TSP++ Version 3.0

TSP++ User's Guide

Building 3rd Party TAPI Service Providers for Windows NT 4.0. Windows 2000 and Windows XP

TSP+++ Version 3.046

Table of Contents

Introduction

TAPI is an acronym that stands for **T**elephony **A**pplications **P**rogramming **I**nterface.

> **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 **W**indows **O**pen **S**ervices **A**rchitecture (**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.

Last year Microsoft 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 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 announced that it will introduce TAPI 3.0 with Windows 2000 (previously called Windows NT 5.0). TAPI 3.0 will add media streaming and control functions into the TAPI model and introduce a new COM model for application development.

Since TSP++ Version 3.0 is designed for server based telephony under TAPI 2.1 and 3.0, this manual will not discuss 16-bit TAPI unless required for historical purposes.

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 **TAPISRV.EXE** to provide telephony support for the application.

TAPI32.DLL – This is the library that 32-bit applications can link to in order to call telephony functions. It communicates with **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).

TAPISRV.EXE – This is the 32-bit service manager under Windows NT that provides the support for Windows 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.

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.

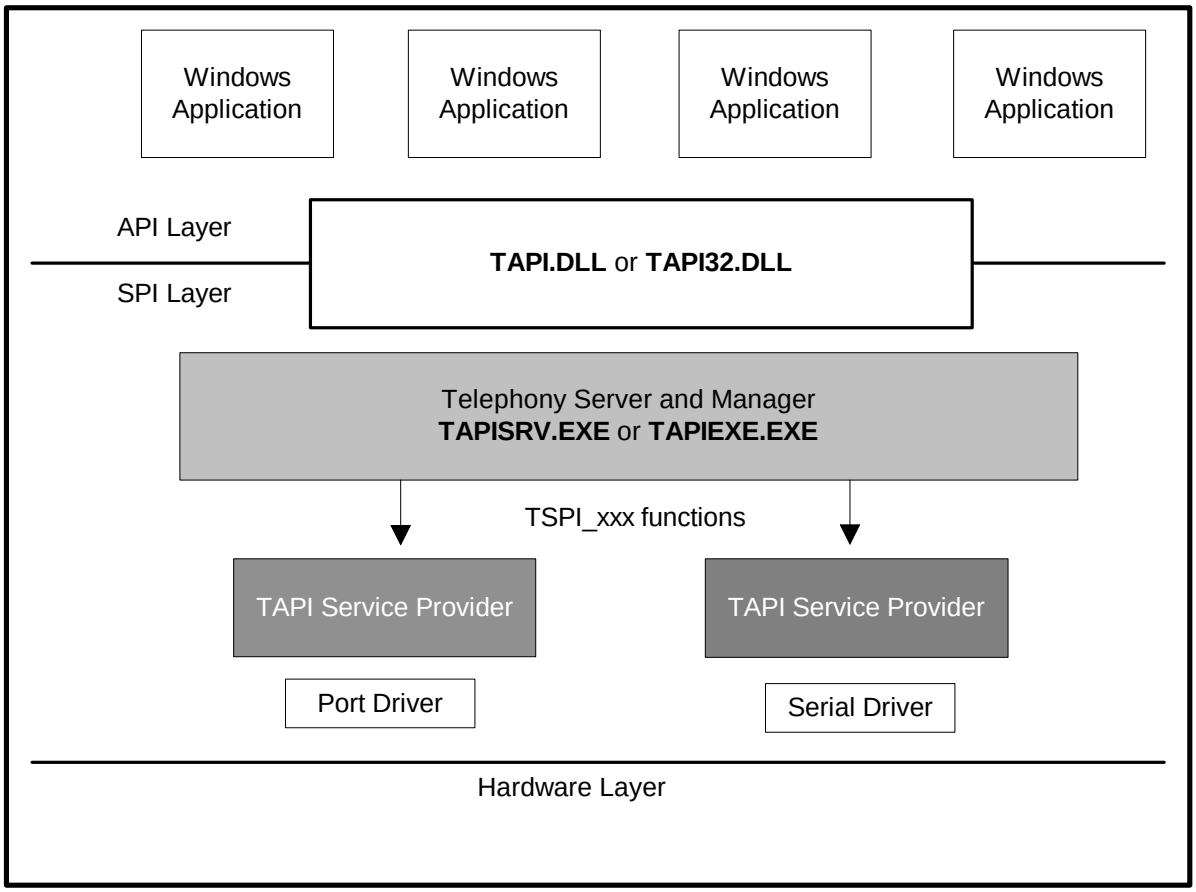
Each TSP (TAPI Service Provider) is a dynamic link library (DLL) that carries out low-level device-specific actions needed to complete the telephony tasks. The TSP is always called by one of the TAPI system components, never by an application directly, and is generally connected to some piece of telephone 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 **TAPI32.DLL**.



