **LINEDEVCAPS** structure and returned using **TSPI_lineGetID**.

Return Value

Index array of added structure.

CTSPIDevice::CloseDevice

```
virtual BOOL CloseDevice();
```

Remarks

This method is called when **TSPI_lineClose** or **TSPI_phoneClose** is called by TAPI.

The default behavior is to call the **CTSPIDevice::CloseDevice** method for processing.

Return Value

TRUE if the connection device closed.

FALSE if the close failed.

CTSPICConnection::CompleteCurrentRequest

```
BOOL CompleteCurrentRequest(LONG IResult = 0, BOOL fTellTAPI = TRUE,  
    BOOL fRemove = TRUE);
```

<i>IResult</i>	Final result code for the head request in our list.
<i>fTellTAPI</i>	TRUE if TAPI should be notified that the request is complete.
<i>fRemove</i>	TRUE if we should delete the request from the list.

Remarks

This method locates the current running request (the first request in the asynchronous list) and calls the **CTSPIRequest::Complete** method to complete the request. It then removes the request from the list if the **fRemove** parameter is **TRUE**.

Return Value

TRUE if the request was completed or updated with new information.

FALSE if the completion failed.

CTSPICConnection::CompleteRequest

```
BOOL CompleteRequest(CTSPIRequest* pReq, LONG IResult = 0,  
    BOOL fTellTAPI = TRUE, BOOL fRemove = TRUE);
```

<i>pReq</i>	Request object to complete
<i>IResult</i>	Final result code for the head request in our list.
<i>fTellTAPI</i>	TRUE if TAPI should be notified that the request is complete.
<i>fRemove</i>	TRUE if we should delete the request from the list.

Remarks

This method calls the **CTSPIRequest::Complete** method to complete the request. It then removes the request from the list if the **fRemove** parameter is **TRUE**.

Return Value

TRUE if the request was completed or updated with new information.

FALSE if the completion failed.

CTSPICConnection::FindRequest

CTSPIRequest* FindRequest(**CTSPICallAppearance*** pCall, **WORD** wReqType);

<i>pCall</i>	Call object associated with request (NULL for any)
<i>wReqType</i>	REQUEST_xxx key to locate.

Remarks

This method searches the request list looking for the specified request. It returns the first located request.

Return Value

Located **CTSPIRequest** object or NULL if no request matching the criteria was found.

CTSPICConnection::GetConnInfo

LPCTSTR GetConnInfo() const;

Remarks

This method returns the PBX switch or phone information for this line/phone device. This information can be supplied to the object using the **SetConnInfo** method.

Return Value

Pointer to string contents for PBX information.

CTSPICConnection::GetCurrentRequest

virtual CTSPIRequest* GetCurrentRequest() const;

Remarks

This method returns the current request which is being processed by the line or phone device. The default behavior is to return the first request in the array.

Return Value

Pointer to the current request.

CTSPConnection::GetDeviceClass*32bit-only*

```
DEVICECLASSINFO* GetDeviceClass(LPCTSTR pszClass);
```

pszClass Device class to search for (e.g. "tapi/line", etc.)

Remarks

This method returns the device class structure associated with the specified class name. If you associate class information with an object, it is automatically reported back through the **LINEDEVCAPS** structure and returned using **TSPI_lineGetID**.

Return Value

DEVICECLASSINFO structure which represents the specified text name or NULL if no association has been performed.

CTSPConnection::GetDeviceID

```
DWORD GetDeviceID() const;
```

Remarks

This method returns the device identifier associated with this device. This is a combination of the permanent provider id associated to the parent **CTSPIDevice** object, and the index into the array where this line or phone is.

This device identifier is used to associate a command to the line or phone with the companion applications in the 16-bit library.

Return Value

Unique identifier to the line or phone within the TSP.

CTSPConnection::GetDeviceInfo

```
CTSPIDevice* GetDeviceInfo() const;
```

Remarks

This method returns the device parent for this line or phone device.

Return Value

Parent **CTSPIDevice** object for the line or phone.

CTSPConnection::GetName

LPCTSTR GetName() const;

Remarks

This method returns the name of the line or phone device that was assigned during the creation of the object.

Return Value

String pointer representing ASCII or UNICODE name of line/phone device.

CTSPConnection::GetNegotiatedVersion

DWORD GetNegotiatedVersion() const;

Remarks

This method returns the negotiated TAPI version decided on when the line or phone was initially opened by TAPI.

This function is only valid when the line or phone device is open and has running requests.

Return Value

TAPI version negotiated by TAPI system components and this TSP.

CTSPConnection::GetRequest

CTSPRequest* GetRequest(int iPos) const;

iPos Index in the request list for the request being accessed.

Remarks

This method allows random-access direct retrieval of requests within the request list. The passed index should be within the current valid ranges (0 - **GetRequestCount**).

Return Value

Request object at the specified position or NULL if no request was found.

CTSPICConnection::GetRequestCount

int GetRequestCount() const;

Remarks

This method returns the current count of pending requests in the asynchronous request list.

Return Value

Current count of pending/active requests.

CTSPICConnection::Init

virtual void Init(CTSPIDevice* pDeviceOwner, DWORD dwDeviceID);

pDeviceOwner
dwDeviceID

Device owner object for this line/phone.
Unique identifier for the line/phone within the TSP.

Remarks

This method initializes the **CTSPICConnection** object and is called directly after the constructor. It is documented here simply for overriding purposes.

CTSPICConnection::IsLineDevice

BOOL IsLineDevice() const;

Remarks

This method returns whether the **CTSPICConnection** object is representing a line or phone device. Another method of doing this would be through MFCs run-time type information.

Return Value

TRUE if this is a line device.
FALSE if this is a phone device.

CTSPICConnection::IsPhoneDevice

BOOL IsPhoneDevice() const;

Remarks

This method returns whether the **CTSPICConnection** object is representing a line or phone device. Another method of doing this would be through MFCs run-time type information.

Return Value

TRUE if this is a phone device.

FALSE if this is a line device.

CTSPICconnection::OnCancelRequest

```
virtual void OnCancelRequest (CTSPIRequest* pReq);
```

pReq Request object which is about to be canceled.

Remarks

This method is called by the connection object when a request is being canceled due to a **REQUEST_DROP** being inserted into the queue.

The default behavior is to pass control to the **CTSPIDevice::OnCancelRequest** method.

CTSPICconnection::OnNewRequest

```
virtual BOOL OnNewRequest (CTSPIRequest* pReq);
```

pReq Request object which is about to be added to the list.

Remarks

This method is called by the connection object each time a new request packet is added to the request list. It is called before the request is officially added to the list and allows the derived connection object to manipulate the list before insertion.

The default behavior is to pass control to the **CTSPIDevice::OnNewRequest** method.

Return Value

TRUE if the connection object should continue to add the request.

FALSE if the request is to be canceled.

CTSPConnection::OnRequestComplete

Protected

```
virtual void OnRequestComplete (CTSPRequest* pReq, LONG IResult);
```

<i>pReq</i>	Request object which has completed.
<i>IResult</i>	Final result code for request.

Remarks

This method is called when a request completes in our request list. It is derived in the **CTSPLineConnection** and **CTSPPhoneConnection** objects to cleanup the request once it is completed (or failed).

CTSPConnection::OnTimer

```
virtual void OnTimer();
```

Remarks

This method is called by the **CTSPDevice** object when an interval timer has been received from either the companion application (16-bit) or interval timer thread (32-bit). It gives the object an opportunity to process idle-time tasks or other cleanup work.

The default behavior is to pass control to the **CServiceProvider::ProcessIntervalTimer** method.

CTSPConnection::OpenDevice

```
virtual BOOL OpenDevice ();
```

Remarks

This method is called from the **CTSPLineConnection** and **CTSPPhoneConnection** objects to open the line or phone device.

If the function is not overridden, it will forward the request onto **CTSPDevice::OpenDevice**.

Return Value

TRUE/FALSE success indicator. If FALSE is returned, the **TSPI_lineOpen** or **TSPI_phoneOpen** which generated the request will fail and return an error.

CTSPICConnection::ReceiveData

```
virtual BOOL ReceiveData (DWORD dwData=0, const LPVOID lpBuff=NULL,
                        DWORD dwSize=0);
```

<i>dwData</i>	32-bit numeric data response.
<i>lpBuff</i>	Pointer to buffer with received data.
<i>dwSize</i>	Size of the above buffer

Remarks

This method is called from the **CTSPIDevice** object owner when it receives data from either the companion application (16-bit) or the input thread (32-bit).

Default behavior of this function is to pass control onto **CServiceProvider::ProcessData**.

Return Value

TRUE/FALSE success indicator.

CTSPICConnection::RemoveDeviceClass

32-bit only

```
BOOL RemoveDeviceClass (LPCTSTR pszClass);
```

<i>pszClass</i>	Device class key to remove.
-----------------	-----------------------------

Remarks

This method removes the specified device class information from this object. It will no longer be reported as available to TAPI.

Return Value

TRUE if device class information was located and removed.

CTSPICConnection::RemovePendingRequests

```
void RemovePendingRequests(CTSPICallAppearance* pCall = NULL,
                          WORD wReqType = REQUEST_ALL,
                          LONG lResult=TAPIERR_REQUESTCANCELLED);
```

<i>pCall</i>	Call object to remove requests for (NULL for all)
<i>wReqType</i>	Request type to locate and remove (REQUEST_ALL for all)
<i>lResult</i>	Result code to send to TAPI.

Remarks

This method completes all pending requests which match the criteria without processing the request and sends the specified error code back to TAPI. Only requests which match the criteria passed in are processed by this function.

CTSPICConnection::RemoveRequest

```
BOOL RemoveRequest(CTSPIRequest* pReq, BOOL fDelete=FALSE);
```

<i>pReq</i>	Request to remove from the list.
<i>fDelete</i>	Whether to delete the request

Remarks

This method completes all pending requests that match the criteria without processing the request and sends the specified error code back to TAPI. Only requests that match the criteria passed in are processed by this function.

Return Value

TRUE/FALSE success indicator.

CTSPICConnection::SendData

```
virtual BOOL SendData (LPCVOID lpBuff, DWORD dwSize);
```

<i>lpBuff</i>	Buffer to send out to the hardware device
<i>dwSize</i>	Size of the buffer to transmit.

Remarks

This method can be called to send data to the device. In the 16-bit library, this function sends data to the companion application. In the 32-bit application, this function essentially does nothing and must be implemented.

If the function is not overridden, it will forward the request onto **CTSPIDevice::SendData**.

Return Value

TRUE/FALSE success indicator.

CTSPIDevice::SendString

virtual BOOL SendString (LPCTSTR lpszBuff);

lpszBuff Null-terminated buffer to send out to the hardware device

Remarks

This method can be called to send string data to the device. In the 16-bit library, this function sends data to the companion application. In the 32-bit application, this function essentially does nothing and must be implemented.

If the function is not overridden, it will forward the request onto **CTSPIDevice::SendData**.

Return Value

TRUE/FALSE success indicator.

CTSPIDevice::SetConnInfo

void SetConnInfo(LPCTSTR lpszInfo);

lpszInfo Null-terminated buffer to use as

Remarks

This function sets a string into the object which will be returned as the PBX or phone information in the **PHONECAPS** and **LINECAPS** structure.

CTSPIDevice::SetDeviceID

Protected

void SetDeviceID(DWORD dwID);

dwID New device line/phone id to associate to this object.

Remarks

This function changes the line/phone device id to another value. This is used by the **CServiceProvider** class when a new line or phone is created using the **CTSPIDevice::CreateLine** or **CTSPIDevice::CreatePhone**. Once TAPI finalizes the line or phone addition, it calls the provider back and hands in a new device identifier for the line/phone.

CTSPICConnection::SetName

```
void SetName(LPCTSTR lpszName);
```

lpszName Name to associate with this line/phone.

Remarks

This function changes the name which is reported back to TAPI in the **PHONECAPS** and **LINECAPS** structures. It should be called during the setup of the line/phone.

CTSPICConnection::WaitForAllRequests

```
void WaitForAllRequests(CTSPICallAppearance* pCall=NULL,  
                        WORD wRequest=REQUEST_ALL);
```

pCall Call appearance to look for (NULL indicates ALL)
wRequest Request id to look for (**REQUEST_ALL** indicates ALL).

Remarks

This function pauses the calling thread until all requests that match the specified criteria have completed or failed.

Warning: Do not put the processing thread to sleep unless you are certain that all the requests will complete!

CTSPICConnection::WaitForRequest

```
virtual void WaitForRequest(DWORD dwTimeout, CTSPICRequest* pReq);
```

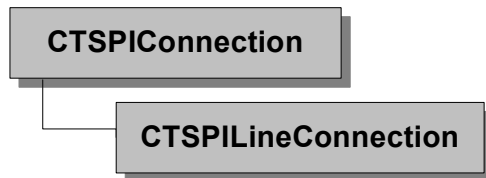
dwTimeout Timeout to wait for the the request to complete in milliseconds. Zero indicates to use the standard request timeout (see **CServiceProvider::SetTimeout**).
pReq Request to wait for.

Remarks

This function pauses the calling thread until the specified request completes. If the specified timeout occurs, the thread is automatically released even if the request isn't done.

Warning: Do not put the processing thread to sleep unless you are certain that the request will complete!

CTSPILineConnection



The **CTSPILineConnection** object represents a line device in the TSP++ library. It is owned by a **CTSPIDevice** object and manages a set of asynchronous requests that are pending for the line it represents. Each object maintains the state and capabilities of the line connection.

Line Devices

A line device is any telephony control device which is connected to an actual telephone line, such as a fax device, modem, or an ISDN card. Line devices support telephony capabilities by allowing applications to send or receive information to or from a telephone network. TAPI itself views the line as a point of entry that leads to the H/W switch. Each line device contains a set of one or more addresses which can be used to establish calls. In some cases, a service provider will model a line as having only a single address (POTS), and in others, multiple addresses are required (ISDN or digital network).

Line connection initialization

The line objects are created by the **CTSPIDevice** object during initialization of the service provider, or when the service provider calls the **CTSPIDevice::CreateLine** method to dynamically add a line device. After the constructor is complete, the virtual method **Init** is called to actually initialize the object. This method initializes known fields in the **LINEDEVCAPS** and **LINEDEVSTATUS** structures and returns. The service provider is responsible for adding address channels to the line during this initialization process by either overriding the object and its **Init** method, or doing the work during the **CServiceProvider** method **providerInit** after the base class is finished.

Line Open/Close events

The line is not considered open until a **lineOpen** is received in the **CServiceProvider** class. At this time, the **CTSPILineConnection::Open** method will be called for the line, setting up a TAPI opaque handle and returning the pointer to the **CTSPILineConnection** as our handle back to TAPI. For more information on this, see the section titled *Opaque handles*. A line will only be opened by TAPI one (regardless of how many applications request it). When the open call is received, the **CTSPICConnection::OpenDevice** method will be invoked. The reverse is true of the **Close** method. In this case, the **CTSPICConnection::CloseDevice** will be invoked.

Capabilities

The line capabilities of the line device are maintained in the **CTSPILineConnection** object. Generally, the capabilities are assigned during the initialization of the service provider. TAPI stores and expects the line device capabilities to be in a **LINEDEVCAPS** structure. The line object maintains a static copy of this structure for easy access. A pointer to this structure can be obtained through the **GetLineDevCaps** method. During the initialization process, the service provider should set any required fields that the base class doesn't fill out or may default to an incorrect value. These include:

Structure member	Description
dwDevCapFlags	Should be filled with applicable LINEDEVCAPFLAGS_
dwMaxNumActiveCalls	Should be adjusted if more than one call may be active on an address, or multiple addresses are shared on a channel.
dwBearerModes	If LINEBEARERMODE_PASSTHROUGH is supported, it needs to be or'ed into the bearer modes for the line. <i>DO NOT</i> include it with an address. The address bearer mode should always be a single mode.
dwAnswerMode	Should be filled if existing calls will not be dropped when answering a new call. This uses the LINEANSWERMODE_ bit flags
dwRingModes	Defaults to one.
dwUIAcceptSize	Defaults to zero.
dwUIAnswerSize	Defaults to zero.
dwUIMakeCallSize	Defaults to zero.
dwUIDropSize	Defaults to zero.
dwUISendUserInfoSize	Defaults to zero.
dwUICallInfoSize	Defaults to zero.
MinDialParams	Defaults to min/max parameters passed on CreateAddress .
MaxDialParams	Defaults to min/max parameters passed on CreateAddress .
DefaultDialParams	Must be supplied by derived class.

All other fields are filled in either during initialization of the line object, or by the methods **CreateAddress** and **AddTerminal** which add capabilities to the device. If the field isn't listed, then it is set to a default value.

Device Specific Extensions

If the line device supports device-specific features, then these must be added either by overriding the **CServiceProvider** method **lineGetDevCaps**, or by overriding the **CTSPILineConnection** method **GatherCapabilities**. To add device-specific extensions to the **LINEDEVSTATUS** record, the **CServiceProvider::lineGetDevStatus**, or **CTSPILineConnection::GatherStatus** may be overridden.

Device function extensions can be supported by either overriding the **CServiceProvider** method **lineDevSpecific**, **lineDevSpecificFeature**, or the **CTSPILineConnection** method **DevSpecificFeature**.

Warning:

*Do not add the device specific information into the static **LINEDEVCAPS**, there is no extra-allocated space!*

Status information

Status information for the line device is also contained within the **CTSPILineConnection** object. The TAPI dynamic link library expects line status to be reported in a **LINEDEVSTATUS** structure. This structure is embedded in our line object to maintain the current state of the line. As different methods change the state of the line, TAPI is notified through the asynchronous callback of each changed element. Most notifications are supported completely by the library. The methods that change the status record are: **SetTerminalModes**, **SetRingMode**, **SetBatteryLevel**, **SetSignalLevel**, **SetRoamMode**, and **SetDeviceStatusFlags**. All fields except the device-specific information are automatically filled out by the line device and tracked through various callbacks by the address and call objects, and by using the above methods to change the information in the status record. The current line device status record may be obtained through the **GetLineDevStatus** method. Other caveats about status notifications are:

Ring Events

To support ring events in the line, the service provider class must call the **OnRingDetected** method. Each ring should call the function, and when the ringing stops, the function should be called one last time with a TRUE value set for the stop ringing flag. The ring count is managed automatically by the library.

Unsupported notifications

The notifications that are not directly supported by the library are: **LINEDEVSTATE_MAINTENANCE**, **LINEDEVSTATE_DEVSPECIFIC**, **LINEDEVSTATE_REINIT**, **LINEDEVSTATE_CAPSCHANGE**, **LINEDEVSTATE_CONFIGCHANGE**, **LINEDEVSTATE_TRANSLATECHANGE**, and **LINEDEVSTATE_COMPLCANCEL**. These can be supplemented by a derived service provider class and passed to TAPI by calling the **OnLineStatusChange** method.

Channels and Addresses

Each line has one or more *channels* on which incoming or outgoing calls may be placed. In a standard home telephone system (POTS), only one channel exists on a line, and it is used exclusively for voice or data traffic. With ISDN, at least three (up to 30+) channels can exist on a line simultaneously.

In the TSP++ library, each channel is assigned its own address. An address corresponds to a telephone directory number. The library assigns each address a unique identifier called an Address ID to make it easy to identify. The identifiers range from zero to the total number of addresses present minus one. If two addresses share a channel with DID (direct inward dialing) or distinctive ring, two address objects would still be added, but only one call could be active at any given time. Otherwise, the addresses work independently of each other and each address may have one or more active calls on the line. To support bridged addresses, some special settings need to be adjusted by the derived classes. See the section titled *Address Sharing* for more information on this topic.

The line object creates an address in response to the **CreateAddress** method. This method needs to be called during initialization of the service provider to add each address to the list of available addresses.

Terminals

The telephony hardware may have one or more *terminals* on which telephony signals are routed. This can include a phone device with lamps, buttons, display, and some form of voice I/O (handset, speakerphone, etc.), a connection to a PC sound card, or some other hardware routing mechanism for manipulating and interpreting the audio/telephony signals. A TAPI application can control where the information is routed by using the **lineSetTerminal** function. This would impact the normal routing between the switch and the station, controlled by the TSP itself. This allows the application to turn specific audio devices on and off based on how the user wishes to interact with a call. For example, a piece of hardware might have a ringer and a headset associated with the line. The application can turn the ringer off and instead notify the user through a dialog that there is an incoming call. If the user decides to answer the call, the application can then cause all events to be routed to the headset, turning the speaker and microphone on and then asking the switch to route the audio signal to that terminal. The library supports the reporting and management of terminals through a series of methods, **AddTerminal**, **RemoveTerminal**, **GetTerminalCount**, and **GetTerminalInformation**. TAPI is automatically informed about all terminal devices which are added using these APIs. When the command to route terminal information is invoked by an application using **lineSetTerminal**, an asynchronous request of type **REQUEST_SETTERMINAL** is placed into the queue.

When the service provider processes the request, the **CTSPILineConnection::SetTerminalModes** function should be used to inform the class library about the new assignments for each terminal.

This will cause the class library to send all the appropriate notifications and adjust each of the active calls on the line to reflect the proper terminal information.

Request completions

When a request completes on the line, the **OnRequestComplete** method is called. The default line connection object takes the following actions when processing requests:

REQUEST_SETTERMINAL If the request completes successfully, the line connection will alter the internal terminal map and notify TAPI that the terminal information has changed in the **LINEDEVCAPS** record. This change will be cascaded to the address and calls that are owned by the line.

REQUEST_FORWARD If the request completes successfully, forwarding record(s) will be added to the **LINEDEVCAPS** structure and TAPI will be notified that the information has changed. All addresses on the line will be adjusted as well.

REQUEST_COMPLETECALL If the request completes successfully, a call completion request will be added to an internal array for notification of the final completion. It may be located using the **FindCallCompletionRequest** and deleted using the **RemoveCallCompletionRequest** methods.

REQUEST_UNCOMPLETECALL If the request completes successfully, the call completion request which was queued originally will be removed and deleted. TAPI is not notified since this is not considered a *device* request, rather a user request.

Operations - Public Members

In addition to all the members which are available from the base class **CTSPICConnection**, the following members are specific to the line object:

AddTerminal	Add a new terminal device to the line (<i>initialization only!</i>)
CreateAddress	Create a new address on the line (<i>initialization only!</i>)
CreateUIDialog	Creates a new asynchronous user-interface dialog associated with an open line (32-bit only).
FindCallCompletionRequest	Locate a call completion request by call, switch information, or call completion ID.
FindCallByState	Locates a call appearance on this line based on a call state.
ForceClose	Forces the line to close immediately
GetAddress	Locate an address object by its dialable address or id.
GetAddressCount	Returns the number of addresses on this line.
GetDefaultMediaDetection	Returns the default media types detected
GetLineHandle	Returns the TAPI opaque line handle for this device.
GetLineDevCaps	Returns a pointer to the LINEDEVCAPS for this line.
GetLineDevStatus	Returns a pointer to the LINEDEVSTATUS for this line
GetTerminalCount	Returns the count of terminals on this line.
GetTerminalInformation	Returns the information associated with a terminal on this line.
GetUIDialog	Returns a handle to the open dialog user-interface object (32-bit only).
OnRingDetected	This method needs to be called whenever a ring is detected on this line.
RemoveCallCompletionRequest	Remove and delete a call completion request.
RemoveTerminal	Remove an existing terminal from the line.
SendDialogInstanceData	Sends data to an open user-interface object (32-bit only).
Send_TAPI_Event	This method sends TAPI an event record through the asynchronous callback supplied by TAPI on the Open.call .
SetBatteryLevel	This method sets the current battery level in the LINEDEVSTATUS record.
SetDeviceStatusFlags	This method sets the current device flags in the

SetRingMode	LINEDEVSTATUS record. This method sets the current ring mode listed in the LINEDEVSTATUS record.
SetRoamMode	This method sets the current roaming mode in the LINEDEVSTATUS record.
SetLineFeatures	This allows the service provider to adjust the line features in the LINEDEVSTATUS record.
SetSignalLevel	This method sets the current signal level in the LINEDEVSTATUS record.
SetTerminalModes	This method sets the terminal routing information. This is called by the library when a REQUEST_SETTERMINAL completes successfully.

Overridables - Public Members

CanSupportCall	Return whether the call-type can be supported on this line. This is used internally to validate LINECALLPARAMS .
FindAvailableAddress	Locate an address suitable for carrying a call type.
GetPermanentDeviceID	Returns the permanent device ID assigned by the class library to this line device. This device ID is a combination of the provider ID and the position within the device array of the CTSPIDevice object owner.
OnMediaConfigChanged	This is called when the media configuration information is changed.
OnLineCapabilitiesChanged	This is called when any data in the LINEDEVCAPS record changes.
OnLineStatusChange	This is called when any data in our LINEDEVSTATUS record changes to notify TAPI.
ValidateMediaControlList	Validate the media control list against our LINEDEVCAPS . This is used internally during lineSetMediaControl .

Operations - Protected Members

CanHandleRequest	This calls the CServiceProvider class to see if a request can be handled by this line.
FreeDialogInstance	Frees an open user-interface object. (32-bit only).
IsConferenceAvailable	Returns a BOOL indicating whether there is an active conference call on this line.
IsTransferConsultAvailable	Returns a BOOL indicating whether there is a consultation transfer pending on this line.

Overridables - Protected Members

Init	This method is called directly after the constructor by the CTSPIDevice owner.
OnAddressFeaturesChanged	This is called when address features are changed within an owned address object on this line.
OnCallDeleted	Called when a CTSPICallAppearance object is being deleted (deallocated) by an address. The default implementation removes any references to the call from any pending or working asynchronous requests.
OnCallFeaturesChanged	This is called by the address object when a call on the address changes its call features.
OnCallStateChange	This is called by call appearances on this line when the call state information changes.
OnConnectedCallCountChange	This is called when the connected call count changes in any of the

OnLineFeaturesChanged	lines address objects. This is called when the line features are changed by the class library.
OnRequestComplete	This method is called as a request on this line completes. It performs various cleanup procedures required for the request based on whether the request finished successfully or not.
OnTimer	This is called on a periodic interval to do various cleanup and event checks.

Overridables - TAPI Members

Close	This method is called for lineClose . Default implementation decrements the device count and waits for all Drop requests to finish.
ConditionalMediaDetection	This method is called for lineConditionalMediaDetection . Handled completely within the library.
ConfigDialog	This method is called for lineConfigDialog and lineConfigDialogEdit . Default implementation returns Not Supported . (16-bit only).
DevSpecificFeature	This method is called for lineDevSpecificFeature . Default implementation returns Not Supported .
Forward	This method is called for lineForward . Default implementation issues a REQUEST_FORWARD request.
GatherCapabilities	This method is called for lineGetDevCaps . Handled completely within the library.
GatherStatus	This method is called for lineGetDevStatus . Handled completely within the library.
GenericDialogData	This is called to process lineGenericDialogData requests on the line.
GetAddressID	This method is called for lineGetAddressID . Managed completely within the library.
GetDevConfig	This method is called for lineGetDevConfig . Default implementation returns Not Supported .
GetIcon	This method is called for lineGetIcon . Default implementation returns Not Supported .
GetID	This method is called for lineGetID . Default implementation supports the tapi/line and tapi/phone extensions.
MakeCall	This method is called for lineMakeCall . Default implementation locates an available call appearance and issues a REQUEST_MAKECALL request.
Open	This method is called for lineOpen . Default implementation increments the device count and swaps handles with TAPI.
SetDefaultMediaDetection	This method is called for lineSetDefaultMediaDetection . Handled completely within the library.
SetDevConfig	This method is called for lineSetDevConfig . Default implementation returns Not Supported .
SetMediaControl	This method is called for lineSetMediaControl . Default implementation watches for the event(s) and when seen, issues a REQUEST_SETMEDIA request.
SetLineDevStatus	This method is called in response to a lineSetLineDevStatus function.
SetStatusMessages	This method is called for lineSetStatusMessages . Handled completely within the library.
SetTerminal	This method is called for lineSetTerminal . Default implementation issues a REQUEST_SETTERMINAL request.
UncompleteCall	Remove a call completion request for this line.

CTSPILineConnection::AddTerminal

```
int AddTerminal(LPCTSTR lpszName, LINETERMCAPS& termCaps,  
               DWORD dwModes = 0L);
```

lpszName ASCII name of the terminal device
termCaps Terminal capabilities to associate with the terminal
dwModes *LINETERMMODE_*xxx flags to associate with terminal

Remarks

This method creates a new terminal device for the line. A terminal device is an end-point where all line-oriented telephony signals are routed. This can include buttons, tones, audio signals, display information, etc.

This function automatically adds the structures necessary to report the terminal information back to TAPI, and if **lineSetTerminal** is invoked on the line, the information passed will be checked against the available added terminals.

TAPI is notified that the number of terminals has changed through the **LINEDEVSTATE_TERMINALS** event.

Return Value

Terminal identifier - this is simply the position of the entry within the terminal array. If the function fails, (-1) is returned.

CTSPILineConnection::CanHandleRequest

Protected

```
BOOL CanHandleRequest(WORD wRequest, DWORD dwData = 0);
```

wRequest Request type to check (**REQUEST_**xxx from *splib.h*).
dwData Optional data to pass through library.

Remarks

This method is called to verify that a specific request type can be processed by the TSP. Eventually, it ends up calling **CServiceProvider::CanHandleRequest**, which is a virtual function. The optional data member is not used by the class library, but can be used by the TSP when overriding the **CServiceProvider** member.

This function is called in response to any request made by TAPI against the service provider. It verifies that the request is valid at that moment.

Return Value

TRUE/FALSE success code. FALSE is returned if the function is not supported. This generally will cause the request for which this function was called to fail.

CTSPILineConnection::CanSupportCall

```
virtual LONG CanSupportCall (  
    const LPLINECALLPARAMS lpCallParams) const;  
  
lpCallParams        LINECALLPARAMS structure to verify.
```

Remarks

This method is called to verify that a call of a specified type may be supported by the line object. The function by default runs through all the **CTSPIAddressInfo** objects and calls the **CanSupportCall** method of each address.

Return Value

This function returns a standard TAPI return code. If any of the address objects report that the call can be made, then the function returns successful.

If none of the addresses can handle the call based on the information in the structure, then the function returns either a specific error as to what cannot be supported, or **LINEERR_INVALIDCALLPARAMS**.

CTSPILineConnection::Close()

```
virtual LONG Close();
```

Remarks

This method is invoked by **TAPI** when the line is closed by all applications. It calls the **CTSPIDevice::CloseDevice** method and resets all the line handle information. Once this function completes, the line will have no further interaction with TAPI until it is opened again.

Return Value

This function returns a standard TAPI return code.

CTSPILineConnection::ConditionalMediaDetection

```
virtual LONG ConditionalMediaDetection(DWORD dwMediaModes,  
    const LPLINECALLPARAMS lpCallParams);  
  
dwMediaModes    Media modes which are being checked by LINEMAPPER.  
lpCallParams    LINECALLPARAMS structure to verify.
```

Remarks

This method is invoked by **TAPI** when the application requests a line open using the **LINEMAPPER**. This method will check the requested media modes and return an acknowledgement based on whether the line can monitor all the requested modes and can support the request call type based on the **LINECALLPARAMS** structure.

Return Value

This function returns a standard TAPI return code. It calls the **CServiceProvider::ProcessCallParameters** function to validate the call parameters. The media mode item within the call parameters will be the media modes passed into this function.

If the call parameters are supported, zero is returned.

CTSPILineConnection::CreateAddress

```
DWORD CreateAddress (LPCTSTR lpszDialableAddr = NULL,
                    LPCTSTR lpszAddrName = NULL,
                    BOOL fAllowIncoming = TRUE,
                    BOOL fAllowOutgoing = NULL,
                    DWORD dwAvailMediaModes = LINEMEDIAMODE_UNKNOWN,
                    DWORD dwBearerMode = LINEBEARERMODE_VOICE,
                    DWORD dwMinRate = 0L, DWORD dwMaxRate = 0L,
                    LPLINEDIALPARAMS lpDialParams = NULL,
                    DWORD dwMaxNumActiveCalls = 1,
                    DWORD dwMaxNumOnHoldCalls = 0,
                    DWORD dwMaxNumOnHoldPendCalls = 0,
                    DWORD dwMaxNumConference = 0,
                    DWORD dwMaxNumTransConf = 0);
```

<i>lpszDialableAddr</i>	Dialable phone number of the address.
<i>lpszAddrName</i>	ASCII name reported back in ADDRESSCAPS .
<i>fAllowIncoming</i>	TRUE if incoming calls are allowed on this address.
<i>fAllowOutgoing</i>	TRUE if outgoing calls are allowed on this address.
<i>dwAvailMediaModes</i>	Available media modes on this address.
<i>dwBearerMode</i>	Single LINEBEARERMODE_xxx flag.
<i>dwMinRate</i>	Minimum data rate reported in ADDRESSCAPS .
<i>dwMaxRate</i>	Maximum data rate reported in ADDRESSCAPS .
<i>lpDialParams</i>	Dialing parameters (NULL to use line information).
<i>dwMaxNumActiveCalls</i>	Max number of calls in a Connected state.
<i>dwMaxNumOnHoldCalls</i>	Max number of calls in a Hold state.
<i>dwMaxNumOnHoldPendCalls</i>	Max number of calls waiting for Transfer/Conference .
<i>dwMaxNumConference</i>	Max number of calls conferenced together.
<i>dwMaxNumTransConf</i>	Max number of calls conferenced from a transfer event.

Remarks

This method is used to create a new address on a line. It should be used during the initialization of the line object (**CTSPILineConnection::Init**). The information given to each address is used to determine the capabilities of the line itself. For instance, all the media modes for each added address object are collected and returned in the **LINEDEVcaps** of the line object.

Return Value

The address identifier assigned to the address. This corresponds to the position in the address array where the object was added.

CTSPILineConnection::CreateUIDialog

```
HTAPIDIALOGINSTANCE CreateUIDialog (DRV_REQUESTID dwRequestID,  
    LPVOID lpParams = NULL, DWORD dwSize = 0L,  
    LPCTSTR lpszUIDLLName = NULL);
```

<i>dwRequestID</i>	Request ID which identifies the process owner of the dialog.
<i>lpParams</i>	Optional pointer to a parameter block
<i>dwSize</i>	Size of the above parameter block
<i>lpszUIDLLName</i>	Name of the user-interface DLL to load (NULL for default).

Remarks

This method causes TAPI to load a unsolicited dialog in the process space of the application that sponsored the **dwRequestID** parameter. The request ID must be valid at the time of this API being called (i.e. the request cannot have been completed yet). The parameter block is user-defined, so any information may be passed to the dialog. If no DLL name is supplied, then the name passed to the constructor of the **CServiceProvider** object is used.

Return Value

Handle to the dialog created or NULL if there was an error.

CTSPILineConnection::CTSPILineConnection

```
CTSPILineConnection()
```

Remarks

This is the constructor for the line connection object.

CTSPILineConnection::~~CTSPILineConnection

```
~CTSPILineConnection()
```

Remarks

This is the destructor for the line connection object.

CTSPILineConnection::DevSpecificFeature

```
virtual LONG DevSpecificFeature(DWORD dwFeature,
    DRV_REQUESTID dwRequestID,
    LPVOID lpParams, DWORD dwSize);
```

<i>dwFeature</i>	Feature code passed to lineDevSpecificFeature .
<i>dwRequestID</i>	Asynchronous request ID associated with this request.
<i>lpParams</i>	Optional pointer to a parameter block
<i>dwSize</i>	Size of the above parameter block

Remarks

This method is called by the library when an application calls **lineDevSpecificFeature** and identifies this line object in the **HLINE** parameter. The **TSPI_lineDevSpecificFeature** function must be exported from the provider. This function enables service providers to provide access to features not typically offered by TAPI. The meaning of these extensions are device specific and known only to the service provider and applications which are written to take advantage of them. This function is not handled directly by the library and is provided simply for overriding purposes.

Return Value

You must override this function to provide device-specific functionality. The library returns **LINEERR_OPERATIONUNAVAIL** if you do not override this function.

CTSPILineConnection::FindAvailableAddress

```
virtual CTSPIDressInfo* FindAvailableAddress (
    const LPLINECALLPARAMS lpCallParams,
    DWORD dwFeature = 0) const;
```

<i>lpCallParams</i>	LINECALLPARAMS structure which identifies the type of call.
<i>dwFeature</i>	Feature which will use the address (LINEADDRFEATURE_xxx)

Remarks

This method is used in the library to locate a specific address to use for an outgoing call. Either the address identified by the **LINECALLPARAMS.dwAddressID** is returned, or the information within the **LINECALLPARAMS** structure is used to find the best suitable address by bearer mode and media mode information.

Return Value

A **CTSPIDressInfo** object which can handle the call type presented within the **LINECALLPARAMS** structure.

CTSPILineConnection::FindCallByState

```
CTSPICallAppearance* FindCallByState(DWORD dwState);
```

dwState Call state to locate.

Remarks

This method is used in the library to locate any call on the line which is in the given state. The first call found is returned. Each address on the line is checked in order to find a call.

Return Value

The first **CTSPICallAppearance** found which is in the given state.

CTSPILineConnection::FindCallCompletionRequest

```
TSPICOMLETECALL* FindCallCompletionRequest (DWORD dwSwitchInfo,
      LPCTSTR pszSwitchInfo);
TSPICOMLETECALL* FindCallCompletionRequest(
      CTSPICallAppearance* pCall);
TSPICOMLETECALL* FindCallCompletionRequest(DWORD dwCompletionID);
```

dwSwitchInfo Switch information which was associated with a completion request.

pszSwitchInfo ASCII Switch information which was associated with a completion request.

pCall Call object which is associated with completion request.

dwCompletionID Completion identifier associated with completion request.

Remarks

These methods are provided to locate call completion requests which have been setup on the line. Call completion requests specify how a call that could not be connected normally should be completed instead. They are issued by the application using **lineCompleteCall**. The network or switch might be unable to complete a call because the destination is busy, doesn't answer, or because there are no outgoing lines available on the PBX. The completion request is created by the **CompleteCall** asynchronous request and stays in an array owned by the line object.

This function allows the provider to locate a completion request once the call has come back to the station - when the PBX is signaling the final completion of the call. A new call will typically be created and offered to the application using **CTSPIAddressInfo::CreateCallAppearance** with the reason code being **LINECALLREASON_CALLCOMPLETION**, and the completion id filled out from the **TSPICOMLETECALL** record.

The completion record should be destroyed using the **RemoveCallCompletionRequest** by the service provider when the completion is detected and offered on the PBX.

Return Value

The **TSPICOMLETECALL** record which matches the given search criteria.

CTSPILineConnection::ForceClose

```
void ForceClose();
```

Remarks

This method may be called by the service provider to close the line device immediately regardless of it being in-use or not. It should be used when the device is going offline for some hardware reason or because a component or network connection has been lost.

CTSPILineConnection::Forward

```
virtual LONG Forward (DRV_REQUESTID dwRequestID,  
    CTSPIDAddressInfo* pAddr,  
    TSPILINEFORWARD* lpForwardInfo,  
    HTAPICALL htConsultCall,  
    LPHDRVCALL lphdConsultCall);
```

<i>dwRequestID</i>	Asynchronous request ID associated with this forward.
<i>pAddr</i>	Address object to forward (NULL means all addresses)
<i>lpForwardInfo</i>	Forward Information structure (see the section on <i>Asynchronous Request Data Structures</i>).
<i>htConsultCall</i>	TAPI opaque call handle for the creation of a consultation call as a result of this operation.
<i>lphdConsultCall</i>	TSP call handle to match to the created consultation call created for the forwarding.

Remarks

This method is called when the line is forwarded using **lineForward**. The **TSPILineForward** function must be exported from the provider. The **TSPILINEFORWARD** structure contains all the parameters which were sent by the application. The consultation call parameters are used only if the network or switch creates a consultation call as a result of a forwarding request. In this case, the service provider is responsible for creating a new call appearance using **CTSPIDAddressInfo::CreateCallAppearance** passing in the **htConsultCall**, and assigning the resultant pointer to the **lphdConsultCall** pointer.

The default behavior of the line object is to verify that the line/address can be forwarded using the **CTSPIDAddressInfo::CanForward** function, and then add an asynchronous request of type **REQUEST_FORWARD** into the queue.

Return Value

TAPI Error code or asynchronous request ID if the request was queued.

CTSPILineConnection::FreeDialogInstance

LONG FreeDialogInstance(HTAPIDIALOGINSTANCE htDlgInst);

htDlgInst Dialog instance to deallocate.

Remarks

This function is called when a unsolicited user-interface dialog which was created using **CreateUIDialog** is being deallocated due to it being destroyed. This function removes all references to the dialog and deletes the internal representation of the dialog from the provider.

Return Value

TAPI Error code or zero if the function was successful.

CTSPILineConnection::GatherCapabilities

**virtual LONG GatherCapabilities (DWORD dwTSPIVersion,
 DWORD dwExtVer, LPLINEDEVCAPS lpLineCaps);**

<i>dwTSPIVersion</i>	The TAPI version which structures should reflect.
<i>dwExtVer</i>	The extension version which structures should reflect.
<i>lpLineCaps</i>	The structure to fill with line capabilities.

Remarks

This function is called when an application requests line capabilities using the **lineGetDevCaps** function. The class library automatically fills in all the known information about the provider using the line, address, and call objects which are associated with the current line object.

The service provider can adjust the capabilities returned by using the **GetLineDevCaps** function and modifying the returning structure. This should typically be done when the line is first initialized as capabilities normally do not change during the life of the provider.

If the structure is modified through the **GetLineDevCaps** function, TAPI is not automatically notified.

This function should be overridden if the service provider supports device extensions and wishes to return the information in the **LINEDEVCAPS** structure.

Return Value

TAPI Error code or zero if the function was successful.

CTSPILineConnection::GatherStatus

virtual LONG GatherStatus (LPLINEDEVSTATUS lpStatus);

lpStatus The structure to fill with line status.

Remarks

This function is called when an application requests the current line status using the **lineGetDevStatus** function. The class library automatically fills in all the known information about the provider using the line, address, and call objects which are associated with the current line object.

The service provider can adjust the capabilities returned by using the **GetLineDevStatus** function and modifying the returning structure. Some of the information can be modified using other **CTSPILineConnection** methods. These include **SetBatteryLevel**, **SetSignalLevel**, **SetRoamMode**, **SetDeviceStatusFlags**, **SetRingMode**, and **SetTerminalModes**. Using these methods is recommended as TAPI is notified about the information changing through a **LINEDEVSTATUS** callback.

If the structure is modified through the **GetLineDevStatus** function, TAPI is not automatically notified.

This function should be overridden if the service provider supports device extensions and wishes to return the information in the **LINEDEVSTATUS** structure.

Return Value

TAPI Error code or zero if the function was successful.

CTSPILineConnection::GenericDialogData

32-bit only

**virtual LONG CTSPILineConnection::GenericDialogData (LINEUIDIALOG* pLineDlg
LPVOID lpParam, DWORD dwSize);**

<i>pLineDlg</i>	LINEUIDIALOG structure which is associated with dialog.
<i>lpParam</i>	Parameter from the UI dialog.
<i>dwSize</i>	Size of the passed parameter block.

Remarks

This method is called when the user-interface component of the TSP is sending data back to the service provider, and the object type specified in the **TSPI_providerGenericDialogData** function was set to **TUISPIDLL_OBJECT_LINEID**.

Return Value

Standard TAPI return code. FALSE indicates success.

CTSPILineConnection::GetAddress

```
CTSPIAddressInfo* GetAddress (DWORD dwAddressID) const;  
CTSPIAddressInfo* GetAddress (LPCTSTR lpszDialableAddr) const;
```

<i>dwAddressID</i>	Numeric address id (zero-based) to locate.
<i>lpszDialableAddr</i>	Dialable phone number of the address to locate.

Remarks

This function searches the address array associated with the line connection and locates the address associated with the given parameter.

Return Value

Address object associated with the given parameter, NULL if not found.

CTSPILineConnection::GetAddressCount

```
DWORD GetAddressCount() const;
```

Remarks

This function returns the number of addresses which were created on the line using the **CreateAddress** function.

Return Value

Number of addresses on the line, this number will be one larger than the largest index which may be passed into the **GetAddress** method.

CTSPILineConnection::GetAddressID

```
virtual LONG GetAddressID(LPDWORD lpdwAddressId,  
                          DWORD dwAddressMode, LPCTSTR lpszAddress, DWORD dwSize);
```

<i>lpdwAddressID</i>	Returning address id which was found.
<i>dwAddressMode</i>	Type of address specified in lpszAddress .
<i>lpszAddress</i>	Dialable phone number of the address to locate.
<i>dwSize</i>	Size of the lpszAddress string.

Remarks

This function returns the address identifier associated with an address in the specified format on the line. This function is called in response to an application calling **lineGetAddressID**.

The default behavior for this function is to enumerate through all the addresses on the line object and locate the one which has the specified dialable address.

Return Value

TAPI error code or zero if the function was successful.

CTSPILineConnection::GetDefaultMediaDetection

DWORD GetDefaultMediaDetection() const;

Remarks

This function should be used by the service provider to determine which media modes that TAPI is interested in being notified about. Offering calls should not be presented to TAPI unless the media mode is present in the return value of this function.