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, through a system RPC. 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 **TAPI32.DLL** (or **TAPI.DLL**) to determine which services are available on the given computer. The TAPI server 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.

These TAPI requests are forwarded from **TAPI32.DLL** to the TAPI server/manager (**TAPISRV.EXE**). This manager runs as a Windows NT 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.

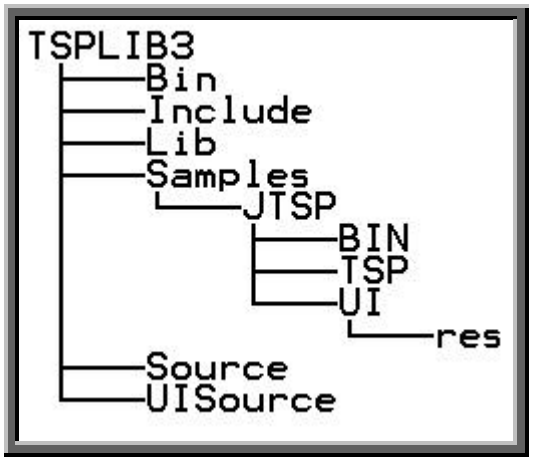
When a request is made from an application, it is sent to **TAPISRV.EXE** through an RPC facility by **TAPI32.DLL**. The request is verified and then is passed through to the appropriate entry-point in some TSP to manage the request. **TAPISRV.EXE** will then reply to the RPC, releasing the original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

Using TSP++ Version 3.0, you can quickly build robust client/server TAPI service providers that can handle anything from a large office PBX or ACD system to a small key telephone system.

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.

Binary files (BIN)

Filename	Description
JTAProxy.exe	ACD Proxy application – see the section on *Agent Support*.
JTTspTrc.dll	TSP Tracing support – see the section on debugging notes
TSPWizard.awx	The TAPI Service Provider AppWizard – see the section on *Creating a new Service Provider*. [1]
TSPWizard.hlp	TSP AppWizard help file.[1]

[1] This file may have been placed into the Visual Studio 6.0 Template directory with all other custom AppWizard files. If this directory was located during installation, then the file will not be in the **BIN** directory as indicated above.

Include files (INCLUDE)

Filename	Description
poolmgr.h	Contains the Thread Pool Manager template class
regstream.h	Contains the registry **TStream** derived class used to share information between the SPLUI library and TSP++.
spacd.h	Internal ACD definitions shared with **JTAPROXY** component.
spagent.h	Agent and ACD structures and objects
spbase.h	Basic structures and objects for all of TSP++
spbstrm.h	Base **TStream** class for persistent data storage
splib.h	Main include file for TSP++
splibui.h	User interface support for TSP++
spreq.h	TAPI request object definitions

Source Files (SRC)

Filename	Description
address.cpp	Implements the address level class (CTSPIAddressInfo).
call.cpp	Implements the call level class (CTSPICallAppearance).
callhub.cpp	Implements the call hub class (CTSPICallHub).
confcall.cpp	Implements the derived conference call class (CTSPIConferenceCall).
conn.cpp	Implements the basic connection object (CTSPIConnection).
critsec.cpp	Implements the internal critical section object (CintCriticalSection)
debug.cpp	Debugging interfaces used in TSP++
device.cpp	Implements the device level class (CTSPIDevice).
display.cpp	Implements the telephone display used internally (CPhoneDisplay).

dllmain.cpp	Has the DLLMain function for the TSP.
lineconn.cpp	Implements the line level class (CTSPILineConnection).
misc.cpp	Miscellaneous class methods.
phoneconn.cpp	Implements the telephone level class (CTSPIPhoneConnection).
poolmgr.cpp	Implements the static portion of the pool manager template.
request.cpp	Implements the asynchronous request object (CTSPIRequest).
serialize.cpp	Implements the serialization support to read configuration information from the registry.
sp.cpp	Basic service provider class with all general methods (CServiceProvider)
spdll.cpp	Indirection layer used for debugging service providers.
tsplayer.cpp	Main TSP_xxx entrypoint handlers and validation layer.

Pre-built Libraries (LIB)

The library files were built using Visual C++ 6.0. If you receive linker errors, 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
SPLIB32.lib	Retail build (single-byte)
SPLIB32D.lib	DEBUG build (single-byte)
SPLIB32U.lib	Retail build (Unicode)
SPLIB32UD.lib	DEBUG build (Unicode)
SPLIBUI.lib	Retail build of the user interface library (single-byte)
SPLIBUID.lib	DEBUG build of the user interface library (single byte)
SPLIBUIU.lib	Retail build of the user interface library (Unicode)
SPLIBUIDU.lib	DEBUG build of the user interface library (Unicode)

Sample Source code (SAMPLES)

This directory contains all the sample code for the library. One main sample is provided to show how to implement a third party service provider.

JTSP

This directory contains all the source code and pre-built binaries for the JulMar 3rd Party TAPI Service Provider. It supports the majority of the TAPI 2.1 specification and runs under Windows NT 4.0 or 5.0. For more information on this part of the SDK, consult the related documents concerning the TSP sample and JPBX simulator that it communicates with.

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.

Base Object Class

All of the main telephony objects are derived from a basic **CTSPIBaseObject** class, which provides support for thread synchronization and a 4-byte user-data area that may be used by developers. Most of the objects in the library may be overriden, which 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*.

STL usage

The TSP++ library uses the *Standard Template Libraries* for managing object containers and string storage. Most of the library wraps STL internally so that the derived provider doesn't need any specific knowledge to manipulate the data structures. The one area that is an exception is in string management.

String Management

TSP++ uses the **std::string** class to manage strings (note that when building a Unicode provider, this is replaced with **std::wstring**). A typedef is included in the main *splib.h* header file which casts this to the value **TString**. This **TString** object is passed to and from from many of the TSP++ methods.

Normally this wouldn't be an issue except that **std::string** does not provide an automatic cast for the **LPCTSTR** type. Instead to get access to the string buffer, you must call the **std::string.c_str** method.

Debug Stream

The debug facility uses the standard iostream library to build output intended for the debug window. Specifically, the **std::stringstream** (or **std::wstringstream** when built for Unicode) class is used to build debug output. The library typedef for this is **TStringStream**. All of the **Dump** methods return **TString** objects containing the entire dump of the object. This has the advantage of keeping the output together (rather than sending it one line at a time where it might be split up by another thread, TAPISRV, or another provider doing output at that moment.

Array Management

Arrays of objects are managed using the **std::vector** container. Any data structure which requires non-keyed random access (a [] operator) uses this container to store its data.

TSP++ has an extension called **advector** defined in *spbase.h* which performs *auto-deleting* on the contents of the array when it is deleted. In addition, when an array is copied through the copy constructor, the contained objects are duplicated with their copy constructor. This container class requires that the elements being contained were allocated on the heap using the C++ **new** operator.