When writing for third party telephony under TAPI 2.x, all calls should be offered to TAPI, and this function should only be used to see which ones have active owners.

Return Value

Last known media mode which was set by TAPI through the **TSPI_lineSetDefaultMediaDetection**.

CTSPILineConnection::GetDevConfig

**virtual LONG GetDevConfig(CString& strDeviceClass,
LPVARSTRING lpDeviceConfig);**

<i>strDeviceClass</i>	Device class which is being asked for.
<i>lpDeviceConfig</i>	Returning device configuration information.

Remarks

This function is called in response to an application calling **lineGetDevConfig**. It is used for device-specific extensions which are not provided through the general TAPI interface. It is not directly supported in the class library.

The service provider must override this function in order to provide the device-specific capability. The default implementation of the function returns an error.

Return Value

The class library returns **LINEERR_OPERATIONUNAVAIL**.

CTSPILineConnection::GetIcon

virtual LONG GetIcon (CString& strDevClass, LPHICON lphIcon);

<i>strDeviceClass</i>	Device class which is being asked for.
<i>lphIcon</i>	Returning icon handle.

Remarks

This function is called in response to an application calling **lineGetIcon**. It is used to return an icon specific to the line. It is not directly supported in the class library.

The service provider must override this function in order to provide the icon. The default implementation of the function returns an error. Note that TAPI itself will return an icon to the application if the service provider fails the function or does not support it.

Return Value

The class library returns **LINEERR_OPERATIONUNAVAIL**.

CTSPILineConnection::GetID

**virtual LONG GetID (CString& strDevClass, LPVARSTRING lpDeviceID,
HANDLE hTargetProcess);**

<i>strDeviceClass</i>	Device class which is being asked for.
<i>lpDeviceID</i>	Returning device information.
<i>hTargetProcess</i>	Process requesting data (32-bit only).

Remarks

This function is called in response to an application calling **lineGetID**. It is used to return device information to the application based on a string device class key. The 32-bit class library implements this function internally and returns any device information which was added using **CTSPILineConnection::AddDeviceClass**.

In the 32-bit version of the library, any handle which was given to the **AddDeviceClass** function is automatically duplicated in the given process for TAPI 2.x using the **DuplicateHandle** Win32 function.

Return Value

TAPI error code or zero if the function was successful.

CTSPILineConnection::GetLineDevCaps

LPLINEDEVCAPS GetLineDevCaps();

Remarks

This function returns the **LINEDEVCAPS** structure which is maintained for the line and returned by the **GatherCapabilities** method. Any of the non-pointer information may be read or modified through the resulting pointer, although TAPI is not notified if the information is modified directly. To make sure TAPI is correctly updated, use the appropriate member function of the **CTSPILineConnection** object to modify the capabilities of the line.

WARNING: Do not attempt to add information to the end of the structure, use the provided **CTSPILineConnection** methods to associate the additional pointer information with the line.

Return Value

Pointer to the **LINEDEVCAPS** structure.

CTSPILineConnection::GetLineDevStatus

LPLINEDEVSTATUS GetLineDevStatus();

Remarks

This function returns the **LINEDEVSTATUS** structure which is maintained for the line and returned by the **GatherStatus** method. Any of the non-pointer information may be read or modified through the resulting pointer, although TAPI is not notified if the information is modified directly. To make sure TAPI is correctly updated, use the appropriate member function of the **CTSPILineConnection** object to modify the status of the line.

If the service provider *does* modify the structure, and wants to notify TAPI, the **CTSPILineConnection::OnLineStatusChange** method should be used.

WARNING: Do not attempt to add information to the end of the structure, use the provided **CTSPILineConnection** methods to associate the additional pointer information with the line.

Return Value

Pointer to the **LINEDEVSTATUS** structure.

CTSPILineConnection::GetLineHandle

HTAPILINE GetLineHandle() const;

Remarks

This function returns the line handle which was assigned to this line device by TAPI when the line was opened.

Return Value

Line handle assigned by TAPI, or (-1) if the line is not currently open.

CTSPILineConnection::GetTerminalCount

```
int GetTerminalCount() const;
```

Remarks

This function returns the number of terminals which were added to this line using the **AddTerminal** function. For more information on terminals, see the section on *Terminals* in the line object description above.

Return Value

Count of terminals which are associated with this line object (zero if none).

CTSPILineConnection::GetTerminalInformation

```
DWORD GetTerminalInformation (int iTerminalID) const;
```

iTerminalID Terminal to return information for.

Remarks

This function returns the **LINETERMMODE_xxx** constants which are associated with the specified terminal identifier. The passed identifier should be between zero and the total count of terminals on the line.

Return Value

The **LINETERMMODE_xxx** constants which were associated with the terminal when it was added to the line.

CTSPILineConnection::GetUIDialog

```
LINEUIDIALOG* GetUIDialog (HTAPIDIALOGINSTANCE htDlgInstance);
```

htDlgInstance TAPI dialog instance to locate

Remarks

This function locates and returns the **LINEUIDIALOG** structure which is associated with the specified unsolicited user-interface dialog specified. The dialog will have been created using the **CreateUIDialog** function. This function is used internally and is documented in case the service provider needs access to the data structures associated with the user-interface system.

Return Value

The **LINEUIDIALOG** structure that is associated with the given TAPI handle, or NULL if no dialog was found.

CTSPILineConnection::Init

Protected

```
virtual VOID Init(CTSPIDevice* pDevice, DWORD dwLineDeviceID,  
                DWORD dwPosition);
```

<i>pDevice</i>	The device object that owns this line
<i>dwLineDeviceID</i>	The TAPI line device ID associated with this line
<i>dwPosition</i>	The position the line was added to in the device line array.

Remarks

This function is called directly after the constructor to initialize the line connection object. It is called during the **CServiceProvider::providerInit** function by the device owner (during the **CTSPIDevice::Init**), or when a new line is created using the **CTSPIDevice::CreateLine** function.

The service provider should override this function in order to adjust capabilities and add any addresses which are part of the line.

CTSPILineConnection::IsConferenceAvailable

Protected

```
BOOL IsConferenceAvailable(CTSPICallAppearance* pCall);
```

<i>pCall</i>	Call appearance which is looking for a conference.
--------------	--

Remarks

This method determines whether there is a conference call active on the same address and line as the call appearance passed. If the flag **LINEDEVCAPFLAGS_CROSSADDRCONF** is OR'd into the **LINEDEVCAPS.dwDevCapFlags** field, then the function expands the search across all the addresses on the line.

Return Value

TRUE/FALSE if a conference was found on either the same address or the same line (if cross address conferencing is available).

CTSPILineConnection::IsTransferConsultAvailable

Protected

```
BOOL IsTransferConsultAvailable(CTSPICallAppearance* pCall);
```

<i>pCall</i>	Call object to locate consultant call for
--------------	---

Remarks

This determines whether there is a consultant call or some other call which we could transfer to right now. It is used internally by the class library to determine basic call features when call states are changing.

Return Value

TRUE/FALSE if there is an available consultation call available on the same address or line as the given call.

CTSPILineConnection::MakeCall

```
virtual LONG MakeCall (DRV_REQUESTID dwRequestID, HTAPICALL htCall,  
                      LPHDRVCALL lpHdCall, TSPIMAKECALL* lpMakeCall);
```

<i>dwRequestID</i>	The asynchronous request ID to associate with this request.
<i>htCall</i>	The opaque TAPI handle to associate with the created call.
<i>lpHdCall</i>	The pointer to an area to store our created call handle.
<i>lpMakeCall</i>	The TSPIMAKECALL structure which details the call to make.

Remarks

This function is called in response to an application calling **lineMakeCall**. The **TSPILineMakeCall** function must be exported from the provider. The default behavior is to locate an appropriate address through the **CTSPILineConnection::FindAvailableAddress** function and the **LINECALLPARAMS** structure which is part of the passed **TSPIMAKECALL** structure. It will then create a call appearance on the address using **CTSPIAddressInfo::CreateCallAppearance**, and finally pass control off to the **CTSPICallAppearance::MakeCall** function to complete the processing of the request.

The final result of this function is that an asynchronous request of type **REQUEST_MAKECALL** will be generated and placed into the queue detailing the call appearance and dialing information.

Return Value

TAPI error code or the asynchronous request ID if the function was successful in creating the request.

CTSPILineConnection::OnAddressFeaturesChanged

Protected

```
virtual DWORD OnAddressFeaturesChanged (CTSPIAddressInfo* pAddr,  
                                       DWORD dwFeatures)
```

<i>pAddr</i>	The address object which changed its features.
<i>dwFeatures</i>	The new LINEADDRFEATURE_xxx bits which are valid.

Remarks

This function is called when an address associated with this line object changes its address features. The return value will be used as the final features reported to TAPI.

This function may be overridden to change features reported as necessary.

Return Value

Updated feature flags to report back to TAPI. The default behavior is to return the given **dwFeatures** passed to the function.

CTSPILineConnection::OnCallDeleted

Protected

```
virtual VOID OnCallDeleted(CTSPICallAppearance* pCall);
```

pCall The call appearance which is being destroyed.

Remarks

This function is called when a call associated with this line object is being deleted due to it being deallocated by all owner/monitor applications. Once this function returns, the call object pointer will be invalid and unavailable.

CTSPILineConnection::OnCallFeaturesChanged

Protected

```
virtual DWORD OnCallFeaturesChanged(CTSPICallAppearance* pCall,  
                                     DWORD dwCallFeatures);
```

pCall The call appearance which is changing.
dwCallFeatures The call features which are now in effect for the call.

Remarks

This function is called when a call associated with this line object is changing its feature information in the **LINECALLSTATUS** record. The return value will be used as the final features reported to TAPI.

This function may be overridden to change features reported as necessary.

Return Value

Updated feature flags to report back to TAPI. The default behavior is to return the given **dwCallFeatures** passed to the function.

CTSPILineConnection::OnCallStateChange

Protected

```
virtual VOID OnCallStateChange (CTSPIAddressInfo* pAddr,  
                                CTSPICallAppearance* pCall, DWORD dwNewState, DWORD dwOldState);
```

<i>pAddr</i>	The address that the call is associated with.
<i>pCall</i>	The call appearance which is being destroyed.
<i>dwNewState</i>	The new call state that the call is moving to.
<i>dwOldState</i>	The previous call state for this call.

Remarks

This function is called when a call associated with this line object is changing call states. It is called *before* any of the call features are updated. The class library uses this function to adjust the number of active, onHold, pending and idle call counts.

CTSPILineConnection::OnLineCapabilitiesChanged

```
virtual VOID OnLineCapabilitiesChanged();
```

Remarks

This function is called internally in the library when any of the information in the **LINEDEVCAPS** structure is changed for any reason. It notifies TAPI using a **LINEDEVSTATE_CAPSCHANGE** notification.

It should be called by the service provider if the **LINEDEVCAPS** structure is modified directly through the **CTSPILineConnection::GetDevCaps** function.

CTSPILineConnection::OnLineFeaturesChanged

Protected

```
virtual DWORD OnLineFeaturesChanged(DWORD dwLineFeatures);
```

<i>dwLineFeatures</i>	New line features for the line.
-----------------------	---------------------------------

Remarks

This function is called when any of the line features are altered on the line by the class library. The return value will be placed into the **LINEDEVSTATUS.dwLineFeatures** field and reported back to TAPI.

Return Value

Updated feature flags to report back to TAPI. The default behavior is to return the given **dwLineFeatures** passed to the function.

CTSPILineConnection::OnLineStatusChange

```
virtual VOID OnLineStatusChange (DWORD dwState, DWORD dwP2 = 0L,
                                DWORD dwP3 = 0L);
```

<i>dwState</i>	The new line state.
<i>dwP2</i>	Optional parameter depending on the new state.
<i>dwP3</i>	Optional parameter depending on the new state.

Remarks

This function is used internally in the library when any of the information in the **LINEDEVSTATUS** structure is changed using one of the **CTSPILineConnection** methods or through a child (address or call) object changing. It checks to see if the status change is being monitored by TAPI and then calls TAPI to notify it about the change to the device status if required.

It should be called by the service provider if the **LINEDEVSTATUS** structure is modified directly through the **CTSPILineConnection::GetDevStatus** function.

CTSPILineConnection::OnMediaConfigChanged

```
virtual VOID OnMediaConfigChanged();
```

Remarks

This function is used internally in the library when any of the media configuration information in the **LINEDEVSTATUS** structure is changed. It should be called by the service provider if the **LINEDEVSTATUS** structure is modified directly through the **CTSPILineConnection::GetDevStatus** function.

CTSPILineConnection::OnRequestComplete**Protected**

```
virtual VOID OnRequestComplete (CTSPIRequest* pReq, LONG IResult);
```

<i>pReq</i>	The request which is being completed.
<i>IResult</i>	The final return code for the request.

Remarks

This function is called when any request which is being managed by the line is completed by the service provider. It gives the line an opportunity to do any cleanup required with the request after it has terminated.

The default implementation of the class library manages several requests:

REQUEST_SETTERMINAL If the request completes successfully, the line connection will alter the internal terminal map and notify TAPI that the terminal information has changed in the **LINEDEVCAPS** record. This change will be cascaded to the address and calls that are owned by the line.

REQUEST_FORWARD If the request completes successfully, forwarding record(s) will be added to the **LINEDEVCAPS** structure and TAPI will be notified that the information has changed. All addresses on the line will be adjusted as well.

REQUEST_COMPLETECALL If the request completes successfully, a call completion request will be added to an internal array for notification of the final completion. It may be located using the **CTSPILineConnection::FindCallCompletionRequest** method and deleted using the **CTSPILineConnection::RemoveCallCompletionRequest** method.

REQUEST_UNCOMPLETECALL If the request completes successfully, the call completion request which was queued originally will be removed and deleted. TAPI is not notified since this is not considered a *device* request, rather a user request.

CTSPILineConnection::OnRingDetected

```
VOID OnRingDetected (DWORD dwRingMode,
                     BOOL fStopRinging = FALSE);
```

<i>dwRingMode</i>	The ring mode which was detected.
<i>fStopRinging</i>	TRUE if the final ring has been detected and there will be no more forthcoming. It will reset the ring count.

Remarks

This function should be called by the service provider when a ring event is detected on the line device. It will report the ring event through the **LINEDEVSTATE_RINGING** TAPI event. The ring count is maintained internally based on the ring mode (each ring mode supported on the line has a separate ring counter). The ring count should be reset either when the final ring is seen (if the PBX reports it), or as the first ring detected so that it is counted correctly.

The function is not called by the library - it is intended to be used by the service provider device code to report ring functionality.

CTSPILineConnection::Open

```
virtual LONG Open(HTAPILINE htLine, LINEEVENT lpfnEventProc,
                  DWORD dwTSPIVersion);
```

<i>htLine</i>	The opaque TAPI handle to assign to the line object.
<i>lpfnEventProc</i>	The callback to associate with this line.
<i>dwTSPIVersion</i>	The version the line negotiated at to shear structures to.

Remarks

This function is called by TAPI when the line device is opened. The default library implementation records the information associated with the line calls the **CTSPIDevice::OpenDevice** method to open the line. If that function fails, **LINEERR_RESOURCEUNAVAIL** is returned, otherwise, a success code is returned.

The **LINEDEVSTATUS** structure is updated to reflect an additional open and TAPI is notified that the **dwNumOpens** field has changed.

Return Value

TAPI error code or zero if the line was opened successfully.

CTSPILineConnection::RemoveCallCompletionRequest

```
VOID RemoveCallCompletionRequest(DWORD dwCompletionID,  
    BOOL fNotifyTAPI = FALSE);
```

<i>dwCompletionID</i>	The completion ID which is being removed/completed.
<i>fNotifyTAPI</i>	Whether TAPI should be notified about the removal.

Remarks

This function is called to remove a call completion record from the line. For more information on call completions, see the description for **CTSPILineConnection::FindCallCompletion**. If the service provider cancels a completion due to the PBX failing, it should notify TAPI through the **fNotifyTAPI** field. This will cause the class library to issue a **LINEDEVSTATE_COMPLCANCEL** event to TAPI. If the request is canceled through the initial **REQUEST_COMPLETECALL**, then TAPI is not notified of a call completion cancellation.

TAPI is automatically notified that the number of call completions has changed through a **LINEDEVSTATE_NUMCOMPLETIONS** event.

CTSPILineConnection::RemoveTerminal

```
VOID RemoveTerminal (int iTerminalID);
```

<i>iTerminalID</i>	The terminal identifier to remove.
--------------------	------------------------------------

Remarks

This function should be called by the service provider to remove a terminal which was added to the line through the **CTSPILineConnection::AddTerminal** function.

TAPI is notified that the number of terminals has changed through the **LINEDEVSTATE_TERMINALS** event.

This function will cause the terminal identifiers to be adjusted throughout the class library. All objects are notified through the **OnTerminalCountChanged** method and are adjusted accordingly.

CTSPILineConnection::SendDialogInstanceData

```
VOID SendDialogInstanceData (HTAPIDIALOGINSTANCE htDlgInstance,
                             LPVOID lpParams = NULL, DWORD dwSize = 0L);
```

<i>htDlgInstance</i>	Dialog instance to send the data to.
<i>lpParams</i>	Data to send to the dialog.
<i>dwSize</i>	Size of the data block to send to the dialog.

Remarks

This function is used to communicate with a unsolicited user-interface dialog which was created using the **CTSPILineConnection::CreateUIDialog** function. The **lpParams** buffer is automatically thunked to the process where the dialog is.

Note: Embedded pointers are not supported in the **lpParams** buffer.

CTSPILineConnection::Send_TAPI_Event

```
VOID Send_TAPI_Event(CTSPICallAppearance* pCall, DWORD dwMsg,
                     DWORD dwP1 = 0L, DWORD dwP2 = 0L, DWORD dwP3 = 0L);
```

<i>pCall</i>	Call object to reference in the event.
<i>dwMsg</i>	Event to send to TAPI (LINE_xxx message from <i>tapi.h</i>).
<i>dwP1</i>	Parameter dependant on the message.
<i>dwP2</i>	Parameter dependant on the message.
<i>dwP3</i>	Parameter dependant on the message.

Remarks

This function is used by the class library to notify TAPI about various events happening on the line. It issues events to the line callback which was supplied by TAPI when the line was opened.

It can be called by the service provider to support the various line notifications which are not directly supported by the class library (such as **LINE_DEVSPECIFIC**, or **LINE_DEVSPECIFICFEATURE**).

The function expects the following information for the following notifications:

LINE_CREATEDIALOGINSTANCE - The provider handle is taken from the input to **CServiceProvider::providerEnumDevices**. The function must be exported from the service provider.

LINE_SENDDIALOGINSTANCEDATA - The dialog instance is taken from the **dwP3** parameter.

CTSPILineConnection::SetBatteryLevel

VOID SetBatteryLevel (DWORD dwBattery);

dwBattery Battery level for the cellular or portable device. Should be between zero and (0xffff).

Remarks

This function is used to set the current battery level in the **LINEDEVSTATUS** structure. It will notify TAPI that the battery level has changed through a **LINEDEVSTATE_BATTERY** event.

It is not called internally by the library.

CTSPILineConnection::SetDefaultMediaDetection

Virtual LONG SetDefaultMediaDetection (DWORD dwMediaModes);

dwMediaModes Media modes TAPI is interested in.

Remarks

This function is called by TAPI when the media mode(s) being monitored for is changing. The library keeps track of the current requested media modes and implements this function automatically. The current media mode(s) is returned through the **CTSPILineConnection::GetDefaultMediaDetection** function.

The passed media mode flags are validated against the **LINEDEVCAPS.dwMediaModes** value which is obtained through the media modes supported by all the addresses on the line. If any of the passed media modes are not supported, **LINEERR_INVALIDMEDIAMODE** is returned.

Return Value

TAPI error code or zero if the function was successful.

CTSPILineConnection::SetDevConfig

**virtual LONG SetDevConfig(CString& strDeviceClass,
const LPVOID lpDeviceConfig, DWORD dwSize);**

strDeviceClass Device class which is being set.
lpDeviceConfig Device configuration information.
dwSize Size of the passed device configuration.

Remarks

This function is called in response to an application calling **lineSetDevConfig**. It is used for device-specific extensions which are not provided through the general TAPI interface. It is not directly supported in the class library.

The service provider must override this function in order to provide the device-specific capability. The default implementation of the function returns an error.

Return Value

The class library returns **LINEERR_OPERATIONUNAVAIL**.

CTSPILineConnection::SetDeviceStatusFlags

VOID SetDeviceStatusFlags (DWORD dwStatus);

dwStatus Device status flags to set on the line.

Remarks

This function can be called by the service provider to adjust the **LINEDEVSTATUS.dwDevStatusFlags** field. It automatically compares the existing flags with the newly passed flags and notifies TAPI for each change found in the state.

The following device status events can be reported based on changes to the device status flags:

LINEDEVSTATE_DISCONNECTED, **LINEDEVSTATE_CONNECTED**, **LINEDEVSTATE_MSGWAITOFF**, **LINEDEVSTATE_MSGWAITON**, **LINEDEVSTATE_INSERTSERVICE**, **LINEDEVSTATE_OUTOFSERVICE**, and **LINEDEVSTATE_LOCK**.

CTSPILineConnection::SetLineDevStatus

**virtual LONG SetLineDevStatus (DRV_REQUESTID dwRequestID,
 DWORD dwStatusToChange, BOOL fStatus);**

<i>dwRequestID</i>	Asynchronous request ID to associate with this request.
<i>dwStatusToChange</i>	Device status flags (LINEDEVSTATUSFLAGS_xxx) to set on the line device.
<i>fStatus</i>	TRUE or FALSE to turn the status on or off.

Remarks

This function is called in response to an application calling the **lineSetDevStatus** function. The **TSPI_lineSetDevStatus** function must be exported from the provider. The default implementation validates the **dwStatusToChange** against the **LINEDEVCAPS.dwSettableDevStatus** field and creates a **REQUEST_SETDEVSTATUS** asynchronous request and inserts it into the queue.

When the service provider processes the request, it should use the appropriate **CTSPILineConnection** methods to change the status of the device so that TAPI is notified correctly.

Return Value

TAPI error code or the asynchronous result code if the request was queued/started.

CTSPILineConnection::SetMediaControl

virtual LONG SetMediaControl (TSPI MEDIACONTROL* lpMediaControl);

lpMediaControl Media control structure associated with this request.

Remarks

This function is called in response to an application calling the **lineSetMediaControl** function. The **TSPI_lineSetMediaControl** function must be exported from the provider. It is responsible for enabling and disabling control actions on the media stream associated with this line and all addresses/calls present here. Media control actions can be triggered by the detection of specified digits, media modes, custom tones, and call states. The new specified media controls replace all the ones that were in effect for this line, address, or call prior to this request.

The default behavior of the class library is to invoke the **CTSPIAddressInfo::SetMediaControl** method on each address present on the line.

Return Value

TAPI error code zero if the function was successful.

CTSPILineConnection::SetRingMode

VOID SetRingMode (DWORD dwRingMode);

dwRingMode New ring mode for this line.

Remarks

This worker function can be called by the service provider to adjust the current ring mode for the line device. It adjusts the **LINEDEVSTATUS.dwRingMode** value and notifies TAPI through a **LINEDEVSTATE_OTHER** event. It is not invoked directly by the class library.

CTSPILineConnection::SetRoamMode

```
VOID SetRoamMode (DWORD dwRoamMode);
```

dwRoamMode New roam mode for this line.

Remarks

This worker function can be called by the service provider to change the roam mode for the line device. It adjusts the **LINEDEVSTATUS.dwRoamMode** value and notifies TAPI through a **LINEDEVSTATE_ROAMMODE** event. It is not invoked directly by the class library.

CTSPILineConnection::SetSignalLevel

```
VOID SetSignalLevel (DWORD dwSignal);
```

dwSignalLevel New signal level for this line (0 - 0xffff).

Remarks

This worker function can be called by the service provider to change the signal level for the line device. It adjusts the **LINEDEVSTATUS.dwSignalLevel** value and notifies TAPI through a **LINEDEVSTATE_SIGNAL** event. It is not invoked directly by the class library.

CTSPILineConnection::SetStatusMessages

```
virtual LONG SetStatusMessages(DWORD dwLineStates,  
                                DWORD dwAddressStates);
```

dwLineStates **LINEDEVSTATE_***xxx* events to route to TAPI.

dwAddressStates **LINEADDRESSSTATE_***xxx* events to route to TAPI.

Remarks

This function is called in response to an application calling **lineSetStatusMessages**. The **TSPI_lineSetStatusMessages** function must be exported from the provider. TAPI will keep track of all events being asked for and make a single call to this function with all requested events across all applications. The class library keeps track of this information and references it when notifying TAPI about any changes to the **LINEDEVCAPS** or **LINEDEVSTATUS** structures.

Return Value

TAPI error code zero if the function was successful.

CTSPILineConnection::SetTerminal

```
virtual LONG SetTerminal (DRV_REQUESTID dwReqID,  
    TSPILINESETTERMINAL* lpLine);
```

<i>dwReqID</i>	Asynchronous request ID to associate with this request.
<i>lpLine</i>	TSPILINESETTERMINAL structure with all the request information gathered and validated by the class library.

Remarks

This function is called in response to an application calling **lineSetTerminal**. The **TSPILineSetTerminal** function must be exported from the provider. The default function of the library is to create and insert a **REQUEST_SETTERMINAL** request into the queue. When the service provider processes this request and completes the request with a zero return code, the class library will store and set the new terminal modes within the library data structures using the **CTSPILineConnection::SetTerminalModes** function.

Return Value

TAPI error code, or the asynchronous request ID if the request was queued.

CTSPILineConnection::SetTerminalModes

```
VOID SetTerminalModes (int iTerminalID, DWORD dwTerminalModes,  
    BOOL fRouteToTerminal);
```

<i>iTerminalID</i>	Terminal identifier to adjust (0 to GetTerminalCount).
<i>dwTerminalModes</i>	Terminal mode(s) (LINETERMMODE_xxx) to adjust.
<i>fRouteToTerminal</i>	Whether the terminal is added or removed from modes.

Remarks

This is the function which should be called when a **REQUEST_SETTERMINAL** is completed by the service provider. This stores or removes the specified terminal from the terminal modes given, and then forces it to happen for any existing calls on the line by routing the notification through the **CTSPIAddressInfo::SetTerminalModes**.

TAPI is notified about the change through a **LINEDEVSTATE_TERMINALS** event.

CTSPILineConnection::UncompleteCall

```
virtual LONG UncompleteCall (DRV_REQUESTID dwRequestID,  
    DWORD dwCompletionID);
```

<i>dwRequestID</i>	Request ID which identifies this request.
<i>dwCompletionID</i>	Assigned completion identifier.

Remarks

This function is called when an application calls **lineUncompleteCall** to stop a call completion request. The **TSPI_lineUncompleteCall** function must be exported from the provider. The default library behavior is to validate the completion id and create an asynchronous request **REQUEST_UNCOMPLETECALL** with the original **TSPICOMPLETECALL** structure. Once the request is completed by the service provider with a zero return code, the call completion record is removed using **RemoveCallCompletionRequest** automatically.

Return Value

TAPI error code, or the asynchronous request ID if the request was queued.

CTSPILineConnection::ValidateMediaControlList

```
virtual LONG ValidateMediaControlList(  
    TSPIMEDIACONTROL* lpMediaControl) const;
```

lpMediaControl Structure with all the media control information validated.

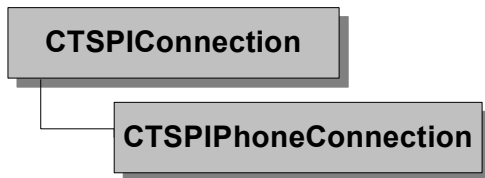
Remarks

This function is called when an application calls **lineSetMediaControl**. It is used to validate that the media control parameters are valid for this line device. It validates the information contained within the structure using the **LINEDEVCAPS** structure and TAPI rules concerning media control lists.

Return Value

TAPI error code, or zero if the media control structure is valid.

CTSPiPhoneConnection



A **CTSPiPhoneConnection** object represents a phone device in the library. This object is maintained in a list by the device object. There will generally only be a single phone device per service provider, but the library itself imposes no limitations on the number of phone objects added to the device object. A phone device is defined by the TAPI specification as a device that has some or all of the following capabilities:

Transducer - This is a means for audio input/output. TAPI recognizes that a phone device may have several transducers, which can be taken off-hook or placed on-hook under application or manual user control. TAPI supports three basic types of hook switch devices common to many phone sets:

1. *Handset* - The traditional mouth-and-ear piece combination on most phones.
2. *Speakerphone* - Enables the user to conduct hands-free calls.
3. *Headset* - Standard operator-style headset.

A hook switch must be off-hook to allow audio data to be sent to and/or received by the corresponding transducer.

Volume/Gain/Mute - Each hook switch device is a pairing of a speaker and a microphone. The service provider can provide for volume control and muting of speaker components and for gain control or muting of microphone components if the device supports it.

Ringer - A means for alerting users, usually through a bell or audible tone. A phone device may be able to ring in a variety of modes or patterns.

Display - A mechanism for visually presenting messages to the user.

Buttons - An array of push-buttons on the phone. Button-Lamp IDs identify a button and lamp pair. It is possible to have button-lamp pairs with either no button or no lamp. Button-lamp IDs are values that range from 0 to the maximum number of button-lamps available on the phone device, minus one. Each button belongs to a button class. Classes include:

1. Call appearances
2. Feature buttons (such as Forward, or Transfer)
3. Digit keypad buttons (0-9, *, #)
4. Local buttons which control speakerphone volume and other phone set options.

Lamps - An array of lamps individually controllable from the service provider. Lamps can be lit in different modes by varying the on and off frequency. The button-lamp ID identifies the lamp. As a lamp changes states, the service provider needs to notify TAPI.

Data Areas - Memory areas in the phone device where instruction code or data can be downloaded to and/or uploaded from.

The TAPI dynamic link library allows client applications to monitor and control elements of the phone device. The most useful elements for an application are the hook switch devices. The phone set can act as an audio I/O device to the computer, with volume control, gain control, a ringer, data areas (for programming the phone), and a display. Some of these items may not be present, or all of them might be present.

There is no guaranteed core set of services supported by all phone devices. Therefore, the capabilities of the phone device may be very minimal, or have all the TSPI functions supported. These telephony capabilities will vary based on the configuration (client versus client-server or the telephone hardware).

Phone Initialization

During the initialization of a **CTSPiPhoneConnection** object, the initial phone capabilities are setup in the **PHONECAPS** record stored in the object. The capabilities initialized are very minimal, and the derived service provider is responsible for supplying the remainder of the information, either through the **AddXXX** methods, or by adjusting the information in the **PHONECAPS** structure directly. The fields that will most commonly be adjusted will be:

Structure member	Description
dwNumRingModes	This is initialized to one. If more ring patterns are supported on the device, this needs to be set the correct value.

Additional information in the record is set using the **AddUploadBuffer**, **AddDownloadBuffer**, **SetupDisplay**, **AddButton**, **SetPhoneInfo**, and **AddHookSwitchDevice**. These should be called during the **CServiceProvider::providerInit** method so that the information is ready when TAPI opens the phone device.

Status Information

Status information for the phone device is stored in an embedded **PHONESTATUS** structure. The library manages all elements of the record. A pointer to this record may be obtained through the **GetPhoneStatus** member function.

Device Specific Extensions

If the phone device supports device-specific extensions to the **PHONECAPS** or **PHONESTATUS** records, then they must be added either by overriding the **CServiceProvider** method **phoneGetDevCaps**, or by overriding the **CTSPiPhoneConnection** method **GatherCapabilities**.

Device specific functions may be added by overriding either **CServiceProvider::phoneDeviceSpecific** or **CTSPiPhoneConnection::DeviceSpecific**.

Warning:

*Do not add the device specific information into the static **LINEDEVCAPS**, there is no extra-allocated space!*

Operations - Public Members

AddButton	This adds a new button or lamp (or both) to the button list maintained in the phone object. The PHONECAPS structure will be modified appropriately.
AddDisplayChar	This adds a character to the display at the current cursor position. The cursor position is then adjusted.
AddDisplayString	This adds a string to the display, automatically moving the cursor (both X and Y when necessary).
AddDownloadBuffer	This will add a new download buffer to our phone object. This will adjust the PHONECAPS structure appropriately. The download buffer can be manipulated through the phoneGetPhoneData method.
AddHookSwitchDevice	This adds a hookswitch device to our list maintained by the phone object. The PHONECAPS structure will be modified appropriately.
AddUploadBuffer	This will add a new upload buffer to our phone object. This will adjust the PHONECAPS structure appropriately. The upload buffer can be manipulated through the phoneSetPhoneData method.

ClearDisplayLine	This clears (blanks with spaces) a line on the display.
ForceClose	This forces the phone to be closed immediately.
GetButtonCount	This returns the count of buttons on the phone.
GetButtonInfo	This returns a pointer to the CPhoneButtonInfo object which represents a button on the phone. The pointer is declared const and therefore cannot be used to change the button information. Use the SetButtonInfo method for changes to the button.
GetCursorPos	This returns the X/Y position of the cursor on the display. The top-left coordinates are always (0,0).
GetDisplayBuffer	This returns a buffer representing our current display state.
GetLampMode	This returns the lamp mode for a particular lamp/button identifier.
GetPhoneCaps	This returns a pointer to our PHONECAPS record. The data may be modified through this pointer.
GetPhoneHandle	Returns the TAPI opaque phone handle associated with this phone device. If the device is not open, (-1) will be returned.
GetPhoneStatus	This returns a pointer to our PHONESTATUS record. The data may be modified through this pointer.
ResetDisplay	This method resets the display to spaces, and sets the cursor position to top-left.
Send_TAPI_Event	This method sends an event request to TAPI.
SetButtonInfo	This is called to modify the button information for a button in our button list. This should only be called if the device changes the button information itself (without the service provider). The library will automatically call this if a REQUEST_SETBUTTONINFO asynchronous request completes successfully. TAPI is notified through the callback.
SetButtonState	This changes the state of a button in our button list. This should be called if the button state is changed by the device (i.e. a user pressing a function button on the physical phone). TAPI is notified through the callback.
SetDisplay	This allows the service provider to set the entire phone display to a value.
SetDisplayChar	This allows a specific character position on the display to be modified. The cursor position is not changed.
SetDisplayCursorPos	This changes the cursor position to the specified X/Y coordinate location. No data is changed on the display.
SetupDisplay	This allocates a buffer which will be used to represent the phone display. The display is buffered in the phone object so that no request to the phone will be required when TAPI requests the display information. If a display is present, this method should be called. The PHONECAPS record will be modified appropriately.
SetGain	This changes the microphone gain of a hookswitch device in the PHONESTATUS record. This should only be called when the device has changed the hookswitch gain. The library will automatically call this when a REQUEST_SETHOOKGAIN completes successfully. TAPI is notified through the callback.
SetHookSwitch	This changes the state of a hookswitch in the PHONESTATUS record. This should only be called when the device has changed the hookswitch state. The

SetLampState	library will automatically call this when a REQUEST_SETHOOKSWITCH completes successfully. TAPI is notified through the callback. This changes the state of a lamp in our button list. This should only be called if the lamp state is changed by the device directly. This is automatically called by the library when a REQUEST_SETLAMP asynchronous request completes successfully. TAPI is notified through the callback.
SetPhoneFeatures	This changes the PHONESTATUS phone features available for TAPI.
SetRingMode	This changes the ring mode in the PHONESTATUS record. This should only be called when the device changes the ring mode. The library will automatically call this method when a REQUEST_SETRING completes. TAPI is notified through the callback.
SetRingVolume	This changes the current ringer volume in the phone object. This should only be called when the device changes the ring volume. The library will automatically call this method when a REQUEST_SETRING completes. TAPI is notified through the callback.
SetStatusFlags	This changes the dwStatusFlags field in the PHONESTATUS record. TAPI is notified through the callback.
SetVolume	This changes the volume of a hookswitch device in the PHONESTATUS record. This should only be called when the device has changed the hookswitch volume. The library will automatically call this when a REQUEST_SETHOOKSWITCHVOL completes successfully. TAPI is notified through the callback.

Operations - Protected Members

CanHandleRequest	This method calls the CServiceProvider::CanHandleRequest method.
-------------------------	---

Overridables - Protected Members

Init	This method is called directly after the constructor to initialize the phone object.
OnButtonStateChange	This method is called when any button changes states (i.e. UP/DOWN). It notifies TAPI about the change through the asynchronous callback.
OnPhoneCapabilitiesChanged	This is called when any information in the PHONECAPS record changes.
OnPhoneFeaturesChanged	This is called when the phone features change in the PHONESTATUS record.
OnPhoneStatusChange	This is called when any information in our PHONESTATUS record changes. It notifies TAPI about the change through the asynchronous callback.
OnRequestComplete	This method is called when any asynchronous request for this phone device has completed. It gives the device an opportunity to examine the return code and change settings in the phone object.

Overridables - TAPI functions

Close	This is called to close the phone device. It will decrement the usage of its parent CTSPIDevice object.
ConfigDialog	This is called to invoke a UI window to configure the phone device in response to a phoneConfigDialog request. The default implementation returns OperationUnavail .
DevSpecificFeature	This is called to invoke device-specific commands on the phone in response to a phoneDevSpecific request. The default implementation returns OperationUnavail .
GatherCapabilities	This method gathers the PHONECAPS record for a phoneGetDevCaps request. It is handled completely by the library.
GatherStatus	This method gathers the PHONESTATUS record for a phoneGetDevStatus request. It is handled completely by the library.
GenericDialogData	This method is called to handle user-interface dialog data being passed from a UI event.
GetButtonInfo	This method is called in response to a phoneGetButtonInfo request. It is handled completely by the library.
GetData	This method is called in response to a phoneGetData request. It generates a REQUEST_GETPHONEDATA request.
GetDisplay	This method is called in response to a phoneGetDisplay request. It is handled completely by the library.
GetGain	This method is called in response to a phoneGetGain request. It is handled completely by the library.
GetHookSwitch	This method is called in response to a phoneGetHookSwitch request. It is handled completely by the library.
GetIcon	This function is called in response to a phoneGetIcon request. The default implementation returns OperationUnavail .
GetID	This is called in response to a phoneGetID request. It returns resource handle and device information about the phone based on a passed device class. The default implementation supports the tapi/phone class.
GetLamp	This method is called in response to a phoneGetLamp request. It is handled completely by the library.
GetRing	This method is called in response to a phoneGetRing request. It is handled completely by the library.
GetVolume	This method is called in response to a phoneGetVolume request. It is handled completely by the library.
Open	This is called to open the phone device. It will increment the usage of its parent CTSPIDevice object.
SetButtonInfo	This method is called in response to a phoneSetButtonInfo request. It generates an asynchronous request for the worker code to complete. If the request completes successfully, the information will be stored back to our button object.
SetData	This method is called in response to a phoneSetData request. It generates a REQUEST_SETPHONEDATA request.
SetDisplay	This method is called in response to a phoneSetDisplay

	request. It generates a REQUEST_SETDISPLAY request.
SetGain	This method is called in response to a phoneSetGain request. It generates a REQUEST_SETHOOKSWITCHGAIN request.
SetHookSwitch	This method is called in response to a phoneSetHookSwitch request. It generates a REQUEST_SETHOOKSWITCH request.
SetLamp	This method is called by the phoneSetLamp method to change the status of a lamp indicator. It generates a REQUEST_SETLAMP asynchronous request.
SetRing	This method is called in response to a phoneSetRing request. It generates a REQUEST_SETRING request.
SetStatusMessages	This method restricts the messages which will be sent by the library for changes to the PHONESTATUS record. It is handled completely by the library.
SetVolume	This method is called in response to a phoneSetVolume request. It generates a REQUEST_SETHOOKSWITCHVOL request.

CTSPiPhoneConnection::AddButton

```
int AddButton (DWORD dwFunction, DWORD dwMode,  
              DWORD dwAvailLampStates, DWORD dwLampState, LPCTSTR lpszText);
```

<i>dwFunction</i>	Button function (PHONEBUTTONFUNCTION_ xxx).
<i>dwAvailLampStates</i>	Available lamp states (PHONELAMPMODE_ xxx).
<i>dwLampState</i>	Current lamp state (from available states).
<i>lpszText</i>	ASCII Text for button.

Remarks

This function adds a button to the phone model presented to TAPI. This should be called when the phone is added to the system (such as during **CTSPiPhoneConnection::Init**). Each button added to the phone will be reported through the **PHONEDEVCAPS** and TAPI button information functions.

Return Value

The position of the button within the internal button array.

CTSPiPhoneConnection::AddDisplayChar

```
VOID AddDisplayChar (TCHAR cChar);
```

<i>cChar</i>	Character to add to the display.
--------------	----------------------------------

Remarks

This function adds a new character to the display at the current cursor position. If the provider needs to control the position that the character is inserted at, use the **CTSPiPhoneConnection::SetDisplayChar** function.

CTSPiPhoneConnection::AddDisplayString

```
VOID AddDisplayString (LPCTSTR pszDisplay);
```

<i>pszDisplay</i>	NULL terminated string to add to the display.
-------------------	---

Remarks

This function adds a new string to the display at the current cursor position. The linefeed character given in the **CTSPiPhoneConnection::SetupDisplay** function is interpreted as a movement to the next row, first column.

CTSPiPhoneConnection::AddDownloadBuffer

```
int AddDownloadBuffer (DWORD dwSizeOfBuffer);
```

dwSizeOfBuffer Size of the download buffer to add to the phone.

Remarks

This function adds a download buffer to the phone model presented to TAPI. This should be called when the phone is added to the system (such as during **CTSPiPhoneConnection::Init**). Each buffer added to the phone will be reported through the **PHONEDEVcaps** structure.

To add a download buffer, the TSP must export either the **TSPI_phoneGetData** or the **TSPI_phoneSetData** function.

Return Value

The position of the buffer within the internal tracking array.

CTSPiPhoneConnection::AddHookSwitchDevice

```
VOID AddHookSwitchDevice (DWORD dwHookSwitchDev,  
    DWORD dwAvailModes, DWORD dwCurrMode,  
    DWORD dwVolume =-1L, DWORD dwGain =-1L,  
    DWORD dwSettableModes =-1L, DWORD dwMonitoredModes =-1L);
```

<i>dwHookSwitchDev</i>	Device type (PHONEHOOKSWITCHDEV_XXX).
<i>dwAvailModes</i>	Hookswitch Modes supported (PHONEHOOKSWITCHMODE_XXX).
<i>dwVolume</i>	Current volume (-1 if volume level changes are not supported).
<i>dwGain</i>	Current gain (-1 if gain level changes are not supported).
<i>dwSettableModes</i>	Hookswitch modes which can be set.
<i>dwMonitoredModes</i>	Hookswitch modes which can be monitored.

Remarks

This function adds a hookswitch device to the phone model presented to TAPI. This should be called when the phone is added to the system (such as during **CTSPiPhoneConnection::Init**). Each hookswitch added to the phone will be reported through the **PHONEDEVcaps** structure.

CTSPiPhoneConnection::AddUploadBuffer

```
int AddUploadBuffer (DWORD dwSizeOfBuffer);
```

dwSizeOfBuffer Size of the upload buffer to add to the phone.

Remarks

This function adds an upload buffer to the phone model presented to TAPI. This should be called when the phone is added to the system (such as during **CTSPiPhoneConnection::Init**). Each buffer added to the phone will be reported through the **PHONEDEVCAPS** structure.

To add an upload buffer, the TSP must export either the **TSPI_phoneGetData** or the **TSPI_phoneSetData** function.

Return Value

The position of the buffer within the internal tracking array.

CTSPiPhoneConnection::CanHandleRequest

Protected

```
BOOL CanHandleRequest(WORD wRequest, DWORD dwData = 0);
```

wRequest Request type to check (**REQUEST_XXX** from *splib.h*).
dwData Optional data to pass through library.

Remarks

This method is called to verify that a specific request type can be processed by the TSP. Eventually, it ends up calling **CServiceProvider::CanHandleRequest**, which is a virtual function. The optional data member is not used by the class library, but can be used by the TSP when overriding the **CServiceProvider** member.

This function is called in response to any request made by TAPI against the service provider. It verifies that the request is valid at that moment.

Return Value

TRUE/FALSE success code. FALSE is returned if the function is not supported. This generally will cause the request for which this function was called to fail.

CTSPiPhoneConnection::ClearDisplayLine

```
VOID ClearDisplayLine (int iRow);
```

iRow Row on the display to clear.

Remarks

This method clears a single row on the display.

CTSPiPhoneConnection::Close()

```
virtual LONG Close();
```

Remarks

This method is invoked by TAPI when the phone is closed by all applications. It calls the **CTSPiDevice::CloseDevice** method and resets all the phone handle information. Once this function completes, the phone will have no further interaction with TAPI until it is opened again.

Return Value

This function returns a standard TAPI return code.

CTSPiPhoneConnection::CTSPiPhoneConnection()

```
CTSPiPhoneConnection();
```

Remarks

This is the constructor to the phone object.

CTSPiPhoneConnection::~~CTSPiPhoneConnection()

```
~CTSPiPhoneConnection();
```

Remarks

This is the destructor to the phone object.