Some defined array types include:

Typedef	Contents
TUIntArray	An array of unsigned integers
TDWordArray	An array of DWORD values

The TSP++ objects maintained in vector or advector arrays include:

Object Type	Typedef array
CTSPIDevice	TDeviceArray
CTSPILineConnection	TLineArray
CTSPIAddressInfo	TAddressArray
CTSPIPhoneConnection	TPhoneArray
DIALINFO	TDialStringArray
TSPIFORWARDINFO	TForwardInfoArray
TIMEREVENT	TTimerEventArray

TERMINALINFO	TTerminalInfoArray
USERUSERINFO	TUserInfoArray
DEVICECLASSINFO	TDeviceClassArray
LINEGENERATETONE	TGenerateToneArray
TSPITONEMONITOR	TMonitorToneArray

List Management

Linked-lists of objects are managed using the **std::list** container. Any data structure which requires non-keyed sequential access or tends to come and go quickly uses this container to store its data.

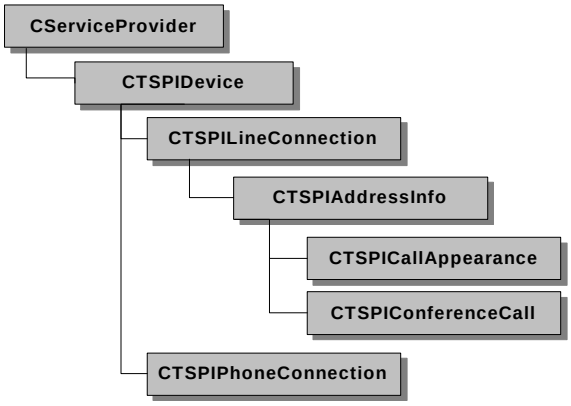
TSP++ has an extension called **adlist** defined in *spbase.h* which performs *auto-deleting* on the contents of the array when it is deleted. In addition, when an list is copied through the copy constructor, the contained objects are duplicated with their copy constructor. This container class requires that the elements being contained were allocated on the heap using the C++ **new** operator.

The TSP++ objects maintained in vector or advector arrays include:

Object Type	Typedef array
CTSPICallAppearance	TCallList
CTSPICallHub	TCallHubList
CTSPIRequest	TRequestList

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, most of the objects are not related to each other in C++ terms.



The class hierarchy starts with the **CServiceProvider** class. This object contains globl methds and tables which any device in the TSP might make use of. For example, it contains the debug log event mechanism which developers may override to create a physical file log of TSP events. This object is the first object created and is required to be a global variable in the TSP (which means there cannot be more than one of these objects).

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 if the hardware has some form of plug & play or event notification for new lines or stations.

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 (device, line, address, telephone, 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.

In order to allow for the most flexibility, the TSP++ library has a built in facility for 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. By storing the runtime information for the each of the telephony object types, the library can create derivative objects of which it is not completely aware. Using the **CServiceProvider::SetRuntimeObjects** method and the **DECLARE_TSPI_OVERRIDE** macro, all the basic objects used internally in the library may be overridden and enhanced.

> Tip: You do not need to use this mechanism to override the **CServiceProvider** object since you *must* create one of these. Simply define a new object derived from **CServiceProvider** and then instantiate a global of that object in any of your source files.

Method for overriding an object

If you decide to override a library object (generally required for lines and phones), first use the **DECLARE_TSPI_OVERRIDE** macro to define the RTTI structures necessary for dynamic creation. This macro creates a *factory* for your derived object, which may then be used to create an object of that type. This definition should be performed in a CPP source file using global scope (i.e. not defined in any function).

Then, in the constructor for your service provider object, call the **SetRuntimeObjects** method and give it the ASCII name <u>*not*</u> Unicode of each of the objects you are overriding.

```
// Tell the library about our derived objects. CMyDevice should be defined
// in a header file included by this and should be derived from CTSPIDevice.
// Note: this must be global!
DECLARE_TSPI_OVERRIDE(CMyDevice);

// Constructor for CMyProvider.. derived from CServiceProvider
CMyProvider::CMyProvider() :
        CServiceProvider(_T("MYUI.DLL"), _T("Sample"), TAPIVER_30)
{
        // Setup our line device and phone device object override.
        SetRuntimeObjects (“CMyDevice”, NULL, NULL, NULL, NULL, NULL);
}
```

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 server. 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 server will send **HTAPI**xxx handles where appropriate and expect the service provider to return **HDRV**xxx 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 server never passes the handles outside of itself and the service provider, so the need for verification of the handle on return is minimal. The library performs automatic verification on the handle's memory address before allowing the handle to be used by the derived provider code.