CTSPISPhoneConnection::DevSpecificFeature

```
virtual LONG DevSpecificFeature(DRV_REQUESTID dwRequestID,  
                                LPVOID lpParams, DWORD dwSize);
```

<i>dwRequestID</i>	Asynchronous request ID associated with this request.
<i>lpParams</i>	Optional pointer to a parameter block
<i>dwSize</i>	Size of the above parameter block

Remarks

This method is called by the library when an application calls **phoneDevSpecificFeature** and identifies this phone object in the **HPHONE** parameter. The **TSPI_phoneDevSpecificFeature** function must be exported from the provider. This function enables service providers to provide access to features not typically offered by TAPI. The meaning of these extensions are device specific and known only to the service provider and applications which are written to take advantage of them. This function is not handled directly by the library and is provided simply for overriding purposes.

Return Value

You must override this function to provide device-specific functionality. The library returns **PHONEERR_OPERATIONUNAVAIL** if you do not override this function.

CTSPISPhoneConnection::ForceClose

```
void ForceClose();
```

Remarks

This method may be called by the service provider to close the phone device immediately regardless of it being in-use or not. It should be used when the device is going offline for some hardware reason or because a component or network connection has been lost.

CTSPiPhoneConnection::GatherCapabilities

**virtual LONG GatherCapabilities (DWORD dwTSPIVersion, DWORD dwExtVer,
LPPHONECAPS lpPhoneCaps);**

<i>dwTSPIVersion</i>	The TAPI version which structures should reflect.
<i>dwExtVer</i>	The extension version which structures should reflect.
<i>lpPhoneCaps</i>	The structure to fill with phone capabilities.

Remarks

This function is called when an application requests the phone capabilities using the **phoneGetDevCaps** function. The class library automatically fills in all the known information about the phone using the phone and all the associated button, lamp, and display information.

The service provider can adjust the capabilities returned by using the **GetPhoneCaps** function and modifying the returning structure. This should typically be done when the phone is first initialized as capabilities normally do not change during the life of the provider.

If the structure is modified through the **GetPhoneCaps** function, TAPI is not automatically notified.

This function should be overridden if the service provider supports device extensions and wishes to return the information in the **PHONECAPS** structure.

Return Value

TAPI Error code or zero if the function was successful.

CTSPIPhoneConnection::GatherStatus

virtual LONG GatherStatus (LPPHONESTATUS IpStatus);

lpStatus The structure to fill with line status.

Remarks

This function is called when an application requests the current line status using the **phoneGetStatus** function. The class library automatically fills in all the known information about the phone using the phone and all the associated button, lamp, and display information.

The service provider can adjust the capabilities returned by using the **GetPhoneStatus** function and modifying the returning structure. Some of the information can be modified using other **CTSPiPhoneConnection** methods. These include **SetButtonInfo**, **SetLampState**, **SetButtonState**, **SetStatusFlags**, **SetRingMode**, **SetRingVolume**, **SetHookSwitch**, **SetVolume**, and **SetGain**. Using these methods is recommended as TAPI is notified about the information changing through a **PHONEDEVSTATUS** callback.

If the structure is modified through the **GetPhoneStatus** function, TAPI is not automatically notified.

This function should be overridden if the service provider supports device extensions and wishes to return the information in the **PHONESTATUS** structure.

Return Value

TAPI Error code or zero if the function was successful.

CTSPIPhoneConnection::GenericDialogData

32-bit only

```
virtual LONG CTSPiPhoneConnection::GenericDialogData (LPVOID lpParam,
    DWORD dwSize);
```

<i>lpParam</i>	Parameter from the UI dialog.
<i>dwSize</i>	Size of the passed parameter block.

Remarks

This method is called when the user-interface component of the TSP is sending data back to the service provider, and the object type specified in the **TSPi_providerGenericDialogData** function was set to **TUISPIDLL OBJECT PHONEID**.

Return Value

Standard TAPI return code. FALSE indicates success.

CTSPiPhoneConnection::GetButtonCount

```
int GetButtonCount() const;
```

Remarks

This method can be used by the service provider to determine the count of buttons which have been added to the phone using the **CTSPiPhoneConnection::AddButton** function.

Return Value

Count of buttons.

CTSPiPhoneConnection::GetButtonInfo

```
const CPhoneButtonInfo* GetButtonInfo(int iButtonID) const;
```

iButtonID Button index (zero-based)

Remarks

This method retrieves information about the specified button. The index runs from zero to the count of buttons (see **CTSPiPhoneConnection::GetButtonCount**). The returning object is an internal class library representation of a button.

Return Value

Pointer to a **CPhoneButtonInfo** object which represents this button.

CTSPiPhoneConnection::GetButtonInfo

```
virtual LONG GetButtonInfo (DWORD dwButtonId,  
                             LPPHONEBUTTONINFO lpButtonInfo);  
dwButtonId                      Lamp/Button identifier  
lpButtonInfo                    TAPI PHONEBUTTONINFO structure.
```

Remarks

This method is called when an application calls **phoneGetButtonInfo**. It is responsible for returning the button information for the specified lamp/button identifier. The default implementation returns all the required information from the internal button information object.

Return Value

TAPI error code or zero if the function was successful.

CTSPiPhoneConnection::GetCursorPos

CPoint GetCursorPos() const;

Remarks

This method can be used by the service provider to determine where the current cursor position on the display is. This should only be called if a display has been setup by the phone object (see **CTSPiPhoneConnection::SetupDisplay**).

Return Value

TAPI error code or zero if the function was successful.

CTSPiPhoneConnection::GetData

virtual LONG GetData (TSPIPHONEDATA* pPhoneData);

pPhoneData Structure which identifies buffer to retrieve data from.

Remarks

This method is called when an application uses the **phoneGetData** function to retrieve data from a configured phone buffer. The default implementation generates a **REQUEST_GETPHONEDATA** request.

Note that this is a special case since the function is *not* asynchronous according to the TAPI specification. Instead, the library issues the request and then blocks the thread until the request completes.

Return Value

TAPI error code or zero if the function was successful.

CTSPiPhoneConnection::GetDisplay

virtual LONG GetDisplay (LPVARSTRING lpVarString);

lpVarString Returning data pointer for display.

Remarks

This method is called when an application uses the **phoneGetDisplay** function to retrieve the display information for the phone. The class library fills in the buffer with the current representation of the display based on what has been added/updated using the **CTSPiPhoneConnection::AddDisplayChar** and **CTSPiPhoneConnection::AddDisplayString** functions.

Return Value

TAPI error code or zero if the function was successful.

CTSPiPhoneConnection::GetDisplayBuffer

CString GetDisplayBuffer() const;

Remarks

This method can be used by the service provider to read the current representation of the display buffer. The returning string is read-only. Any changes made to it will *not* be reflected in the display. If the service provider needs to make changes to the display, it should use the various update methods such as **CTSPiPhoneConnection::AddDisplayChar** and **CTSPiPhoneConnection::AddDisplayString**.

Return Value

TAPI error code or zero if the function was successful.

CTSPISPhoneConnection::GetGain

```
virtual LONG GetGain (DWORD dwHookSwitchDevice,  
                     LPDWORD lpdwGain);
```

dwHookSwitchDevice
lpdwGain

Hookswitch device to query the gain for.
Buffer to fill in with current gain value.

Remarks

This method is called when an application uses the **phoneGetGain** function. It is implemented in the class library to retrieve the current gain setting for the specific hook switch device. If the hook switch device doesn't support gain, an error is returned.

Return Value

TAPI error code or zero if the function was successful.

CTSPISPhoneConnection::GetHookSwitch

```
virtual LONG GetHookSwitch (LPDWORD lpdwHookSwitch);
```

lpdwHookSwitch

Returning hookswitch values.

Remarks

This method is called when an application uses the **phoneGetHookSwitch** function. It is implemented in the class library to check the **PHONESTATUS.dwHandsetHookSwitchMode**, **PHONESTATUS.dwHeadsetHookSwitchMode**, and **PHONESTATUS.dwSpeakerHookSwitchMode** and return all the hookswitch devices which are currently active on the phone device. If any of the devices are available and have either MIC, SPEAKER, or both active, then the hookswitch device is returned in the **lpdwHookSwitch** variable.

Return Value

TAPI error code or zero if the function was successful.

CTSPiPhoneConnection::GetIcon

virtual LONG GetIcon (CString& strDevClass, LPHICON lphIcon);

<i>strDeviceClass</i>	Device class which is being asked for.
<i>lphIcon</i>	Returning icon handle.

Remarks

This function is called in response to an application calling **phoneGetIcon**. It is used to return an icon specific to the phone. It is not directly supported in the class library.

The service provider must override this function in order to provide the icon. The default implementation of the function returns an error. Note that TAPI itself will return an icon to the application if the service provider fails the function or does not support it.

Return Value

The class library returns **PHONEERR_OPERATIONUNAVAIL**.

CTSPiPhoneConnection::GetID

**virtual LONG GetID (CString& strDevClass, LPVARSTRING lpDeviceID,
HANDLE hTargetProcess);**

<i>strDeviceClass</i>	Device class which is being asked for.
<i>lpDeviceID</i>	Returning device information.
<i>hTargetProcess</i>	Process requesting data (32-bit only).

Remarks

This function is called in response to an application calling **phoneGetID**. It is used to return device information to the application based on a string device class key. The 32-bit class library implements this function internally and returns any device information which was added using **CTSPConnection::AddDeviceClass**.

In the 32-bit version of the library, any handle which was given to the **AddDeviceClass** function is automatically duplicated in the given process for TAPI 2.x using the **DuplicateHandle** Win32 function.

Return Value

TAPI error code or zero if the function was successful.

CTSPiPhoneConnection::GetLamp

```
virtual LONG GetLamp (DWORD dwButtonId, LPDWORD lpdwLampMode);
```

dwButtonId Button/Lamp identifier.

lpdwLampMode Returning **PHONELAMPMODE_***xxx* constants.

Remarks

This function is called in response to an application calling **phoneGetLamp**. It is used to return the current state of a specific lamp. The class library implements this using the current setting of the lamps from the internal lamp/button array. The service provider should not call this function, if it needs to know the state of a lamp, use the **CTSPiPhoneConnection::GetLampMode** function.

Return Value

TAPI error code or zero if the function was successful.

CTSPiPhoneConnection::GetLampMode

```
DWORD GetLampMode (int iButtonId);
```

iButtonId Button/Lamp identifier.

Remarks

This function can be used by the service provider to determine what the current state of a particular lamp is. It returns the current **PHONELAMPMODE_***xxx* constant which reflect the current lamp condition.

Return Value

Current lamp mode for the specified lamp/button identifier.

CTSPiPhoneConnection::GetPhoneCaps

LPPHONECAPS GetPhoneCaps();

Remarks

This function returns the **PHONECAPS** structure which is maintained for the phone and returned by the **GatherCapabilities** method. Any of the non-pointer information may be read or modified through the resulting pointer, although TAPI is not notified if the information is modified directly. To make sure TAPI is correctly updated, use the appropriate member function of the **CTSPiPhoneConnection** object to modify the capabilities of the phone.

WARNING: Do not attempt to add information to the end of the structure, use the provided **CTSPiPhoneConnection** methods to associate the additional pointer information with the phone.

Return Value

Pointer to the **PHONECAPS** structure.

CTSPiPhoneConnection::GetPhoneHandle

HTAPIPHONE GetPhoneHandle() const;

Remarks

This function returns the phone handle which was assigned to the phone connection when it was opened by TAPI.

Return Value

TAPI phone handle or (-1) if the phone is not yet open.

CTSPiPhoneConnection::GetPhoneStatus

LPPHONESTATUS GetPhoneStatus();

Remarks

This function returns the **PHONESTATUS** structure which is maintained for the phone and returned by the **GatherStatus** method. Any of the non-pointer information may be read or modified through the resulting pointer, although TAPI is not notified if the information is modified directly. To make sure TAPI is correctly updated, use the appropriate member function of the **CTSPiPhoneConnection** object to modify the status of the phone.

If the service provider *does* modify the structure, and wants to notify TAPI, the **CTSPiPhoneConnection::OnPhoneStatusChange** method should be used.

WARNING: Do not attempt to add information to the end of the structure, use the provided **CTSPiPhoneConnection** methods to associate the additional pointer information with the phone.

Return Value

Pointer to the **PHONESTATUS** structure.

CTSPiPhoneConnection::GetRing

virtual LONG GetRing (LPDWORD lpdwRingMode, LPDWORD lpdwVolume);

lpdwRingMode
lpdwVolume

Returning ring mode for the phone.
Returning ring volume for the phone.

Remarks

This function is called in response to an application calling **phoneGetRing**. The class library implements this function internally by returning the current known ring value. The service provider can adjust this using the **CTSPiPhoneConnection::SetRingMode** and **CTSPiPhoneConnection::SetRingVolume** methods.

Return Value

TAPI error code or zero if successful.

CTSPiPhoneConnection::GetVolume

```
virtual LONG GetVolume (DWORD dwHookSwitchDev, LPDWORD lpdwVolume);
```

dwHookSwitchDev
lpdwVolume

Hookswitch device to query.
Returning speaker volume for the hookswitch.

Remarks

This function is called in response to an application calling **phoneGetVolume**. The class library implements this function internally by returning the current known volume for the specified hookswitch. The service provider can adjust this using the **CTSPiPhoneConnection::SetVolume** method.

Return Value

TAPI error code or zero if successful.

CTSPiPhoneConnection::Init

Protected

```
virtual VOID Init(CTSPiDevice* pDevice, DWORD dwPhoneDeviceID,  
                DWORD dwPosition);
```

<i>pDevice</i>	The device object that owns this line
<i>dwPhoneDeviceID</i>	The TAPI phone device ID associated with this phone
<i>dwPosition</i>	The position the phone was added to in the device phone array.

Remarks

This function is called directly after the constructor to initialize the phone connection object. It is called during the **CServiceProvider::providerInit** function by the device owner (during the **CTSPiDevice::Init**), or when a new phone is created using the **CTSPiDevice::CreatePhone** function.

The service provider should override this function in order to adjust capabilities and add any buttons, lamps, hookswitch devices, and display information to the phone.

CTSPiPhoneConnection::OnButtonStateChange

Protected

```
virtual VOID OnButtonStateChange (DWORD dwButtonID, DWORD dwMode,  
    DWORD dwState);
```

<i>dwButtonID</i>	Button identifier which has changed.
<i>dwMode</i>	The mode for the button.
<i>dwState</i>	The new <i>pressed</i> state for the button.

Remarks

This function is called when the state of a button changes on the phone. The class library informs TAPI that the button has changed using the **PHONE_BUTTON** event.

CTSPiPhoneConnection::OnPhoneCapabilitiesChanged

Protected

```
virtual VOID OnPhoneCapabilitiesChanged();
```

Remarks

This function is called when the **PHONECAPS** structure is changed. The service provider should also call this function when it modifies the **PHONECAPS** structure for any reason. It is used to notify TAPI.

CTSPiPhoneConnection::OnPhoneFeaturesChanged

Protected

```
virtual VOID OnPhoneFeaturesChanged();
```

Remarks

This function is called when the **PHONESTATUS.dwPhoneFeatures** field changes. It is used to notify TAPI.

CTSPiPhoneConnection::OnPhoneStatusChange

Protected

```
virtual VOID OnPhoneStatusChange(DWORD dwState,
                                  DWORD dwParam = 0);
```

<i>dwState</i>	The phone state which has changed
<i>dwParam</i>	Optional parameter based on the state.

Remarks

This function is called when the state of something has changed on the phone. It should be called by the service provider if something changes on the device and there is not support in the library for it. An example would be device specific extensions.

CTSPiPhoneConnection::OnRequestComplete

Protected

```
virtual VOID OnRequestComplete (CTSPIRequest* pReq, LONG IResult);
```

<i>pReq</i>	The request which is being completed.
<i>IResult</i>	The final return code for the request.

Remarks

This function is called when any request which is being managed by the phone is completed by the service provider. It gives the phone an opportunity to do any cleanup required with the request after it has terminated.

The default implementation of the class library manages several requests:

REQUEST_SETBUTTONINFO If the request completes successfully, the button information is recorded in the internal button structure using the **CTSPiPhoneConnection::SetButtonInfo** method.

REQUEST_SETLAMP If the request completes successfully, the lamp information is recorded in the internal lamp structure using the **CTSPiPhoneConnection::SetLampState** method.

REQUEST_SETRING If the request completes successfully, the ring information is recorded using the **CTSPiPhoneConnection::SetRingMode** and **CTSPiPhoneConnection::SetRingVolume** methods.

REQUEST_SETHOOKSWITCH If the request completes successfully, the hookswitch information is recorded using the **CTSPiPhoneConnection::SetHookSwitch** method.

REQUEST_SETHOOKSWITCHGAIN If the request completes successfully, the hookswitch information is recorded using the **CTSPiPhoneConnection::SetGain** method.

REQUEST_SETHOOKSWITCHVOL If the request completes successfully, the hookswitch information is recorded using the **CTSPiPhoneConnection::SetVolume** method.

CTSPISPhoneConnection::Open

```
virtual LONG Open(HTAPIPHONE htPhone, PHONEEVENT lpfnEventProc,  
    DWORD dwTSPIVersion);
```

<i>htPhone</i>	The opaque TAPI handle to assign to the phone object.
<i>lpfnEventProc</i>	The callback to associate with this phone.
<i>dwTSPIVersion</i>	The version the phone negotiated at to shear structures to.

Remarks

This function is called by TAPI when the phone device is opened. The default library implementation records the information associated with the phone and calls the **CTSPIDevice::OpenDevice** method to open the phone. If that function fails, **PHONEERR_RESOURCEUNAVAIL** is returned, otherwise, a success code is returned.

Return Value

TAPI error code or zero if the phone was opened successfully.

CTSPISPhoneConnection::ResetDisplay

```
VOID ResetDisplay();
```

Remarks

This function can be used by the service provider to clear the display and set it to all blanks. The current cursor position is reset to the (0,0) coordinate.

CTSPISPhoneConnection::Send_TAPI_Event

```
VOID Send_TAPI_Event(DWORD dwMsg, DWORD dwP1 = 0L,  
    DWORD dwP2 = 0L, DWORD dwP3 = 0L);
```

<i>dwMsg</i>	Event to send to TAPI (PHONE_xxx message from <i>tapi.h</i>).
<i>dwP1</i>	Parameter dependant on the message.
<i>dwP2</i>	Parameter dependant on the message.
<i>dwP3</i>	Parameter dependant on the message.

Remarks

This function is used by the class library to notify TAPI about various events happening on the phone. It issues events to the phone callback which was supplied by TAPI when the phone was opened.

It can be called by the service provider to support the various phone notifications which are not directly supported by the class library (such as **PHONE_DEVSPECIFIC**).

CTSPiPhoneConnection::SetButtonInfo

```
VOID SetButtonInfo (int iButtonID, DWORD dwFunction, DWORD dwMode,
    LPCTSTR pszName);
```

<i>dwMsg</i>	Event to send to TAPI (PHONE_ <i>xxx</i> message from <i>tapi.h</i>).
<i>dwP1</i>	Parameter dependant on the message.
<i>dwP2</i>	Parameter dependant on the message.
<i>dwP3</i>	Parameter dependant on the message.

Remarks

This worker function should be called by the provider when information associated with a button changes. It is automatically called by the class library when a **REQUEST_SETBUTTONINFO** completes with a zero return code. The button array object is marked as *dirty* when the button information is changed. This allows the provider code to serialize the array out to some persistent storage if desired.

CTSPiPhoneConnection::SetButtonInfo

```
virtual LONG SetButtonInfo (DRV_REQUESTID dwRequestID,
    TSPISETBUTTONINFO* lpButtInfo);
```

<i>dwRequestID</i>	Asynchronous request ID for this request.
<i>lpButtonInfo</i>	Parameter block for the request.

Remarks

This function is called when an application uses **phoneSetButtonInfo** to change the information associated with a *soft* button on the phone device. The **TSPI_phoneSetButtonInfo** function must be exported from the provider. It will validate that the button is a valid lamp/button index within the inserted buttons array (see **AddButton**), and then will insert a **PHONE_SETBUTTONINFO** request into the phone request queue. If the provider completes the request successfully, the **SetButtonInfo** worker function is called automatically to change the information within the button array inside the class library.

Return Value

TAPI error code or the asynchronous request ID if the **PHONE_SETBUTTONINFO** request was inserted into the queue.

CTSPiPhoneConnection::SetButtonState

DWORD SetButtonState (int iButtonId, **DWORD** dwButtonState);

<i>iButtonID</i>	Button identifier to modify
<i>dwButtonState</i>	New button state (PHONEBUTTONSTATE_ xxx).

Remarks

This worker function should be called by the provider when the state of a button changes. It will update the current button information and notify TAPI through a **PHONE_BUTTON** event.

Return Value

Previous button state.

CTSPiPhoneConnection::SetData

virtual LONG SetData (**DRV_REQUESTID** dwRequestID,
TSPiPHONEDATA* pPhoneData);

<i>dwRequestID</i>	Asynchronous request ID for this request.
<i>lpPhoneData</i>	Parameter block for the request.

Remarks

This function is called when an application uses **phoneSetData** to change the information associated with either a download or upload buffer. The **TSPi_phoneSetData** function must be exported from the provider. It validates the buffer identifier against the buffers added using **AddDownloadBuffer** and **AddUploadBuffer**, then inserts a **REQUEST_SETPHONEDATA** request into the phone request queue. The provider is responsible for handling the request - no further work is done in the class library.

Return Value

TAPI error code or the asynchronous request ID if the **PHONE_SETPHONEDATA** request was inserted into the queue.

CTSPiPhoneConnection::SetDisplay

VOID CTSPiPhoneConnection::SetDisplay (**LPCTSTR** pszBuff);

<i>pszBuff</i>	Buffer to set as the display.
----------------	-------------------------------

Remarks

This worker function is used by the provider to set the current display on the phone. It updates the internal representation of the display and notifies TAPI using a **PHONESTATE_DISPLAY** notification event.

CTSPiPhoneConnection::SetDisplay

**virtual LONG SetDisplay (DRV_REQUESTID dwRequestID,
TSPIPHONESETDISPLAY* lpDisplay);**

<i>dwRequestID</i>	Asynchronous request ID for this request.
<i>lpDisplay</i>	Parameter block for the request.

Remarks

This function is called when an application wants to change the phone display and calls **phoneSetDisplay**. The **TSPI_phoneSetDisplay** function must be exported from the provider. It validates the row and column information against the display device (see **AddDisplay**), and inserts a **REQUEST_SETDISPLAY** into the phone request queue. If the provider completes the request with a zero return code, the internal display information is updated automatically using the **SetDisplay** worker function.

Return Value

TAPI error code or the asynchronous request ID if the **REQUEST_SETDISPLAY** request was inserted into the queue.

CTSPiPhoneConnection::SetDisplayChar

VOID SetDisplayChar (int iColumn, int iRow, TCHAR cChar);

<i>iColumn</i>	Column position (X)
<i>iRow</i>	Row position (Y)
<i>cChar</i>	Character to set at the specified position.

Remarks

This worker function can be used by the service provider to set a single character in the display buffer independent of the current cursor position. The function does not change the current cursor position. It is not used internally in the class library. TAPI is notified using a **PHONESTATE_DISPLAY** event notification.

CTSPISPhoneConnection::SetDisplayCursorPos

```
VOID SetDisplayChar (int iColumn, int iRow);
```

<i>iColumn</i>	Column position (X)
<i>iRow</i>	Row position (Y)

Remarks

This worker function can be used by the service provider to set the current position of the cursor. This is used for all functions which add characters to the display buffer. TAPI is notified using a **PHONESTATE_DISPLAY** event notification.

CTSPISPhoneConnection::SetGain

```
VOID SetGain (DWORD dwHookSwitchDev, DWORD dwGain);
```

<i>dwHookSwitchDev</i>	Hook-switch device to adjust
<i>dwGain</i>	New gain value for device.

Remarks

This worker function can be used by the service provider to set the current gain level for a specific hookswitch device. TAPI is notified using a **PHONESTATE_HANDSETGAIN**, **PHONESTATE_SPEAKERGAIN**, or **PHONESTATE_HEADSETGAIN** event notification depending on the type of hookswitch device.

CTSPISPhoneConnection::SetGain

```
virtual LONG SetGain (DRV_REQUESTID dwRequestId,  
                     TSPIHOOKSWITCHPARAM* pHSPParam);
```

<i>dwRequestId</i>	Asynchronous request ID for this request.
<i>lpHSPParam</i>	Parameter block for the request.

Remarks

This function is called when an application wants to change the phone gain for a hookswitch device and calls **phoneSetGain**. The **TSPI_phoneSetGain** function must be exported from the provider. It validates the hookswitch device based on the devices added using **AddHookSwitchDevice**, and inserts a **REQUEST_SETHOOKSWITCHGAIN** into the phone request queue. If the provider completes the request with a zero return code, the internal gain information is updated automatically using the **SetGain** worker function.

Return Value

TAPI error code or the asynchronous request ID if the **REQUEST_SETHOOKSWITCHGAIN** request was inserted into the queue.

CTSPISPhoneConnection::SetHookSwitch

```
VOID SetHookSwitch (DWORD dwHookSwitchDev, DWORD dwMode);
```

<i>dwHookSwitchDev</i>	Hook-switch device to adjust
<i>dwMode</i>	New setting for the device.(PHONEHOOKSWITCHMODE_ value)

Remarks

This worker function can be used by the service provider to set the current state for a specific hookswitch device. TAPI is notified using a **PHONESTATE_HANDSETHOOKSWITCH**, **PHONESTATE_SPEAKERHOOKSWITCH**, or **PHONESTATE_HEADSETHOOKSWITCH** event notification depending on the type of hookswitch device.

CTSPISPhoneConnection::SetHookSwitch

```
virtual LONG SetHookSwitch (DRV_REQUESTID  
dwRequestId, TSPIHOOKSWITCHPARAM* pHSPParam);
```

<i>dwRequestId</i>	Asynchronous request ID for this request.
<i>lpHSPParam</i>	Parameter block for the request.

Remarks

This function is called when an application wants to change the phone hookswitch state and calls **phoneSetHookSwitch**. The **TSPI_phoneSetHookSwitch** function must be exported from the provider. It validates the hookswitch device based on the devices added using **AddHookSwitchDevice**, and inserts a **REQUEST_SETHOOKSWITCH** into the phone request queue. If the provider completes the request with a zero return code, the internal hookswitch information is updated automatically using the **SetHookSwitch** worker function.

Return Value

TAPI error code or the asynchronous request ID if the **REQUEST_SETHOOKSWITCH** request was inserted into the queue.

CTSPISPhoneConnection::SetLamp

```
virtual LONG SetLamp (DRV_REQUESTID dwRequestID,  
                     TSPISETBUTTONINFO* lpButtInfo);
```

<i>dwRequestID</i>	Asynchronous request ID for this request.
<i>lpButtInfo</i>	Parameter block for the request.

Remarks

This function is called when an application wants to change the lamp state and calls **phoneSetLamp**. The **TSPI_phoneSetLamp** function must be exported from the provider. It validates the lamp-id based on the buttons and lamps added using **AddButton**, and inserts a **REQUEST_SETLAMP** into the phone request queue. If the provider completes the request with a zero return code, the internal lamp information is updated automatically using the **SetLampState** worker function.

Return Value

TAPI error code or the asynchronous request ID if the **REQUEST_SETLAMP** request was inserted into the queue.

CTSPISPhoneConnection::SetLampState

```
DWORD SetLampState (int iButtonLampId, DWORD dwLampState);
```

<i>iButtonLampId</i>	Button/Lamp identifier for the lamp to change
<i>dwLampState</i>	New lamp state setting for the device.

Remarks

This worker function can be used by the service provider to set the current state for a specific lamp on the phone device. TAPI is notified using a **PHONESTATE_LAMP** event notification.

CTSPISPhoneConnection::SetPhoneFeatures

```
void SetPhoneFeatures (DWORD dwFeatures);
```

<i>dwFeatures</i>	New features for the phone device.
-------------------	------------------------------------

Remarks

This worker function can be used by the service provider to set the current phone features into the **PHONESTATUS.dwPhoneFeatures** field. TAPI is notified through a **PHONESTATUS_OTHER** event.

CTSPISPhoneConnection::SetRing

```
virtual LONG SetRing (DRV_REQUESTID dwRequestID,  
    TSPIRINGPATTERN* pRingPattern);
```

<i>dwRequestID</i>	Asynchronous request ID for this request.
<i>lpRingPattern</i>	Parameter block for the request.

Remarks

This function is called when an application wants to change the ring pattern for the phone and calls **phoneSetRing**. The **TSPI_phoneSetRing** function must be exported from the provider. It validates the ring mode based on the **PHONEDEVCAPS.dwNumRingModes** value, and inserts a **REQUEST_SETRING** into the phone request queue. If the provider completes the request with a zero return code, the internal ring-mode information is updated automatically using the **SetRingMode** and **SetRingVolume** worker functions.

Return Value

TAPI error code or the asynchronous request ID if the **REQUEST_SETRING** request was inserted into the queue.

CTSPISPhoneConnection::SetRingMode

```
VOID SetRingMode (DWORD dwRingMode);
```

<i>dwRingMode</i>	New ring mode the phone ringer is using.
-------------------	--

Remarks

This worker function can be used by the service provider to set the current ring-mode for the phone ringer device. The ring mode is updated in the **PHONECAPS.dwNumRingModes** field and TAPI is notified using a **PHONESTATE_RINGMODE** event notification.

CTSPISPhoneConnection::SetRingVolume

```
VOID SetRingVolume (DWORD dwRingVolume);
```

<i>dwRingVolume</i>	New ring volume the phone ringer is using.
---------------------	--

Remarks

This worker function can be used by the service provider to set the current ring-volume for the phone ringer device. The ring volume is updated in the **PHONESTATUS.dwRingVolume** field and TAPI is notified using a **PHONESTATE_RINGVOLUME** event notification.

CTSPPhoneConnection::SetStatusFlags

DWORD SetStatusFlags (DWORD dwStatus);

dwStatus Status of the phone device.

Remarks

This worker function can be used by the service provider to set the current status of the phone device. This function adjusts the **PHONESTATUS.dwStatusFlags** field and then notifies TAPI about what changed. The flags are simply changed to the passed values, so if the provider needs to save what is currently there and *update* it, then it must OR or AND the bits out appropriately.

Based on what changed in the status field, the following notifications are sent:

Operation	Bit field	Notification sent to TAPI
Added	PHONESTATUSFLAGS_CONNECTED	PHONESTATE_CONNECTED
Removed	PHONESTATUSFLAGS_CONNECTED	PHONESTATE_DISCONNECTED
Added	PHONESTATUSFLAGS_SUSPENDED	PHONESTATE_SUSPEND
Removed	PHONESTATUSFLAGS_SUSPENDED	PHONESTATE_RESUME

Return Value

Previous contents of the **PHONESTATUS.dwStatusFlags** field.

CTSPISPhoneConnection::SetStatusMessages

```
virtual LONG SetStatusMessages(DWORD dwPhoneStates,  
                                DWORD dwButtonModes, DWORD dwButtonStates)
```

<i>dwPhoneStates</i>	PHONESTATE_xxx messages to forward to TAPI.
<i>dwButtonModes</i>	PHONEBUTTONMODE_xxx messages to forward to TAPI.
<i>dwButtonStates</i>	PHONEBUTTONSTATE_xxx messages to forward to TAPI.

Remarks

This function is called when TAPI wants to change the notification messages that it should receive from the phone device. This adjusts which **PHONE_STATUS** messages are sent to TAPI as internal data structures change. The class library implements this function completely. It is documented so that it may be overridden by the service provider.

Return Value

TAPI error code or zero if the function was successful.

CTSPISPhoneConnection::SetupDisplay

```
VOID SetupDisplay (int iColumns, int iRows, char cLineFeed = _T("\n"));
```

<i>iColumns</i>	Number of columns in display.
<i>iRows</i>	Number of rows in display
<i>cLineFeed</i>	The character which will be used to divide lines.

Remarks

This worker function creates a new display device on the phone. Only one display device may be created on the phone device, multiple calls to this function will fail. As a result of this function, the **PHONEDEVCAPS** will be altered to show the number of rows/columns available on the phone display, and the **PHONEFEATURE_GETDISPLAY** feature will be added to the **PHONEDEVCAPS.dwPhoneFeatures**. If the **TSPI_phoneSetDisplay** function is exported, the **PHONEFEATURE_SETDISPLAY** feature will also be added.

All display APIs will fail until this function is called to setup the characteristics of the display. You should not call any display functions until this is done.

CTSPIPhoneConnection::SetVolume

```
VOID SetVolume (DWORD dwHookSwitchDev, DWORD dwVolume);
```

dwHookSwitchDev Hookswitch device to adjust the volume on.

dwVolume New volume value device is using.

Remarks

This worker function may be called by the service provider to adjust the internal value for the volume of a hookswitch device on the phone. It adjusts the applicable volume field in the **PHONESTATUS** structure and reports a **PHONESTATE_HANDSETVOLUME**, **PHONESTATE_SPEAKERVOLUME**, or **PHONESTATE_HEADSETVOLUME** notification event back to TAPI based on the hookswitch device that changed.

CTSPIPhoneConnection::SetVolume

```
virtual LONG SetVolume (DRV_REQUESTID dwRequestId,  
                        TSPIHOOKSWITCHPARAM* pHSPParam);
```

dwRequestId Asynchronous request id associated with request.

pHSPParam Parameter structure associated with request.

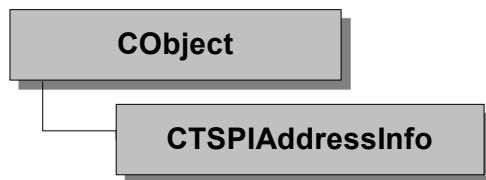
Remarks

This function is called when an application uses the **phoneSetVolume** function to change the volume of a hookswitch device. The provider must export the **TSPI_phoneSetVolume** function. The class library validates the hookswitch device and creates a **REQUEST_SETVOLUME** request block and inserts it into the phone request queue. If the provider completes the request with a zero return code, the class library will set the new volume into the **PHONESTATUS** structure using the **SetVolume** worker function.

Return Value

TAPI error code or the asynchronous request ID if the **REQUEST_SETVOLUME** request was inserted into the phone queue.

CTSPIAddressInfo



Addresses

An address corresponds to a telephone directory number, and is assigned twice: first by the telephone company at the switch, and second by the user while configuring the local system. Each address on the line is considered a channel of the line, and is represented by a **CTSPIAddressInfo** object. This object has a unique dialable address (telephone network address), and a unique number from zero to the total addresses on the line. This number (called an Address ID) is used in conjunction with the line handle to uniquely identify the address to TAPI. Since an address depends on its line to exist, the address's ID is only meaningful in the context of the associated line device. Each **CTSPIAddressInfo** object represents a single address on a line. The address object stores the capabilities, status, and call appearance information which is active for this address. Each request from the TAPI dynamic link library which is targeted for an address will eventually get to one of these objects.

Address sharing

In general, there is only one address per channel on a line. In the TSP++ library, a channel is generally treated synonymously with an address. Some installations support the assignment of more than one address to a single channel. In a home line configuration (POTS), this is made possible through "distinctive ring" which is an extra-fee service provided by the telephone company. An example of this would be parents who use one address, a child which uses another, and the fax which uses a third address. When an incoming call appears for any of these addresses, all the phones connected to the line ring, but the ring pattern will be different depending on the number dialed by the calling party. The people in the house can then determine who the incoming call is meant for, and the fax machine answers its calls by recognizing its own ringing pattern.

Many large corporations use DID, or "direct-inward dialing" for incoming calls. Before a call is connected, its destination extension number is signaled to the PBX which causes the extension to ring instead of the operators phone. In ISDN systems, the various "B" channels might not have separate addresses. Because these "B" channels might be on the same address, it is the service provider which must route the call to the appropriate channel through media detection.

Address Configuration

The relationship of an address to a line and to other addresses is known as its configuration. The network or switch can configure addresses to line assignments in many ways. The types of address configurations which are supported by TAPI are:

- | | |
|------------------|---|
| Private | The address is assigned to one line device only. An inbound call is notified at one line device. This is the default configuration for TSP++. |
| Bridged | A bridged address is a single address assigned to more than one line device. An incoming call will be offered to all lines associated with the address. |
| Monitored | The line indicates the busy or idle status of the address, but no calls may be |

placed or accepted on the address.

The default configuration for the library is *private*. If this needs to be overridden, then the **ADDRESSCAPS** of the **CTSPIAddressInfo** will need to be adjusted to reflect this. Again, if the channel is shared across addresses, such as a bridged configuration, then special adjustments need to be made. See the above section on *Address Sharing*.

Address initialization

The address objects are created by the **CTSPILineConnection** object in response to the **CreateAddress** method. As part of this method, the line connection owner, address id, dialable address, address name, media modes, bearer modes, data rates, and call appearance information are setup for the address. The main function of the address initialization is to initialize all the fields in the **LINEADDRESSCAPS** and **LINEADDRESSSTATUS** records.

Address Capabilities

The capabilities of each address are stored inside the **CTSPIAddressInfo** object. As each address is created and added to a line (through the **CreateAddress** method of the **CTSPILineConnection** object), the passed parameters will give a default set of capabilities to the address. Additionally, if any capabilities need to be adjusted, the **CTSPIAddressInfo** method **GetAddressInfo** will return a pointer to the **ADDRESSCAPS** record where this information is stored. Most of the fields are initialized for the address automatically. The following fields may need to be changed or supplied by the service provider:

Address Call Completion - If call completion is supported, then the **AddCompletionMessage** method should be called to add all the completion messages supported for display on the line. This will fill out the appropriate fields in the **LINEADDRESSCAPS** structure.

Adjusting the dwMaxNum fields - If any of the **dwMaxNum** call fields in the **ADDRESSCAPS** record are changed, then the **dwMaxNumActiveCalls** in the **CTSPILineConnection** **LINEDEVcaps** may also need to be adjusted. This is not done automatically since there is no way to detect modifications done through the **GetAddressInfo** method of **CTSPIAddressInfo**.

Structure member	Description
dwAddressSharing	Store the appropriate LINEADDRESSSHARING_ if not private.
dwSpecialInfo	Report any of the special signaling available on the address.
dwMaxNumActiveCalls	If the service provider supports more than one active call appearance at a time, then change this value.
dwMaxNumOnHoldCalls	This should be replaced by the total number of calls available on the address. It defaults to 1 if Hold is supported.
dwMaxNumOnHoldPendCalls	This should be replaced by the total number of calls available on the address. It defaults to 1 if conferencing or transfer is supported.
dwAddrCapFlags	Fill in with the required address capabilities. Default is supplied, but may not be adequate for all systems.
dwRemoveFromConfCaps	If conferencing is supported, this needs to be filled in.
dwRemoveFromConfState	If conferencing is supported, this needs to be filled in.
dwTransferModes	If transfer is supported, this needs to be filled in.
dwParkModes	If call park is supported, this needs to be filled in.
dwForwardModes	If forwarding is supported, this needs to be filled in.
dwMaxForwardEntries	If forwarding is supported, this needs to be filled in.
dwMaxSpecificEntries	If forwarding is supported, this needs to be filled in.
dwMinFwdNumRings	If forwarding is supported, this needs to be filled in.
dwMaxFwdNumRings	If forwarding is supported, this needs to be filled in.
dwMaxCallCompletions	If call completion is supported, this needs to be filled in.
dwCallCompletionConds	If call completion is supported, this needs to be filled in.
dwCallCompletionModes	If call completion is supported, this needs to be filled in.

Device Specific Extensions

If the address supports device-specific information being returned in the **LINEADDRESSCAPS** record, then these must be added either by overriding the **CServiceProvider** method **lineGetAddressCaps**, or by overriding the **CTSPIAddressInfo** method **GatherCapabilities**. If the address supports this extension information for the **LINEADDRESSSTATUS** record, then it must be added either by overriding the **CServiceProvider** method **lineGetAddressStatus**, or by overriding the **CTSPIAddressInfo** method **GatherStatusInformation**. Also, the **dwAddressStates** member of the **LINEADDRESSCAPS** structure will need to have the **LINEADDRESSSTATE_DEVSPECIFIC** bit flag added to it.

Warning:

*Do not add the device specific information into the static **LINEADDRESSCAPS**, there is no extra-allocated space!*

Address Status

The state of the address is managed completely by the **CTSPIAddressInfo** object. The TAPI dynamic link library expects the status of an address to be returned in a **LINEADDRESSSTATUS** structure during a call to **lineGetAddressStatus**. A pointer to the structure maintained by the library may be retrieved using the **GetAddressStatus** method. All fields except the device-specific information are filled out.

Request completions

When a request completes on the line, the **OnRequestComplete** method is called. The default line connection object takes the following actions when processing requests:

REQUEST_SETTERMINAL If the request completes successfully, the address will alter the internal terminal map and notify TAPI that the terminal information has changed in the **LINEADDRESSCAPS** record. This change will be cascaded to the calls which are owned by the address.

REQUEST_FORWARD If the request completes successfully, the address will add a forwarding record to the **LINEADDRESSCAPS** record and notify TAPI that forwarding information has changed. If the request failed, then the consultation call created will be destroyed.

REQUEST_SETUPXFER If the request fails, the consultation call created will be destroyed.

REQUEST_SETUPCONF If the request fails, the consultation call created will be destroyed.

REQUEST_PREPAREADDCONF If the request fails, the consultation call created will be destroyed.

REQUEST_COMPLETEXFER If the request fails, the consultation call created for a conference will be destroyed.

REQUEST_MAKECALL If the request fails, the call appearance will be deleted if no call state information was ever reported to TAPI. Otherwise, the call will be moved to the **Idle** state.

REQUEST_PICKUP If the request fails, the call appearance will be deleted.

REQUEST_UNPARK If the request fails, the call appearance will be deleted.

Operations - Public Members

AddCallTreatment	Stores a call treatment information block with the LINEADDRESSCAPS structure (32-bit only).
AddCompletionMessage	Add a completion message (<i>initialization only</i>)
AddDeviceClass	Adds a new device class object to the LINEADDRESSCAPS structure (32-bit only).
AddForwardEntry	Manually add a forwarding request into this address. This should be used at initialization if the device is <i>already</i> forwarded to some destination so that the class library may be kept in synch with the device.
CanAnswerCalls	Returns TRUE/FALSE if this address can answer offering calls.

CanMakeCalls	This information is provided when the address is added to the line. Returns TRUE/FALSE if this address can ever make outgoing calls. This information is provided when the address is added to the line.
CreateCallAppearance	Create a new call appearance on this address
CreateConferenceCall	Create a new conference call appearance on this address.
FindAttachedCall	This locates a call attached to the specified call on this address.
FindCallByCallID	Find a call appearance based on the dwCallID field of the LINECALLINFO record.
FindCallByHandle	Find a call appearance based on an opaque TAPI handle.
FindCallByState	Find a call appearance based on a LINECALLSTATE_
GetAddressCaps	This returns a pointer to the LINEADDRESSCAPS record. The information within the record may be changed.
GetAddressID	This returns the Address ID assigned to this address. It is a value from zero to the number of addresses on the line minus one.
GetAddressStatus	This returns a pointer to the LINEADDRESSSTATUS record. The information within the record may be changed.
GetAvailableMediaModes	This returns the media modes supported on this address. The available media modes are set by the CTSPILineConnection::CreateAddress method.
GetBearerMode	Returns the current bearer mode of this address. This is set during the creation of the address in the CTSPILineConnection::CreateAddress method.
GetCallCount	This returns the current count of calls on this address.
GetCallInfo	Return the CTSPICallAppearance for a call based on an index.
GetCallTreatmentName	Returns a call treatment block from the LINEADDRESSCAPS structure (32-bit only).
GetCompletionMessageCount	Return the number of completion messages
GetCompletionMessage	Return a completion message based on an index.
GetCurrentRate	This returns the current data rate for the address. This is set during the creation of the address in the CTSPILineConnection::CreateAddress method.
GetDeviceClass	Returns a device class object from the LINEADDRESSCAPS structure (32-bit only).
GetDialableAddress	This returns the dialable telephone network address assigned to this object.
GetLineOwner	This returns a pointer to the owning CTSPILineConnection object.
GetName	This returns the address name. This name will be used as the caller-id name when outgoing calls are placed on this address.
RemoveCallAppearance	Remove and delete a call appearance.
RemoveCallTreatment	Removes a call treatment record from the LINEADDRESSCAPS structure.
RemoveDeviceClass	Removes a device class from the ADDRESSCAPS structure (32-bit only).
SetAddressFeatures	This is used to change the current address features reported to TAPI.
SetCurrentRate	This sets the current data rate reported on the address. This new rate will then be used as the maximum rate for any new calls created on the address.
SetDialableAddress	This changes the dialable address associated with this object which is reported as caller-id information.
SetName	This changes the name associated with the address which is reported as caller-id information.
SetNumRingsNoAnswer	Sets the number of rings before considered a “no answer” condition. This is stored in the LINEADDRESSSTATUS record.