TAPI Notifications

When any event occurs within 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 that 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
LINEADDRESSTATE_DEVSPECIFIC	Not generated by the TSP library
LINEADDRESSSTATE_INUSEZER	OnCallStateChange
LINEADDRESSTATE_INUSEONE	OnCallStateChange
LINEADDRESSTATE_INUSEMANY	OnCallStateChange
LINE_ADDRESSSTATE_NUMCALLS	CreateConference CreateCallAppearance RemoveCallAppearance OnCallStateChange
LINEADDRESSTATE_FORWARD	SetNumRingsNoAnswer OnRequestComplete
LINEADDRESSTATE_TERMINALS	SetTerminalModes OnTerminalCountChanged
LINEADDRESSTATE_CAPSCHANGE	OnAddressCapabiltiesChanged

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 TAPI applications are finished with it or not. This is used when the line goes out of service and the provider cannot control it any longer.

The **CTSPILineConnection::ForceClose** method may be used to send this event.

LINE_DEVSPECIFIC

This notification is used in supporting device-specific extensions that are not part of the formal TAPI specification. 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 that 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 send 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	DevStatusConnected SetDeviceStatusFlags
LINEDEVSTATE_DISCONNECTED	DevStatusConnected SetDeviceStatusFlags
LINEDEVSTATE_MSGWAITON	DevMsgWaiting SetDeviceStatusFlags
LINEDEVSTATE_MSGWAITOFF	DevMsgWaiting SetDeviceStatusFlags
LINEDEVSTATE_INSERVICE	DevStatusInService SetDeviceStatusFlags
LINEDEVSTATE_OUTOFSERVICE	DevStatusInService 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	DevStatusLocked SetDeviceStatusFlags
LINEDEVSTATE_CAPSCHANGE	OnLineCapabiltiesChanged
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**.

User-Interface Events

As discussed in the previous sections, 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, TAPI defines a seperate telephony service provider UI DLL interface. The TAPI32.DLL module to 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*.

A secondary library *SPLUI* is used to generate the UI DLL for a service provider project. This library contains support functions and overrides specific to the user-interface process. Unlike the 1st party implementation of TSP++, the UI DLL is maintained in a separate DLL package rather than within the main TSP code body. This keeps the TSP from being loaded into multiple processes and allows the UI DLL to be redistributed to client machines if Microsoft ever allows the UI dialogs to be redirected to client machines (spontaneous dialogs). The *SPLUI* library uses MFC for it's object management, so MFC may be used to generate the UI as well. For more information on this topic, refer to the document on the *TSP++ User Interface Library*.

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
DialableNumber	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. The plus sign (+) is a valid character in dialable strings. It indicates that the phone number is a fully qualified international number.
Subaddress	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.
Name	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 library is done with the Microsoft CRT memory functions and is tracked for memory leaks by the C runtime memory allocator package. If the TSP is built with debugging information turned on, a dump of orphan memory will accompany the shutdown of the service provider. There should not be any objects listed in the output. The TSP++ library makes sure to clean up even the global objects before the dump occurs.

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. It is recommended that if you are creating objects or structures, you use the C++ **new** operator so that constructors get called appropriately.

Command	Description
AllocMem	Allocate a generic non-object block of memory
FreeMem	Free a block of memory allocated with AllocMem
CopyVarString	Copies a buffer into a VARSTRING pointer.
DwordAlignPtr	DWORD aligns an address (32-bit only).
AddDataBlock	Adds a new VARSTRING block to a buffer.

Agent Support

TAPI 2.0 introduced full support for agent capabilities through a *proxy application* that fulfills agent requests for a TSP. The interface was designed so that a call center could implement their own agent proxy for a TSP, which they got from a manufacturer. On most switches however, the agent process management happens on the same link as call control – this implies that the agent support should be directly in the TSP since it controls the data link.

Agent Proxy Service application

TSP++ Version 3.0 adds support for a pre-built binary agent proxy application, called **JTAPROXY**, that is tied to the service provider and passes data to/from the provider using an internal interface to TSP++. This allows for the proxy to pass requests to the service provider, which may need to gather information from the ACD or PBX before continuing with the request. The proxy application is designed to work with the TSP library to process agent requests in a high-performance fashion and forwards each request onto the TSP to process.

This shipped proxy application may be shipped with any TSP built with TSP++ royalty free to support agent requests. It must run on the same NT server as the TSP as the mechanism used to pass data to and from the service provider is only supported as a local IPC. ***It will not run properly under Windows 95 or Windows 98***.

TSP++ does not require that you use the pre-built proxy application. The agent features may be invoked by any mechanism desired and may even be seperated from the TSP using the standard TAPI architecture (although if this is done, no support is given from TSP++ for agent management). This pre-built proxy application provided simply to speed up the majority of cases that require agent support.

JTAPROXY is an NT service application – it runs in the context of the NT service manager, just as the TAPI server does. It is a separate application and therefore has it's own 32-bit process space seperated from the TAPI server. To communicate with the TSP driver, JTAPROXY allocates two blocks of *shared memory* which it uses to send and receive requests from the TSP, as if they were sharing the same address space. This is the fastest IPC mechanism available in Windows NT, and is the mechanism used to implement all the other IPCs when used within the same computer.

The proxy application has a command line interface, which may be used to interact with it. The following flags are supported:

AUTOSTART	This causes the proxy server to be started with the system (Automatic vs. Manual startup). It must be accompanied by the **REGISTER** keyword
USER=	This allows you to specify the user-id to impersonate when making requests. This should be the same user-id used with TAPISRV.EXE.

PASSWORD=	This is the password for the above user-id.
QUIET	This turns off the notification message boxes for successful/unsuccessful installations.
CONSOLE	This causes the program to start as a console application and output its information to stdout.
REGISTER	This is used to install the agent proxy as an NT service on the machine. The **USERID** and **PASSWORD** keywords should accompany it to associate a user account on the domain.
UNREGISTER	This de-installs the proxy server from the NT system. It removes the program from the service managers control table.

Testing Agent Support

To test agent support, run **JTAPROXY** as a console application. This is done by invoking the proxy application from a command line prompt with the following syntax:

```
C:\> JTAPROXY –CONSOLE
```

This will start the proxy, which will in turn start the TAPI server (as it makes calls into TAPI). The service provider should load (if it isn't loaded already) and the proxy will connect to the service provider and open any lines which have executed the **EnableAgentSupport** method.

Once this is completed, agent support is enabled for all the lines which have requested it.

Using Agent Support as a service

Once the agent support in the TSP has been tested, use the agent proxy as a service to perform the final testing. Install **JTAPROXY** on the NT server using the following syntax:

```
C:\> JTAPROXY –AUTOMATIC –REGISTER USER=Administrator PASSWORD=adminpassword
```

Note that case and order of the parameters is unimportant.

It is recommended that you install the agent proxy under an account that has Administrator privilages on the domain This is also recommended for **TAPISRV**. The proxy application does not need to interact with the desktop.

The primary reason for using a higher privilege level is to ensure that security level required for opening the memory blocks between **JTAPROXY** and **TAPISRV** is sufficient.

Stopping the agent proxy

To shut down the agent proxy, simply press CTRL+C in the command window where it is running, or if it is running as a service, use the NT service manager.

Installing the proxy agent as part of the TSP installation.

To install the proxy agent as part of a full TSP installation, have the install program copy the **JTAPROXY.EXE** to the WINNT\SYSTEM32 directory and execute the registration command line adding the **QUIET** keyword.

```
JTAPROXY –QUIET –AUTOMATIC –REGISTER USER=Administrator PASSWORD=adminpassword
```

This will cause the registration to happen silently.