SetTerminalModes This method sets the terminal routing information. This is called by the library when a **REQUEST_SETTERMINAL** completes successfully.

Overridables - Public Members

CanForward Validates a forwarding information request for this address.
CanSupportCall Returns whether the call type can be carried on this address.
CanSupportMediaModes Called to determine if a particular set of media modes can be supported on this address
OnAddressCapabilitiesChange This should be called by the service provider if any of the fields in the **LINEADDRESSCAPS** record change after the service provider has been initialized.
OnAddressStateChange This method is called when any of the data in our **LINEADDRESSSTATUS** record has changed.

Operations - Protected Members

AddAsynchRequest This queues a request into the device asynchronous list.
CanHandleRequest This calls the **CServiceProvider** object to determine if a request may be processed on this address.
DeleteForwardingInfo This method is used to delete the forwarding information associated with this address.
GetTerminalInformation Returns the terminal information for this address.

Overridables - Protected Members

Init This method is called directly after the constructor to initialize the address object.
OnAddressFeaturesChanged This method is used to adjust the address features associated with this address and allow the service provider to change them before TAPI is notified.
OnCallFeaturesChanged This method is called by the call appearance if it changes its features.
OnCallStateChange Called by call appearances owned by this address when any call state information changes.
OnCreateCall This method is called when any new call appearance is created on this address.
OnRequestComplete This method is called when any request completes which was associated with this address.
OnTerminalCountChanged This method is called by the line owner when any of the terminal counts change due to a **AddTerminal** or **RemoveTerminal** method call in the **CTSPILineConnection..**
OnTimer This method is called periodically to provide support for event and monitoring.

Overridables - TAPI Members

CompleteTransfer This method is called for **lineCompleteTransfer**. The default implementation issues a **REQUEST_COMPLETEXFER** request.
Forward This method is called for **lineForward**. If the request completes successfully, it will update the forwarding information for the address automatically.
GatherCapabilities This method is called for the **lineGetAddressCaps** function. It is handled completely within the library.
GatherStatusInformation This method is called for **lineGetAddressStatus**. It is handled completely within the library.

GetID	This method is called for lineGetID . Default implementation returns Not Supported .
Pickup	This method is called for linePickup . The default implementation issues a REQUEST_PICKUP request.
SetMediaControl	This method is called for lineSetMediaControl when an address ID is specified. Each call appearance will get this media control list when created.
SetStatusMessages	This sets the status messages TAPI wishes to be notified about on this address. This is called by the line owner when a lineSetStatusMessages is received.
SetTerminal	This method is called for lineSetTerminal when an address ID is specified. If the request completes successfully, it will update the terminal information in the LINEADDRESSSTATE record automatically.
SetupConference	This method is called for lineSetupConference . The default implementation creates the conference handles and issues a REQUEST_SETUPCONF request.
SetupTransfer	This method is called for lineSetupTransfer . The default implementation issues a REQUEST_SETUPXFER request.
Unpark	This method is called for lineUnpark . The default implementation issues a REQUEST_UNPARK request.

CTSPIAddressInfo::AddAsynchRequest

Protected

```
CTSPIRequest* AddAsynchRequest(WORD wReqId, DRV_REQUESTID dwReqId = 0,
                                LPCVOID lpBuff = NULL, DWORD dwSize = 0);
```

<i>wReqId</i>	REQUEST_XXX value which will be assigned to the request.
<i>dwReqId</i>	TAPI asynchronous request identifier
<i>lpBuff</i>	Optional 32-bit pointer to buffer for request.
<i>dwSize</i>	Size of above buffer.

Remarks

This method inserts a new request into the pending request list. The **CTSPIRequest** object is created, initialized, and inserted into the list associated with the owning **CTSPILineConnection** object. If there are no running requests in the list, then this request is actually started and given the opportunity to run the **STATE_INITIAL** processing before this function returns

Return Value

Request object that was created, or NULL if function failed.

CTSPIAddressInfo::AddCallTreatment

```
VOID AddCallTreatment (DWORD dwCallTreatment, LPCTSTR pszName);
```

<i>dwCallTreatment</i>	Call treatment type
<i>pszName</i>	ASCII Name of the call treatment.

Remarks

This method creates an entry in the Call Treatment handler for this address. The call treatment entry will be returned in the **LINEADDRESSCAPS** structure. The call treatment type can either be one of the TAPI-specified entries (**LINECALLTREATMENT_XXX**), or it can be a service-provider defined value.

The current call treatment indicates the sounds a party on a call that is unanswered or on hold hears.

CTSPIAddressInfo::AddCompletionMessage

```
int AddCompletionMessage (LPCTSTR pszBuff);
```

<i>pszBuff</i>	Completion message to add.
----------------	----------------------------

Remarks

This method creates an entry in the Call Completion Message handler for this address. The call completion entry will be returned in the **LINEADDRESSCAPS** structure.

Return Value

Position within the call completion message array for this entry. The array is sequential, starting at zero.

CTSPIAddressInfo::AddDeviceClass

32bit-only

```
int CTSPIDeviceClass::AddDeviceClass (LPCTSTR pszClass, DWORD dwData);
int CTSPIDeviceClass::AddDeviceClass (LPCTSTR pszClass, HANDLE hHandle,
    LPCTSTR lpszBuff);
int CTSPIDeviceClass::AddDeviceClass (LPCTSTR pszClass, HANDLE hHandle,
    LPVOID lpBuff, DWORD dwSize);
int CTSPIDeviceClass::AddDeviceClass (LPCTSTR pszClass, DWORD dwFormat,
    LPVOID lpBuff, DWORD dwSize,
    HANDLE hHandle = INVALID_HANDLE_VALUE);
int CTSPIDeviceClass::AddDeviceClass (LPCTSTR pszClass, LPCTSTR pszBuff,
    DWORD dwType = -1L);
```

<i>pszClass</i>	Device class to add/update for (e.g. "tapi/line", etc.)
<i>dwData</i>	DWORD data value to associate with device class.
<i>hHandle</i>	Win32 handle to associate with device class.
<i>lpszBuff</i>	Null-terminated string to associate with device class.
<i>lpBuff</i>	Binary data block to associate with device class
<i>dwSize</i>	Size of the binary data block
<i>dwType</i>	STRINFORMAT of the <i>lpszBuff</i> parameter.

Remarks

This method adds an entry to the device class list, associating it with the specified class name. If you associate class information with an object, it is automatically reported back through the **LINEADDRESSCAPS** structure and returned using **TSPI_lineGetID**.

Return Value

Index array of added structure.

CTSPIAddressInfo::AddForwardEntry

```
int AddForwardEntry (DWORD dwForwardMode, LPCTSTR pszCaller,
    LPCTSTR pszDestination, DWORD dwDestCountry);
```

<i>dwForwardMode</i>	Forwarding mode for the address.
<i>pszCaller</i>	Caller to be forwarded.
<i>pszDestination</i>	Destination to forward this address to.
<i>dwDestCountry</i>	Country code to forward to.

Remarks

This worker function allows the direct addition of forwarding information on this address. This should only be used if the service provider can detect that the address is already forwarded on initialization.

If the service provider is forwarding the phone due to TAPI requesting the forward, this function is called automatically by the class library when the **REQUEST_FORWARD** is completed. This function should only be called by the service provider to adjust the forwarding information for existing forwarding conditions, or due to unsolicited forwarding requests by the phone device or switch.

To delete the forwarding information, pass a zero in for forward mode.

Return Value

Index array of forwarding information.

CTSPIAddressInfo::CanAnswerCalls

```
BOOL CanAnswerCalls() const;
```

Remarks

This method returns the value given to the **CTSPIAddressInfo::Init** function regarding whether the address is capable of receiving incoming calls or not. This will almost always be **TRUE** unless the address is a *dial-out* address only.

CTSPIAddressInfo::CanForward

```
virtual LONG CanForward(TSPILINEFORWARD* lpForwardInfo, int iCount);
```

lpForwardInfo
iCount

Forwarding information from a **lineForward** call.
Number of addresses being forwarded (zero is all).

Remarks

This function is called to verify that this address can forward given the specified forwarding information. All addresses being forwarded in a group will be given a chance to check the forwarding request before the **Forward** function is actually invoked to insert the asynchronous request.

Return Value

TAPI error code or zero if the forwarding information is acceptable for the address.

CTSPIAddressInfo::CanHandleRequest

Protected

```
BOOL CanHandleRequest(WORD wRequest, DWORD dwData = 0);
```

wRequest Request type to check (**REQUEST_***xxx* from *splib.h*).
dwData Optional data to pass through library.

Remarks

This method is called to verify that a specific request type can be processed by the TSP. Eventually, it ends up calling **CServiceProvider::CanHandleRequest**, which is a virtual function. The optional data member is not used by the class library, but can be used by the TSP when overriding the **CServiceProvider** member.

This function is called in response to any request made by TAPI against the service provider. It verifies that the request is valid at that moment.

Return Value

TRUE/FALSE success code. FALSE is returned if the function is not supported. This generally will cause the request for which this function was called to fail.

CTSPIAddressInfo::CanMakeCalls

```
BOOL CanMakeCalls() const;
```

Remarks

This method returns the value given to the **CTSPIAddressInfo::Init** function regarding whether the address is capable of creating outgoing calls or not. This will almost always be **TRUE** unless the address is hardwired to receive calls only.

CTSPIAddressInfo::CanSupportCall

```
virtual LONG CanSupportCall (const LLINECALLPARAMS lpCallParams) const;
```

lpCallParams **LINECALLPARAMS** structure to validate.

Remarks

This method checks the contents of the **LINECALLPARAMS** structure and returns whether or not the call can be supported on this address. The media-mode and call parameter flags are checked against the **LINEADDRESSCAPS** fields.

Return Value

TAPI error code if the call cannot be supported indicating reason.

CTSPiAddressInfo::CanSupportMediaModes

virtual BOOL CanSupportMediaModes (DWORD dwMediaModes) const;

dwMediaModes LINEMEDIAMODE_xxx Media mode flags to validate.

Remarks

This method returns whether or not the specified media modes are supported on this address. This is done by using the values given to the address when it was created.

Return Value

TRUE or FALSE whether the specified media modes are valid for this address.

CTSPiAddressInfo::CompleteTransfer

virtual LONG CompleteTransfer (DRV_REQUESTID dwRequestId,
TSPITRANSFER* lpTransfer, HTAPICALL htConfCall,
LPHDRVCALL lphdConfCall);

<i>dwRequestId</i>	Asynchronous request ID associated with this request.
<i>lpTransfer</i>	Data block associated with original transfer request.
<i>htConfCall</i>	TAPI opaque handle if the transfer results in a conference.
<i>lphdConfCall</i>	Return pointer for TSP handle to resulting conference.

Remarks

This method is called when an application completes a conference using **lineCompleteTransfer**. The TSP must export the **TSPi_lineCompleteTransfer** function. Some telephony devices allow a transfer to complete into a three-way conference call, for this reason, TAPI sends a call handle to use if this happens. The class library validates that the call is in a valid callstate, validates the transfer type (to conference or regular) based on the **LINEADDRESSCAPS**, creates a conference call appearance if the transfer is moving into a conference, and finally, submits a **REQUEST_COMPLETEXFER** asynchronous request into the request queue.

Return Value

TAPI error code or the asynchronous request ID if a **REQUEST_COMPLETEXFER** request was placed into the request queue.

CTSPIAddressInfo::CreateCallAppearance

```
CTSPICallAppearance* CreateCallAppearance(HTAPICALL htCall = NULL,
    DWORD dwCallParamFlags = 0,
    DWORD dwOrigin = LINECALLORIGIN_UNKNOWN,
    DWORD dwReason = LINECALLREASON_UNKNOWN,
    DWORD dwTrunk = 0xffff, DWORD dwCompletionID = 0);
```

<i>htCall</i>	TAPI opaque handle to represent this call (NULL if the call is being generated by the TSP. This will cause the TSP to ask TAPI for a new call handle).
<i>dwCallParamFlags</i>	Initial LINECALLINFO.dwCallParamFlags settings.
<i>dwOrigin</i>	Initial LINECALLORIGIN_xxx value.
<i>dwReason</i>	Initial LINECALLREASON_xxx value.
<i>dwTrunk</i>	Initial external trunk which was seized for this call.
<i>dwCompletionID</i>	Completion identifier for a newly completed call. This should match the original completion request ID given by TAPI when the lineCompleteCall function was called.

Remarks

This method creates a new call appearance on the address. The appropriate data structures are established, and the **LINEADDRESSSTATUS** fields are updated to reflect a new call. The call state starts out in the **LINECALLSTATE_UNKNOWN** state until the first time it is changed by the service provider. If a call appearance using the specified **htCall** parameter already exists, it is returned. If the **htCall** parameter is **NULL**, then a new call appearance is established in TAPI using a **LINE_NEWCALL** event.

Return Value

New call appearance pointer, **NULL** if the call could not be created.

CTSPIAddressInfo::CreateConferenceCall

```
CTSPICallAppearance* CreateConferenceCall(HTAPICALL htCall = NULL);
```

<i>htCall</i>	TAPI opaque handle to represent this call (NULL if the call is being generated by the TSP. This will cause the TSP to ask TAPI for a new call handle).
---------------	--

Remarks

This method creates a new conference call appearance on the address. The appropriate data structures are established, and the **LINEADDRESSSTATUS** fields are updated to reflect a new call. If a call appearance using the specified **htCall** parameter already exists, it is returned. If the **htCall** parameter is **NULL**, then a new call appearance is established in TAPI using a **LINE_NEWCALL** event. The bearer mode, rate, origin, reason, and trunk are all established from the values in the parent address object.

Return Value

New conference call appearance pointer, **NULL** if the call could not be created.

CTSPIAddressInfo::CTSPIAddressInfo

```
CTSPIAddressInfo();
```

Remarks

This is the constructor for the address object. It should only be called by the class library.

CTSPIAddressInfo::~~CTSPIAddressInfo

```
virtual ~CTSPIAddressInfo();
```

Remarks

This is the destructor for the address object. It should only be called by the class library. It may be overridden to delete any additional data added to the object.

CTSPIAddressInfo::DeleteForwardingInfo

```
VOID DeleteForwardingInfo();
```

Remarks

This method is used to delete the information in the address objects forwarding array. This array holds the forwarding information reported to TAPI during any **lineGetAddressStatus**. This function is called during the destruction of the object and when new forwarding information replaces the existing information.

CTSPIAddressInfo::FindAttachedCall

```
CTSPICallAppearance* FindAttachedCall (CTSPICallAppearance* pSCall) const;
```

pSCall Source call.

Remarks

This method returns a pointer to the call which is attached to the specified source call. This allows a chain of attached calls. This feature is primarily used to track consultation calls and conferences internally in the library. Generally, the attachment is *one-way*, i.e. the attached call is not linked back to its attachee call.

Return Value

Call which is *attached* to the given call object. NULL if no call is attached.

CTSPIAddressInfo::FindCallByCallID

CTSPICallAppearance* FindCallByCallID (DWORD dwCallID) const;

dwCallID Call Identifier to search the call list for.

Remarks

This method locates a call by the **LINECALLINFO.dwCallID** field. This function may be used by service providers to match up a call appearance on an address to a device call which has been identified and placed into the **dwCallID** field of the **LINECALLINFO** record. The **dwCallID** field is not used by the class library, and may be used for any purpose by the service provider.

Typically, in the third-party telephony environment, the field is used to tag call objects to switch objects. The application can then determine if two call objects represent the same physical call by comparing the **dwCallID** field.

Return Value

Call which has the specified value in the **LINECALLINFO.dwCallID** field. NULL if no call on this address has that value.

CTSPIAddressInfo::FindCallByHandle

CTSPICallAppearance* FindCallByHandle (HTAPICALL htCall) const;

htCall TAPI opaque call handle to locate.

Remarks

This method locates a call by the TAPI opaque call handle. Each call has a unique handle assigned by TAPI itself to represent the call appearance. It is associated with the call object when the object was created. This function allows the service provider to locate a call appearance using that handle from TAPI.

Return Value

Call which has the specified handle value. NULL if no call on this address matches the specified TAPI call handle.

CTSPIAddressInfo::FindCallByState

CTSPICallAppearance* FindCallByState(DWORD dwCallState) const;

dwCallState **LINECALLSTATE_**xxx to locate.

Remarks

This method locates a call in the specified call state. The first call in the specified state is returned. If multiple calls exist in the same state, then the service provider must enumerate through the calls using **GetCallCount** and **GetCallInfo**.

Return Value

First call found which is currently in the specified state. NULL if no calls on this address are in the specified state.

CTSPiAddressInfo::Forward

```
virtual LONG Forward (DRV_REQUESTID dwRequestId,
                    TSPILINEFORWARD* lpForwardInfo, HTAPICALL htConsultCall,
                    LPHDRVCALL lphdConsultCall);
```

<i>dwRequestId</i>	Asynchronous request ID associated with this request.
<i>lpForwardInfo</i>	Data block associated with forwarding request.
<i>htConsultCall</i>	TAPI opaque handle if forward generates consultation call.
<i>lphdConsultCall</i>	Return pointer for TSP handle to resulting consultation call.

Remarks

This method is called when an application forwards the address using **lineForward**. The TSP must export the **TSPi_lineForward** function. Some telephony devices transition to the dialtone state when the forwarding function is invoked, for this reason, TAPI sends a call handle to use if this happens. The class library validates that the call is in a valid callstate, creates a conference call appearance if the forwarding request generates a consultation call, and finally, submits a **REQUEST_FORWARD** asynchronous request into the request queue.

Return Value

TAPI error code or the asynchronous request ID if a **REQUEST_FORWARD** request was placed into the request queue.

CTSPiAddressInfo::GatherCapabilities

```
virtual LONG GatherCapabilities (DWORD dwTSPIVersion,
                                DWORD dwExtVer, LPLINEADDRESSCAPS lpAddressCaps);
```

<i>dwTSPIVersion</i>	The TAPI version which structures should reflect.
<i>dwExtVer</i>	The extension version which structures should reflect.
<i>lpAddressCaps</i>	The structure to fill with address capabilities.

Remarks

This function is called when an application requests the address capabilities using the **lineGetAddressCaps** function. The class library automatically fills in all the known information about the provider using the line, address, and call objects which are associated with the current address object.

The service provider can adjust the capabilities returned by using the **GetAddressCaps** function and modifying the returning structure. This should typically be done when the address is first initialized as capabilities normally do not change during the life of the provider.

If the structure is modified through the **GetAddressCaps** function, TAPI is not automatically notified.

This function should be overridden if the service provider supports device extensions and wishes to return the information in the **LINEADDRESSCAPS** structure.

Return Value

TAPI Error code or zero if the function was successful.

CTSPIAddressInfo::GatherStatusInformation

```
virtual LONG GatherStatusInformation (  
    LPLINEADDRESSSTATUS lpStatus);
```

lpStatus The structure to fill with the current address status.

Remarks

This function is called when an application requests the current address status using the **lineGetAddressStatus** function. The class library automatically fills in all the known information about the provider using the line, address, and call objects which are associated with the current address object.

The service provider can adjust the capabilities returned by using the **GetAddressStatus** function and modifying the returning structure. Some of the information can be modified using other **CTSPIAddressInfo** methods. These include **SetNumRingsNoAnswer**, **SetTerminalModes**, **SetCurrentRate**, and **SetAddressFeatures**. Using these methods is recommended as TAPI is notified about the information changing through a **LINEADDRESSSTATUS** callback.

If the structure is modified through the **GetAddressStatus** function, TAPI is not automatically notified.

This function should be overridden if the service provider supports device extensions and wishes to return the information in the **LINEADDRESSSTATUS** structure.

Return Value

TAPI Error code or zero if the function was successful.

CTSPIAddressInfo::GetAddressCaps

LPLINEADDRESSCAPS GetAddressCaps();

Remarks

This function returns the structure which maps the capabilities for the address. Any information which is not an offset/size buffer may be modified or read from this structure.

If the structure is modified using this function, TAPI is not automatically notified.

Return Value

Pointer to the structure maintained by the address object.

CTSPIAddressInfo::GetAddressID

DWORD GetAddressID() const;

Remarks

This function returns the address identifier assigned to this address object. The number returned will always be between zero and the total number of addresses on the line owner object.

Return Value

Array index of the address object in question within the line owner array.

CTSPIAddressInfo::GetAddressStatus

LPLINEADDRESSSTATUS GetAddressStatus();

Remarks

This function returns the structure which maps the current status for the address. Any information which is not an offset/size buffer may be modified or read from this structure.

If the structure is modified using this function, TAPI is not automatically notified.

Return Value

Pointer to the structure maintained by the address object.

CTSPIAddressInfo::GetAvailableMediaModes

DWORD GetAvailableMediaModes () const;

Remarks

This function is a quick inline function to return the **LINEADDRESSCAPS.dwAvailableMediaModes** value. This was set by the creation of the address object through the media modes passed into the **CTSPILineConnection::CreateAddress** function.

Return Value

The **dwAvailableMediaModes** value in the **LINEADDRESSCAPS** structure.

CTSPIAddressInfo::GetBearerMode

DWORD GetBearerMode() const;

Remarks

This function is a quick inline function to return the bearer mode (**LINEBEARERMODE_xxx** value). This was set by the creation of the address object through the bearer mode passed into the **CTSPILineConnection::CreateAddress** function.

Return Value

The bearer mode of the address object.

CTSPIAddressInfo::GetCallCount

int GetCallCount() const;

Remarks

This function is used to return a current count of **CTSPICallAppearance** objects being tracked and owned by this address object. Since the call count changes dynamically while the provider runs this count should not be relied upon for any length of time.

Return Value

The number of call appearance objects owned by this address.

CTSPIAddressInfo::GetCallInfo

CTSPICallAppearance* GetCallInfo(int iPos) const;

iPos Index of call to retrieve from array.

Remarks

This function returns the call appearance object which is at the specified array position. The position element should be between zero and **GetCallCount**.

Return Value

The call appearance object at the specified position. NULL if no call exists at that position.

CTSPIAddressInfo::GetCallTreatmentName

CString GetCallTreatmentName (DWORD dwCallTreatment) const;

dwCallTreatment Call treatment identifier to return name for.

Remarks

This function returns the call treatment name associated with the specified call treatment identifier. The standard call treatments (**LINECALLTREATMENT_xxx**) or service-provider specific treatments can be added to the address using **AddCallTreatment**.

Return Value

The name of the specified call treatment.

CTSPIAddressInfo::GetCompletionMessage

LPCTSTR GetCompletionMessage (int iPos) const;

iPos Array position of the completion message.

Remarks

This function returns the completion message associated with the specified array position index. Completion messages may be added to the address using **AddCompletionMessage**.

Return Value

The description of the specified completion message.

CTSPIAddressInfo::GetCompletionMessageCount

```
int GetCompletionMessageCount() const;
```

Remarks

This function returns the count of completion messages associated with the address. Completion messages may be added to the address using **AddCompletionMessage**.

Return Value

The total number of completion messages on the address. Zero if no completion messages exist.

CTSPIAddressInfo::GetCurrentRate

```
DWORD GetCurrentRate() const;
```

Remarks

This inline function returns the current data rate associated with the address. It is initialized during the address initialization to the minimum data rate and may be adjusted using **CTSPIAddressInfo::SetCurrentRate**.

Return Value

The current data rate associated with the address object.

CTSPIAddressInfo::GetDeviceClass

32bit-only

```
DEVICECLASSINFO* GetDeviceClass(LPCTSTR pszClass);
```

pszClass Device class to search for (e.g. "tapi/line", etc.)

Remarks

This method returns the device class structure associated with the specified class name. If you associate class information with an object, it is automatically reported back through the **LINEDEVCAAPS** structure and returned using **TSPI_lineGetID**.

Return Value

DEVICECLASSINFO structure which represents the specified text name. NULL if no association has been performed.

CTSPIAddressInfo::GetDialableAddress

LPCTSTR GetDialableAddress() const;

Remarks

This inline function returns the dialable phone number of the address object. This *dialable address* is associated with the address object when it was created through the **CTSPILineConnection::CreateAddress** function.

Return Value

The dialable address of the address object.

CTSPIAddressInfo::GetID

virtual LONG GetID (CString& strDevClass, LPVARSTRING lpDeviceID,
HANDLE hTargetProcess);

<i>strDeviceClass</i>	Device class which is being asked for.
<i>lpDeviceID</i>	Returning device information.
<i>hTargetProcess</i>	Process requesting data (<i>32-bit only</i>).

Remarks

This function is called in response to an application calling **lineGetID**. It is used to return device information to the application based on a string device class key. The 32-bit class library implements this function internally and returns any device information which was added using **CTSPIAddressInfo::AddDeviceClass**.

In the 32-bit version of the library, any handle which was given to the **AddDeviceClass** function is automatically duplicated in the given process for TAPI 2.x using the DuplicateHandle Win32 function.

Return Value

TAPI error code or zero if the function was successful.

CTSPIAddressInfo::GetLineOwner

CTSPILineConnection* GetLineOwner() const;

Remarks

This inline function returns the owning line object for this address.

Return Value

Pointer to the owner **CTSPILineConnection** object of this address.

CTSPIAddressInfo::GetName

LPCTSTR GetName() const;

Remarks

This inline function returns the name of the address object. The name was established when the address object was created through the **CTSPILineConnection::CreateAddress** function. It may be changed using the **CTSPIAddressInfo::SetName** function.

This name is returned in the **LINEADDRESSCAPS** structure and is used by many applications to represent the address on dialogs.

Return Value

Pointer to buffer with name of the address object.

CTSPIAddressInfo::GetTerminalInformation

Protected

DWORD GetTerminalInformation (int iTerminalID) const;

iTerminal Terminal identifier to retrieve information for.

Remarks

This inline function returns the terminal information which applies to the address. The terminal information is automatically copied from the line owner object when the address is created and is kept in synch with any changes to the line or calls below it.

Return Value

Pointer to buffer with name of the address object.

CTSPIAddressInfo::Init

Protected

```
virtual VOID Init (CTSPILineConnection* pLine, DWORD dwAddressID,
    LPCTSTR lpszAddress, LPCTSTR lpszName, BOOL flncoming,
    BOOL fOutgoing, DWORD dwAvailMediaModes,
    DWORD dwlBearerMode, DWORD dwMinRate, DWORD dwMaxRate,
    DWORD dwMaxNumActiveCalls, DWORD dwMaxNumOnHoldCalls,
    DWORD dwMaxNumOnHoldPendCalls, DWORD dwMaxNumConference,
    DWORD dwMaxNumTransConf);
```

<i>pLine</i>	Line owner object for this address.
<i>dwAddressID</i>	Address index for this object
<i>lpszAddress</i>	Dialable phone number of the address.
<i>lpszName</i>	ASCII name reported back in ADDRESSCAPS .

<i>fIncoming</i>	TRUE if incoming calls are allowed on this address.
<i>fOutgoing</i>	TRUE if outgoing calls are allowed on this address.
<i>dwAvailMediaModes</i>	Available media modes on this address.
<i>dwBearerMode</i>	Single LINEBEARERMODE_xxx flag.
<i>dwMinRate</i>	Minimum data rate reported in ADDRESSCAPS .
<i>dwMaxRate</i>	Maximum data rate reported in ADDRESSCAPS .
<i>dwMaxNumActiveCalls</i>	Max number of calls in a Connected state.
<i>dwMaxNumOnHoldCalls</i>	Max number of calls in a Hold state.
<i>dwMaxNumOnHoldPendCalls</i>	Max number of calls waiting for Transfer/Conf .
<i>dwMaxNumConference</i>	Max number of calls conferenced together.
<i>dwMaxNumTransConf</i>	Max number of calls conferenced from a transfer event.

Remarks

This function is used to initialize an address object. It is called directly after the constructor of the **CTSPIAddressInfo** object in response to a **CTSPILineConnection::CreateAddress** call. If the service provider overrides this function, it *must* call the base class implementation.

CTSPIAddressInfo::OnAddressCapabilitiesChanged

Protected

```
virtual VOID OnAddressCapabilitiesChanged();
```

Remarks

This method is called when any information within the **LINEADDRESSCAPS** structure changes. It is called by the class library when any of the **SetXXX** worker functions are called which change information in the **LINEADDRESSCAPS** structure. It should be called by the service provider if any of the data within the structure is changed through the **GetAddressCaps()** function.

The default behavior is to notify TAPI through a **LINEADDRESSSTATE_CAPSCHANGE** event.

CTSPIAddressInfo::OnAddressFeaturesChanged

Protected

```
virtual DWORD OnAddressFeaturesChanged (DWORD dwFeatures);
```

dwFeatures New features bitmask for the address (**LINEADDRFEATURE_x**).

Remarks

This method is called when the **LINEADDRESSCAPS.dwAddressFeatures** bits are about to be changed by the class library. It gives the derived provider an opportunity to adjust the capabilities before TAPI is notified.

Return Value

Adjusted feature flags for the address. The default return value is the passed in **dwFeatures** bitmask.

CTSPIAddressInfo::OnAddressStateChange

Protected

```
virtual VOID OnAddressStateChange (DWORD dwAddressState);
```

dwAddressState Address state change to send to TAPI.

Remarks

This method is used by the class library to notify TAPI about events occurring on the address. It checks to see if the address state notification type is being monitored by TAPI and then sends a **LINE_ADDRESSTATE** event notification if it is.

CTSPIAddressInfo::OnCallFeaturesChanged

Protected

```
virtual DWORD OnCallFeaturesChanged(CTSPICallAppearance* pCall,  
                                     DWORD dwCallFeatures);
```

pCall Call appearance on this address which has changed.
dwCallFeatures New call feature bitmask for the call (**LINECALLFEATURE_x**).

Remarks

This method is called by the child call appearance object when the feature list for the call is about to be changed by the class library. It gives the address object an opportunity to adjust its own feature list based on what is now available on the call.

The default class library behavior is to call the **CTSPILineConnection::OnCallFeaturesChanged** method.

Return Value

Adjusted feature flags for the call. The default return value is the passed in **dwCallFeatures** bitmask.

CTSPIAddressInfo::OnCallStateChanged

```
virtual VOID OnCallStateChange (CTSPICallAppearance* pCall,  
                                DWORD dwState, DWORD dwOldState);
```

<i>pCall</i>	Call appearance on this address which has changed.
<i>dwState</i>	New call state for the call.
<i>dwOldState</i>	Previous call state for the call.

Remarks

This method is called by the child call appearance object when the call state of the call is about to be changed. It gives the address object an opportunity to adjust its feature list based on what is now available on the call.

The default class library behavior is to call the **CTSPILineConnection::OnCallStateChanged** method.

CTSPIAddressInfo::OnCreateCall

Protected

```
virtual VOID OnCreateCall (CTSPICallAppearance* pCall);
```

<i>pCall</i>	New Call appearance on this address.
--------------	--------------------------------------

Remarks

This method is called when any call appearance (conference, consultant or normal) is created on the address.

The default class library behavior is to ignore the event.

CTSPIAddressInfo::OnRequestComplete

Protected

```
virtual VOID OnRequestComplete (CTSPIRequest* pReq, LONG IResult);
```

<i>pReq</i>	Request which has completed
<i>IResult</i>	Final return code for the request.

Remarks

This method is called when a request is completed on the owner address/line. It gives the address object an opportunity to cleanup any data or information associated with the request.

The default implementation of the class library manages several requests:

REQUEST_SETTERMINAL If the request completes successfully, the terminal information is updated in the address object using the **CTSPIAddressInfo::SetTerminalModes** method.

REQUEST_FORWARD If the request completes successfully, the forwarding information is updated on the address. If the function failed, the consultation call created for the forward is deleted if it never transitioned to any state (i.e. still in the **Unknown** state).

REQUEST_SETUPXFER If the request fails, then the consultation call which was created for a transfer is deleted or idled depending on whether TAPI was ever told about it.

REQUEST_COMPLETEXFER If the request fails, the consultation call which was created for the transfer is idled. In addition, the relationship setup between the consultation call and the original call appearance is removed using **CTSPICallAppearance::SetRelatedCallID**.

REQUEST_SETUPCONF If the request fails, then the consultation call which was created for a transfer is deleted or idled depending on whether TAPI was ever told about it.

REQUEST_PREPAREADDCONF If the request fails, then the consultation call which was created for a transfer is deleted or idled depending on whether TAPI was ever told about it.

REQUEST_MAKECALL If the request fails, then the consultation call which was created for a transfer is deleted or idled depending on whether TAPI was ever told about.

REQUEST_PICKUP If the request fails, then the consultation call which was created for a transfer is deleted or idled depending on whether TAPI was ever told about.

REQUEST_UNPARK If the request fails, then the consultation call which was created for a transfer is deleted or idled depending on whether TAPI was ever told about.

CTSPIAddressInfo::OnTerminalCountChanged

Protected

```
virtual VOID OnTerminalCountChanged (BOOL fAdded, int iPos,
    DWORD dwMode = 0L);
```

<i>fAdded</i>	Whether the terminal mode was ADDED or REMOVED.
<i>iPos</i>	Position of the affected terminal.
<i>dwMode</i>	Terminal mode which was added or removed.

Remarks

This method is used by the class library to synchronize the terminal modes between the line and address objects. When the terminal information is adjusted through TAPI using the **lineSetTerminal** function, this method adjusts each of the addresses on the line to match the new terminal modes. The terminal mode is either *added* to the existing terminal, or *removed* depending on the first parameter.

CTSPIAddressInfo::OnTimer

Protected

```
virtual VOID OnTimer();
```

Remarks

This method is used by the class library to allow for a periodic timer to cycle through the address objects. The default implementation in the class library is to send the timer down to each of the call objects. See the **CTSPICallAppearance::OnTimer** for more information on this aspect of the timers. The timer interval may be adjusted using **CServiceProvider::SetTimerInterval**.

CTSPIAddressInfo::Pickup

```
virtual LONG Pickup (DRV_REQUESTID dwRequestID, HTAPICALL htCall,  
                    LPHDRVCALL lphdCall, TSPILINEPICKUP* lpPickup);
```

<i>dwRequestID</i>	Asynchronous request ID for this request.
<i>htCall</i>	New call appearance handle from TAPI.
<i>lphdCall</i>	Returning call handle from TSP.
<i>lpPickup</i>	Data structure for request.

Remarks

This function is called in response to an application calling **linePickup**. It is used to pick up a call alerting at the specified destination address and returns a call handle for the picked-up call. The class library will validate the basics of the request and create a **REQUEST_PICKUP** request packet and insert it into the line owner queue.

Return Value

TAPI error code or asynchronous request ID if the **REQUEST_PICKUP** packet was added to the queue.

CTSPIAddressInfo::RemoveCallAppearance

```
VOID RemoveCallAppearance(CTSPICallAppearance* pCall);
```

<i>pCall</i>	Call appearance to remove from the array.
--------------	---

Remarks

This method is used by the class library to remove and delete calls from the address call array. The function deletes the call object passed so once this returns, the specified call object memory address will be invalid. It is typically called when TAPI deletes a call appearance through **TSPi_lineCloseCall**, or when the service provider fails a request packet containing a consultation call.

CTSPIAddressInfo::RemoveCallTreatment

VOID RemoveCallTreatment (DWORD dwCallTreatment);

dwCallTreatment Call treatment index to remove.

Remarks

This function may be used remove any added call treatment entries. Call treatment entries are added using **CTSPiAddressInfo::AddCallTreatment**. The current call treatment indicates the sounds a party on a call that is unanswered or on hold hears.

CTSPiAddressInfo::RemoveDeviceClass

32-bit only

BOOL CTSPiAddressInfo::RemoveDeviceClass (LPCTSTR pszClass);

pszClass Device class key to remove.

Remarks

This method removes the specified device class information from this object. It will no longer be reported as available to TAPI.

Return Value

TRUE if device class information was located and removed.

CTSPiAddressInfo::SetAddressFeatures

VOID SetAddressFeatures(DWORD dwFeatures);

dwFeatures Address features which are now available (**LINEADDRFEATURE_xx**).

Remarks

This method sets the current features available on the address. It does *not* invoke the **CTSPiAddressInfo::OnAddressFeaturesChanged** function.

CTSPIAddressInfo::SetCurrentRate

VOID SetCurrentRate (DWORD dwRate);

dwRate Current data rate which is being used on the address.

Remarks

This method sets the current data rate being used on the address. This should generally be called for a modem-style device or ISDN address where a known data rate is set on the device. This will change the initial data rate of any call created on the address after this completes.

CTSPIAddressInfo::SetDialableAddress

VOID SetDialableAddress(LPCTSTR pszAddress);

pszAddress New dialable address for this object.

Remarks

This method changes the dialable address of the address object. This typically will not be used as the dialable address normally doesn't change during the life of a provider. If the extension *can* change, this function can be used to adjust the **LINEADDRESSCAPS.dwDialableAddr** fields. TAPI is notified that the **LINEADDRESSCAPS** structure has changed.

CTSPIAddressInfo::SetMediaControl

virtual LONG SetMediaControl (TSPIMEDIACONTROL* lpMediaControl);

lpMediaControl Data structure for request.

Remarks

This function is called in response to an application calling **lineSetMediaControl**. The **TSPi_lineSetMediaControl** must be exported by the service provider to support this function. It enables and disables control actions on the media stream associated with the address. Media control actions can be triggered by the detection of specified digits, media modes, custom tones, and call states. The class library will save off the new media control packet and then enumerate through each of the call appearances on the address and notify them that the media control information has changed using the **CTSPICallAppearance::SetMediaControl** function.

Return Value

TAPI error code or zero if the function was successful.

CTSPIAddressInfo::SetName

VOID SetName (LPCTSTR pszName);

pszName ASCII name for the address.

Remarks

This method changes the name reported in the caller id field for any outgoing calls.

CTSPIAddressInfo::SetNumRingsNoAnswer

VOID SetNumRingsNoAnswer (DWORD dwNumRings);

dwNumRings Number of rings before answering.

Remarks

This method may be used by the service provider to change the **LINEADDRESSSTATUS.dwNumRingsNoAnswer** field which is used for forwarding information. It notifies TAPI that the field has changed using a **LINEADDRESSSTATE_FORWARD** event notification.

CTSPIAddressInfo::SetStatusMessages

virtual VOID SetStatusMessages(DWORD dwStates);

dwStates New status messages to send to TAPI.

Remarks

This function is called in response to an application calling **lineSetStatusMessages**. It changes the notifications which the service provider forwards onto TAPI. It is completely implemented within the class library and only requires that the service provider exports **TSPI_lineSetStatusMessages**.

Return Value

TAPI error code or zero if the function was successful.

CTSPIAddressInfo::SetTerminal

```
virtual LONG SetTerminal (DRV_REQUESTID dwReqID,  
    TSPILINESETTERMINAL* lpLine);
```

<i>dwRequestID</i>	Asynchronous request ID for this request.
<i>lpLine</i>	Data structure for request.

Remarks

This function is called in response to an application calling **lineSetTerminal** and specified only a single address rather than the complete line. The **TSPI_lineSetTerminal** function must be exported from the provider. The default function of the library is to create and insert a **REQUEST_SETTERMINAL** request into the queue. When the service provider completes this request successfully, the class library will use the worker function **CTSPIAddressInfo::SetTerminalModes** function to adjust the final terminal destinations.

Return Value

TAPI error code, or the asynchronous request ID if the request was queued.

CTSPIAddressInfo::SetTerminalModes

```
VOID SetTerminalModes (int iTerminalID, DWORD dwTerminalModes,  
    BOOL fRouteToTerminal);
```

<i>iTerminalID</i>	Terminal identifier to adjust (0 to GetTerminalCount).
<i>dwTerminalModes</i>	Terminal mode(s) (LINETERMMODE_xxx) to adjust.
<i>fRouteToTerminal</i>	Whether the terminal is added or removed from modes.

Remarks

This is the function which is called when a **REQUEST_SETTERMINAL** is completed by the service provider. This stores or removes the specified terminal from the terminal modes given, and then forces it to happen for any existing calls on the address by routing the notification through the **CTSPICallAppearance::SetTerminalModes**.

TAPI is notified about the change through a **LINEDEVSTATE_TERMINALS** event.

CTSPAddressInfo::SetupConference

```
virtual LONG SetupConference (DRV_REQUESTID dwRequestID,
                             TSPICONFERENCE* lpConf, HTAPICALL htConfCall,
                             LPHDRVCALL lphdConfCall, HTAPICALL htConsultCall,
                             LPHDRVCALL lphdConsultCall);
```

<i>dwRequestID</i>	Asynchronous request ID for this request.
<i>lpConf</i>	Data structure for request.
<i>htConfCall</i>	New TAPI handle for created conference call.
<i>lphdConfCall</i>	Returning call handle for created conference call.
<i>htConsultCall</i>	New TAPI handle for consultation call.
<i>lphdConsultCall</i>	Returning call handle for created consultation call.

Remarks

This function is called in response to an application calling **lineSetupConference**. The **TSPi_lineSetupConference** function must be exported from the provider. The address object validates the call which is starting the conference, validates the information **TSPICONFERENCE** structure (number of parties, etc.), and creates the new conference call and possibly, the consultation call. It then attaches the consultation call to the conference using **CTSPICallAppearance::AttachCall** and submits a **REQUEST_SETUPCONF** request into the request queue.

Return Value

TAPI error code, or the asynchronous request ID if the request was queued.

CTSPAddressInfo::SetupTransfer

```
virtual LONG SetupTransfer(DRV_REQUESTID dwRequestID,
                           TSPITRANSFER* lpTransfer, HTAPICALL htConsultCall,
                           LPHDRVCALL lphdConsultCall);
```

<i>dwRequestID</i>	Asynchronous request ID for this request.
<i>lpTransfer</i>	Data structure for request.
<i>htConsultCall</i>	New TAPI handle for consultation call.
<i>lphdConsultCall</i>	Returning call handle for created consultation call.

Remarks

This function is called in response to an application calling **lineSetupTransfer**. The **TSPi_lineSetupTransfer** function must be exported from the provider. This function is used for managing a supervised transfer where an intermediate call is placed by the application. The class library validates the parameters in the **TSPITRANSFER** data structure, creates the consultation call, and then submits a **REQUEST_SETUPXFER** request into the queue.

Return Value

TAPI error code, or the asynchronous request ID if the request was queued.

CTSPIAddressInfo::Unpark

```
virtual LONG Unpark (DRV_REQUESTID dwRequestID, HTAPICALL htCall,  
                    LPHDRVCALL lphdCall, CADObArray* parrAddresses);
```

<i>dwRequestID</i>	Asynchronous request ID for this request.
<i>htCall</i>	New TAPI handle for unparked call.
<i>lphdCall</i>	Returning call handle for unparked call.
<i>parrAddresses</i>	Addresses to unpark call from.

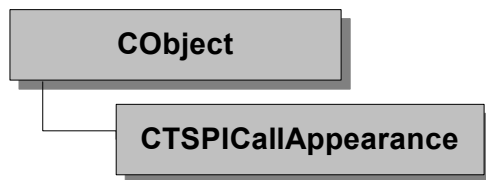
Remarks

This function is called in response to an application calling **lineUnpark**. The **TSPI_lineUnpark** function must be exported from the provider. This function is used to retrieve a parked call at the address specified in the address array. The class library validates that another call can be created, creates a call appearance to model the unparked call, and then calls the **CTSPICallAppearance::Unpark** function to retrieve the call.

Return Value

TAPI error code, or the asynchronous request ID if the request was queued.

CTSPICallAppearance



Each **CTSPICallAppearance** object represents a call on an address. The calls are maintained in a list by the **CTSPIAddressInfo** object, and are dynamically created and destroyed as needed. The call appearance object stores the capabilities and status for the call. Each request from the TAPI dynamic link library which is targeted for an address or a call will eventually get to one of these objects. The call appearance object generally represents an end-point to a conversation being maintained on the address. It doesn't need to have a direct connection to a physical call on the telephone network although it can, and generally will. In some situations, multiple call appearances may be present on a single address, although in general, only one of these calls will be *active* at any given point. The actual connection between the call appearance and the physical hardware is left up to the service provider to establish and determine (the field **dwCallID** in the **LINECALLINFO** structure may be used to store any information required).

Call Appearances

Unlike lines and addresses, call appearances are dynamic. A call appearance represents a connection between two or more addresses. The originating address is the caller. The destination address identifies the remote endpoint or station which the originator dialed (the called). Zero, one, or more calls can exist on a single address at any given time. A good example of multiple calls on a single address is call waiting: during a conversation with one party, an address with call waiting is notified (through some mechanism) that another party is attempting to call. The phone is then "flashed" to answer the second call which places the first party on hold. The person can then toggle between the two by flashing and talk to either party. In this example, the person has two calls using a single address. Since the telephone handset can only talk to one remote party at a time, only one call appearance may be active at any given point, the other being placed on hold at the switch.

Call Handles

TAPI identifies a specific call by a *call handle*. One call handle exists for each call owned or monitored by the TAPI dynamic link library. Each call created will get assigned a TAPI handle, and when new calls are received on the device, the service provider should tell the address to create a new call appearance with the **CreateCallAppearance** method of the **CTSPIAddressInfo** object, passing a NULL in for the TAPI handle. This will indicate to ask TAPI for a new call handle, and the resulting call appearance object will be connected to it. When a call is deleted through a **lineDeallocateCall**, the address object will automatically delete the call appearance and remove it from the call array.