Uninstalling the proxy agent

To uninstall the proxy agent, use the **UNREGISTER** command line option:

```
JTAPROXY –QUIET–UNREGISTER
```

This will remove all references of the agent proxy from the computer. Note that it will *not* stop running as a result of this. You must use the service manager API or control panel applet to stop the service, then it may be deleted from the system.

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 TAPI server 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. Some people don't like to use Unicode because it causes the executable image to grow in size considerably if many strings are included. To accomidate all, TSP++ 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 single byte, 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**

You may use either library to generate your TSP. Note that Windows 95 and 98 do not support Unicode internally and therefore you *must* use the non-Unicode library if you intend to ship your TSP on that platform.

If your target platform is NT only, then JulMar recommends using the Unicode library since it will provide the best performance on the Windows NT platform.

Using multiple threads

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

CEnterCode synchronization object

There is a special object defined in the TSP++ library called **CEnterCode**. It takes any of the TSP++ library objects as its first parameter and locks the object for updates. Each of the TSP++ objects have an internal synchronization object 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* use this object to synchronize data access if you internally modify an object because other threads may be accessing it simultaneously. This is especially important if the target server is a multi-processor system.

Calling any of the TSP object methods will automatically lock the object and release it before returning.

Detecting deadlocks

As with any multi-threaded software, a complicated series of inter-related objects sometimes has dependancies that are not obvious. In many cases, when an event is reported by an object, the object's internal synch object has been set to stop other threads from modifying the object. This is especially true of the call object, which changes constantly.

It is important to not call too many other parent objects during the processing of object events (such as **CTSPICallAppearance::OnCallStateChange**) since this may cause a deadlock scenerio.

If you run into a deadlock scenerio and believe it is caused by TSP++, notify JulMar Technology immediately (see the section on *Technical support*) and send a TSP trace log so that the problem may be verified and corrected.

Debugging Notes

Debugging service providers can sometimes be 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. In addition, 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 a particular location using soft breakpoints.

> **Number one rule:** always build the internal versions of the provider using **DEBUG** mode. The TSP++ library has a lot of extra debugging validations that are performed to insure that it is being used properly.

Since the 32-bit TSP drivers under Windows NT run in the context of **TAPISRV.EXE**, any output generated through the tracing facility cannot be seen unless a debugger is attached to the **TAPISRV** process.

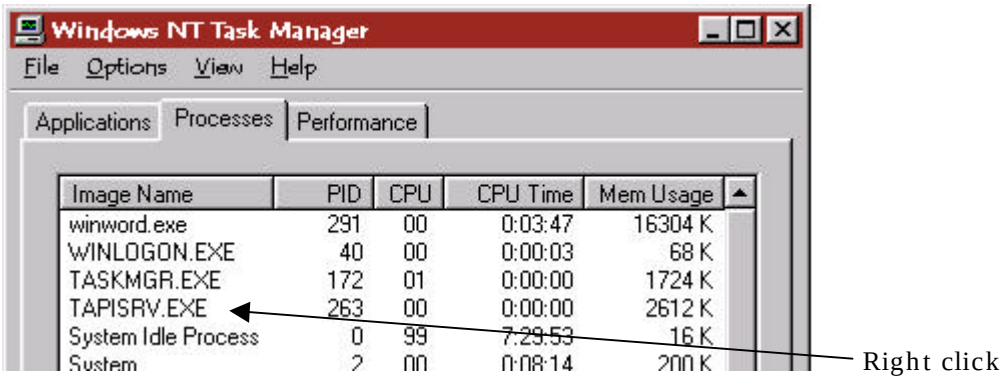
You can override the **CServiceProvider::TraceOut** method and dump the debug text to another medium, or use system-wide debug monitor (such as **DebugView** available for free from http://www.sysinternals.com/) in order to globally see debug events no matter which process they occur in.

Debugger recommendations

It is 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.