Call States

Each call created has a *call state*. This state defines to TAPI applications what is happening on the call and which functions are currently available. Initially the call will be in the **Unknown** state. As different events are performed on the call, it will transition through various call states, eventually ending up in the **Idle** state. At this point, the call is considered "dead" until it is de-allocated by TAPI through a **lineDeallocateCall** request. The call appearance should *never* transition out of an **Idle** state. As the state of the call changes, TAPI is automatically notified by

the class library, and the other objects (line and address) are also told about the call and adjust the available features for their own object based on what state the call is in. The state of the call is generally not changed by the class library, it is left up to the derived service provider class to adjust the call state as the device notifies it about changes on the call.

Call states can change as a result of the TAPI dynamic link library performing some action with the call, an unsolicited event caused by the switch or telephone network, or the user or remote party pressing buttons on the phone. Also, different call states can indicate that connections exist to different parts of the switch. For example, a *dial tone* is a particular state of a switch that means the computer is ready to receive digits to dial.

Whenever a call changes state, the TAPI dynamic link library is notified and will report the new state to the application in a callback. This call-state notification tells the application what the call's new state is, instead of reporting the occurrence of specific events and assuming that the application will be able to deduce the transitions between two states.

Some of the call states and events defined by TAPI are exclusive to inbound or outbound call processing, while others occur in both cases. Several of these call states provide additional information that can be used by the application. For example, the *busy* state signifies that a call cannot be completed because a resource between the originator and the destination is unavailable, as when an intermediate switch has reached its capacity and cannot handle an additional call. Information supplied with the *busy* state includes *station busy* or *trunk busy*. Station busy means that the destination's station is busy (the phone is off-hook), while trunk busy means that a circuit in the switch or network is busy.

Bearer and Media Modes

The *bearer mode* of a call corresponds to the quality of service requested from the network for establishing a call. The *media mode* of a call describes the type of information that is exchanged over a specific call of a given bearer mode. As an example, the analog telephone network provides only 3.1 kHz voice-grade quality of service - this is its bearer mode. However, a call with this bearer mode can support a variety of different media modes such as voice, fax, or data modem. In other words, media modes require certain bearer modes. The bearer mode of a call is specified when the call is set up, or is provided when the call is offered. With line devices able to represent channel pools, it is possible for a service provider to allow calls to be established with wider bandwidth, this could be possible with ISDN for instance by tying the "A" and "B" channels together.

New Call Media mode identification

When the service provider detects a new call on the network, it generally will offer the call to the TAPI dynamic link library. This occurs by creating a new call appearance on the address - using the **CTSPiAddressInfo::CreateCallAppearance** and passing a NULL for the call handle. Then the call state should be set to **LINECALLSTATE_OFFERING** to indicate that the call is requesting pickup by the station. When the call is offered, a suggested media mode must be passed to indicate which application should receive ownership of the call. No call may exist in TAPI without an owner, if a call is handed to TAPI in the **Connected** state, and no owner is found, it will be dropped. TAPI will select the media modes it is watching for by calling the **lineSetDefaultMediaDetection** method. This in turn can be queried by the service provider by the **CTSPiLineConnection::GetDefaultMediaDetection** method. Any call offered to TAPI which has a media mode not in the selected list will be ignored or dropped. It is suggested, though not necessary, that the service provider keep track of the call, instead of offering it to TAPI, and if the **lineSetMediaDetection** is called again with the selected media mode present, *then* offer the call to TAPI if it is still available. This cuts down on the process that the TAPI dynamic link library must perform to determine ownership.

When the service provider detects a new call, it may be able to single out a media mode, or it may only be able to narrow down the possibilities to a certain few. These first media mode settings

are called *initial media modes*, and are passed to TAPI using the **SetCallState** method when the new call is set to **LINECALLSTATE_OFFERING**. If the media mode can be positively identified, only one flag will be set in the initial media mode - that of the call being offered. Otherwise, the multiple flags will be set, and the **LINEMEDIAMODE_UNKNOWN** bit will also be set indicating that the media mode has not been completely determined.

The service provider can use several methods to narrow down the media modes of an offering call. Many of them revolve around configuration issues. For instance, the service provider could be configured to:

- ✓ Work with only a single media mode, or certain media modes.
- ✓ Associate particular called addresses with particular media modes. This can be accomplished using **Direct Inward Dialing** (DID).
- ✓ Associate particular caller addresses with particular media modes. This can be accomplished using Caller ID.
- ✓ Identify the ring pattern of the incoming call and match it to a predetermined pattern (of several possibilities) that is reserved for a particular media mode. For example, if the incoming call is using ring pattern 2, the service provider could be configured to recognize it to be a fax call.

Also, depending on the intelligence of the network, the service provider might be able to analyze the call's protocol frames to determine the media mode. Or, the provider might automatically answer the call and perform probing on the line to determine the media mode. In this scenario, the provider would pass the call to TAPI in the **Connected** state.

Once the call is answered by an owner application, the TAPI application is responsible for media probing. If during this process, the service provider detects a media change on the line, it should use the **OnDetectedNewMediaModes** method, adding whatever flags are necessary. For example, an incoming call comes in on a line, and the service provider determines that the media could be **InteractiveVoice**, **Fax**, or **DataModem**. In this case, it will create a new call appearance (through the **CreateCallAppearance** method of the **CTSPIAddressInfo** which the call is being offered on), and send an initial callstate change of **Offering** with the initial media modes being **InteractiveVoice**, **Fax**, **DataModem**, and **Unknown** since it cannot positively identify the media mode. The TAPI dynamic link library will then begin a laborious process of finding a call owner for this call based on a priority of applications and media-type handoffs (for more information on this, see the *Windows SDK Reference: TAPI*). Since **InteractiveVoice** is the highest priority media mode, it will be tested first. If the application conclusively determines that there is a human caller at the destination, it will call **lineSetMediaMode** to change the media mode field of the **LINECALLINFO** record. But, if during the probing being performed by the application, the media state suddenly becomes known to the service provider, then the provider will call the **OnDetectedNewMediaModes** and send TAPI any monitoring information it needs. As an example, the application might play an outgoing "leave a message" voice message while the incoming call starts sending a fax calling tone. By calling the **OnDetectedNewMediaModes**, the media monitoring will be checked and potentially inform TAPI of the new incoming media mode. Note that the **LINECALLINFO** record is *not* changed in this case. Once the initial media mode is determined, the service provider should never change this field unless requested by the TAPI dynamic link library.

Call Appearance Initialization

The call appearance objects are dynamically created by the **CTSPIAddressInfo** object as needed. They can be created due to a variety of reasons, for instance, a new call being placed on the address (**lineMakeCall**) or a new called being offered on the address. Also, several of the TAPI functions may require the usage of a *consultation call* in order to complete. As part of the initialization of a call appearance, the default settings for the **LINECALLINFO** and **LINECALLSTATUS** records are filled out.

Call Information

The current information for a call is stored in the **CTSPICallAppearance** object in a **LINECALLINFO** structure. TAPI will ask for this information periodically through the **lineGetCallInfo** method of the service provider. Most of the fields are handled automatically by the base class, and some are supplied to the **CreateCallAppearance** method when the call is created on the address. It is the responsibility of the service provider in most cases to supply the call reason. Since many of the reasons are for incoming calls, the library cannot determine this information and therefore the service provider should supply that information when the call is created on the address.

The pieces of information not supported by the base class include comments, user-to user information, high and low level compatibility information, charging information, and device-specific extensions. All of these use the variable portions of the data structure, and therefore to supply them, the service provider will have to either override **lineGetCallInfo** in the **CServiceProvider** class, or derive a new call appearance and override **GatherCallInformation**.

Call Status

The current call status is also stored in the **CTSPICallAppearance** object. It is stored in a **LINECALLSTATUS** structure which is embedded in the class. All of the required information within the structure is maintained completely within the library. The only fields not completed in the library are the device-specific extension fields. Since this field is variable, to be supported in the service provider will require either an override of the **lineGetCallStatus** method of the **CServiceProvider** class, or a derivation of a new call appearance and overriding the **GatherCallStatus** method.

Device Specific Extensions

If the **LINECALLINFO** or **LINECALLSTATUS** records can supports device-specific information, then these must be added either by overriding the **CServiceProvider** method **lineGetCallInfo/lineGetCallStatus**, or by overriding the **CTSPICallAppearance** methods **GatherCallInformation/GatherStatusInformation**.

Warning:

*Do not add the device specific information into the static **LINECALLINFO** or **LINECALLSTATUS** structures, there is no extra allocated space!*

Request completions

When a request completes on the line, the **OnRequestComplete** method is called. The default line connection object takes the following actions when processing requests:

REQUEST_SETTERMINAL If the request completes successfully, the call appearance will alter the internal terminal map and notify TAPI that the terminal information has changed in the **LINECALLINFO** record.

REQUEST_SETCALLPARAMS If the request succeeds, the bearer mode and dialing parameters of our call information record will be altered to reflect the new changes.

REQUEST_SECURECALL If the request succeeds, the call information record will be altered to reflect that the call is now *secure*.

REQUEST_DROPCALL If the request succeeds, then the call state will be changed to **Idle** if it isn't already set that way.

REQUEST_GENERATEDIGITS If the request succeeds, TAPI will be notified about the digit generation completion through a **LINE_GENERATE** message.

REQUEST_GENERATETONE If the request succeeds, TAPI will be notified about the tone generation completion through a **LINE_GENERATE** message.

Operations - Public Members

AddDeviceClass	Adds a new device class structure to the LINECALLINFO structure.
AttachCall	This attaches a call appearance to this call appearance for conferencing or consultation.
CreateConsultationCall	Creates a new consultation call and attaches it to this call object.
DetachCall	This method removes the association between two call appearances.
GetAddressInfo	Return the owner CTSPIAddressInfo object.
GetAttachedCall	Return the pointer to the CTSPICallAppearance which is attached to this call. This is used for consultation call management within the library. Anytime a consultation call is created, it is attached to the call it was created for. It is automatically detached when either call goes Idle .
GetCallHandle	Return the opaque TAPI handle for this call appearance.
GetCallInfo	Return a pointer to the LINECALLINFO structure. The pointer may be used to modify any of the data within the structure <i>except</i> offset fields.
GetCallState	Return the current callstate (LINECALLSTATE_) from the LINECALLSTATUS record.
GetCallStatus	Return a pointer to the LINECALLSTATUS structure. The pointer may be used to modify any of the data within the structure <i>except</i> offset fields.
GetCallType	Return the call type (CALLTYPE_). This is setup by the library to identify consultant and conference calls on an address.
GetConsultationCall	Returns any associated consultation call to this call object.
GetDeviceClass	Returns a DEVICECLASS structure from the LINECALLINFO structure.
GetLineConnectionInfo	Return the CTSPILineConnection this call is part of.
OnDigit	This method should be called when a digit (DTMF or PULSE) is detected on the call.
OnDetectedNewMediaModes	This method should be called when a new media mode is detected on the call.
OnReceivedUserUserInfo	This method should be called when User to User information is received from the network. The data is copied into local storage and will be automatically returned in the LINECALLINFO record. TAPI is notified.
OnTone	This method should be called when a tone (composed of a set of frequencies) is detected on the call.
RemoveDeviceClass	This removes a device class structure from the LINECALLINFO structure. It must have been placed there used AddDeviceClass .
SetAppSpecificData	This method is called to change the application specific field of the call. This is called in response to a lineSetAppSpecific call. TAPI is notified.
SetBearerMode	This method should be called to change the bearer mode of the call. TAPI will be notified.
SetCalledIDInformation	This method fills in the called information. TAPI is notified.
SetCallerIDInformation	This method fills in the caller information. TAPI is notified.
SetCallFeatures	This method adjusts the current call features available to the call.
SetCallID	This method can be called to change the CALLID field of the call appearance. This may be used to store service provider information about the call. It is not used by the library. TAPI is notified.
SetCallOrigin	This method changes the call origin for the call appearance. TAPI is notified.

SetCallParametersFlag	This method should be called to change the current call parameters. The current call parameters are overwritten. This should be called by the service provider to support lineSetSecure . TAPI is notified
SetCallReason	This method changes the call reason for this call appearance. TAPI is notified.
SetCallState	This method should be called to change the call state of the call. All call state changes should come through this method. TAPI is notified.
SetCallData	This sets the 32-bit DWORD value associated with the call. TAPI is notified (32-bit only).
SetCallTreatment	This sets the current call treatment. TAPI is notified. (32-bit only).
SetCallType	Sets the internal call type. This is used to identify consultant and conference calls.
SetConsultationCall	Creates a consultation relationship between this call and another.
SetDataRate	This method should be called to change the data rate of the call. TAPI will be notified.
SetDestinationCountry	This method changes the destination country for this call appearance. TAPI is notified.
SetDialParameters	This method sets the current dialing parameters for the call appearance. TAPI is notified.
SetDigitMonitor	This method establishes the digit monitor process for this call.
SetQualityOfService	This sets the current known quality of service for the call. TAPI is notified (32-bit only).
SetRedirectionIDInformation	This method fills in the redirection id information. TAPI is notified.
SetRedirectingIDInformation	This method fills in the redirecting id information. TAPI is notified.
SetRelatedCallID	This method is used to change the related CALLID field of the call appearance. This is used during conference calls and consultant calls to relate the call to another call object. DO NOT CHANGE THIS FIELD.
SetTerminalModes	This method is called to change the terminal modes of the call appearance. This should be invoked once the device has completed a lineSetTerminal request. This will automatically be called for any changes to the address or line. TAPI is notified.
SetToneMonitor	This method establishes tone monitoring on the call.
SetTrunkID	This method sets the TRUNK ID of the call appearance. TAPI is notified.

Operations - Static Members

IsActiveCallState	Returns whether the supplied callstate is in an active state according to TAPI rules.
IsConnectedCallState	This method returns whether the supplied callstate is <i>connected</i> and is taking up bandwidth on the address.

Operations - Protected Members

AddAsynchRequest	Allocate and add a new request to our device asynchronous request list.
CanHandleRequest	Calls the CServiceProvider class to determine if a specific request can be handled by the call.
CompleteDigitGather	This method is called to end a digit gathering process.
DeleteToneMonitorList	This method is used to delete the active tone monitor list from the call.

Overridables - Protected Members

Init	This is called directly after the constructor to initialize the various fields of the LINECALLINFO and LINECALLSTATUS structures.
OnCallInfoChange	This method is called whenever any data within the LINECALLINFO record is changed.
OnCallStatusChange	This method is invoked each time a field in our LINECALLSTATUS record changes.
OnConsultantCallIdle	This is called when the consultation call attached to a conference which is pending an add request is transitioned to the idle state. This by default does nothing, but allows the derived class to perform whatever action the switch normally takes (i.e. move back to the connected state, establish a new consultation call, etc.)
OnMediaControl	This method is called when a media control event is detected. This is only used when media control monitoring is initiated via lineSetMediaControl . This generates an asynchronous request.
OnRelatedCallStateChange	This method is called when any call appearance which is related to this call appearance (through the dwRelatedCallID field in the LINECALLINFO record) changes call states.
OnRequestComplete	This method is called when a request is completed and is associated with this call.
OnTerminalCountChanged	This is invoked by the address object when the count of terminals has changed at the line level. This adds or removes a terminal from the terminal id list.
OnTimer	This is periodically called to check event lists.
OnToneMonitorDetect	This method is called when a tone event which matches a monitor record is detected. This is only used when a tone list is set up via lineMonitorTones . This generates a TAPI notification.

Overridables - TAPI Methods

Accept	This is called by the lineAccept method. Default implementation issues a REQUEST_ACCEPT request.
Answer	This is called by the lineAnswer method. Default implementation issues a REQUEST_ANSWER request.
BlindTransfer	This is called by the lineBlindTransfer method. Default implementation issues a REQUEST_BLINDXFER request.
Close	This is called by the lineClose method. Default implementation deletes the call appearance.
CompleteCall	This is called by the lineCompleteCall method. Default implementation issues a REQUEST_COMPLETECALL request.
Dial	This is called by the lineDial method. Default implementation issues a REQUEST_DIAL request.
Drop	This is called by the lineDrop method. Default implementation issues a REQUEST_DROP request.
GatherCallInformation	This is called by the lineGetCallStatus method. This is handled completely within the library.
GatherDigits	This is called by the lineGatherDigits method. This is handled completely within the library as long as the OnDigit method is called when digits are detected.
GatherStatusInformation	This is called by the lineGetCallInfo method. This is handled completely within the library.
GenerateDigits	This is called by the lineGenerateDigits method. Default implementation issues a REQUEST_GENERATEDIGITS request.

GenerateTone	This is called by the lineGenerateTone method. Default implementation issues a REQUEST_GENERATETONE request.
GetID	This method is called for lineGetID . Default implementation returns Not Supported .
Hold	This is called by the lineHold method. Default implementation issues a REQUEST_HOLD request.
MakeCall	This is called by the lineMakeCall method. Default implementation issues a REQUEST_MAKECALL request.
MonitorDigits	This is called by the lineMonitorDigits method. This is handled within the library if the OnDigit method is called when a digit is detected.
MonitorMedia	This is called by the lineMonitorMedia method. This is handled within the library if the OnDetectedNewMediaModes method is called when new media modes are seen.
MonitorTones	This is called by the lineMonitorTones method. This is handled within the library if the OnTone method is called when a tone is detected.
Park	This is called by the linePark method. Default implementation issues a REQUEST_PARK request.
Pickup	This is called by the linePickup method. Default implementation issues a REQUEST_PICKUP request.
Redirect	This is called by the lineRedirect method. Default implementation issues a REQUEST_REDIRECT request.
ReleaseUserUserInfo	This is called by the lineReleaseUserUserInfo method. Default implementation deletes any existing user to user information reported by OnReceivedUserUserInformation .
Secure	This is called by the lineSecure method. Default implementation issues a REQUEST_SECURECALL request.
SendUserUserInfo	This is called by the lineSendUserUserInfo method. Default implementation issues a REQUEST_SENDUSERINFO request.
SetCallData	This is called by the lineSetCallData method. Default implementation issues a REQUEST_SETCALLDATA request. The values are placed into the call appearance automatically when the request completes with a zero return code.
SetCallTreatment	This is called by the lineSetCallTreatment method. Default implementation issues a REQUEST_SETCALLTREATMENT request. The values are placed into the call appearance automatically when the request completes with a zero return code.
SetCallParams	This is called by the lineSetCallParams method. Default implementation issues a REQUEST_SETCALLPARAMS request. The values are placed into the call appearance automatically when the request completes with a zero return code.
SetMediaControl	This is called by the lineSetMediaControl method. This watches for an event and if seen, issues a REQUEST_SETMEDIA request.
SetMediaMode	This is called by the lineSetMediaMode method. This changes the dwMediaMode value in the CALLINFO record.
SetQualityOfService	This is called by the lineSetQualityOfService method. Default implementation issues a REQUEST_SETQOS request. The values are placed into the call appearance automatically when the request completes with a zero return code.
SetTerminal	This is called by the lineSetTerminal method. Default implementation issues a REQUEST_SETTERMINAL request. The values are placed into the call appearance when the request completes with a zero return code.

SwapHold

This is called by the **lineSwapHold** method. Default implementation issues a **REQUEST_SWAPHOLD** request.

Unhold

This is called by the **lineUnhold** method. Default implementation issues a **REQUEST_UNHOLD** request.

Unpark

This is called by the **lineUnpark** method. Default implementation issues a **REQUEST_UNPARK** request.

CTSPICallAppearance::Accept

```
virtual LONG Accept(DRV_REQUESTID dwRequestID,
                   LPCSTR lpszUserUserInfo, DWORD dwSize);
```

<i>dwRequestID</i>	Asynchronous request ID for this request.
<i>lpszUserUserInfo</i>	User to User information to send with the Accept .
<i>dwSize</i>	Size of the user to user information block.

Remarks

This function is called in response to an application calling **lineAccept**. The **TSPI_lineAccept** function must be exported from the provider. The function is used to accept an incoming offering call from the switch. On some PBX switch systems, the station receives the offering call before any alerting is performed on the station. The accept function allows the station to begin to ring or notify the user in some other fashion. The class library builds a **REQUEST_ACCEPT** packet and inserts it into the request queue.

Return Value

TAPI error code, or the asynchronous request ID if the request was queued.

CTSPICallAppearance::AddAsynchRequest**Protected**

```
CTSPIRequest* AddAsynchRequest (WORD wReqId,
                                DRV_REQUESTID dwReqId = 0,
                                LPCVOID lpBuff = NULL, DWORD dwSize = 0)
```

<i>wReqId</i>	Request ID to use for the generated request (REQUEST_xx).
<i>dwReqId</i>	TAPI asynchronous request ID to associate with the request.
<i>lpBuff</i>	Optional buffer for the request - based on the REQUEST_xx .
<i>dwSize</i>	Size of the buffer associated with the request.

Remarks

This method inserts a new request into the pending line request list. If there are no running requests in the list, then this request is actually started and given the opportunity to run the **STATE_INITIAL** processing before this function returns.

By default, this function inserts the request at the end of the line request array.

Return Value

Created request object which was inserted into the line request queue.

CTSPICallAppearance::AddDeviceClass*32bit-only*

```

int CTSPICConnection::AddDeviceClass (LPCTSTR pszClass, DWORD dwData);
int CTSPICConnection::AddDeviceClass (LPCTSTR pszClass, HANDLE hHandle,
    LPCTSTR lpszBuff);
int CTSPICConnection::AddDeviceClass (LPCTSTR pszClass, HANDLE hHandle,
    LPVOID lpBuff, DWORD dwSize);
int CTSPICConnection::AddDeviceClass (LPCTSTR pszClass, DWORD dwFormat,
    LPVOID lpBuff, DWORD dwSize,
    HANDLE hHandle = INVALID_HANDLE_VALUE);
int CTSPICConnection::AddDeviceClass (LPCTSTR pszClass, LPCTSTR pszBuff,
    DWORD dwType = -1L);

```

<i>pszClass</i>	Device class to add/update for (e.g. "tapi/line", etc.)
<i>dwData</i>	DWORD data value to associate with device class.
<i>hHandle</i>	Win32 handle to associate with device class.
<i>lpszBuff</i>	Null-terminated string to associate with device class.
<i>lpBuff</i>	Binary data block to associate with device class
<i>dwSize</i>	Size of the binary data block
<i>dwType</i>	STRINFORMAT of the lpszBuff parameter.

Remarks

This method adds an entry to the device class list, associating it with the specified class name. If you associate class information with an object, it is automatically reported back through the **LINECALLINFO** structure and returned using **TSPI_lineGetID**.

Return Value

Index array of added structure.

CTSPICallAppearance::Answer

```

virtual LONG Answer(DRV_REQUESTID dwReq,
    LPCSTR lpszUserUserInfo, DWORD dwSize);

```

<i>dwRequestID</i>	Asynchronous request ID for this request.
<i>lpszUserUserInfo</i>	User to User information to send with the Answer .
<i>dwSize</i>	Size of the user to user information block.

Remarks

This function is called in response to an application calling **lineAnswer**. The **TSPI_lineAnswer** function must be exported from the provider. The function is used to pickup an incoming offering call from the switch. The class library verifies that the call is in the **LINECALLSTATE_OFFERING** state and builds a **REQUEST_ANSWER** packet and inserts it into the request queue.

Return Value

TAPI error code, or the asynchronous request ID if the request was queued.

CTSPICallAppearance::AttachCall

```
void AttachCall(CTSPICallAppearance* pCall);
```

pCall Call to attach this call.

Remarks

This method associates two call appearances together using an internal call object field. This *attachment* is used in keeping track of conference calls and consultation calls created for various events such as transfers, forwards, or conferences. The attached call may be retrieved using **CTSPICallAppearance::GetAttachedCall**. Anytime the call state for an attached call changes, the other call is notified using the **CTSPICallAppearance::OnRelatedCallStateChange** method.

CTSPICallAppearance::BlindTransfer

```
virtual LONG BlindTransfer(DRV_REQUESTID dwRequestId,  
                           CADObservable* parrDestAddr, DWORD dwCountryCode);
```

<i>dwRequestId</i>	Asynchronous request ID for this request.
<i>parrDestAddr</i>	Address to transfer the call to.
<i>dwCountryCode</i>	Country code for the above dialable address.

Remarks

This function is called in response to an application calling **lineBlindTransfer**. The **TSPI_lineBlindTransfer** function must be exported from the provider. The function is used to perform an unsupervised transfer for an existing call on the switch. The class library verifies that the call is in the **LINECALLSTATE_CONNECTED** state and builds a **REQUEST_BLINDXFER** packet and inserts it into the request queue.

Return Value

TAPI error code, or the asynchronous request ID if the request was queued.

CTSPICallAppearance::CanHandleRequest

Protected

```
BOOL CanHandleRequest(WORD wRequest, DWORD dwData = 0);
```

wRequest Request type to check (**REQUEST_***xxx* from *splib.h*).
dwData Optional data to pass through library.

Remarks

This method is called to verify that a specific request type can be processed by the TSP. Eventually, it ends up calling **CServiceProvider::CanHandleRequest**, which is a virtual function. The optional data member is not used by the class library, but can be used by the TSP when overriding the **CServiceProvider** member.

This function is called in response to any request made by TAPI against the service provider. It verifies that the request is valid at that moment.

Return Value

TRUE/FALSE success code. FALSE is returned if the function is not supported. This generally will cause the request for which this function was called to fail.

CTSPICallAppearance::Close

```
virtual LONG Close();
```

Remarks

This function is called in response to all applications which have references to a call destroying the handles through **lineDeallocateCall**. It is invoked through the **TSPI_lineCloseCall** function which must be exported from the provider. The function deallocates the call object and removes it from the address owner array. After this call is complete, the current object will be invalid and should not be referenced.

Return Value

TAPI error code, or zero if the call was deallocated.

CTSPICallAppearance::CompleteCall

```
virtual LONG CompleteCall (DRV_REQUESTID dwRequestId,  
                           LPDWORD lpdwCompletionID, TSPICOMLETECALL * lpCompCall);
```

<i>dwRequestId</i>	Asynchronous request ID for this request.
<i>lpdwCompletionID</i>	Completion ID to assign to this request.
<i>lpCompCall</i>	Data structure associated with this request.

Remarks

This function is called in response to an application calling **lineCompleteCall**. It requires that the service provider export the **TSPI_lineCompleteCall** function. It is used to specify how a call that could not be connected normally should be completed instead. The network or switch may not be able to complete a call because the network resources are busy, or the remote station is busy or doesn't answer.

If the service provider completes the function successfully, the line object will copy the information associated with the completion request into an internal array.

Once the switch acknowledges the completion request and sends back an offering call representing the newly completed call, the service provider should use the **CTSPILineConnection::FindCallCompletionRequest** function and pass the appropriate completion identifier within the **TSPICOMLETECALL** data structure into the **CTSPIAddressInfo::CreateCallAppearance** function so that TAPI knows that it is a completion request. In addition, the call reason should be either the default, or specifically set to **LINECALLREASON_CALLCOMPLETION**.

Return Value

TAPI error code or the asynchronous request ID if the request was queued.

CTSPICallAppearance::CompleteDigitGather

```
VOID CompleteDigitGather (DWORD dwReason);
```

<i>dwReason</i>	Reason that the digit gathering process is being stopped (LINEGATHERTERM_XX).
-----------------	---

Remarks

This function completes a digit gathering session and deletes and resets the digit gather. It is called when a termination digit is found, the buffer is full, or a gather request is cancelled.

CTSPICallAppearance::CreateConsultationCall*32bit-only*

```
CTSPICallAppearance* CreateConsultationCall(HTAPICALL htCall,  
      DWORD dwCallParamFlags)
```

htCall TAPI opaque call handle (NULL if none)
dwCallParamFlags Call parameter flags to associate with this new call.

Remarks

This function creates a new call appearance and associates it to an existing call. It is used to create *consultation* relationships between two calls.

Return Value

Pointer to the consultation call object that was created and attached to this call.

CTSPICallAppearance::CTSPICallAppearance

```
CTSPICallAppearance();
```

Remarks

This is the constructor for the call object. Any derived class cannot have any parameters as it is created within the class library.

CTSPICallAppearance::~~CTSPICallAppearance

```
~CTSPICallAppearance();
```

Remarks

This is the destructor for the call object.

CTSPICallAppearance::DeleteToneMonitorList

Protected

```
VOID DeleteToneMonitorList();
```

Remarks

This internal method is used to delete the list of tones which are being monitored for by TAPI.

CTSPICallAppearance::DetachCall

```
VOID DetachCall();
```

Remarks

This internal method is used to remove the attachment from this call appearance to another. The calls were attached using **CTSPICallAppearance::AttachCall**. This call attachment is used in consultation and conference calls.

CTSPICallAppearance::Dial

```
virtual LONG Dial (DRV_REQUESTID dwRequestID,  
                  CADObArray* parrAddresses, DWORD dwCountryCode);
```

<i>dwRequestID</i>	Asynchronous request ID for this request.
<i>parrAddresses</i>	Dialable addresses to dial.
<i>dwCountryCode</i>	Country code to use for addresses.

Remarks

This function is called in response to an application calling **lineDial**. It requires that the service provider export the **TSPI_lineDial** function. It is used on existing call appearances to continue the dialing process after the **TSPI_lineMakeCall** is completed.

The class library validates that the call is not **Idle** or **Disconnected**, adjusts the caller id and called id information to include the digits if the call is in a known dialing state, and submits a **REQUEST_DIAL** request on the line.

Return Value

TAPI error code or the asynchronous request ID if the request was queued.

CTSPICallAppearance::Drop

```
virtual LONG Drop(DRV_REQUESTID dwRequestId=0,  
                  LPCSTR lpszUserUserInfo = NULL, DWORD dwSize = 0);
```

<i>dwRequestId</i>	Asynchronous request ID for this request.
<i>lpszUserUserInfo</i>	User to user information passed with the drop request.
<i>dwSize</i>	Size of the user to user information block.

Remarks

This function is called in response to an application calling **lineDrop**. TAPI invokes the **TSPI_lineDial** function which must be exported. It is used to drop a connection on a call appearance. Once this function completes, the call appearance should be in the **Idle** state.

The class library submits a **REQUEST_DROP** request if the call appearance isn't already in the **Idle** state.

Return Value

TAPI error code or the asynchronous request ID if the request was queued.

CTSPICallAppearance::GatherCallInformation

```
virtual LONG GatherCallInformation (LPLINECALLINFO lpCallInfo);
```

<i>lpCallInfo</i>	TAPI LINECALLINFO structure to fill.
-------------------	---

Remarks

This function is called in response to an application calling **lineGetCallInfo**. TAPI invokes the **TSPI_lineGetCallInfo** function which must be exported. It is used to return information about the call appearance to TAPI and invoking applications.

The class library gathers all the call information from internal data structures and the internal **LINECALLINFO** structure maintained within the call object. This function may be overridden by the service provider if device specific features are implemented by the provider.

The derived provider can adjust the return values by using the provider **SetXXX** methods or using the **CTSPICallAppearance::GetCallInfo** function and adjusting the returning structure.

Return Value

TAPI error code or zero if the function was successful.

CTSPICallAppearance::GatherDigits

```
virtual LONG GatherDigits (TSPIDIGITGATHER* lpGather);
```

<i>lpGather</i>	Data structure with information for request.
-----------------	--

Remarks

This function is called in response to an application calling **lineGatherDigits**. The service provider must export the **TSPI_lineGatherDigits** function. It is used to allocate and assign a buffer for all DTMF or pulse digits noticed by the service provider on the media stream.

It initiates the buffered gathering of digits on the specified call. The application specifies a buffer in which to place the digits and the maximum number of digits to be collected.

The class library implements this function completely - the service provider simply needs to export the **TSPI_lineGatherDigits** function and then call the **CTSPICallAppearance::OnDigit** method when any digit is seen on the media stream.

Return Value

TAPI error code or zero if the function was successful.

CTSPICallAppearance::GatherStatusInformation

```
virtual LONG GatherStatusInformation(LPLINECALLSTATUS lpCallStatus);
```

lpCallStatus TAPI **LINECALLSTATUS** structure to fill.

Remarks

This function is called in response to an application calling **lineGetCallStatus**. TAPI will call the **TSPI_lineGetCallStatus** function which must be exported by the service provider. It is used by the application to see the current status features and abilities of the call.

The class library implements this function completely and returns all the known information about the call from the internal data structures. The derived provider can adjust the return values by using the provider **SetXXX** methods or using the **CTSPICallAppearance::GetCallStatus** function and adjusting the returning structure.

Return Value

TAPI error code or zero if the function was successful.

CTSPICallAppearance::GenerateDigits

```
virtual LONG GenerateDigits (TSPIGENERATE* lpGenerate);
```

lpGenerate Data structure with parameters for this function.

Remarks

This function is called in response to an application calling **lineGenerateDigits**. The service provider must export the **TSPI_lineGenerateDigits** function.

The method initiates the generation of the specified digits on the specified call as inband tones using the specified signaling mode. Invoking this function with a NULL value for **lpDzDigits** aborts any digit generation currently in progress. An application which invokes **lineGenerateDigits** while

digit generation is in progress aborts the current digit generation and initiates the generation of the most recently specified digits.

The class library validates the parameters within the **TSPIGENERATE** data structure, removes any pending **REQUEST_GENERATEDIGITS** request, and creates and inserts a new **REQUEST_GENERATEDIGITS** request into the line queue.

Return Value

TAPI error code or zero if the function added the **REQUEST_GENERATEDIGITS** request.

CTSPICallAppearance::GenerateTone

```
virtual LONG CTSPICallAppearance::GenerateTone (  
    TSPIGENERATE* lpGenerate);
```

lpGenerate Data structure with parameters for this function.

Remarks

This function is called in response to an application calling **lineGenerateTone**. The service provider must export the **TSPI_lineGenerateTone** function.

This method generates the specified inband tone over the specified call. Invoking this function with a zero for **lpGenerate->dwToneMode** aborts the tone generation currently in progress on the specified call. An application which invokes **lineGenerateTone** while tone generation is in progress aborts the current tone generation and initiates the generation of the newly specified tone.

The class library validates the parameters within the **TSPIGENERATE** data structure, removes any pending **REQUEST_GENERATETONE** request, and creates and inserts a new **REQUEST_GENERATETONE** request into the line queue.

Return Value

TAPI error code or zero if the function added the **REQUEST_GENERATETONE** request.

CTSPICallAppearance::GetAddressInfo

```
CTSPIAddressInfo* GetAddressInfo() const;
```

Remarks

This method may be used by the service provider to determine which address object the call belongs to.

Return Value

The **CTSPIAddressInfo** object which owns the call.

CTSPICallAppearance::GetAttachedCall

CTSPICallAppearance* GetAttachedCall();

Remarks

This method may be used by the service provider to determine if any call appearances are attached to this call. Attached calls are used internally in the library to maintain connections between consultation and conference calls.

Return Value

The **CTSPICallAppearance** object which is attached to the current call or NULL if there is no attached call.

CTSPICallAppearance::GetCallHandle

HTAPICALL GetCallHandle() const;

Remarks

This method may be used by the service provider to get the TAPI opaque handle which represents this call in the TAPI system. For more information on opaque handles, see the section on *TAPI handles* in the beginning of the manual.

Return Value

The opaque TAPI handle which represents the current call object in the TAPI system DLLs.

CTSPICallAppearance::GetCallInfo

LPLINECALLINFO GetCallInfo();

Remarks

This function returns the structure which maps the current call information for the call object. Any information which is not an offset/size buffer may be modified or read from this structure.

If the structure is modified using this function, TAPI is not automatically notified.

Return Value

Pointer to the structure maintained by the call object.

CTSPICallAppearance::GetCallState

DWORD GetCallState() const;

Remarks

This function returns the current TAPI call state of the call object. The call state reflects the current state of the connection between this call and the destination or source. The value returned from here will always match the current value in the **LINECALLSTATE.dwCallState** element. The call state starts at **LINECALLSTATE_UNKNOWN** and may be adjusted by the service provider using the **CTSPICallAppearance::SetCallState**.

Return Value

The current TAPI call state.

CTSPICallAppearance::GetCallStatus

LPLINECALLSTATUS GetCallStatus();

Remarks

This function returns the structure which maps the current status for the call object. Any information which is not an offset/size buffer may be modified or read from this structure.

If the structure is modified using this function, TAPI is not automatically notified.

Return Value

Pointer to the structure maintained by the call object.

CTSPICallAppearance::GetCallType

int GetCallType() const;

Remarks

This function returns the type of call this call object represents.

CALLTYPE_NORMAL	Normal call appearance - created using TSPI_lineMakeCall , TSPI_linePickup , or TSPI_lineUnpark .
CALLTYPE_CONFERENC	Conference call - really a CTSPIConferenceCall object.
CALLTYPE_CONSULTANT	Consultant call - attached to a normal call, represents a call created on the switch for a pending transfer/forward/conference event.

Return Value

Type of call (**CALLTYPE_xxx**) for this object.

CTSPICallAppearance::GetConsultationCall*32bit-only***CTSPICallAppearance* GetConsultationCall() const;****Remarks**

This method returns any attached consultation call. It is the same as [GetAttachedCall](#)

Return Value

Pointer to the consultation call object attached to this call.

CTSPICallAppearance::GetDeviceClass*32bit-only***DEVICECLASSINFO* GetDeviceClass(LPCTSTR pszClass);**

pszClass Device class to search for (e.g. "tapi/line", etc.)

Remarks

This method returns the device class structure associated with the specified class name. If you associate class information with an object, it is automatically reported back through the **LINEDEVCAPS** structure and returned using **TSPI_lineGetID**.

Return Value

DEVICECLASSINFO structure which represents the specified text name. NULL if no association has been performed.

CTSPICallAppearance::GetID

```
virtual LONG GetID (CString& strDevClass, LPVARSTRING lpDeviceID,  
HANDLE hTargetProcess);
```

<i>strDeviceClass</i>	Device class which is being asked for.
<i>lpDeviceID</i>	Returning device information.
<i>hTargetProcess</i>	Process requesting data (<i>32-bit only</i>).

Remarks

This function is called in response to an application calling **lineGetID** and specifying a specific call handle. It is used to return device information to the application based on a string device class key. The 32-bit class library implements this function internally and returns any device information which was added using **CTSPICallAppearance::AddDeviceClass**.

In the 32-bit version of the library, any handle which was given to the **AddDeviceClass** function is automatically duplicated in the given process for TAPI 2.x using the **DuplicateHandle** Win32 function.

Return Value

TAPI error code or zero if the function was successful.

CTSPICallAppearance::GetLineConnectionInfo

```
CTSPILineConnection* GetLineConnectionInfo() const;
```

Remarks

This inline function returns the owning line object for this call object.

Return Value

Pointer to the owner **CTSPILineConnection** object of this call object.

CTSPICallAppearance::Hold

```
virtual LONG Hold (DRV_REQUESTID dwRequestID);
```

<i>dwRequestID</i>	TAPI Asynchronous request ID for the request.
--------------------	---

Remarks

This function is called in response to an application calling **lineHold**. The service provider must export the **TSPI_lineHold** function. The class library checks the call state to make sure it is in **Connected**, **Proceeding**, **Dialing**, or **Dialtone**, and submits a **REQUEST_HOLD** request into the line queue.

Return Value

TAPI error code or the request ID if the asynchronous request was added to the queue.

CTSPICallAppearance::Init

Protected

```
virtual VOID Init (CTSPIAddressInfo* pAddress, HTAPICALL htCall,
    DWORD dwBearerMode = LINEBEARERMODE_VOICE,
    DWORD dwRate = 0, DWORD dwCallParamFlags = 0,
    DWORD dwOrigin = LINECALLORIGIN_UNKNOWN,
    DWORD dwReason = LINECALLREASON_UNKNOWN,
    DWORD dwTrunk = 0xffff, DWORD dwCompletionID = 0,
    BOOL fNewCall = FALSE);
```

<i>pAddress</i>	Address owner for this call.
<i>htCall</i>	TAPI opaque handle for this call.
<i>dwBearerMode</i>	LINEBEARERMODE_ <i>xxx</i> from the address.
<i>dwRate</i>	Data rate for this call (current rate on address).
<i>dwCallParamFlags</i>	Call parameter flags passed to address.
<i>dwOrigin</i>	LINECALLORIGIN_ <i>xxx</i> from the address.
<i>dwReason</i>	LINECALLREASON_ <i>xxx</i> from the address.
<i>dwTrunk</i>	Trunk identifier for call.
<i>dwCompletionID</i>	Completion identifier from a previous CompleteCall .
<i>fNewCall</i>	TRUE if this is a call created by the TSP.

Remarks

This function is called right after the constructor to initialize the call appearance. The values passed to this function correspond directly to the values passed to the **CTSPIAddressInfo::CreateCallAppearance** method.

The **fNewCall** parameter indicates whether the service provider used the TAPI **LINE_NEWCALL** facility to create the call (previously being unknown to TAPI).

CTSPICallAppearance::IsActiveCallState

```
static BOOL IsActiveCallState(DWORD dwState);
```

<i>dwState</i>	Call state in question.
----------------	-------------------------

Remarks

This function can be used by the service provider to determine if the given call state is considered *active* by the class library and TAPI. A call state is considered active if it is not **Idle**, **OnHold** (for any reason), or **Disconnected**.

Return Value

TRUE or FALSE if the call state is considered active.

CTSPICallAppearance::IsConnectedCallState

```
static BOOL IsConnectedCallState(DWORD dwState);
```

dwState Call state in question.

Remarks

This function can be used by the service provider to determine if the given call state is considered *connected* by the class library. A call state is considered active if it is tying up a channel on the line (i.e. no other call can be present due to this call). This would *not* include things such as **OnHold** or **Offering**. The function is subtly different than **CTSPICallAppearance::IsActiveCallState**, incoming calls are not treated as connected, but they *are* treated as active.

Return Value

TRUE or FALSE if the call state is considered connected.

CTSPICallAppearance::MakeCall

```
virtual LONG MakeCall (DRV_REQUESTID dwRequestID,  
                      TSPIMAKECALL* lpMakeCall);
```

dwRequestID Asynchronous request ID for this request.
lpMakeCall Parameter buffer for this request.

Remarks

This function is called in response to an application calling **lineMakeCall**. The service provider must export the **TSPi_lineMakeCall** function.

The function is called by the **CTSPILineConnection::MakeCall** method as part of the new call processing. It sets up the origin, reason and country code within the call object, copies the **LINECALLPARAMS** buffer if it was given by the application, established caller id and called id information, and submits a **REQUEST_MAKECALL** request to the line queue.

Return Value

TAPI error code or the request ID if the asynchronous request was added to the queue.

CTSPICallAppearance::MonitorDigits

virtual LONG MonitorDigits (DWORD dwDigitModes);

dwDigitModes Digit modes to monitor for.

Remarks

This function is called in response to an application calling **lineMonitorDigits**. The service provider must export the **TSPI_lineMonitorDigits** function.

It enables and disables the unbuffered detection of digits received on the call. Each time a digit of the specified digit mode(s) is detected, a message is sent to the application indicating which digit has been detected. The function is completely implemented within the class library, all the service provider needs to do is call the **CTSPICallAppearance::OnDigit** method each time a digit is detected on the device.

Return Value

TAPI error code or zero if the new digit modes will be monitored for.

CTSPICallAppearance::MonitorMedia

virtual LONG MonitorMedia (DWORD dwMediaModes);

dwMediaModes Media modes to monitor for.

Remarks

This function is called in response to an application calling **lineMonitorMedia**. The service provider must export the **TSPI_lineMonitorMedia** function.

It enables and disables the detection of media modes on the specified call. When a media mode is detected, a message is sent to the application.. The function is completely implemented within the class library, all the service provider needs to do is call the **CTSPICallAppearance::OnDetectedNewMediaModes** method when a new media mode is detected on the media stream.

Return Value

TAPI error code or zero if the new media modes will be monitored for.

CTSPICallAppearance::MonitorTones

virtual LONG MonitorTones (TSPITONEMONITOR* lpMon);

lpMon Data structure with tones to monitor for.

Remarks

This function is called in response to an application calling **lineMonitorTones**. The service provider must export the **TSPI_lineMonitorTones** function.

It enables and disables the detection of inband tones on the call. Each time a specified tone is detected, a message is sent to the application.. The function is completely implemented within the class library, all the service provider needs to do is call the **CTSPICallAppearance::OnTone** method when a new tone is detected on the media stream.

If the tones generated are not specific or do not come in three frequency sets, the provider may override the **CServiceProvider::MatchTones** function to manage the comparison between detected tones and monitoring tones.

Return Value

TAPI error code or zero if the new tones will be monitored for.

CTSPICallAppearance::OnCallInfoChange**Protected**

```
virtual VOID OnCallInfoChange (DWORD dwCallInfo);
```

dwCallInfo **LINECALLINFO** field which changed.

Remarks

This worker function is called whenever information in our **LINECALLINFO** record has changed. It notifies TAPI about the change to the call object if necessary. If the service provider changes data directly in the **LINECALLINFO** record using **CTSPICallAppearance::GetCallInfo**, it should invoke this method to tell TAPI.

CTSPICallAppearance:: OnConsultantCallIdle**Protected**

```
virtual VOID OnConsultantCallIdle(CTSPICallAppearance* pCall);
```

pCall Consultation call which is now IDLE.

Remarks

This notification is called when a consultant call attached to this call (which may be a conference call) goes IDLE. The default implementation does nothing.

CTSPICallAppearance::OnDetectedNewMediaModes

```
VOID OnDetectedNewMediaModes (DWORD dwMediaModes);
```

dwMediaModes New media mode(s) seen on the media stream.

Remarks

The service provider should call this function when a change on the media stream is detected. This would happen if the device supported more than one media mode, and was capable of doing tone detection on the line. This function drives the tone monitoring support in the class library.

Note that the service provider should include *all* media modes which have been detected since this replaces the existing media modes in the **LINECALLSTATUS** structure.

CTSPICallAppearance::OnDigit

```
VOID OnDigit (DWORD dwType, char cDigit);  
VOID OnDigit (DWORD dwType, TCHAR cDigit);
```

dwType Type of digit detected (**LINEDIGITMODE_xxx**).
cDigit Specific digit detected.

Remarks

The service provider should call this function when a DTMF or pulsed digit is detected on the connection. This function drives the digit monitoring and gathering support in the class library.

CTSPICallAppearance::OnMediaControl**Protected**

```
virtual VOID OnMediaControl (DWORD dwMediaControl);
```

dwMediaControl Media control event which just fired.

Remarks

This notification function is called when a media control event was activated due to a media monitoring event being detected by the class library. The default implementation inserts a **REQUEST_MEDIACONTROL** request into the line queue so the service provider may perform work in the context of its worker thread.

CTSPICallAppearance::OnReceivedUserUserInfo

VOID OnReceivedUserUserInfo (LPVOID lpBuff, DWORD dwSize);

dwMediaControl Media control event which just fired.

Remarks

The service provider may call this method if user-to-user information is received by the device from the underlying switch network. The data is copied into an internal buffer and presented to TAPI when requested.

CTSPICallAppearance::OnRelatedCallStateChange**Protected**

**virtual VOID OnRelatedCallStateChange (CTSPICallAppearance* pCall,
DWORD dwState, DWORD dwOldState);**

pCall Call appearance which changed state.
dwState New call state of the call.
dwOldState Previous call state of the call.

Remarks

This notification function is called whenever a call which is related to this call changes state. The call relationship is made through the **CTSPICallAppearance::AttachCall** and **LINECALLINFO.dwRelatedCallID** fields and is used by conference and consultation calls to relate them to a call appearance.

CTSPICallAppearance::OnRequestComplete**Protected**

virtual VOID OnRequestComplete (CTSPIRequest* pReq, LONG lResult);

pReq Request which has completed
lResult Final return code for the request.

Remarks

This method is called when a request is completed on the owner address/line. It gives the call object an opportunity to cleanup any data or information associated with the request.

The default implementation of the class library manages several requests:

REQUEST_SETTERMINAL If the request completes successfully, the terminal information is updated in the address object using the **CTSPIAddressInfo::SetTerminalModes** method.

REQUEST_SETCALLPARAMS If the request completes successfully, the bearer mode and dialing parameters are copied into the call object.

REQUEST_DROP If the request completes successfully, all the pending requests for the call are removed from the line queue.

REQUEST_SECURECALL If the request completes successfully the **LINECALLPARAMFLAGS_SECURE** flag is added to the **LINECALLINFO.dwCallParamFlags** field and TAPI is notified of the change.

REQUEST_SWAPHOLD If the request completes successfully, the call types between the **onHold** call and the **Connected** call are swapped (consultation for normal).

REQUEST_GENERATEDIGITS If the request completes successfully, then TAPI is told about the request through a **LINE_GENERATE** notification.

REQUEST_GENERATETONE If the request completes successfully, then TAPI is told about the request through a **LINE_GENERATE** notification.

REQUEST_SETCALLDATA If the request completes successfully, then the call data associated with the object is updated using **SetCallData** from the request.

REQUEST_SETQOS If the request completes successfully, then the quality of service call data associated with the object is updated using **SetQualityOfService** from the request.

CTSPICallAppearance::OnTerminalCountChanged

Protected

```
virtual VOID OnTerminalCountChanged (BOOL fAdded, int iPos,
                                     DWORD dwMode = 0L);
```

<i>fAdded</i>	Whether the terminal mode was ADDED or REMOVED.
<i>iPos</i>	Position of the affected terminal.
<i>dwMode</i>	Terminal mode which was added or removed.

Remarks

This method is used by the class library to synchronize the terminal modes between the line, address and call objects. When the terminal information is adjusted through TAPI using the **lineSetTerminal** function, this method adjusts each of the calls on the address to match the new terminal modes. The terminal mode is either *added* to the existing terminal, or *removed* depending on the first parameter. TAPI is notified of the change through a **LINECALLINFOSTATE_TERMINAL** event notification.

CTSPICallAppearance::OnTimer

Protected

```
virtual void OnTimer();
```

Remarks

This internal function is called by the address when our interval timer is set. It is used by the class library to check the call digit gathering process to insure that it has not timed out.

CTSPICallAppearance::OnTone

VOID OnTone (DWORD dwFreq1, DWORD dwFreq2 = 0, DWORD dwFreq3 = 0);

Remarks

This function should be called by the service provider when a frequency is detected on the connection represented by this call. It drives the tone monitoring capabilities in the class library. Also, check out the **CServiceProvider::MatchTones** function.

CTSPICallAppearance::OnToneMonitorDetect

**virtual VOID OnToneMonitorDetect (DWORD dwToneListID,
DWORD dwAppSpecific);**

<i>dwToneListID</i>	Tone list identifier which was detected
<i>dwAppSpecific</i>	Application specific data passed from TAPI.

Remarks

This function is called when a monitored tone has been detected. It notifies TAPI using the **LINE_MONITORTONE** event.

CTSPICallAppearance::Park

virtual LONG Park (DRV_REQUESTID dwRequestID, TSPILINEPARK* lpPark);

<i>dwRequestID</i>	TAPI asynchronous request ID.
<i>lpPark</i>	Data structure for request.

Remarks

This function is called in response to an application calling **linePark**. The service provider must export the **TSPI_linePark** function. It is used to park an active call at a destination address on the switch. The class library verifies that the call is in the **Connected** state, validates the park mode in the **TSPILINEPARK** structure, and submits a **REQUEST_PARK** asynchronous request.

Return Value

TAPI error code or the asynchronous request ID if the request was added to the queue.

CTSPICallAppearance::Pickup

```
virtual LONG Pickup (DRV_REQUESTID dwRequestID,  
                    TSPILINEPICKUP* lpPickup);
```

<i>dwRequestID</i>	TAPI asynchronous request ID.
<i>lpPickup</i>	Data structure for request.

Remarks

This function is called in response to an application calling **linePickup**. The service provider must export the **TSPI_linePickup** function. It is used to pick up a call alerting at the specified destination address and return a call handle for the picked up call. If invoked with a NULL for the **lpPickup->lpszDestAddr** parameter, a group pickup is performed. If required by the device capabilities, **lpPickup->lpszGroupID** specifies the group ID to which the alerting station belongs..

The class library sets the caller id information appropriately and submits a **REQUEST_PICKUP** request.

Return Value

TAPI error code or the asynchronous request ID if the request was added to the queue.

CTSPICallAppearance::Redirect

```
virtual LONG Redirect (DRV_REQUESTID dwRequestID,  
                      CADObservable* parrAddresses,  
                      DWORD dwCountryCode);
```

<i>dwRequestID</i>	TAPI asynchronous request ID.
<i>parrAddresses</i>	Address to redirect call to.
<i>dwCountryCode</i>	Country code associated with dialable address.

Remarks

This function is called in response to an application calling **lineRedirect**. The service provider must export the **TSPI_lineRedirect** function. It is used to redirect an offering call to another station or dialable address.

The class library verifies that the call is **Offering**, sets the redirection id information appropriately and submits a **REQUEST_REDIRECT** request.

Return Value

TAPI error code or the asynchronous request ID if the request was added to the queue.

CTSPICallAppearance::ReleaseUserUserInfo

```
virtual LONG ReleaseUserUserInfo(DRV_REQUESTID dwRequestID)
```

dwRequestID TAPI asynchronous request ID.

Remarks

This function is called in response to TAPI releasing user-user information which is present in the **LINECALLINFO** record. The service provider must export the **TSPI_lineReleaseUserUserInfo** function. The function is implemented completely within the library and frees the data structure established by the **CTSPICallAppearance::OnReceivedUserUserInfo**.

Return Value

Zero.

CTSPICallAppearance::RemoveDeviceClass

32-bit only

```
BOOL CTSPICallAppearance::RemoveDeviceClass (LPCTSTR pszClass);
```

pszClass Device class key to remove.

Remarks

This method removes the specified device class information from this object. It will no longer be reported as available to TAPI.

Return Value

TRUE if device class information was located and removed.

CTSPICallAppearance::Secure

```
virtual LONG Secure (DRV_REQUESTID dwRequestID);
```

dwRequestID TAPI asynchronous request ID.

Remarks

This function is called in response to an application calling **lineSecure**. The service provider must export the **TSPI_lineSecure** function. The function secures the call from any interruptions or interference that may affect the call's media stream. The class implementation validates that the call is not secure already, verifies that the call is active, and submits a **REQUEST_SECURECALL** request to the queue.

When the service provider completes the request with a zero result, the call object will set the **LINECALLPARAMFLAGS_SECURE** flag in the **LINECALLINFO.dwCallParamFlags** field.

Return Value

TAPI error code or the asynchronous request ID if the request was added to the queue.

CTSPICallAppearance::SendUserUserInfo

**virtual LONG SendUserUserInfo (DRV_REQUESTID dwRequestID,
LPCSTR lpszUserUserInfo, DWORD dwSize);**

<i>dwRequestID</i>	TAPI asynchronous request ID.
<i>lpszUserUserInfo</i>	Information block to send to the other party
<i>dwSize</i>	Size of the information block to send.

Remarks

This function is called in response to an application calling **lineSendUserUserInfo**. The service provider must export the **TSPI_lineSendUserUserInfo** function. The class implementation validates that the call is active, and submits a **REQUEST_SENDUSERINFO** request to the queue.

If the size of the block to send exceeds the size of a network packet, the service provider is responsible for breaking the block up.

Return Value

TAPI error code or the asynchronous request ID if the request was added to the queue.

CTSPICallAppearance::SetAppSpecificData

VOID SetAppSpecificData(DWORD dwAppSpecific);

<i>dwAppSpecific</i>	Data to set into the structure.
----------------------	---------------------------------

Remarks

This worker function may be used by the service provider to set the current **LINECALLINFO.dwAppSpecific** field. The function is called by the **lineSetAppSpecific** handler in the **CServiceProvider** class.

TAPI is notified of the change using a **LINECALLINFOSTATE_APPSPECIFIC** event.

CTSPICallAppearance::SetBearerMode

VOID SetBearerMode(DWORD dwBearerMode);

<i>dwBearerMode</i>	New bearer mode for the call.
---------------------	-------------------------------

Remarks

This worker function may be used by the service provider to set the current **LINECALLINFO.dwBearerMode** field. It is used by the class library when a **REQUEST_SETCALLPARAMS** completes successfully.

TAPI is notified of the change using a **LINECALLINFOSTATE_BEARERMODE** event.

CTSPICallAppearance::SetCallData

```
virtual LONG SetCallData (DRV_REQUESTID dwRequestID,  
                          TSPICALLDATA* pCallData)
```

<i>dwRequestID</i>	TAPI asynchronous request ID.
<i>pCallData</i>	CALLDATA information to associate with the call.

Remarks

This function is called in response to an application calling **lineSetCallData**. The service provider must export the **TSPi_lineSetCallData** function. The class implementation validates that the call is active, can support call data of the specified size, and submits a **REQUEST_SETCALLDATA** request to the queue.

When the service provider completes the request with a zero result, the call object will set the call data using the worker function **CTSPICallAppearance::SetCallData**.

Return Value

TAPI error code or the asynchronous request ID if the request was added to the queue.

CTSPICallAppearance::SetCallData

```
VOID SetCallData (LPVOID lpvCallData, DWORD dwSize);
```

<i>lpvCallData</i>	Call data to set
<i>dwSize</i>	Size of the call data

Remarks

This worker function may be used by the service provider to set the current **LINECALLINFO.dwCallDataXXX** fields. It is used by the class library when a **REQUEST_SETCALLDATA** completes successfully.

The memory used by the call data is allocated internally within the library and freed when either it is replaced by a subsequent call to **SetCallData**, or when the call is destroyed.

TAPI is notified of the change using a **LINECALLINFOSTATE_CALLDATA** event.

CTSPICallAppearance::SetCalledIDInformation

```
VOID SetCalledIDInformation (DWORD dwFlags, LPCTSTR lpszPartyID = NULL,  
                             LPCTSTR lpszName = NULL, DWORD dwCountryCode = 0);
```

<i>dwFlags</i>	LINECALLPARTYID_ xxx indicating information available.
<i>lpszPartyID</i>	Dialable number to set into call.
<i>lpszName</i>	Name of party.
<i>dwCountryCode</i>	Country code if available.

Remarks

This worker function may be used by the service provider to set the current called id information for the call. The called party corresponds to the originally addressed party.

TAPI is notified of the change using a **LINECALLINFOSTATE_CALLEDID** event.

CTSPICallAppearance::SetCallerIDInformation

```
VOID SetCallerIDInformation (DWORD dwFlags, LPCTSTR lpszPartyID = NULL,  
                             LPCTSTR lpszName = NULL, DWORD dwCountryCode = 0);
```

<i>dwFlags</i>	LINECALLPARTYID_ xxx indicating information available.
<i>lpszPartyID</i>	Dialable number to set into call.
<i>lpszName</i>	Name of party.
<i>dwCountryCode</i>	Country code if available.

Remarks

This worker function may be used by the service provider to set the current caller id information for the call. The caller party corresponds to the originator of the call.

TAPI is notified of the change using a **LINECALLINFOSTATE_CALLERID** event.

CTSPICallAppearance::SetCallFeatures

VOID SetCallFeatures (DWORD dwFeatures);

dwFeatures New **LINECALLFEATURE_xx** codes which are available.

Remarks

This worker function may be used by the service provider to set the current available features for the call appearance. It does not invoke the **CTSPIAddressInfo::OnCallFeaturesChanged** function.

CTSPICallAppearance::SetCallIID

VOID SetCallIID (DWORD dwCallIID);

dwCallIID Unique call identifier to use for this call.

Remarks

In some telephony environments, the switch or service provider may assign a unique identifier to each call. This allows the call to be tracked across transfers, forwards, or other events. The field is not used in the base class and is available for derived service providers to use. This function allows the field to be changed.

TAPI is notified through a **LINECALLINFOSTATE_CALLID** event.

CTSPICallAppearance::SetCallOrigin

VOID SetCallOrigin(DWORD dwOrigin);

dwCallOrigin New **LINECALLORIGIN_xx** value for the call.

Remarks

This changes the call origin of the call. This field is located in the **LINECALLINFO** structure. TAPI is notified through a **LINECALLINFOSTATE_ORIGIN** event.

CTSPICallAppearance::SetCallParameterFlags

VOID SetCallParameterFlags (DWORD dwFlags);

dwFlags New **LINECALLPARAMFLAG_**xxx values for the call

Remarks

This changes the call parameter flags for the call. These are located in the **LINECALLINFO** structure. TAPI is notified through a **LINECALLINFOSTATE_OTHER** event.

CTSPICallAppearance::SetCallParams

**virtual LONG SetCallParams (DRV_REQUESTID dwRequestID,
 TSPICALLPARAMS* lpCallParams);**

dwRequestID TAPI asynchronous request ID.
lpCallParams **TSPICALLPARAMS** information to associate with the request.

Remarks

This function is called in response to an application calling **lineSetCallParams**. The service provider must export the **TSPI_lineSetCallParams** function. The class implementation validates that the call is active and submits a **REQUEST_SETCALLPARAMS** request to the queue.

When the service provider completes the request with a zero result, the call object will set the new bearer mode and dialing parameter flags into the call object.

Return Value

TAPI error code or the asynchronous request ID if the request was added to the queue.

CTSPICallAppearance::SetCallReason

VOID SetCallReason(DWORD dwReason);

dwReason New **LINECALLREASON_**xxx value for the call

Remarks

This changes the call reason flag for the call. It is located in the **LINECALLINFO** structure. TAPI is notified through a **LINECALLINFOSTATE_REASON** event.

CTSPICallAppearance::SetCallState

```
VOID SetCallState(DWORD dwState, DWORD dwMode = 0L,
                  DWORD dwMediaMode = 0L, BOOL fTellTapi = TRUE);
```

<i>dwState</i>	New call state for the call (LINECALLSTATE_ xxx).
<i>dwMode</i>	Call state mode (LINExxxMODE_ xxx).
<i>dwMediaMode</i>	Detected media mode for call (required if first state transition).
<i>fTellTapi</i>	TRUE if TAPI should be notified of the state change.

Remarks

This changes the call state of the given call appearance. When the call is first transitioned out of the **LINECALLSTATE_UNKNOWN** state (which is the initial state), a media mode must be given for the call. The class library changes feature information within the call, address, and line objects when any call changes states. In addition, conferences and attached calls may also react to a call changing state.

TAPI is notified if the **fTellTapi** flag is TRUE using the **LINE_CALLSTATE** event.

CTSPICallAppearance::SetCallTreatment

```
virtual LONG SetCallTreatment(DRV_REQUESTID dwRequestID,
                              DWORD dwCallTreatment);
```

<i>dwRequestID</i>	TAPI asynchronous request ID.
<i>dwCallTreatment</i>	New call treatment (LINECALLTREATMENT_ xx).

Remarks

This function is called in response to an application calling **lineSetCallTreatment**. The service provider must export the **TSPI_lineSetCallTreatment** function. The function sets the sounds a party on a call that is unanswered or on hold hears.

If the call is currently in a state where the call treatment is relevant, then it goes into effect immediately. Otherwise, the treatment will take effect the next time the call enters a relevant state. The class implementation submits a **REQUEST_SETCALLTREATMENT** request to the service provider in case any work needs to be done there. If the service provider code completes the request with a zero return code, then the specified call treatment value is stored off in the **LINECALLINFO** record and TAPI is notified of the change.

Return Value

TAPI error code or the asynchronous request ID if the request was added to the queue.

CTSPICallAppearance::SetCallTreatment

```
VOID SetCallTreatment(DWORD dwCallTreatment);
```

dwCallTreatment New **LINECALLTREATMENT_xxx** value for the call

Remarks

This changes the call treatment flag for the call. It is located in the **LINECALLINFO** structure. TAPI is notified through a **LINECALLINFOSTATE_TREATMENT** event.

CTSPICallAppearance::SetCallType

```
VOID SetCallType (int iCallType);
```

iCallType New call type (**CALLTYPE_xx**) for this call.

Remarks

This changes the call type of the call. The call type is used to identify the type of call appearance this call object represents, a normal call, a conference call, or a consultant call. See **CTSPICallAppearance::GetCallType** for more information.

CTSPICallAppearance::SetConnectedIDInformation

```
VOID SetConnectedIDInformation (DWORD dwFlags,  
                                LPCTSTR lpszPartyID = NULL, LPCTSTR lpszName = NULL,  
                                DWORD dwCountryCode = 0);
```

<i>dwFlags</i>	LINECALLPARTYID_xxx indicating information available.
<i>lpszPartyID</i>	Dialable number to set into call.
<i>lpszName</i>	Name of party.
<i>dwCountryCode</i>	Country code if available.

Remarks

This worker function may be used by the service provider to set the current connected id information for the call. The connected party corresponds to the final party the call connected to, which in most cases will be the called id.

TAPI is notified of the change using a **LINECALLINFOSTATE_CONNECTEDID** event.

CTSPICallAppearance::SetConsultationCall*32bit-only*

```
void SetConsultationCall(CTSPICallAppearance* pCall);
```

pCall Consultation call to associate with this call.

Remarks

This method establishes a consultation relationship between the two calls. The passed call is turned into a consultation call (GetCallType will return **CALLTYPE_CONSULT**) and then attached to the owner object.

CTSPICallAppearance::SetDataRate

```
VOID SetDataRate(DWORD dwDataRate);
```

dwDataRate New data rate for this call.

Remarks

This worker function changes the data rate of the call. The data rate starts initially as the current data rate of the address owner. It is not adjusted directly by the class library. The function adjusts the **dwDataRate** field in the **LINECALLINFO** structure and TAPI is notified through a **LINECALLINFOSTATE_RATE** event.

CTSPICallAppearance::SetDestinationCountry

```
VOID SetDestinationCountry (DWORD dwCountryCode);
```

dwCountryCode Country code for call (zero if unknown).

Remarks

This worker function changes the country code associated with the destination party of the call. It is initially setup on an outgoing call by the function which created the call (**MakeCall**, **Unpark**, etc.). It is not established for incoming calls unless the service provider uses this function to set it. The function adjusts the **dwCountryCode** field in the **LINECALLINFO** structure and TAPI is notified through a **LINECALLINFOSTATE_OTHER** event.

CTSPICallAppearance::SetDialParameters

VOID SetDialParameters (LINEDIALPARAMS& dp);

dp LINEDIALPARAMS to associate with this call appearance.

Remarks

This worker function changes the dialing parameters associated with the call. Unless these parameters are set by either **MakeCall** or **SetCallParams**, their values will be the same as the defaults used in the **LINEDEVCAPS** of the owning line device. The function adjusts the dialing parameters in the **LINECALLINFO** structure and TAPI is notified through a **LINECALLINFOSTATE_DIALPARAMS** event.

CTSPICallAppearance::SetDigitMonitor

VOID SetDigitMonitor(DWORD dwDigitModes);

dwDigitModes New digit modes to monitor the device for.

Remarks

This worker function changes the digit modes that the call is monitoring the media for. The value passed in should be a set of **LINEDIGITMODE_xxx** flags. The function adjusts the **LINECALLINFO.dwMonitorDigitModes** field and notifies TAPI through a **LINECALLINFOSTATE_MONITORMODES** event.

CTSPICallAppearance::SetMediaControl

virtual LONG SetMediaControl (TSPIMEDIACONTROL* lpMediaControl);

lpMediaControl Parameter block associated with request.

Remarks

This function enables and disables control actions on the media stream associated with the specified call. Media control actions can be triggered by the detection of specified digits, media modes, custom tones, and call states. The new specified media controls replace all the ones that were in effect for this line, address, or call prior to this request. The class library removes the previous media control structure and associates itself with the passed media control structure.

Return Value

TAPI error code or zero if the media control structure was associated with the call.

CTSPICallAppearance::SetMediaMode

virtual LONG SetMediaMode (DWORD dwMediaMode);

dwMediaMode New media mode for the call.

Remarks

This function is called in response to an application calling **lineSetMediaMode**. The service provider must export the **TSPI_lineSetMediaMode** function. The function is completely implemented within the class library.

The class library validates the passed media mode, and then changes the current media mode of the call to match the passed media mode. TAPI is notified through a **LINECALLINFOSTATE_MEDIAMODE** event.

Return Value

TAPI error code or zero if the media mode was set into the call.

CTSPICallAppearance::SetMediaMonitor

VOID SetMediaMonitor(DWORD dwModes);

dwModes New media modes to monitor the device for.

Remarks

This worker function changes the media modes that the call is monitoring the media for. The value passed in should be a set of **LINEMEDIAMODE_xxx** flags. The function adjusts the **LINECALLINFO.DwMonitorMediaModes** field and notifies TAPI through a **LINECALLINFOSTATE_MONITORMODES** event.

CTSPICallAppearance::SetQualityOfService

32-bit only

**virtual LONG SetQualityOfService (DRV_REQUESTID dwRequestID,
TSPIQOS* pQOS);**

dwRequestID TAPI asynchronous request ID for this request.
pQOS Quality of Service information to associate with call.

Remarks

This function is called in response to an application calling **lineSetCallQualityOfService**. The service provider must export the **TSPI_lineSetCallQualityOfService** function.

One of the new features in TAPI 2.x is the ability to request, negotiate, renegotiate, and receive indications of Quality of Service (QOS) parameters on incoming and outgoing calls. QOS

information is exchanged between applications and service providers in **FLOWSPEC** structures that are defined in Windows Sockets 2.0.

The class library verifies that the call is not **Idle**, and submits a **REQUEST_SETQOS** request to the line queue.

If the service provider completes the request successfully, then the new QOS information is set into the **LINECALLINFO** structure using the worker function **CTSPICallInfo::SetQualityOfService**.

Return Value

TAPI error code or the asynchronous request ID if the request was added to the queue.

CTSPICallAppearance::SetQualityOfService

```
VOID SetQualityOfService (LPVOID lpSendingFlowSpec, DWORD dwSendSize,
                          LPVOID lpReceivingFlowSpec, DWORD dwReceiveSize);
```

<i>lpSendingFlowSpec</i>	FLOWSPEC structure for sending information.
<i>dwSendSize</i>	Size of the sending structure.
<i>lpReceivingFlowSpec</i>	FLOWSPEC structure for receiving information.
<i>dwReceiveSize</i>	Size of the receiving structure

Remarks

This worker function changes current Quality of Service (QOS) information associated with a call appearance. It should be called when the QOS information is renegotiated on a call by the hardware or the other side of the connection. The function deletes the existing QOS information, and copies the new information into the call appearance. TAPI is then notified through a **LINECALLINFO_QOS** event.

CTSPICallAppearance::SetRedirectingIDInformation

```
VOID SetRedirectingIDInformation (DWORD dwFlags,
                                  LPCTSTR lpszPartyID = NULL, LPCTSTR lpszName = NULL,
                                  DWORD dwCountryCode = 0);
```

<i>dwFlags</i>	LINECALLPARTYID_XXX indicating information available.
<i>lpszPartyID</i>	Dialable number to set into call.
<i>lpszName</i>	Name of party.
<i>dwCountryCode</i>	Country code if available.

Remarks

This worker function may be used by the service provider to set the current redirecting id information for the call. The redirecting party corresponds to the party which directed the call to its destination.

TAPI is notified of the change using a **LINECALLINFOSTATE_REDIRECTINGID** event.

CTSPICallAppearance::SetRedirectionIDInformation

```
VOID SetRedirectionIDInformation (DWORD dwFlags,
    LPCTSTR lpszPartyID = NULL, LPCTSTR lpszName = NULL,
    DWORD dwCountryCode = 0);
```

<i>dwFlags</i>	LINECALLPARTYID_xxx indicating information available.
<i>lpszPartyID</i>	Dialable number to set into call.
<i>lpszName</i>	Name of party.
<i>dwCountryCode</i>	Country code if available.

Remarks

This worker function may be used by the service provider to set the current redirected id information for the call. The redirected party corresponds to the party where the call diverted to as a result of a forward, transfer, etc.

TAPI is notified of the change using a **LINECALLINFOSTATE_REDIRECTIONID** event.

CTSPICallAppearance::SetRelatedCallID

```
VOID SetRelatedCallID (DWORD dwCallID);
```

<i>dwCallID</i>	Call ID to which this call is <i>related</i> .
-----------------	--

Remarks

This worker function is used to associate the current call with another call. It is used within the class library to associate consultation calls with the original call that created it. This applies to transfer and conference events.

TAPI is notified of the change using a **LINECALLINFOSTATE_RELATEDCALLID** event.

CTSPICallAppearance::SetTerminal

```
virtual LONG SetTerminal (DRV_REQUESTID dwReqID,
    TSPILINESETTERMINAL* lpLine);
```

<i>dwRequestID</i>	Asynchronous request ID for this request.
<i>lpLine</i>	Data structure for request.

Remarks

This function is called in response to an application calling **lineSetTerminal** and specified a specific call. The **TSPICallAppearance::lineSetTerminal** function must be exported from the provider. The default

function of the library is to create and insert a **REQUEST_SETTERMINAL** request into the queue. When the service provider completes this request with a zero return code, the class library will adjust the final terminal information in the call using the worker function **CTSPICallAppearance::SetTerminalModes**.

Return Value

TAPI error code, or the asynchronous request ID if the request was queued.

CTSPIAddressInfo::SetTerminalModes

```
VOID SetTerminalModes (int iTerminalID, DWORD dwTerminalModes,  
                      BOOL fRouteToTerminal);
```

<i>iTerminalID</i>	Terminal identifier to adjust (0 to GetTerminalCount).
<i>dwTerminalModes</i>	Terminal mode(s) (LINETERMMODE_xxx) to adjust.
<i>fRouteToTerminal</i>	Whether the terminal is added or removed from modes.

Remarks

This is the function which is called when the terminal routing is adjusted by either TAPI or through the hardware. This stores or removes the specified terminal from the terminal modes given.

TAPI is notified about the change through a **LINECALLINFO_TERMINAL** event.

CTSPIAddressInfo::SetTrunkID

```
VOID SetTrunkID (DWORD dwTrunkID);
```

<i>dwTrunkID</i>	New trunk ID for the call.
------------------	----------------------------

Remarks

This worker function changes the current external trunk which is associated with the call. It may be used by the service provider to set the known trunk after the call has been created.

TAPI is notified about the change through a **LINECALLINFO_TRUNK** event.

CTSPICallAppearance::SwapHold

```
virtual LONG SwapHold(DRV_REQUESTID dwRequestID,  
    CTSPICallAppearance* pCall);
```

<i>dwRequestID</i>	Asynchronous request ID for this request.
<i>pCall</i>	Call appearance to swap with.

Remarks

This function is called in response to an application calling **lineSwapHold**. The **TSPI_lineSwapHold** function must be exported from the provider. The default function of the library is to verify both calls to insure the proper call states. Then to create and insert a **REQUEST_SWAPHOLD** request into the queue.

Return Value

TAPI error code, or the asynchronous request ID if the request was queued.

CTSPICallAppearance::Unhold

```
virtual LONG Unhold (DRV_REQUESTID dwRequestID);
```

<i>dwRequestID</i>	Asynchronous request ID for this request.
--------------------	---

Remarks

This function is called in response to an application calling **lineUnhold**. The **TSPI_lineUnhold** function must be exported from the provider. The default function of the library is to verify that the call is **onHold**, then to create and insert a **REQUEST_UNHOLD** request into the queue.

Return Value

TAPI error code, or the asynchronous request ID if the request was queued.

CTSPICallAppearance::Unpark

```
virtual LONG Unpark (DRV_REQUESTID dwRequestID,  
    CADOArray* parrAddresses);
```

<i>dwRequestID</i>	Asynchronous request ID for this request.
<i>parrAddresses</i>	Address to retrieve the call from.

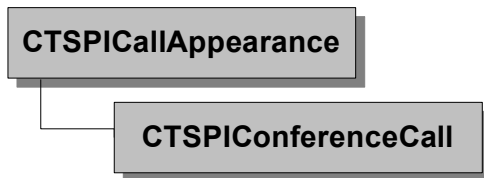
Remarks

This function is called in response to an application calling **lineUnpark**. The **TSPI_lineUnpark** function must be exported from the provider. The class library verifies that an address is given in the array, and submits a **REQUEST_UNPARK** request into the line queue.

Return Value

TAPI error code, or the asynchronous request ID if the request was queued.

CTSPIConferenceCall



A conference call in the TSP++ library is represented by a **CTSPIConferenceCall** object. This object is derived from the **CTSPICallAppearance** object and therefore behaves in the same fashion as any other call appearance. The main difference is that the conference call object is a placeholder for a series of call appearances that comprise the conference. As each call is added to the conference, this object will add the call appearance to a list of active conferenced calls. When this object is deleted or the state is changed to **Idle**, it will automatically de-associate all the call appearances which were previously part of the conference. Each of the calls within the conference are attached to this call object through their *related callid field* in the **LINECALLINFO** record. As each changes call state, the conference call object gets notified and performs various actions.

Conferences

Conference calls are call appearances that include more than two parties simultaneously. A conference call can be established in a number of ways, depending on hardware device capabilities. The capabilities of the line device may limit the number of parties conferenced into a single call and whether or not a conference must start with a connected call. It can also affect how the conference is broken down when it ends. The standard methods supported by TAPI to create a conference are:

- A conference call may need to be established with **lineSetupConference** without an existing two-party call. This returns a handle for the conference call and allocates a consultation call. After a period of consultation, the consultation call can be added with **lineAddToConference**.
- A conference call may begin as a regular two-party call, inbound or outbound. Calling **lineSetupConference** establishes the conference call. This operation takes the original two-party call as input, allocates a conference call, connects the original call to the conference, and allocates a consultation call whose handle is returned to the application. The original call is placed into the **onHoldPendingConf** state, and the new consultation call should be in the **Dialtone** state. The **lineDial** method can then be used on the consultation call to establish a connection to the next party to be added. Once connected to another party, the **lineAddToConference** method adds the new call to the conference.
- A three-way conference call can be established by resolving a transfer request into three-way conference. In this scenario, a two-party call is established as either an inbound or outbound call. Next the call is placed on transfer hold with the function **lineSetupTransfer** which returns a consultation call handle. After a period of consultation, the application may have the option to resolve the transfer setup by selecting the three-way conference option which conferences all three parties together in a conference call with **lineCompleteTransfer** with the **conference** option. Under this option, a conference call handle representing the conference call is allocated and returned to TAPI.

To add additional parties to an existing conference call, TAPI will invoke the **linePrepareAddToConference** method.. When calling this function, the application supplies the handle of an existing conference call. The method allocates a consultation call that can later be added to the conference call and returns a consultation call handle to TAPI. This conference call

is then transitioned to the **onHoldPendingConf** state. Once the consultation call exists, it can be added to the existing conference call with **lineAddToConference**. In some cases, it may be allowable to take any existing call and conference it into an existing conference by placing the call on hold and using **lineAddToConference**.

Once a call becomes a member of a conference call, the member's call state reverts to **Conferenced**. The state of the conference call typically becomes **Connected**. The call handle to the conference call and all the added parties remain valid as individual calls. As each member disconnects from the conference by hanging up, the appropriate call state messages are sent by the library to inform TAPI of this fact. TAPI may also direct for a call to be removed from the conference via the **lineRemoveFromConference** method. The **LINEDEVCAPS** in the **CTSPILineConnection** object describe how removal from a conference may occur.

Other notes about conferencing with TSP++

Since conference calls are generally hardware specific implementations, not much support is provided in the library for actual management of the conference other than the automatic tying together of all the calls (call association). When a **Drop** request is issued for a conference call handle, the derived service provider will probably have to perform some work with the calls within the conference. The **GetConferenceCount** and **GetConferenceCall** methods allow access to each call which transitions into the **Conference** state.

Operations - Public Members

AddToConference	This method manually builds a conference by adding a single call.
GetConference	Return a specific CTSPICallAppearance which has been conferenced in and is in the Conferenced state.
GetConferenceCount	Return the count of conferenced calls within this conference.
RemoveConferenceCall	Remove a call appearance from the conference.

Operations - Protected Members

CanRemoveFromConference	Returns a TRUE/FALSE result indicating whether the specified call may be removed from the owner conference.
IsCallInConference	Return a TRUE/FALSE result indicating whether the specified call appearance exists in our conference call list.

Overridables - Protected Members

OnCallStateChange	Overridden from the CTSPICallAppearance class in order to break-down a conference call in the proper order. When the conference call goes idle, any existing calls attached to the conference will transition to the idle state automatically <i>before</i> TAPI is notified about the conference call going idle.
OnRelatedCallStateChange	Overridden from the CTSPICallAppearance class to receive notifications when related calls change call state. This method is responsible to addition and deletion of calls from the conference.
OnRequestComplete	This method is used to monitor requests which complete and are associated with this conference.

Overridables - TAPI Members

AddToConference	Add an existing consultation call to an existing conference. Submits an asynchronous request. This is called by the lineAddToConference method.
PrepareAddToConference	Prepare to add a new call to the conference. Submits an asynchronous request. This is called by the linePrepareAddToConference method.

RemoveFromConference

Removes a call appearance which is conferenced in. This is call by the **lineRemoveFromConference** method.

CTSPIConferenceCall::AddToConference

```
virtual LONG AddToConference (DRV_REQUESTID dwRequestID,  
                             CTSPICallAppearance* pCall, TSPICONFERENCE* lpConf);
```

<i>dwRequestID</i>	Asynchronous request ID for this request.
<i>pCall</i>	Call to add to the conference
<i>lpConf</i>	Conference information structure

Remarks

This function is called in response to an application calling **lineAddToConference**. The **TSPI_lineAddToConference** function must be exported from the provider. The class library checks the call which is being added to the conference to make sure that either it is the consultant call created by a call to **TSPI_linePrepareAddToConference**, or that the provider can handle addition of *any* call to a conference (the **LINEADDRESSCAPS.dwAddrCapFlags** should have the **LINEADDRCAPFLAGS_CONFERENCEMAKE** flag set to zero if so).

Next, the library checks to see if the call is on the same address - if not, the **LINEDEVCAPS.dwDevCapFlags** must have the **LINEDEVCAPFLAGS_CROSSADDRCONF** flag set.

The callstate of the conference and call to add are checked, the **dwRelatedCallID** field of the consultation call is set to the conference call, and a **REQUEST_ADDTOCONF** request is submitted to the line.

Return Value

TAPI error code, or the asynchronous request ID if the request was queued.

CTSPIConferenceCall::AddToConference

```
void AddToConference(CTSPICallAppearance* pCall);
```

<i>pCall</i>	Call to add to the conference.
--------------	--------------------------------

Remarks

This worker function adds the specified call directly to the conference. It may be used by the service provider to build a conference directly off the hardware responses - specifically when a conference is being built through an external source, such as a phone or another extension.

CTSPIConferenceCall::CanRemoveFromConference

Protected

```
BOOL CanRemoveFromConference(CTSPICallAppearance* pCall) const;
```

pCall Call to remove from the conference.

Remarks

This worker function checks to determine if the specified call may be removed from the conference. It checks to see how a conference call must be disassembled according to the rules set into the **LINEADDRESSCAPS.dwRemoveFromConfCaps** field.

Return Value

TRUE if the call may be removed from the conference now. FALSE otherwise.

CTSPIConferenceCall:: CTSPIConferenceCall

```
CTSPIConferenceCall();
```

Remarks

This is the constructor for the conference call object.

CTSPIConferenceCall::~ ~CTSPIConferenceCall

```
~CTSPIConferenceCall();
```

Remarks

This is the destructor for the conference call object.

CTSPIConferenceCall:: GetConferenceCall

```
CTSPICallAppearance* GetConferenceCall(int iPos);
```

iPos Position within the conference array (0 to count).

Remarks

This worker function retrieves a call appearance which is part of the given conference call. Conference calls are stored in an array managed by the conference owner. The total count of members within the conference may be retrieved through the **CTSPIConferenceCall::GetConferenceCount** function.

Return Value

Pointer to the **CTSPICallAppearance** which is at the specified position in the conference array.
NULL if no call is at that position.

CTSPIConferenceCall:: GetConferenceCount

```
int GetConferenceCount() const;
```

Remarks

This worker function retrieves the count of calls which are members of the conference.

Return Value

Total number of calls that are part of the conference.

CTSPIConferenceCall:: IsCallInConference**Protected**

```
BOOL IsCallInConference(CTSPICallAppearance* pCall) const;
```

pCall Call object in question.

Remarks

This worker function walks the conference array and returns whether or not the specified call is part of the conference.

Return Value

TRUE if the call is currently in the conference. FALSE otherwise.

CTSPIConferenceCall::OnRequestComplete

Protected

```
virtual VOID OnRequestComplete (CTSPIRequest* pReq, LONG IResult);
```

<i>pReq</i>	Request which has completed
<i>IResult</i>	Final return code for the request.

Remarks

This method is called when a request is completed on the owner address/line. It gives the call object an opportunity to cleanup any data or information associated with the request.

The default implementation of the class library manages several requests:

REQUEST_REMOVEFROMCONF If the request completes successfully, the specified call is removed from the conference if it wasn't already by the service provider making the call **Idle**.

CTSPIConferenceCall::PrepareAddToConference

```
virtual LONG PrepareAddToConference(DRV_REQUESTID dwRequestID,  
HTAPICALL htConsultCall, LPHDRVCALL lphdConsultCall,  
TSPICONFERENCE* lpConf);
```

<i>dwRequestID</i>	Asynchronous request ID for this request.
<i>htConsultCall</i>	TAPI opaque handle to created consultation call
<i>lphdConsultCall</i>	TSP returning call handle to consultation call.
<i>lpConf</i>	Conference data structure.

Remarks

This function is called in response to an application calling **linePrepareAddToConference**. The **TSPI_linePrepareAddToConference** function must be exported from the provider. The class library validates the conference parameters, validates the total count within the conference, creates the consultation call and associates it with the conference, and finally, submits a **REQUEST_PREPAREADDDTOCONF** request to the line queue.

Return Value

TAPI error code, or the asynchronous request ID if the request was queued.

CTSPIConferenceCall:: RemoveConferenceCall

```
void RemoveConferenceCall(CTSPICallAppearance* pCall);
```

pCall Call object to remove from the conference

Remarks

This worker function removes the specified call from the conference.

CTSPIConferenceCall::RemoveFromConference

```
virtual LONG RemoveFromConference(DRV_REQUESTID dwRequestID,  
                                  CTSPICallAppearance* pCall, TSPICONFERENCE* lpConf);
```

dwRequestID Asynchronous request ID for this request.
pCall Call to remove from the conference
lpConf Conference data structure.

Remarks

This function is called in response to an application calling **lineRemoveFromConference**. The **TSPIC_lineRemoveFromConference** function must be exported from the provider. The class library validates the conference parameters, makes sure the call may be removed by using the **CTSPIConferenceCall::CanRemoveFromConference** function, and submits a **REQUEST_REMOVEFROMCONF** request into the line queue.

Return Value

TAPI error code, or the asynchronous request ID if the request was queued.

Asynchronous Request Data Structures

Each of the asynchronous requests has a pointer to a data structure with the information necessary to complete the request. In all cases, the line, address, and call appearance which are the owner of the request (or target of the request) can be obtained through the **GetConnectionInfo**, **GetAddressInfo**, and **GetCallInfo** methods of the **CTSPIRequest** object.

The following are the structure definitions for each of the requests:

DEVICEINFO

This structure is used to maintain information about *Device Class* properties that are associated with each object type. These device class structures are returned during the processing for **LINEDEVCAPS** and **TSPI_lineGetID**. This structure is only used in the 32-bit version of the library.

Member	Description
strName	Name of the device ("tapi/line")
dwStringFormat	String format (STRINGFORMAT_XXX)
lpvData	Data associated with device class (may be NULL)
dwSize	Size of data
hHandle	Win32 object handle which is copied to requesting process space.

DIALINFO

This structure is used to manage dial information from TAPI. A **CObject** array of these structures is passed as the data for **REQUEST_BLINDXFER**, **REQUEST_DIAL**, **REQUEST_REDIRECT**, and **REQUEST_UNPARK**.

Member	Description
flsPartialAddress	The address was "partial", and the call should not leave the dialing state. This was indicated in the dialable address by the trailing ";". The ";" is removed from the dial string.
strNumber	The number removed from the dialable address. This is a single address, with all invalid characters (see above list for valid characters) removed.
strName	The name which was attached to the dialable address
strSubAddress	The ISDN sub-address which was attached to the dialable address.

Rather than forcing the derived service provider class to break the information out of the dialable address format, the **CServiceProvider::CheckDialableAddress** method is provided to do all the work. The method is used to break a dialable address into its component parts. The function then returns an object array with potentially multiple **DIALINFO** structures in it. Each **DIALINFO** represents a single address found in a dialable address string. This array is then passed to the requests which need addresses (**lineMakeCall**, **lineRedirect**, **linePark**, etc.)

TSPICALLDATA

This is passed as the data structure for a **REQUEST_SETCALLDATA** command. It is used in the 32-bit library only.

```
class TSPICALLDATA : public CObject
{
    LPVOID lpvCallData;
```

```
    DWORD dwSize;
};
```

Member	Description
lpvCallData	Pointer to an opaque call-data information structure to set into the LINECALLINFO structure.
dwSize	Size of the above data block so it may be passed on a network connection.

TSPICALLPARAMS

This is passed as the data structure for a **REQUEST_SETCALLPARAMS** command.

class TSPICALLPARAMS : public COBJ

```
{
    DWORD dwBearerMode;
    DWORD dwMinRate;
    DWORD dwMaxRate;
    LINEDIALPARAMS DialParams;
};
```

Member	Description
dwBearerMode	New bearer mode for call. It has been validated against valid bearer modes for the address it is on.
dwMinRate	Low boundary for call data rate
dwMaxRate	High boundary for call data rate
DialParams	New dialing parameters

TSPICOMPLETECALL

This structure is passed to manage the **lineCompleteCall** method. It is passed with a **REQUEST_COMPLETECALL** request.

class TSPICOMPLETECALL : public COBJ

```
{
    DWORD dwCompletionMode;
    DWORD dwMessageId;
    DWORD dwSwitchInfo;
    CString strSwitchInfo;
};
```

Member	Description
dwCompletionMode	Completion mode requested (LINECALLCOMPLMODE_XXX). This has been validated against the LINEADDRESSCAPS field dwCallCompletionModes .
dwMessageId	Message id to forward to the station. This is a value from zero to the number of messages - 1. The field is validated against the LINEADDRESSCAPS field dwNumCompletionMessages which is setup by the CTSPIAddressInfo::AddCompletionMessage method.
dwSwitchInfo	Switch information adjusted by the derived service provider. This is a user-definable field to associate this completion request with a H/W request from the address. It is up to the derived service provider to come

up with a unique scheme so that the request may be identified later with **CTSPILineConnection::FindCallCompletionRequest**.
 strSwitchInfo Switch information adjusted by the derived service provider. This is a user-definable field to associate this completion request with a H/W request from the address. It is up to the derived service provider to come up with a unique scheme so that the request may be identified later with **CTSPILineConnection::FindCallCompletionRequest**.

TSPICONFERENCE

This structure is passed to manage the different conference events. It goes with: **REQUEST_ADDTOCONF**, **REQUEST_REMOVEFROMCONF**, and **REQUEST_SETUPCONF**. The requests are always inserted through the conference call object.

class TSPICONFERENCE : public Object

```
{
    CTSPIConferenceCall* pConfCall;
    CTSPICallAppearance* pCall;
    CTSPICallAppearance* pConsult;
    DWORD dwPartyCount;
    LPLINECALLPARAMS lpCallParams;
};
```

Member	Description
pConfCall	Conference call we are working with.
pCall	Call appearance we are adding or removing from the conference. If the request is a REQUEST_SETUPCONF , then this will be the initial call to start the conference off (if required). If the LINEADDRESSCAPS dwAddrCapFlags has the LINEADDRCAPFLAGS_SETUPCONFNULL flag set, then this field will be NULL, otherwise it will have a valid call appearance handle.
pConsult	This is the new consultation call which has been created for this conference call. This will be set to NULL on REQUEST_REMOVEFROMCONF requests.
dwPartyCount	This will be filled on the REQUEST_SETUPCONF request to the number expected in the conference. It is validated against the dwMaxNumConference field in the LINEADDRESSCAPS record.
lpCallParams	Optional LINECALLPARAMS from the application for a REQUEST_SETUPCONF . It will be NULL for all other request types.

TSPIFORWARDINFO

Single forwarding information structure. This is placed into the "arrForwardInfo" array for each forward entry found in the forward list.

class TSPIFORWARDINFO : public COBJect

```
{
    DWORD dwForwardMode;
    DWORD dwDestCountry;
    CADOArray arrCallerAddress;
    CADOArray arrDestAddress;
};
```

Member	Description
dwForwardMode	Forwarding mode for this request (LINEFORWARDMODE_xxx). This was validated against the forwarding types available in the LINEADDRESSCAPS dwForwardModes field.
dwDestCountry	Destination country for the address array
arrCallerAddress	Caller address information for specific forwarding mode. Depending on the forwarding mode, this array might be empty. If it has entries, they will be of type DIALINFO .
arrDestAddress	Destination address information. This should generally always have at least one address (and should generally only have one). The data will be of type DIALINFO .

TSPIGENERATE

This structure is passed to manage digit and tone generation. It is used with the **REQUEST_GENERATEDIGIT** and **REQUEST_GENERATETONE** requests.

class TSPIGENERATE : public CObject

```
{
    DWORD dwEndToEndID;
    DWORD dwMode;
    DWORD dwDuration;
    CString strDigits;
    CPtrArray arrTones;
};
```

Member	Description
dwEndToEndID	Unique identifier of this request to TAPI. This will be passed back to TAPI automatically if the request completes with a zero result code.
dwMode	Digit or Tone mode to generate (LINEDIGITMODE_xxx or LINETONEMODE_xxx). This has been validated against the LINEDEVCAPS field dwGenerateDigitModes and dwGenerateToneModes .
dwDuration	Duration to generate for. This has been adjusted to our minimum and maximum values in the LINEDEVCAPS fields.
strDigits	Digits to generate. Valid digits are 0-9, A-D, #, *.
arrTones	Array of LINEGENERATETONE structures for custom tones.

TSPHOOKSWITCHPARAM

This structure is passed in response to changes in a hookswitch on a phone device. It is passed on a **REQUEST_SETHOOKSWITCH**, **REQUEST_SETHOOKSWITCHVOL** and **REQUEST_SETHOOKSWITCHGAIN** request. When completed with a zero result code, the phone device will store the new values into the **PHONESTATUS** record.

class TSPHOOKSWITCHPARAM : public CObject

```
{
    DWORD dwHookSwitchDevice;
    DWORD dwParam;
};
```

Member	Description
--------	-------------

dwHookSwitchDevice	Hookswitch device to change. This is validated against the number of hookswitch devices added to the phone device through the AddHookSwitchDevice method.
dwParam	New value for the request. This will be a volume field, hookswitch state, or gain value, depending on the request type. It has been validated that this setting may be changed.

TSPILINEFORWARD

This is passed as the data structure for a **REQUEST_FORWARD** command.

class TSPILINEFORWARD : public CObject

```
{
    DWORD dwNumRings;
    CTSPICallAppearance* pCall;
    LPLINECALLPARAMS lpCallParams;
    CObArray arrForwardInfo;
};
```

Member	Description
dwNumRings	Number of rings before “no answer”
pCall	Consultation call appearance created to establish destination for forward. This will only be set to a valid call if the LINEADDRCAPFLAGS_FWDCONSULT flag is set in the LINEADDRESSCAPS dwAddrCapFlags field. Otherwise, it will be NULL.
lpCallParams	Optional calling parameters passed to lineForward , may be NULL.
arrForwardInfo	Array of TSPIFORWARDINFO structures.

TSPILINEPARK

This structure is passed as the data associated with a **REQUEST_PARK** command.

class TSPILINEPARK : public CObject

```
{
    DWORD dwParkMode;
    CADObArray arrAddresses;
    LPVARSTRING lpNonDirAddress;
};
```

Member	Description
dwParkMode	Park mode for this request (LINEPARKMODE_xxx). This has been validated against the LINEADDRESSCAPS dwParkModes field.
arrAddresses	Array of DIALINFO entries to park to. Generally there will only be one address in the array. The array could also be empty if a non-directed park is requested.
lpNonDirAddress	Return buffer for a non-directed park. This parameter needs to be checked before placing any data in it. It is a direct pointer to the structure passed to the linePark method and is in an application context.

TSPILINEPICKUP

This structure is passed as the data associated with a **REQUEST_PICKUP** command.

```
class TSPILINEPICKUP : public CObject
```

```
{
    CADObservable arrAddresses;
    CString strGroupID;
};
```

Member	Description
arrAddresses	Address(s) to pickup call at. This may have no addresses in it if there is a group id to an alerting station supplied. If there is an address, generally, there will be only one.
strGroupID	Group ID to which the alerting station belongs. This may be an empty string.

TSPILINESETTERMINAL

This structure is passed as the data associated with a **REQUEST_SETTERMINAL** command.

```
class TSPILINESETTERMINAL : public CObject
```

```
{
    CTSPILineConnection* pLine;
    CTSPIDialerInfo* pAddress;
    CTSPICallAppearance* pCall;
    DWORD dwTerminalModes;
    DWORD dwTerminalID;
    BOOL bEnable;
};
```

Member	Description
pLine	Line to change terminal information for (may be NULL)
pAddress	Address to change terminal information for (may be NULL)
pCall	Call appearance to change terminal information for (may be NULL)
dwTerminalModes	Terminal modes to change (LINE_TERMINALMODE_xxx).
dwTerminalID	Terminal ID. This is validated against our list of known terminals in the LINEDEVCAPS record. (These are added with the CTSPILineConnection::AddTerminal method).
bEnable	Whether to enable or disable terminal events.

TSPIMAKECALL

This structure is passed as the data associated with a **REQUEST_MAKECALL** command.

```
class TSPIMAKECALL : public CObject
```

```
{
    CADObservable arrAddresses;
    DWORD dwCountryCode;
    LPLINECALLPARAMS lpCallParams;
};
```

Member	Description
--------	-------------

arrAddresses	This field is the list of DIALINFO structures which were broken out of the <i>dialable address</i> passed to lineMakeCall . If multiple addresses are not supported according to LINEDEVCAPS , then only the first address passed will be in the array.
dwCountryCode	Country code for the dialing call.
lpCallParams	Optional call parameters passed from the application for this new call. If supplied, then the address and call appearance selected matches the call parameters or will function with the call parameters. Changes due to the call parameters were already marked on the call appearance (such as Secured , etc.)

TSPIPHONEDATA

This structure is used to set buffers into downloadable areas on a phone. It is passed as the parameter to a **REQUEST_SETPHONEDATA** and **REQUEST_GETPHONEDATA** request.

class TSPIPHONEDATA : public CObject

```
{
    DWORD dwDataID;
    LPVOID lpBuffer;
    DWORD dwSize;
};
```

Member	Description
dwDataID	Buffer to set/retrieve
lpBuffer	Area to store or set data buffer on from to/from.
dwSize	Size of above field

TSPIPHONESETDISPLAY

This structure is passed as the data associated with a **REQUEST_SETDISPLAY** command.

class TSPIPHONESETDISPLAY : public CObject

```
{
    DWORD dwRow;
    DWORD dwColumn;
    LPVOID lpvDisplay;
    DWORD dwSize;
};
```

Member	Description
dwRow	Row in display to modify
dwColumn	Column in display to modify
lpvDisplay	Display changes to make
dwSize	Size of above field

TSPIQOS

This structure is passed in response to the quality of service changing on the line or call. It is used in a **REQUEST_SETQOS** request. When completed with a zero result code, the call will set the quality of service fields in its **LINECALLINFO** structure with the new values.

```
class TSPIQOS : public CObject
{
    LPVOID lpvSendingFlowSpec;
    DWORD dwSendingSize;
    LPVOID lpvReceivingFlowSpec;
    DWORD dwReceivingSize;
};
```

Member	Description
lpvSendingFlowSpec	Pointer to a Windows Sockets FLOWSPEC structure for the quality of sending service along the line.
dwSendingSize	Size of the sending FLOWSPEC structure
lpvReceivingFlowSpec	Pointer to a Windows Sockets FLOWSPEC structure for the quality of receiving service along the line.
dwReceivingSize	Size of the receiving FLOWSPEC structure

TSPIRINGPATTERN

This structure is passed in response to setting changes on our ringer device. It is used in a **REQUEST_SETRING** request. When completed with a zero result code, the phone device will set the **PHONESTATUS** fields with the new values.

```
class TSPIRINGPATTERN : public CObject
{
    DWORD dwRingMode;
    DWORD dwVolume;
};
```

Member	Description
dwRingMode	This is the new ring mode for the phone. It has been validated against the PHONECAPS dwNumRingModes .
dwVolume	Volume for the ringer.

TSPISETBUTTONINFO

This structure is passed as the data associated with a **REQUEST_SETBUTTONINFO** and **REQUEST_SETLAMP** request.

```
class TSPISETBUTTONINFO : public CObject
{
    DWORD dwButtonLampId;
    DWORD dwFunction;
    DWORD dwMode;
    CString strText;
};
```

Member	Description
dwButtonLampId	Button/Lamp id to change. This has been validated against the count of buttons added with the AddButton method.
dwFunction	Function to set button to. This will be zero on a REQUEST_SETLAMP request.
dwMode	Mode for button or lamp.
strText	Text for button. This will be empty on a REQUEST_SETLAMP request.

TSPITRANSFER

This structure is passed to manage the different consultation transfer events. It goes with **REQUEST_SETUPXFER** and **REQUEST_COMPLETEXFER** requests. The call appearance which is the owner of the request is the inbound call which needs to be transferred.

class TSPITRANSFER : public CObject

```
{
    CTSPICallAppearance* pCall;
    CTSPICallAppearance* pConsult;
    CTSPIConferenceCall* pConf;
    DWORD dwTransferMode;
    LPLINECALLPARAMS lpCallParams;
};
```

Member	Description
pCall	Original inbound call which needs to be transferred.
pConsult	New outbound consultation call, or established consultation call for a REQUEST_COMPLETEXFER request.
pConf	New conference call used for REQUEST_COMPLETEXFER requests when they are to transition into a conference call. It will be NULL for a REQUEST_SETUPXFER .
dwTransferMode	Transfer mode (LINETRANSFERMODE_xxxx). This has been validated against the LINEADDRESSCAPS dwTransferModes field.
lpCallParams	Optional LINECALLPARAMS from the application for a REQUEST_SETUPXFER . It will be NULL for a REQUEST_COMPLETEXFER .

Creating a new service provider

Typical Overridden methods

The basic service provider implementation requires that a new class is derived from the **CServiceProvider** class. The derived class will generally always override the following functions of the service provider object:

Method	Description
Constructor	This will generally be overridden in order to allow the basic data objects such as the CTSPIDevice and CTSPILineConnection objects to be overridden.
providerInit	This method is the first one called when the service provider is loaded. If the line or phone device objects are not overridden, then it must perform general initialization such as providing names to all the lines available (via CTSPILineConnection::SetName) and creating all the addresses on the line through the CTSPILineConnection::CreateAddress method. It should always call the base class first. Any adjustments to LINEDEVcaps may be performed here.
providerEnumDevices	This function is called by TAPI to determine the number of phone and line devices supported by the provider. It should be overridden to support Plug and Play under Windows 95 and NT.
providerConfig providerInstall providerRemove lineConfigDialog ProcessData	These methods should be overridden to supply configuration and installation/removal rules.
OnTimer	This function is the workhorse of the service provider. Each service provider generally is a state machine processing responses from its telephony device. This function is called whenever the thread application calls the service provider with data from the device. The device is determined, and this method is passed a CTSPIDevice object along with the data retrieved from the device. This is an optional override, the more preferable approach is to override the line/phone objects and handle the ReceiveData function there. Many service providers require some method to time-out a telephony device. In case the device quits responding, or is detached from the system, a periodic interval timer will be requested from the thread application. This timer will then invoke this method when it is called by the executable.

General work flow

The first step in building a new TSP is to construct the service provider object. The constructor is passed the name of the service provider module used by the **CWinApp** object for any title-bar of message boxes. Next for 16-bit TSPs, it is passed the name of the companion application to start during **providerInit**. The supported TAPI version is passed next - this will be the highest version this provider will negotiate to. Finally, the provider information is last - this will be reported in the **LINEDEVcaps** structure under **dwProviderInfo**. The last two parameters are reversed for 32-bit TSPs.

If any of the basic objects needed overriding, the service provider constructor would be the place to issue a **SetRuntimeObjects** call.

The next override is to the **CTSPILineConnection** object. You can override the **Init** method to automatically add all the addresses necessary for the line to operate. Make sure to pass the request through to the base class first. Each address added is passed an address in dialable format with area code included for North American standards. This will be converted to canonical format when used for caller-id information. Also passed to the address created is a bearer mode. This should be a single mode, not multiple bits, and it will be added to the **LINEDEVCAPS dwBearerModes** field.

Once all the addresses are added, you can then adjust the device capabilities for the line object. No variable areas of the **LINEDEVCAPS** may be altered. The line capabilities and dialing parameters are changed here.

The main processing loop is the **ProcessData** method. This is always called in the context of our companion application or input thread, so any resources (such as serial ports or device driver connections) allocated won't be on an unknown application which could terminate. In the sample **ATSP32**, the input thread always passes the results from the modem as a token code defined in **atspint.h**. The current running request is pulled from the device queue and passed to a state machine designed to manage that request.

Each request type has a specific **processXXXX** function that performs the required hardware interactions to complete the request. As the device generates responses, the input thread will typically send responses from the hardware back through the **ProcessData** function which will perform the *next* step in the state machine.

The **ProcessData** function can be overridden in a multitude of places. The first, and most obvious, is the **CServiceProvider** object. It will see *all* data to and from all devices supported in the provider. The second is to override the **CTSPILineConnection** and **CTSPIPhoneConnection** objects and handle the data and processing for the specific device type (line or phone). This requires slightly more code, but makes a nice logical break in the coding process for the developer. Both methods are presented in the samples, the first is in the 16-bit **ATSP** sample, and the second, more object-oriented method, is in the **ATSP32** sample.

Asynchronous request handing

When a request is generated by a calling application, TAPI will call the appropriate **TSPI_XXXX** handler. The *spdll.cpp* layer will handle that command and call a virtual function in the **CServiceProvider** class. This method will validate the parameters, copy them to local buffers if necessary, and call a method of either the line, address, or call object in question. The lower object will perform more validations specific to itself, and if everything looks OK, initiate that command. For more information on this, see the section on **CTSPIRequest**.

An example of an asynchronous request

As an example, let's look at placing a call:

A TAPI application calls **lineMakeCall**

- TAPI determines that it is our service provider, and invokes the **TSPI_lineMakeCall** entry-point exported from our TSP.
- The *spdll.cpp* module determines which line device is being requested, locates the object and calls the **CServiceProvider::lineMakeCall** method passing it the **CTSPILineConnection** to place the call on.
- The **CServiceProvider::lineMakeCall** method will check various parameters for validity:
 1. It will call **CServiceProvider::CanHandleRequest** to make sure the function is available at this moment. This will by default return **TRUE** since it simply depends on the function being exported.

2. The user to user information (if present) is validated for size.
 3. The **LINECALLPARAMS** block is copied to local storage and the values in the structure are validated using the **CServiceProvider::ProcessCallParameters** method.
 4. The dialable address is verified and split into **DIALINFO** structures using the **CServiceProvider::CheckDialableNumber** method.
 5. A **TSPIMAKECALL** object is created and filled with all the information about the call to place.
 6. Finally, the **CTSPILineConnection::MakeCall** method is called and passed this **TSPIMAKECALL** structure.
- The **CTSPILineConnection::MakeCall** method will then perform additional validations on the information given.
 1. The line connection then validates the address specified in the **LINECALLPARAMS**, and locates the address information object, or uses the call parameters and selects an appropriate address (**CTSPIAddressInfo**) to hold the call based on media mode, bearer mode, etc. This is done using the **FindAvailableAddress** member.
 2. A new call appearance is created on the selected/found address using the **CTSPIAddressInfo::CreateCallAppearance** method.
 3. The **CTSPICallAppearance::MakeCall** method is called and passed the **TSPIMAKECALL** structure.
 - The call appearance then takes over and validates even more information associated with the call.
 1. First, the call state must be **Unknown** - the initial setting.
 2. It verifies that an outgoing call may be placed on its parent address (this may not be true if a specific address was selected through the **LINECALLPARAMS**).
 3. Next, the **LINECALLINFO** record is updated with default settings, or those selected in the **LINECALLPARAMS** information passed by the application.
 4. The caller id and called id information is setup from the address owner, and the destination address specified (only the first **DIALINFO** structure is used if multiple addresses were given).
 5. Finally, the call appearance calls the line device owner to add a **REQUEST_MAKECALL** asynchronous request.
 - When the line connection adds the request packet, if there is no pending request in the list, the connection will call the **StartNextCommand** method to initiate this request (this by default will call the **ProcessData** method with a NULL parameter). If not, it is simply queued at the tail of the list.
 - The asynchronous request id will be passed back from the call appearance, to the line, to the service provider object, and finally back to TAPI to indicate a successful start.
 - When data is received by the input thread or companion application it will invoke the service provider through the **DeviceNotify** function. This in turn will break up the connection id to determine the provider device, and the connection it is from. The **CTSPIDevice::ReceiveData** method will then be called. If it is an internal input thread, it would simply call the appropriate **ReceiveData** member function with the received data.
 - The **CTSPIDevice::ReceiveData** will then either call a specific **CTSPIConnection::ReceiveData** method if a specific connection was supplied by the companion application, or cycle through all the **CTSPIConnection** objects in both the line and phone arrays until one of them returns a TRUE response.

- Each of the target **CTSPICConnection** objects by default will call the **CServiceProvider::ProcessData** method. The first pending request retrieved by **CTSPICConnection::GetCurrentRequest** should be our current **CTSPIRequest** packet. The service provider may stay at this step several times depending on what is required to place a call on the device. As each step is completed, the call appearance should be updated to indicate its current status.
- Once the call has been placed on the device -or- has failed, the **CTSPICConnection::CompleteCurrentRequest** method should be called. This will perform the callback notification to TAPI indicating success/failure, and the request will be freed. If any other pending requests are in our queue, they will then be started with a **StartNextCommand**, and the whole process starts over.

As noted above, the **ProcessData** method is central to the service provider. In most implementations, the service provider will send the hardware a command, and then wait for a response. The response will then trigger a result, good or bad that will eventually complete the pending TAPI request. When data is returned by the executable thread application, it will be processed on that companion application thread's task. Therefore, it is a good idea to break the request into multiple steps in order to minimize the time spent within the service provider. This is especially important if the data input mechanism is time-critical and will not be buffered. In this case, a buffering mechanism in the thread executable might be warranted.

Since TAPI can have several outstanding requests for a connection, one request may affect others later in the queue. To allow for this, the **CTSPICConnection** class has methods to walk the request list (**FindRequest**, **GetRequest**, **GetRequestCount**), and delete a specific request (**RemoveRequest**). Also, when a line/call is dropped, pending requests for that line or call will also be dropped. A method (**RemovePendingRequests**) allows for this to happen based on a number of criteria (line/call/request).

Another potential situation is requiring that a request finish synchronously. This is especially useful if some synchronous operations require multiple steps to complete. A new request type may be generated and inserted into the device list. It would then be processed in the standard **ProcessData** fashion. In this case, the **WaitForRequest** may be called to wait (with optional time-out) for the request to finish. Messages are pumped during this waiting period. This method may require that the processing code be re-entrant since one thread will be blocked at the wait, while the executable thread continues to process data. Special care should be taken to make sure that the execution thread that is servicing the hardware is not paused in this function. When building a 16-bit provider, never use this function on the companion application context thread.

Functions to implement

Depending on the features available in the hardware, different functions will need to be implemented by the service provider. The following chart shows some of the relationships between which request objects should be processed by the **ReceiveData** or **ProcessData** function and which **TSPI_xxx** function needs to be exported from the service provider in order for TAPI to recognize the functionality.

Feature	Export Function	Request To Handle
Make a call	TSPI_lineMakeCall	REQUEST_MAKECALL
Dial on an existing call	TSPI_lineDial	REQUEST_DIAL
Drop a call	TSPI_lineDrop	REQUEST_DROPCALL
Answer an incoming call	TSPI_lineAnswer	REQUEST_ANSWER

Accept an offering call	TSPI_lineAccept	REQUEST_ACCEPT
Place a call on hold	TSPI_lineHold	REQUEST_HOLD
Swap between consultation and holding call	TSPI_lineSwapHold	REQUEST_SWAPHOLD
Take a call off hold	TSPI_lineUnhold	REQUEST_UNHOLD
Transfer a call unsupervised	TSPI_lineBlindTransfer	REQUEST_BLINDXFER
Transfer a call supervised	TSPI_lineSetupTransfer TSPI_lineCompleteTransfer	REQUEST_SETUPXFER REQUEST_COMPLETEXFER
Issue a call completion	TSPI_lineCompleteCall	REQUEST_COMPLETECALL
Cancel a call completion	TSPI_lineUncompleteCall	REQUEST_UNCOMPLETECALL
Forward a line/address	TSPI_lineForward	REQUEST_FORWARD
Park a call	TSPI_linePark	REQUEST_PARK
Unpark a call	TSPI_lineUnpark	REQUEST_UNPARK
Pickup a call from other station	TSPI_linePickup	REQUEST_PICKUP
Redirect a call to other station	TSPI_lineRedirect	REQUEST_REDIRECT
Secure a call from interference	TSPI_lineSecure	REQUEST_SECURECALL
Send message using phone display or other notification	TSPI_lineSendUserToUser	REQUEST_SENDUSERINFO
Change the terminal associated with an extension	TSPI_lineSetTerminal	REQUEST_SETTERMINAL
Monitor standard DTMF pulses on the call	TSPI_lineMonitorDigits	Complete the request REQUEST_MONITORDIGITS with a zero return code and call OnDigit when digits are seen on the call
Monitor Media changes on the line	TSPI_lineMonitorMedia	Complete the request REQUEST_MONITORMEDIA with a zero return code and call OnDetectedNewMediaModes when media changes are seen on the call.
Monitor tones generated on the call	TSPI_lineMonitorTones	Complete the request REQUEST_MONITORTONES with a zero return code and call OnTone when media changes are seen on the call

Gather incoming digits on the call	TSPI_lineGatherDigits	Complete the request REQUEST_GATHERDIGITS with a zero return code and call OnDigit when digits are seen on the call
Generate DTMF or pulse digits on a connected call	TSPI_lineGenerateDigits	REQUEST_GENERATEDIGITS
Generate custom tones on a connected call	TSPI_lineGenerateTone	REQUEST_GENERATETONE
Begin building a conference call with an existing single party call	TSPI_lineSetupConference	REQUEST_SETUPCONF
Prepare to add a call to the conference (create a consultation call)	TSPI_linePrepareAddToConference	REQUEST_PREPAREADDCONF
Add a call to the conference	TSPI_lineAddToConference	REQUEST_ADDTOCONF
Remove a call from the conference	TSPI_lineRemoveFromConference	REQUEST_REMOVECONF

ATSP V2 Sample Service Provider

The 16-bit ATSPV2 sample is a functional modem service provider. It is similar in function to the sample provided in the 16-bit TAPI SDK. It manages a standard HAYES compatible modem and performs general dial-out services.

TSPI Exports

The exports from this sample model the *Basic Telephony* services which are required for any TSP which may be used in Windows Telephony. For information on these exports and their purpose, consult the *Microsoft Telephony Programmer's Guide*.

TSPI_providerConfig	TSPI_lineConditionalMediaDetection	TSPI_lineDial
TSPI_providerInit	TSPI_lineConfigDialog	TSPI_lineDrop
TSPI_providerInstall	TSPI_lineGetAddressCaps	TSPI_lineDropOnClose
TSPI_providerRemove	TSPI_lineGetAddressId	TSPI_lineDropNoOwner
TSPI_providerShutdown	TSPI_lineGetAddressStatus	TSPI_lineGetCallStatus
TSPI_providerEnumDevices	TSPI_lineGetLineDevStatus	TSPI_lineGetDevCaps
TSPI_lineGetCallAddressId	TSPI_lineGetNumAddressIds	TSPI_lineGetId
TSPI_lineGetCallInfo	TSPI_lineNegotiateTSPIVersion	TSPI_lineMakeCall
TSPI_lineSetAppSpecific	TSPI_lineSetDefaultMediaDetection	TSPI_lineOpen
TSPI_lineSetCallParams	TSPI_lineSetStatusMessages	TSPI_lineSetMediaMode
TSPI_lineCloseCall	TSPI_lineClose	

Files

The ATSPV2 project is located in the **ATSP** directory and consists of the following main files:

Service Provider files	
File	Description
STDAFX.CPP	Pre-compiled header support
ATSP.CPP	Main service provider code
CONFIG.CPP	Configuration dialog code
TALKDROP.CPP	Talk/Drop dialog
LINE.CPP	CTSPILineConnection override
REQUEST.CPP	Line request management code
UI.CPP	TSP User interface functions

Companion application files	
File	Description
STDAFX.CPP	Pre-compiled header support
ATSPEXE.CPP	Drives the serial port for the service provider.

Basic class structure

CATSPPProvider

The first class which is overridden in the library is the service provider class **CServiceProvider**. In this case, the ATSP sample derives a class called **CATSPPProvider**. This class handles the overriding of the other objects through the constructor, and overrides the high-level **TSPI_providerXXX** functions.

Method	Purpose
CATSPPProvider	Constructor for the service provider. It gives the CServiceProvider class its required parameters and sets the objects in place to override other TSP++ classes through the SetRuntimeObjects method.
providerEnumDevices	This is not required, but is recommended. It is used in Plug & Play devices which can dynamically determine the number of line and phone devices supported. It always returns a single line.
lineGetID	This provides a simple override to return the 16-bit COMM handle used to talk to the modem. This support is better standardized under Win32.
OpenDevice	This is a new function which sends a command to the companion application to open the serial port. It passes a provider-defined structure which details the port, baud, and a connection id to identify further communication.

CATSPLine

The second class overridden by the sample is the **CTSPILineConnection** class. All of the support for the line device is managed in this override.

Method	Purpose
Init	This function is called after the constructor by the device owner. In our sample, it configures the line and sets up the addresses supported on the line.
OnTimer	This is an override of the interval timer method. It calls into the request handler for timeout conditions.
ReceiveData	This is the main processing loop of the service provider. All device traffic flows through this function.
OpenDevice	This is overridden to stop the device from being opened when the line is opened by TAPI. See the <i>Implementation Notes</i> for more information on this override.
CloseDevice	This is overridden to stop the device from being opened when the line is opened by TAPI. See the <i>Implementation Notes</i> for more information on this override.
processXXX	These are the worker functions which process the individual requests in the line queue. They are called from the ReceiveData method depending on the current request being managed.

Basic Processing Flow

The line object provides a state machine driven from the **ReceiveData** member. The input to this function is twofold. First, all device responses are gathered by the serial communications code in the companion application and passed to this function. Second, the interval timer which goes off every 2.5 seconds sends a **MODEM_INTERVAL_TIMER** event through the function to handle timeout conditions from the modem. The TSP library requests supported by the **ReceiveData** method are:

REQUEST_MAKECALL
REQUEST_SETCALLPARAMS
REQUEST_DIAL
REQUEST_DROPCALL
REQUEST_DROPCALLONCLOSE
REQUEST_DROPNOOWNER

The requests are handled sequentially in this sample. In the case of a modem, only one event can be going on at any given time - i.e. if we are making a call we must wait for the modem to respond to the initial **AT** commands before we can send any further. As the device responds to each **AT** commands sent, the **ReceiveData** function sends the response along with the request packet at the head of the line queue through a **processXXX** function which then divides the command being processed into a state machine applicable for the modem. As the call is placed, the call appearance is altered with **CTSPICallAppearance::SetCallState**, and TAPI updates the applications.

Installation

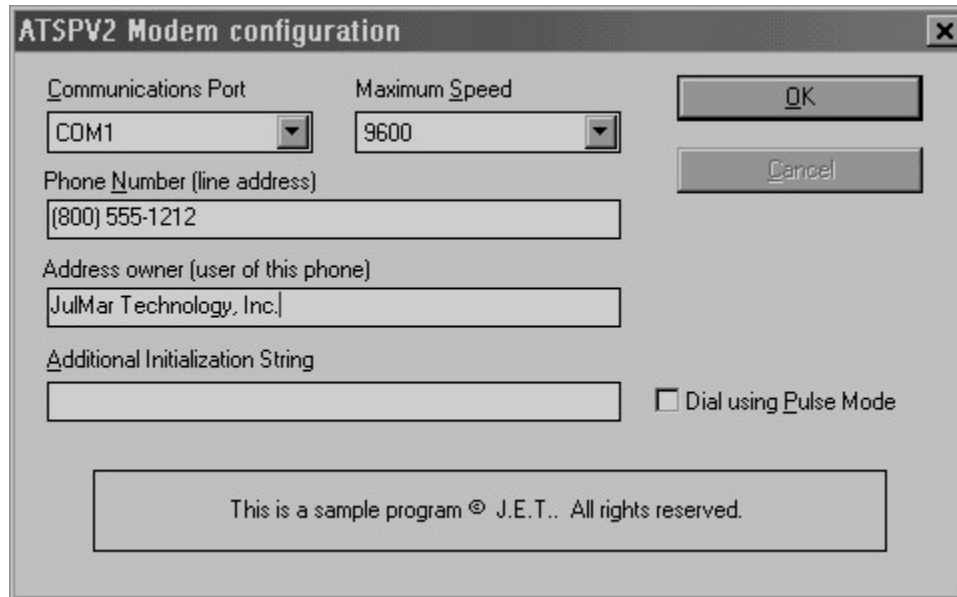
Installation of this service provider is done through the installation applet INSTALL.EXE. The Telephony control panel applet may be used to configure the provider. (**Note:** under Windows 95, the TAPI control panel applet is not automatically installed, but is copied to the **WINDOWS/SYSTEM** directory under the name **telephon.cp\$**. This should be copied to the file **telephon.cpl**).

Simply copy the ATSPV2.TSP and ATV2EXE.EXE files to the Windows SYSTEM directory and run the INSTALL application to add the provider to the system.

Configuration

Once the provider is installed, you may configure it using the Telephony control panel applet.

When configure is clicked, the following screen will be shown.



The image shows a Windows-style dialog box titled "ATSPV2 Modem configuration". It contains several input fields and buttons. The "Communications Port" is a dropdown menu set to "COM1". The "Maximum Speed" is a dropdown menu set to "9600". The "Phone Number (line address)" is a text box containing "(800) 555-1212". The "Address owner (user of this phone)" is a text box containing "JulMar Technology, Inc.". The "Additional Initialization String" is an empty text box. There is a checkbox labeled "Dial using Pulse Mode" which is currently unchecked. At the bottom, there is a large text box containing the text "This is a sample program © J.E.T.. All rights reserved.".

The **Communications Port** is the port that the modem is connected to.

The **Speed** is the baud rate to set the Communications Port to when connecting to the modem.

The **Phone Number** is the dialable phone number of the line the modem is connected to. This will be used for Caller ID reporting when outgoing or incoming calls are processed.

The **Address Owner** is a name to assign to the address. This will be used as the party name for Caller ID reporting.

The **Initialization String** is any special modem commands which are required to support the modem. This typically isn't required, but may be used if a non-standard modem is used or the modem has distinct differences from the Hayes standard.

The **Dial Using Pulse Mode** switch indicates that the TSP should issue all dialing commands using pulse mode vs. tones. This should only be clicked if the line the modem is connected to is not a touch-tone phone.

INI file keys

Configuration information is stored in the TELEPHON.INI, using the **ReadProfileString**, **ReadProfileDWORD**, **WriteProfileString**, and **WriteProfileDWORD** functions. These functions automatically assign the section in the registry. Configuration is driven from the configuration dialog (in **CONFIG.CPP**) and started by the **lineConfig** or **providerConfig** functions (in **UI.CPP**). The keys stored in the TELEPHON.INI are:

Ini File Key	Description
Port	COMM port to use (1-4)
Speed	Speed to open COMM port at
PulseDial	1 = Pulse, 0 = Tone dialing
InitString	Optional initialization string.
LineAddress	Phone # of address

LineName

Name of user on line (caller id)

Implementation Notes for ATSPV2

- When a line or phone is opened in the 16-bit TSP++ library through the **TSPI_lineOpen** or **TSPI_phoneOpen**, the **CTSPILineConnection** or **CTSPIPhoneConnection** object calls its **OpenDevice** method to open the physical device. This normally calls the **CTSPIDevice::OpenDevice** method of the device owning the line or phone. This in turn normally calls **CServiceProvider::OpenDevice** to send a **COMMAND_OPENCONN** message to the companion application.

In this service provider, we manage a physical communications port from our companion application, and don't need to actually open the device until a call is placed on the device (it doesn't support incoming calls). This can be especially important depending on what applications are running - since the communications port is a shared resource, once it is opened by a service provider, all others are locked away from it. So, to override this behavior, our **CATSPLine** class replaces the **OpenDevice** and **CloseDevice** methods with functions that do nothing. Then, when a call is actually placed on the line, the **CServiceProvider::OpenDevice** is called as the first step in placing a call (**REQUEST_MAKECALL**), and **CServiceProvider::CloseDevice** is called when the call terminates.

- Standard AT class modems (which this service provider supports) cannot determine when a call connects unless a carrier is detected. So, unless the destination is a modem, it is not possible to transition the call to a **Connected** state. To correct for this behavior, a dialog is shown as soon as the call transitions to the **Proceeding** state and begins *busy-detection*. This dialog allows the user of the TAPI application which called the **lineMakeCall** function originally to indicate that they have picked up the phone and connected to a call. This *talk-drop* dialog is managed by our **CATSPProvider** object, and created/destroyed dynamically as needed.
- A sample **lineGetID** is provided which allows the communications handle to be retrieved by calling 16-bit applications. This demonstrates how a new device-id function would be implemented.

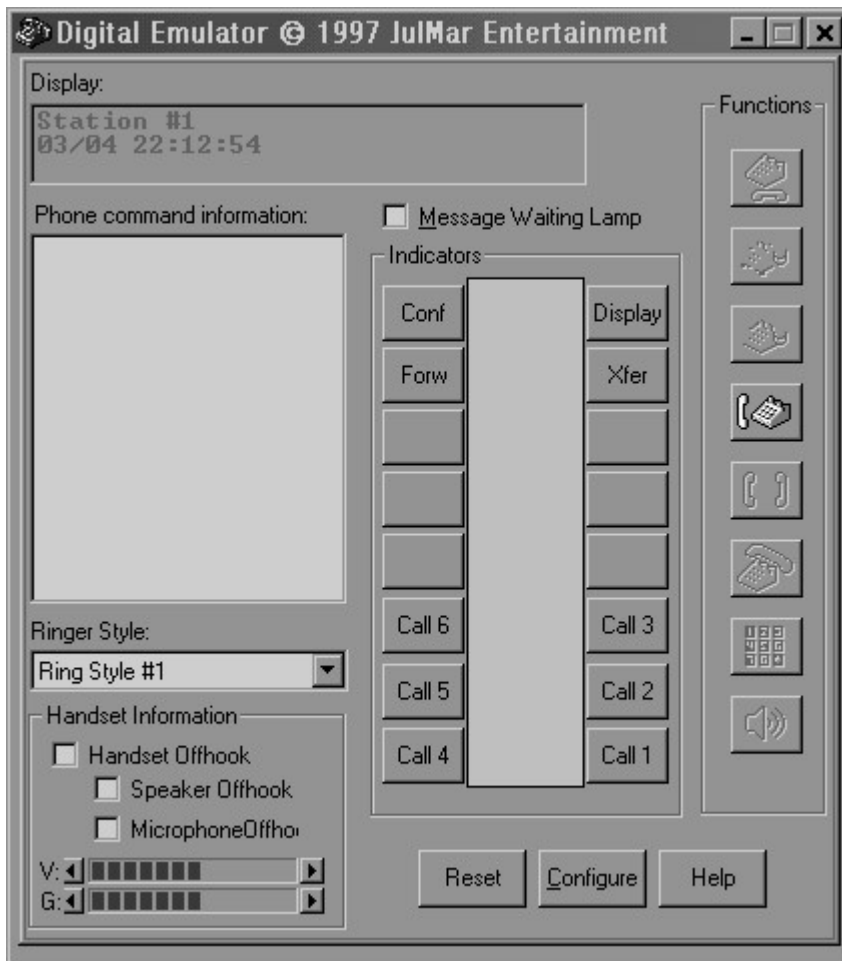
Digital Switch Emulator

The **EMULATOR** program is a component of the **DSSP** and **DSSP32** samples. It acts like a telephone switch and station in a single program. The 16-bit version of the emulator must run on the same workstation as the TSP it is being driven by. The 32-bit version of the emulator which works with **DSSP32** may run on any network machine, but you must have TCP/IP connectivity to use the sample.

For information about the **DSSP** or **DSSP32** samples, see the following sections on each applicable TSP sample.

Running the Emulator

The first step in testing out the **DSSP** samples is always to start the emulator program. It must be running before any TAPI application is started or the service provider will return **LINEERR_NODEVICE** and cause TAPI to issue an error about a provider not starting. The time-out period is five seconds, so the system will seem to pause for about five seconds while the provider searches for the emulator. Once up, the emulator looks like:



Each of the buttons on the emulator causes some event to happen to our *simulated* PBX switch.

- ❑ The **Handset Information** group indicate what state our handset is in. The microphone and gain are controlled separately. If they are clicked upon, TAPI will get notified about a hookswitch change. If TAPI changes the state of the hookswitch device, then the checkboxes will also change.

The volume and gain may be controlled through the buttons located and by the service provider using the **phoneSetVolume** and **phoneSetGain**.

- ❑ The **Message Waiting** button causes our Message Waiting lamp to turn on and off. This also is bi-directional in the service provider.
- ❑ Each of the function buttons (16 in all) running in the center of the phone may be setup as a different function. In the listed picture, 5 are call appearances, and 3 are features of the switch. The others are not set. When the **DSSP** sample first initializes, it will query the emulator about its settings and set itself up appropriately. The button information is retrievable from the TAPI function **phoneGetButtonInfo**. If any of the advanced line features are required (conference, transfer, forward), then a button *must* be defined in this set of 16. Otherwise, a **LINEERR_OPERATIONUNAVAIL** will be returned by the service provider.
- ❑ The buttons along the bottom of the emulator correspond to **Reset**, **Configure**, and **Exit**.. The emulator should be configured while no TAPI devices are running. This insures that the provider and the emulator stay in synch. Once the service provider starts, the emulator will not allow the configuration button to be pressed.
- ❑ The buttons along the right side of the emulator screen correspond to the following functions:



Hold

Place the active call (solid lamp) on hold.



Release

Hang up on the active call and reset the lamp.



Dial

Dial a series of digits on the LOCAL handset.



SimulateCall

Create an incoming call on one of the addresses.



Busy

Simulate a busy signal on outgoing call



Answer

Answer the incoming call from the LOCAL handset.



DialRemote

Simulate dialing events on REMOTE handset



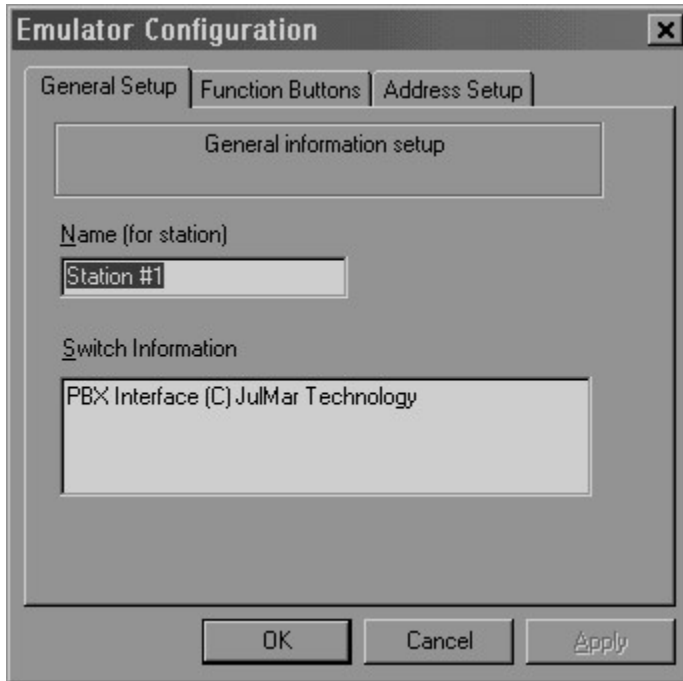
Tones

Simulate tone generation from REMOTE handset

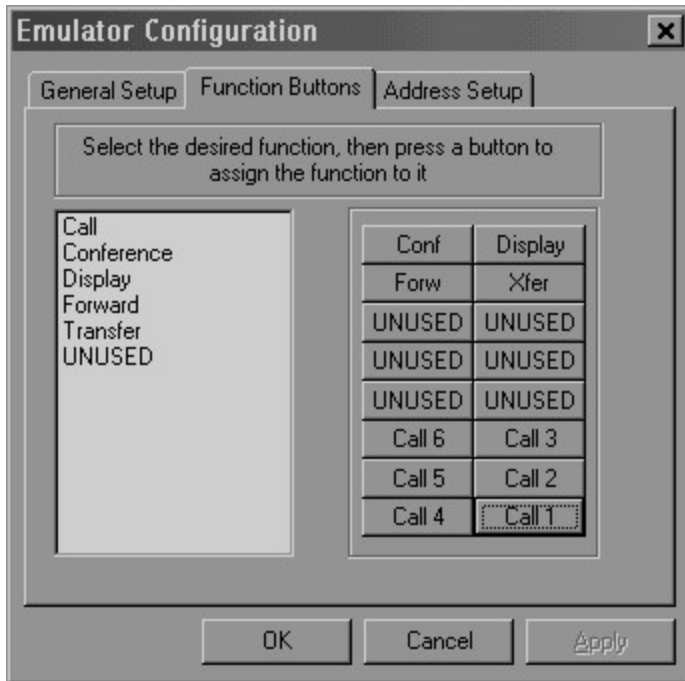
- ❑ The **Phone Command Information area** will contain a list of what the service provider is asking for, and how the emulator is responding. As each command is executed by the emulator program, detailed information about state changes being reported and commands received from the service provider are listed. The color schemes used are:
 - **Green** for service provider commands
 - **Red** for emulator commands
 - **Blue** for status changes.
- ❑ The display will correspond to what the emulator is telling the service provider, and provides an emulation for the **phoneGetDisplay** API. It is currently read-only, although the **phoneSetDisplay** function will return a success result.

Configuring the Emulator

The first step in using the emulator is to configure it. On initial installation, the emulator will have no call appearances available to it. This means that no addresses are available for the line it is attached to, which is generally not a useful phone. The configuration button is the last button on the right of the dialog and will bring up a dialog which looks like:



The general tab allows the name for the station to be entered. This will be the caller ID information reported for any call placed on this address (not received). Also, the switch information is listed here. This information will be returned by the service provider in the **LINEDEVCAPS** and **PHONEDEVCAPS** structures.



The function button tab allows each of the buttons to be configured to a function. To configure a button simply select the function in the listbox on the left, and click the appropriate button on the right. Any button assigned to a call must then be setup with address information using the next tab. The current available functions are:

Call - This is a call appearance. There is no limit (except in buttons) to the number of available call appearances within the emulator. Each is considered a separate address, and only one address may be in an active state at any given time (the others must be inactive or in some hold state).

Display - This is used to update the display. It is available as a feature to the emulator, and is not currently used within the service provider (plans for use are forthcoming).

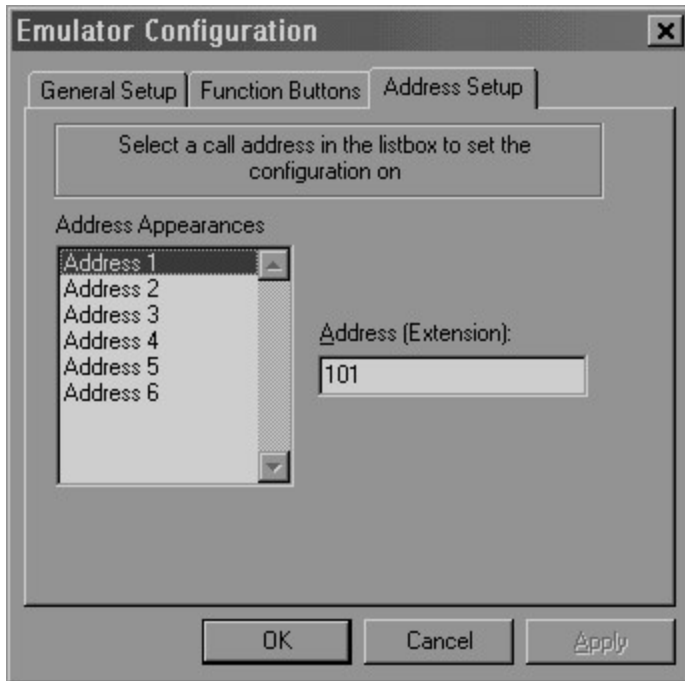
Conference - This implements a simply 3-way conference call. It is not usable within the emulator, but the service provider will require that the button is defined to support conferencing. The conference features of the emulator are currently under work for cross-addressing and more than three parties.

Forward - This implements a station-wide forwarding system. If the button is pressed within the emulator, it will forward to a bogus number (listed in the display). The service provider allows the forwarding to be to any dialable number. Specific call forwarding, internal/external forwarding, and busy forwarding are not supported.

Transfer - This implements a simple transfer facility which can be driven by the service provider. A consultant call is provided in the emulator in the **Dialtone** state, which can then be dialed and transferred. This form of transfer (called a *consultation transfer*) is only available to the service provider. Blind transfer is the default and is used by both the emulator and service provider if the button is pressed directly.

Unused - this is the default state and should be used for any button not defined to a function.

The **Forward**, **Conference**, and **Transfer** buttons must be defined if these features will be available to the service provider.



The address setup tab lists each address which was created by assigning a button to a call appearance. Each address needs an extension provided for it. The extensions should be unique phone numbers which identify the address and will be used in caller ID information.

16-bit Digital Switch Sample Service Provider (DSSP)

The **DSSP** sample is a service provider written for a telephone network with more advanced features than a modem. It works in conjunction with an emulator program that acts like a telephone switch and phone in one (see the previous section on the *DSSP Emulator*).

NOTES:

The provider is a sample, the configuration of the hardware for which a service provider is written really drives the implementation of the provider. For example, in our sample, we assume that if a conference call is dropped, the whole conference is dropped. This may not be the case on the hardware another provider is driving. The code presented here is an example of an implementation - not a guideline for any implementation for a switch.

Important note for Windows 95 and TAPI 2.1

The communication mechanism between the companion application and the emulator uses Windows messages to send and receive data. This mechanism does not work completely under TAPI 2.1 and Windows 95. This is because of a deadlock situation arising due to the 16/32 bit hybrid architecture of Windows 95 and TAPI 2.1 which doesn't exist in Windows 3.1 or under TAPI 1.4. Run the **DSSP32** sample and 32-bit emulator under TAPI 2.1 instead.

Installation

Before any TAPI applications are started, make sure to start the emulator program. Installation of the **DSSP** sample is accomplished by copying the **DSSP.TSP** and **DSSPEXE.EXE** files to the Windows **SYSTEM** directory and running the **INSTALL** program to add the provider to the system.

Configuring DSSP

Configuration information is managed by the emulator and retrieved by the service provider at startup time. To change the configuration, shut down all TAPI applications and then re-configure the emulator.

Using the DSSP Sample Provider

Once the emulator is started and configured (the configuration will only need to be done the first time), the **DSSP** sample provider may be started by invoking any TAPI application. The **Debug Information** window will list any connections made by the **DSSP** sample provider.

Files

The **DSSP** project is located in the **DSSP** directory and consists of the following main files:

Service Provider files located in DSSP\SP	
File	Description
DRV.CPP	This contains the communication methods for directing traffic to and from the companion application.
INIT.CPP	This contains the initialization code.
MISC.CPP	This contains misc. function routines for the service provider.
CONFIG.CPP	This contains the configuration overrides.
DSSP.CPP	This contains the main processing code for the service provider.
LINE.CPP	This contains the CTSPILineConnection override code.

LINEREQ.CPP	This contains the request management code for the line.
PHONE.CPP	This contains the CTSPiPhoneConnection override code.

Companion application files located in DSSPIAPP	
File	Description
DSSPEXE.CPP	Performs the communication with the emulator program using Windows messaging and provides an interval timer to the sample service provider.

Basic class structure

This samples, like the **ATSP** sample above, uses a series of classes and overrides to create the service provider.

CDSPProvider

The derived service provider class is **CDSPProvider**. It provides the basic shell for the service provider which sets up the classes to manage TAPI requests. It contains a constructor which uses the **CServiceProvider::SetRuntimeObjects** method to establish an override for the device, line and phone objects.

Method	Purpose
CDSPProvider	Constructor for the service provider. It gives the CServiceProvider class its required parameters and sets the objects in place to override other TSP++ classes through the SetRuntimeObjects method.
providerEnumDevices	This is not required, but is recommended. It is used in Plug & Play devices which can dynamically determine the number of line and phone devices supported. It always returns a single line and single phone.
lineGetID	This provides a simple override to return WAV devices for input and output to a simulated media stream (the default multimedia input/output devices).

CDSDDevice

The sample also creates a **CDSDDevice** class which manages communication with the actual device through the companion application. It contains driver methods which are called by the line and phone objects to perform work on the emulated switch.

CDSLLine

This overrides the **CTSPiLineConnection** and manages all the initialization and state machines for the line device which is modeled by the TSP. Almost all of the TAPI **TSPI_lineXXX** functions are supported. The restrictions of the line device are:

1. Only unconditional forwarding is supported. The device may not be forwarded unless there are *no* active calls on the device.
2. **linePickup** will return an error result since there is no way to see a call outside the emulator device.

CDSPhone

This overrides the **CTSPiPhoneConnection** and manages all the initialization and state machines for the phone device which is modeled by the TSP. Almost all of the TAPI **TSPI_phoneXXX** functions are supported. The restrictions of the phone device are:

1. **phoneSetDisplay**, **phoneSetButton**, **phoneSetLamp** will all return success but perform no function on the emulator.

Basic Processing Flow

In this sample provider, each request in the line or phone queue is given the opportunity to examine device responses and decide if it is applicable to that request. This is accomplished in the line/phone **ReceiveData** override. As a device response is processed, the line/phone method enumerates through each request and gives it the chance to see the packet. As soon as one of the **processXXX** functions returns an indicator saying that it processed the response, the enumeration stops. As in the **ATSP** sample, each request is sub-divided into a state machine by a **processXXX** function which is applicable for the device and function being invoked.

If no request handled or expected the response, then it is processed by an *unsolicited* event processor. This function handles events that come from the device which were unexpected and typically happened as a result of some outside influence on the device (i.e. the user pressing a button on the emulator).

Implementation Notes

- The interface between the companion application and the service provider follows the same guidelines as the **ATSP** sample, except that on a few of the commands, a structure is passed to the companion application to have the emulator fill out. This is only supported with 16-bit companion application. When a 32-bit companion application is used, the pointers passed using the **SendThreadMessage** method are considered read-only. The data in the structure is copied into the address space of the called 32-bit application, and therefore any alterations will not affect the real structure which was passed. This is a restriction of the **WM_COPYDATA** message which is used to thunk the data. If the provider needs an interface similar to this, then a 16-bit companion application *must* be used. Another possible approach would have been to define new **RESULT_xxx** codes from the companion application, and have the structure passed back through the **DeviceNotify** function of the provider.

32-bit Digital Switch Sample Service Provider (DSSP32)

The **DSSP32** sample is a service provider written for a telephone network with more advanced features than a modem. It works in conjunction with an emulator program that acts like a telephone switch and phone in one. (For more information on this, see the preceding section on the *DSSP Emulator*).

The 32-bit version of the emulator uses Windows Sockets to communicate with the service provider. For this reason, you *must* have TCP/IP installed on the machine that the emulator runs on and the machine that the service provider runs on. They can be the same machine if desired, if this is the case, configure the service provider to use **127.0.0.1** as the IP address of the server.

NOTE:

The provider is a sample, the configuration of the hardware for which a service provider is written really drives the implementation of the provider. For example, in our sample, we assume that if a conference call is dropped, the whole conference is dropped. This may not be the case on the hardware another provider is driving. The code presented here is an example of an implementation - not a guideline for any implementation for a switch.

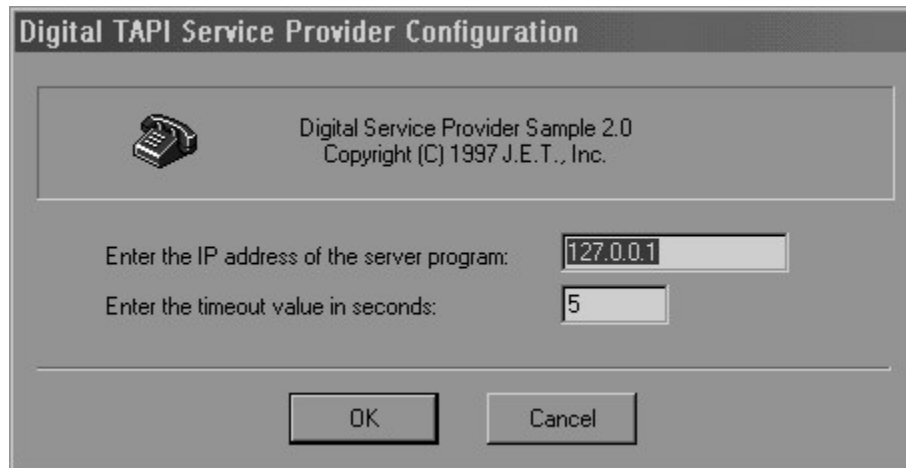
Installation

Before any TAPI applications are started, make sure to start the emulator program. Installation of the **DSSP32** sample is accomplished by copying the **DSSP32.TSP** file into the WINNT\SYSTEM32 (for Windows NT) or WIN95\SYSTEM (for Windows 95 with TAPI 2.1) directory and using the Telephony Control panel applet to add the provider to the system.

When the service provider is first installed, it will prompt for configuration information. See the section on *Configuring the DSSP Sample* for information on what each field means.

Configuring the DSSP32 sample

Since the 32-bit sample is designed to take advantage of client/server technology, it has a configuration screen (unlike its 16-bit predecessor). When you configure the **DSSP32** sample, the following screen will appear:



The **IP Address** indicates where the server **EMULATOR** program is running. If it is running on the same machine as the server provider (local), then enter **127.0.0.1** as the IP address. Otherwise, enter the dotted-decimal number indicating where the server is running.

The timeout value indicates how long the service provider will wait for the server to accept a pending connection during the initialization process. The longer this timeout, the longer the provider waits for the server. This means that if the server isn't running, or the network is down, the service provider will wait *x* seconds and pause the telephony subsystem (which will effectively pause the calling application during the **lineInitialize** function).

Using the DSSP32 Sample Provider

Once the emulator is started and configured (the configuration will only need to be done the first time), the **DSSP** sample provider may be started by invoking any TAPI application. The **Debug Information** window will list any connections made by the **DSSP** sample provider.

Files

The DSSP project is located in the **DSSP** directory and consists of the following main files:

File	Description
DRV.CPP	This contains the communication methods for directing traffic to and from the companion application.
INIT.CPP	This contains the initialization code.
MISC.CPP	This contains misc. function routines for the service provider.
CONFIG.CPP	This contains the configuration overrides.
DSSP.CPP	This contains the main processing code for the service provider.
LINE.CPP	This contains the CTSPILineConnection override code.
LINEREQ.CPP	This contains the request management code for the line.
PHONE.CPP	This contains the CTSPIPhoneConnection override code.

Basic class structure

This samples, like the **ATSP** sample above, uses a series of classes and overrides to create the service provider.

CDSProvider

The derived service provider class is **CDSProvider**. It provides the basic shell for the service provider which sets up the classes to manage TAPI requests. It contains a constructor which uses the **CServiceProvider::SetRuntimeObjects** method to establish an override for the device, line and phone objects.

Method	Purpose
CDSProvider	Constructor for the service provider. It gives the CServiceProvider class its required parameters and sets the objects in place to override other TSP++ classes through the SetRuntimeObjects method.
providerEnumDevices	This is not required, but is recommended. It is used in Plug & Play devices which can dynamically determine the number of line and phone devices supported. It always returns a single line and single phone.
lineGetID	This provides a simple override to return WAV devices for input and output to a simulated media stream (the default multimedia input/output devices).

CDSDevice

The sample also creates a **CDSDevice** class which manages communication with the actual device through the companion application. It contains driver methods which are called by the line and phone objects to perform work on the emulated switch.

CDSLine

This overrides the **CTSPILineConnection** and manages all the initialization and state machines for the line device which is modeled by the TSP. Almost all of the TAPI **TSPI_lineXXX** functions are supported. The restrictions of the line device are:

- Only unconditional forwarding is supported. The device may not be forwarded unless there are *no* active calls on the device.
- linePickup** will return an error result since there is no way to see a call outside the emulator device.

CDSPhone

This overrides the **CTSPIPhoneConnection** and manages all the initialization and state machines for the phone device which is modeled by the TSP. Almost all of the TAPI **TSPI_phoneXXX** functions are supported. The restrictions of the phone device are:

- phoneSetDisplay**, **phoneSetButton**, **phoneSetLamp** will all return success but perform no function on the emulator.

Basic Processing Flow

In this sample provider, each request in the line or phone queue is given the opportunity to examine device responses and decide if it is applicable to that request. This is accomplished in the line/phone **ReceiveData** override. As a device response is processed, the line/phone method

enumerates through each request and gives it the chance to see the packet. As soon as one of the **processXXX** functions returns an indicator saying that it processed the response, the enumeration stops. As in the **ATSP** sample, each request is sub-divided into a state machine by a **processXXX** function which is applicable for the device and function being invoked.

If no request handled or expected the response, then it is processed by an *unsolicited* event processor. This function handles events that come from the device which were unexpected and typically happened as a result of some outside influence on the device (i.e. the user pressing a button on the emulator).

Implementation Notes

1. The **DSSP32** sample uses a secondary thread to communicate with the emulator. The implementation relies on the MFC architecture and creates a *user-interface* thread which basically means it has a **CWnd** handle map associated with this secondary thread. This is simply for the **CSocket** support in MFC. The device thread manages the socket interface to the emulator and passes all responses to the **CTSPIDevice::ReceiveData** function which will then pass it through the line and phone objects.

ATSP32 Sample Service Provider

The 32-bit ATSP sample is a functional modem service provider. It is similar in function to the sample provided in the Win32 SDK. It manages a standard HAYES compatible modem and performs general dial-out/incoming call services using the COMM driver.

Installation

Installation of the **ATSP32** sample is accomplished by copying the **ATSP32.TSP** file into the WINNT\SYSTEM32 (for Windows NT) or WIN95\SYSTEM (for Windows 95 with TAPI 2.1) directory and using the Telephony Control panel applet to add the provider to the system.

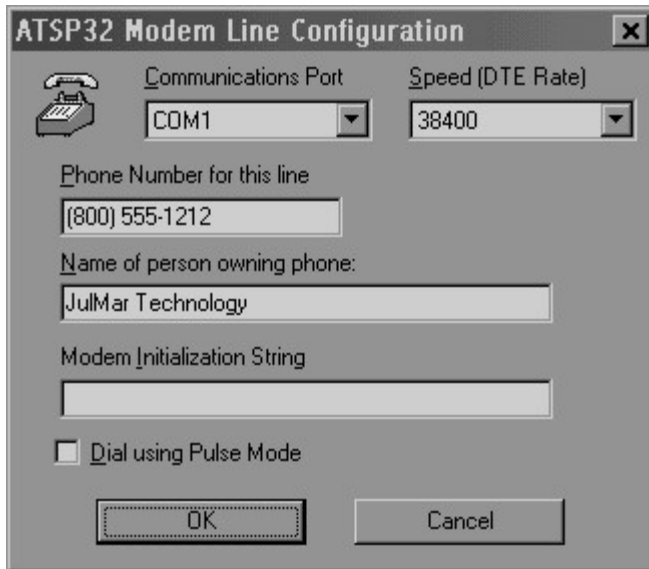
When the service provider is first installed, it will prompt for configuration information. See the section on *Configuring the ATSP32 Sample* for information on what each field means.

Configuring the ATSP32 sample

The **ATSP32** sample supports multiple modes and exposes them to TAPI as a multi-line device. Each modem configured is added as a single line on the provider. The initial configuration screen is to add a single modem to the system.



The first step is to create a new modem by pressing the **Add** button. This will generate the following dialog:



The **Communications Port** is the port that the modem is connected to.

The **Speed** is the baud rate to set the Communications Port to when connecting to the modem.

The **Phone Number** is the dialable phone number of the line the modem is connected to. This will be used for Caller ID reporting when outgoing or incoming calls are processed.

The **Address Owner** is a name to assign to the address. This will be used as the party name for Caller ID reporting.

The **Initialization String** is any special modem commands which are required to support the modem. This typically isn't required, but may be used if a non-standard modem is used or the modem has distinct differences from the Hayes standard.

The **Dial Using Pulse Mode** switch indicates that the TSP should issue all dialing commands using pulse mode vs. tones. This should only be clicked if the line the modem is connected to is not a touch-tone phone.

Once the modem is configured, press **OK** to dismiss the box. The new modem is now ready to be used. To add additional modems, simply repeat the process.

Note: You cannot use multiple lines in this sample on the same COM port address. This is because this sample opens the modem when the **lineOpen** is performed by the application rather than when the call is made. This difference from the 16-bit provider is to support incoming calls.

Registry Information

Configuration information is stored in the registry, using the new **ReadProfileString**, **ReadProfileDWord**, **WriteProfileString**, and **WriteProfileDWord** functions. These functions automatically assign the section in the registry under the **Telephony** key. Configuration is driven from the configuration dialog (in **CONFIG.CPP**), and is called from the UIDLL portion of the provider in **UI.CPP**. The keys used in the registry are:

Ini File Key	Description
ComPort	COMM port to use (1-4)
Speed	Speed to open COMM port at
PulseDial	1 = Pulse, 0 = Tone dialing
InitString	Optional initialization string.
LineAddress	Phone # of address

LineName

Name of user on line (caller id)

Files

The ATSP32 project is located in the **ATSP32** directory and consists of the following main files:

File	Description
ATSP.CPP	Main service provider code
CONFIG.CPP	Modem Configuration dialog code
PCONFIG.CPP	Modem addition/removal dialog code
DEVICE.CPP	COMM port device code
LINE.CPP	Overrides of the CTSPILineConnection object.
QUEUE.CPP	I/O queue code
REQUEST.CPP	Request handlers for the device
TALKDROP.CPP	Code for the Talk/Drop spontaneous dialog.
UI.CPP	User-interface handlers for UIDLL management.

Basic class structure

The service provider derived here is the **CATSPProvider** object. It simply provides the constructor and the **providerEnumDevices** override for specifying the number of supported lines and phones. Note that this override is *required* for a 32-bit TSP.

The device class is overridden to support a modem device. It provides methods to open and close the COMM port device to read/write to the modem. It uses a supplementary class for queue management which is contained within the **QUEUE.CPP** file. All reads/writes to the modem come through this class object.

The line class is overridden to provide an implementation for the **ReceiveData** method rather than processing data in the **CServiceProvider** class as in the other examples. In addition, the **Open** and **Close** methods are overridden to call the device class for opening/closing the COMM device. The supported requests driven from this class are:

REQUEST_ANSWER

REQUEST_DIAL

REQUEST_DROP_CALL

REQUEST_MAKE_CALL

REQUEST_SET_CALL_PARAMS

Each of the appropriate **TSPI_xxxx** functions are exported in **atsp.def**. Each request is then parceled off to a **processXXX** function which then sub-divides the state machine to a smaller state machine applicable for the modem. As the call is placed, the call appearance is altered with **SetCallState**, and TAPI will automatically receive the notifications.

Implementation Notes

- Several threads are used to process input in this example. The first thread is created in the library and is used for an interval timer. This is controlled through the **CServiceProvider::SetIntervalTimer** function (zero turns it off). In addition, a thread is created for each line device created. The dedicated thread reads from the device queue and processes input from the modem for the state machine. It is created in the **CATSPLine::Init** method and runs in the **CATSPLine::InputThread** method. Another thread is created for each monitored COMM port. This COMM thread is created during the first call to the **CATSPDevice::OpenComm** method and is suspended and resumed each time the device is opened and closed. It is responsible for pulling information from the queue and reading/writing it to the physical COMM device.

- Standard AT class modems (which this service provider supports) cannot determine when a call connects unless a carrier is detected. So, unless the destination is a modem, it is not possible to transition the call to a **Connected** state. To correct for this behavior, a dialog is shown as soon as the call transitions to the **Proceeding** state and begins *busy-detection*. This dialog allows the user of the TAPI application which called the **lineMakeCall** function originally to indicate that they have picked up the phone and connected to a call. This *talk-drop* dialog is managed using the UIDLL process under Windows NT, and is run from the **TALKDROP.CPP** file.
- The released version of TAPI 2.0 under Windows NT 4.0 uses two threads to create spontaneous UI dialogs. Since a second thread is used to run the UI event (through **providerGenericDialog**), MFC has not been setup properly on the thread. Rather than writing code to initialize MFC, the sample simply doesn't use it for the talk/drop dialog - it is standard SDK code due to its simplicity. See the section on *User Interface Dialogs* for more information on this problem.
- The standard device class "comm/datamodem" is established in the **lineMakeCall** processing which allows a 32-bit application to read/write the COMM port through the provider. It is removed when the call terminates. This feature utilizes the new **AddDeviceClass** methods.

Main Processing Logic

The first step in **ATSP32** is to construct the service provider object. This object is derived from the **CWinApp** object, which is required by MFC. The constructor is passed the name of the UI DLL to use for any user-interface logic, followed by a descriptive name of the provider - this will be reported in the **LINEDEVCAPS** structure under **dwProviderInfo**. If any of the basic objects needed overriding, this would be the place to issue a **SetRuntimeObjects** call. All 32-bit TSPs must conform to TAPI 2.0 standards, so the version information is not required.

The next step done in the **ATSP32** sample is when TAPI calls the **providerInit** function. This will be the first function call made by TAPI, and it can occur more than once if the provider is supporting multiple devices and is listed more than once in the telephony settings.

The first thing done is to pass the request onto the base library. This will create line and phone objects for the device being initialized. It will start the interval timer thread and return a success indicator. Once this completes, a new device object will be in place and have lines and phone objects attached to it. During this call, the **CTSPILineConnection** and **CTSPIPhoneConnection** objects will be created and initialized.

During the creation of the line objects, **ATSP32** adds the addresses to the line. Each address added is passed an address in dialable format with area code included for North American standards. This will be converted to canonical format when used for caller-id information. Also passed to the address created is a bearer mode. This should be a single mode, not multiple bits, and it will be added to the **LINEDEVCAPS dwBearerModes** field.

Once all the addresses are added, **ATSP32** then adjusts the device capabilities for the line object. No variable areas of the **LINEDEVCAPS** may be altered. The line capabilities and dialing parameters are changed here.

Either zero or the error detected by the library should be returned here. If a non-zero result is returned, then TAPI will unload the service provider and give an error to the user.

The main processing loop in this sample is performed by the line object class. The **ReceiveData** function is called to process input for the line device. By default it calls the **CServiceProvider::ProcessData** function which is managed in the other sample providers. In this sample, the **ReceiveData** function is overridden here and managed on a line by line basis. The line object spawns a thread and monitors the device input queue (maintained by the **CATSPDevice** class) for receipt of information from the modem. When any data is received, it calls the **ReceiveData** function with the token parsed from the input string.

The **ReceiveData** finds the current running **CTSPIRequest** and calls a **processXXX** function to manage the request. Each **processXXXX** function performs the required hardware interactions to complete the request listed in the **CTSPIRequest** object. Multiple calls will be sent to the **processXXX** function as input is sent/received from the device

Testing and Debugging Service Providers

Debugging service providers has always been a painful process. Most service providers are fairly time-critical, meaning that stopping the provider in a debugger is generally not an option when the device expects replies in a timely fashion. Also, since a service provider is a dynamic link library, and dynamically loaded by TAPI as needed, it can be difficult to get it to stop in the provider without a fatal fault.

Number one rule: always build the internal versions of the provider using **DEBUG** mode. Both MFC and the TSP++ library have extra debugging validations that are performed to insure that they are being used properly.

Some Hints for debugging 16-bit TSPs

First of all, to debug the provider, it is generally best to debug it under Windows 3.1 or Windows 95. NT doesn't have support for 16-bit low-level debugging, and that makes it difficult to really determine what is happening. Also, it is generally better to use CodeView for debugging the provider - it allows a DLL to be loaded as part of *any* executable it is debugging - whether that executable uses the DLL doesn't matter. So, by debugging a generic 16-bit program (such as the TAPI browser **tb13.exe**), you can then use the **Run/Load** menu option to force the DLL to load into memory, and then place breakpoints anywhere you like.

Some good functions to remember for breakpoints are:

Function	Description
CTSPIDevice::CompleteRequest	This is called when any asynchronous request completes for whatever reason.
CTSPIDevice::AddAsynchRequest	This is called whenever an asynchronous request is generated anywhere in TSP++.

Just to watch the calls being made, turn on **DBWIN.EXE** which watches all debug output from applications. All TSP calls made will be reflected in the output.

An excellent TAPI program to start testing with is the TAPI Browser (**TB13/TB14/TB20**). It allows direct calls to be made to TAPI, changing the parameters, and even supplying bad parameters to the provider. It is available from Microsoft, and is now a part of the Win32 SDK.

A couple of notes for developing with the 16-bit library:

When in the **ProcessData** method, the service provider will always be on the context of the companion application (if it was called from the library). All requests are inserted in the context of the calling TAPI application.

If no request was pending when a new request was inserted, the library will force a context switch to the companion application and **ProcessData** will be entered to start the request. The requesting TAPI application is blocked until the first state (**STATE_INITIAL**) completes. This means that the request *could* finish, notify TAPI, and be deleted *before* the synchronous response is passed back through to TAPI. This will happen if there is only one state before **CompleteCurrentRequest** is called to delete the request.

Some Hints for debugging 32-bit TSPs

We recommend that you purchase a low-level debugger in order to hook into the address space of the **TAPISRV.EXE** application. An example is NuMega's Soft/Ice for Windows NT. The same functions are used as above, and by using Soft/Ice, you can set break points through the **DebugBreak** API in Win32. If you wish to debug with the developer studio, you can attach the system debugger (MSDEV.EXE, WINDBG.EXE, etc.) through the task manager. Invoke the task manager by right clicking on the task bar at the bottom of the screen. Select the "Processes" tab, Locate the **TAPISRV.EXE** module and right-click on it. Click "Debug". The system debugger specified in the registry will start up and hook the process. This will not allow for debugging the startup code since the process must be running, but will allow for watching the output through the debug window.

Watch what version you pass into the **lineInitialize** function. TAPI negotiates to the *lowest* common version. So, if you say you can handle everything from 1.3 to 2.0, you will end up negotiating to 1.3 when running under Windows 95 or NT. To specifically target a version of TAPI, make sure to specify that version as the **dwLowVersion** parameter.

Debug TAPI components

Another recommendation is to use the debug TAPI components while testing your provider. They are with the other debug symbols and components provided with the Win32 SDK. You can also retrieve them with the TAPI 2.1 installation provided by Microsoft.

Once you have the debug versions of the TAPI system components installed, you can adjust the level of output given by each using the registry and creating/changing DWORD values found under the registry key

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Telephony

The values range from 0 (no debug output) to 4 (maximum debug output). Value names follow the convention "<ModuleName>DebugLevel", e.g., "Tapi32DebugLevel" or "TapisrvDebugLevel".

Other debugging notes

The MFC **TRACE** facility is used by the TSP++ library on the front and back-end of all **TSPI_xxxx** functions called by the TAPI dynamic link library. Before jumping into a debugger, sometimes the problem can be found just by watching the return values for the TSPI functions called. There are some helper functions that are used by TSP++ to dump buffers that can be used by developers as well. These functions are:

Function	Description
DumpMem	Dumps a memory buffer of specified length in HEX and ASCII to the debug terminal window using the MFC TRACE facility.
DumpVarString	Dumps a VARSTRING pointer in HEX and ASCII to the debug terminal window using the MFC TRACE facility.

Note that under Windows NT and TAPI 2.0, you must have a debugger attached to the **TAPISRV.EXE** process in order to see the trace output.

General notes about the library:

- When a **lineClose** occurs, an automatic wait is issued for any pending **lineDrop** requests to insure that all the call handles get closed down. It is *extremely* important to insure that the **Drop** requests are always handled and completed, even if they fail (send back and

- error in this case), but always call the **CompleteRequest** or **CompleteCurrentRequest** method to release any blocked threads.
- Many of the set methods for the objects will store their information back into the object once a request completes successfully. These include most of the **CTSPICallAppearance** methods such as **SetCallTreatment** or **SetCallerIDxxx**. It is passed through the library so that the derived class may do the appropriate changes to the hardware for the information being passed, or fail the request. If the **CompleteRequest** method is called with a result of zero, the information will be considered valid and set into the appropriate structure. TAPI is always notified.

Debug Strings used in the TSP++ Library

The TSP++ Library has a TRACE mechanism that allows the provider to format strings in variable-argument form and output them to the debug terminal. This feature allows for tracing time-related events where a debugger is not possible, or when the developer simply wants to verify that something is happening.

As part of this process, the debug version of the TSP++ library uses the TRACE facility itself to output entry and exit points from the library. Each of the **TSP_xxxx** entry-points which are called by TAPI have a series of TRACE events associated with them that dumps out the contents of input variables and output structures. This allows the developers to monitor which entry-points are being invoked by TAPI and see what the provider did in response to those calls. For information on each of the **TSPI_xxxx** entry-points, please consult the *Microsoft TSPI Reference Guide* available in MSDN.

In addition to the entry/exit point tracing, the TSP++ library also outputs various operations and state changes as it works. The following lists the most common trace events that might be seen in the life of a typical service provider operation.

CreateConferenceCall: SP call=0x#####, TAPI call=0x#####

This is reported in response to the creation of a new conference call within the provider. Specifically, a call to **CTSPIAddressInfo::CreateConferenceCall** will cause this trace event. The *SPCall* parameter is the memory address of the service provider **CTSPICallAppearance** object, and the *TAPICall* parameter is the passed or created **HTAPICALL** which is used by TAPI to represent the call.

CreateCallAppearance: SP call=0x#####, TAPI call=#####

This is reported in response to the creation of a new call within the provider. Specifically, a call to **CTSPIAddressInfo::CreateCallAppearance** will cause this trace event. The *SPCall* parameter is the memory address of the service provider **CTSPICallAppearance** object, and the *TAPICall* parameter is the passed or created **HTAPICALL** which is used by TAPI to represent the call.

Address Call Counts NewCallState==#, OldCallState==#, Active==#, OnHold==#, OnHoldPend==#

This is reported when the number of calls on an address changes. Specifically, when a call changes from some passive state to an active state, is created, or is destroyed, TAPI is notified that the call count has changed on the address. This trace event is a record of that notification. The *NewCallState* and *OldCallState* record the current and previous call state which caused this change, and the *Active/OnHold/OnHoldPend* fields represent the current counted **Active** calls (i.e. in the *Connected* state), the number of *OnHold* calls, and the number of *OnHold* calls which are pending transfer/conference finality. Calls which are *Idle* or *Conferenced* are not reflected in the call counts.

Deleting invalid consultation call <#####>

This is reported when an asynchronous event fails and the TSP++ library is cleaning up a previously created consultation call for the event. This is done only if the call was never reported to any application

(i.e. it never entered any initial call state). The number is the memory address of the service provider **CTSPICallAppearance** object.

LINEADDRCAPS.dwAddressFeatures missing ##### bit
LINEADDRESSCAPS.dwCallFeatures doesn't have ##### in it.

This is reported when the address features are changed through the **CTSPIAddressInfo::SetAddressFeatures** call but the new feature bits are not all part of the capabilities of the address setup by the provider. It is a warning that the service provider developer has incorrectly set the capabilities of the address object. The bit(s) listed are the missing bits from the reported features mask.

Address ID:#### [NAME] DIALABLE_ADDRESS

This is reported when the **CTSPIAddressInfo::Dump** method is called. It reports the address identifier from TAPI, the given name for the address and the DN for the address.

Deleting call appearance #####

This is reported when the TSP++ library is de-allocating a deleted call appearance. The number is the memory address of the service provider **CTSPICallAppearance** object.

WARNING: Call appearance cannot transition out of the IDLE state!

This is reported when a call appearance attempts to transition from the *Idle* state to some other state. This is not allowed in the TAPI model and will generate this warning (and ASSERT).

Call=#####, CallStateChange Notify=1/0, FROM <STATE> to <STATE>

This is reported when a call appearance changes from one state to another. It is representative of the actual event notification made to TAPI if the *Notify* flag is set to '1'. If the *Notify* flag is a zero, TAPI is *not* notified. The number is the memory address of the service provider **CTSPICallAppearance** object.

WARNING: Attempted to dynamically create line without TAPI support
WARNING: Attempted to dynamically create phone without TAPI support

This message is reported when the service provider attempts to dynamically create a line/phone device without the **TSPI_providerCreateLineDevice** or **TSPI_providerCreatePhoneDevice** functions exported.

Dynamically created line object <#####>
Dynamically created phone object <#####>

This is reported when the service provider dynamically adds a line or phone device to the system. The number reported is the memory address of the **CTSPIConnection** object. This will be followed by a call to the appropriate **TSPI_providerCreatexxxDevice** entry-point by TAPI.

Connection <#####> changing device id to #####

This is reported when a previously dynamically added line is assigned a permanent line/phone identifier by TAPI. The connection number is the memory address of the **CTSPIConnection** object, and the reported device id is taken directly from the **TSPI_providerCreatexxxDevice** event.

<#####> created new asynch request: <#####>

This is reported when a line/phone object creates a new asynchronous request. The first number is the memory address of the **CTSPIConnection** object that represents the owner of the request. The second number is the memory address of the created **CTSPIRequest** object.

Request <#####> canceled by OnNewRequest

This is reported when the **CServiceProvider::OnNewRequest** method cancels an asynchronous request. Note that this will never happen by default – the default behavior for this function is to allow creation of the request. The number is the memory address of the **CTSPIRequest** object that was canceled.

Completing request <##### TYPE>, rc=#####, TellTapi=1/0, RemoveRequest=1/0

This is reported when a request is completed using **CompleteRequest** or **CompleteCurrentRequest**. The number is the memory address of the **CTSPIRequest** object that was completed, the *TYPE* is the request type (i.e. *MakeCall*, *Drop*, etc.). The *RC* is the return code reported to TAPI. The *TellTAPI* flag indicates whether TAPI is to be notified of the completion (a zero indicates that the return code is to be simply stored into the request). The *RemoveRequest* indicates whether the request is *really* done and needs to be deleted. Again, a zero indicates that the request is *not* done and will stay in the request queue.

CTSPIRequest: ##### NAME Conn:#####, Call:#####, State:##, Data=#####, TAPISComplete=Yes/No

This is reported when the **CTSPIRequest::Dump** method is used to dump a request object. The first number is the memory address of the **CTSPIRequest** object in question. The *Connection* is the line/phone object address, the *Call* is the **CTSPICallAppearance** object address. The *State* is the current numeric state of the request. The *Data* field is the data buffer address. *TAPISComplete* indicates whether TAPI has been notified that the request has completed.

User-defined request - make sure to delete m_lpData in destructor

This message is a reminder to delete the data buffer associated with user-defined **CTSPIRequest** objects. It is only issued when an unrecognized request is destroyed.

Warning: Request object had invalid data buffer - supposed to be object!

This message indicates that the data buffer associated with a **CTSPIRequest** object is corrupt. It is a warning that something changed the object and that when the destructor was called, the data buffer couldn't be deleted.

Deleting request <#####>***Removing pending request <#####>***

These two messages are reported when a request is deleted, the first because it was completed and then removed. The second because some other previous request caused this particular request to be canceled (a *Drop* for example). The number is the memory address of the **CTSPIRequest** object that was deleted.

Starting next asynch request

This is reported when a request is completed and a new request that was pending in the request list was started.

Line #####, ID:#### [NAME]***Phone #####, ID:#### [NAME]***

This is reported when the **CTSPIConnection::Dump** method is called. It reports the memory address of the line/phone object, the device id assigned by TAPI, and the *NAME* of the device assigned by the provider.

**Device #####, Base Line=#####, Count=#####, Base Phone=#####, Count=#####,
Completion Callback function = #####**

These messages are reported when the **CTSPIDevice** object is created and initialized by the **TSPI_providerInit** event. The reported variables indicate how many lines/phones are created and the device identifiers that they are assigned.

**Adding Line ##### (id #####) to device list
Adding Phone ##### (id #####) to device list**

These messages are reported when the **CTSPIDevice** object creates the line/phone objects which represent the physical lines/phones initialized by the **TSPI_providerInit** function. The first number is the array position within the device object list, and the second number is the device identifier assigned to that line/phone by TAPI.

Line ##### Active=##, OnHold=##, OnHoldPend=##

This is reported when the number of calls on a line changes. Specifically, when a call changes from some passive state to an active state, is created, or is destroyed, TAPI is notified that the call count has changed on the address. This trace event is a record of that notification. The first number is the permanent device identifier of the line. The *Active/OnHold/OnHoldPend* fields represent the current counted **Active** calls (i.e. in the *Connected* state), the number of *OnHold* calls, and the number of *OnHold* calls which are pending transfer/conference finality. Calls which are *Idle* or *Conferenced* are not reflected in the call counts.

**Opening line #####, TAPI handle=#####, SP handle=#####,
Closing line #####, TAPI handle=#####, SP handle=#####,**

These messages are reported when a line is opened or closed using **TSPI_lineOpen/TSPI_lineClose**. It reports the TAPI device identifier, the **HTAPILINE** identifier which TAPI uses to represent the line object, and our service provider handle for the line (which is actually the *address* of the memory object representing the line).

**Send_TAPI_Event: <#####> Line=#####, Call=#####, Msg=##### (NAME),
P1=#####, P2=#####, P3=#####
Send_TAPI_Event: <#####> Phone=#####, Msg=##### (NAME),
P1=#####, P2=#####, P3=#####**

This message is reported when an event is reported to TAPI through the **LINEEVENT** or **PHONEEVENT** callback. The first number is the memory address of the line/phone object reporting the event. The *Line* parameter is the **HTAPILINE** or **HTAPIPHONE** parameter. The *Call* parameter is the **HTAPICALL** parameter. The *Msg* is the **LINE_xxx** or **PHONE_xxx** event which is being reported, and the *P1/P2/P3* parameters are the optional parameters passed with the given message. To determine the meaning of the parameters or the events, consult the *Microsoft TSPI Reference Guide*.

****MSG NOT SENT - TAPI VERSION <x.x****

This message is reported when an event was bubbled up by the service provider but the version of TAPI running on the machine cannot process the message. It is simply a notification that the event was *not* reported. It does not indicate an error of any kind.

lineMakeCall: invalid address id <#####>

This message is reported when the **TSPI_lineMakeCall** function is passed a bad address identifier in the **LINECALLPARAMS** structure.

lineMakeCall: address explicitly specified does not exist

This message is reported when the **TSPI_lineMakeCall** function is passed an unknown address (DN) within the **LINECALLPARAMS** structure.

lineMakeCall: no address available for outgoing call!

This message is reported when the **TSPI_lineMakeCall** function is called and there is no available address to dial out on (i.e. all addresses are either incoming only or have all the bandwidth used up).

LINEDEVCAPS.dwLineFeatures missing ##### bit

This is reported when the line features are changed through the **CTSPILineConnection::SetLineFeatures** call but the new feature bits are not all part of the capabilities of the line setup by the provider. It is a warning that the service provider developer has incorrectly set the capabilities of the line object. The bit(s) listed are the missing bits from the reported features mask.

OnConnectedCallCountChange: Delta=##, New Count=##

This message is reported when the count of calls on a line changes. This is a reflection of all call appearances on the line which are taking up bandwidth and would disrupt functions such as out-dialing. The *Delta* field represents the number of new calls added/removed from the line, and the *count* is the new total count on the line.

***Opening phone ####, TAPI handle=####, SP handle=####
Closing phone ####, TAPI handle=####, SP handle=####***

These messages are reported when a phone is opened or closed using **TSPI_phoneOpen/TSPI_phoeClose**. It reports the TAPI device identifier, the **HTAPIPHONE** identifier which TAPI uses to represent the phone object, and our service provider handle for the phone (which is actually the *address* of the memory object representing the phone).

Thread timed-out, Event=##***Thread waiting for request <##### NAME> to complete, <###>***

These messages indicate that a time-out occurred in **WaitForRequest**. The first number is the memory address of the **CTSPIRequest** object which did not complete within the timeout, and the second number is the handle of the event object that the thread was waiting on.

Thread finished wait, rc=##

This message indicates that a thread was released from the **WaitForRequest** method. The *rc* is the final return code returned by the function.

Multiple addresses listed in dialable address, ignoring all but first

This message indicates that the **CheckDialableNumber** function found multiple addresses within a TAPI dialable number and that the line object owner did not have the **LINEDEVCAPFLAGS_MULTIPLEADDR** bit set indicating that it supports multiple addresses. Only the first address is returned to the caller.

Looking for connection info, Type=0/1, Id=#####

This is reported when a search for a line/phone device occurs by device identifier. The *Type* field indicates whether the search is for a line (0) or phone (1).

Line negotiation failure, device Id out of range
Phone ext negotiation failure, device Id out of range

This is reported when a **TSPI_lineNegotiateAPIVersion** or **TSPI_phoneNegotiateAPIVersion** fails due to the line/phone device id being outside the range given to the provider in **TSPI_providerInit**.

Opening device #####
Closing device #####

These are reported when the **CTSPIDevice::OpenDevice** and **CTSPIDevice::CloseDevice** methods are called by the line/phone objects. The number indicates the TAPI device identifier which requested the open.

Problem starting companion application <NAME>, rc=#####

(16-bit only) This message indicates that the companion application did not start. The executable name is given along with the return code received by **WinExec**.

****ERROR* Unable to allocate memory for Send!***

(16-bit only) This message indicates that the function **GlobalAlloc** failed to allocate a block of memory to send to the companion application.

Forcing context switch to companion application, line device=#####

(16-bit only) This message indicates that the TSP is forcing a context switch from its current thread (the calling TAPI application) to a safe thread owned by the companion application.

DeviceNotify: wCommand=####, ConnID=#####, dwData=####, lpBuff=####, Size=##

(16 bit only) This message is reported when the companion application calls back into the service provider to notify it about an event or due to a context switch/timer. The *wCommand* entry is the **RESPONSE_XXX** event from the *spuser.h* file. The *ConnID* is the line/phone connection identifier which is the target of the event (if known). The *dwData/lpBuff/dwSize* parameters are the data buffers being passed which represent the event itself.

Registering HTASK ####, HWND #### as thread task.

(16 bit only) This indicates that the companion application successfully connected to the TSP library.

MEMORY ALLOCATION FAILURE: Bytes=####

(16 bit only) This indicates that **GlobalAlloc** failed to allocate a buffer.

TAPI ERROR: #####

(16 bit only) This message is reported when one of the **CServiceProvider** methods is returning a TAPI error in response to a **TSPI_XXX** entry-point. It is not used in all cases and should be ignored. Instead, rely on the final result code from the **TSPI_XXXX** traces.

References

---- *Win32 SDK: TAPI*, Microsoft Press, 1995

---- *Chicago Implementation of Windows Telephony*, Microsoft Systems Division, 1994

---- Win32 Implementation of Windows Telephony (TAPI Version 2.0)

---- *Specs: Telephony SPI*, Microsoft Systems Division

---- *Specs: Telephony API*, Microsoft Systems Division

Toby Nixon, *Developing Applications using the Windows Telephony API*, Tech*Ed Microsoft at work, 1994