If you wish to debug with Microsoft's Visual Studio debugger or some other tool and you are using Windows NT, you can attach any process-debugger 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 for this to work, but will allow for watching the output through the debug window.

If you need to debug the startup code of the service provider, you can force an exception using the **DebugBreak** API (or embed an **int 3** into your code) and then handle the exception by debugging the process using the JIT facility in Visual Studio.

Both of these methods use the debugger specified in the Windows Registry under the key:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug\Debugger

If this method doesn't work, consult the Microsoft Knowledge base – especially articles **Q167798** and **Q138786**.

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".

Tracing events

The TSP++ Library has a tracing 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.

Lazy writer logger thread

If you would like to capture the output of the trace event system and write it to some other output destination (such as a file), then override the **CServiceProvider::TraceOut** method and call the global function **ActivateTraceLog** in the constructor of the service provider.

This will create a low priority *logger* thread that will capture the output from the TSP and pass it through the **TraceOut** method where it may be written to a file or anywhere

else. The output is automatically queued by the thread and passed through an event at a time. The string may be modified before output if desired.

If the tracer thread is not activated (by **ActivateTraceLog**) then the **TraceOut** method will not be called.

Output is always directed to the debug terminal window using the Win32 function **OutputDebugString**. This happens in real-time as the events occur so they are mixed correctly with existing trace output (from TAPI for example).

JTTSPTRC.DLL

Tracing of the TSPI entry and exit functions, parameters and structures is accomplished through a secondary DLL. This DLL (JTTSPTRC.DLL) is supplied with the TSP library and may be shipped with derived providers with no royalties.

To use the DLL, simply copy it into the WINNT\SYSTEM32 directory with the service provider and then set the appropriate debug level. The TSPI functions are automatically passed through the DLL and the trace output comes back through the service provider.

To turn off the DLL simply remove it from the directory or set the trace level to something below **TRC_API**.

This tracing dll works with any TSP++ generated TSP of any build type (Unicode, retail, debug, etc.)

Debug Strings used in the TSP++ Library

In addition to the entry/exit point tracing provided by the **JTTSPTRC.DLL**, 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.

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.

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.

AddrID=0x####,LineOwner=0x#### [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.

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.

CallID=0x####,LineOwner=0x####,htCall=0x####,State=STATE,RefCunt=#, htConsult=0x####,htConference=0x####

This is reported when the **CTSPICallAppearance::Dump** method is called. It reports the call id from the switch, the line owner, the TAPI call handle, the current call state, the reference count for this call, the consultation call handle, and the conference owner.

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.

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.

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.

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.

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.

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.

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.

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.

Device ####, Base Line=####, Count=####, Base Phone=####, Count=####

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.

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.

HubCallID=0x####,Count=0x####

This is reported when the **CTSPICallHub::Dump** method is called. It gives the switch call id, count of calls, and then dumps each call out.

htCall=0x####,LineOwner=0x####,State=STATE,RefCount=#,Parties=#

This is reported when the **CTSPIConferenceCall::Dump** method is called. It reports the line owner, the TAPI call handle, the current call state, the reference count for this call and then dumps all the calls which are part of this conference.

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.

LineID=0x####,DeviceID=0x####,htLine=0x####

This is reported by the **CTSPILineConnection::Dump** method and gives information related to the line. It is ok if the htLine value is zero, this simply means TAPI does not have the line open.

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.

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.

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).

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**.

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).

*****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.

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.

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).

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 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 that requested the open.

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).

PhoneID=0x####,DeviceID=0x####,htPhone=0x####

This is reported by the **CTSPIPhoneConnection::Dump** method and gives information related to the phone. It is ok if the htPhone value is zero, this simply means TAPI does not have the phone open.

ProviderID=0x#### [DeviceInfo] AgentProxy is xxx

This is reported when the **CTSPIDeviceInfo::Dump** method is used to dump the device to the debug stream. It reports the provider id, device name, and whether the agent proxy is connected to.

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.

Req=0x#### "Name"
DRV_REQUESTID=0x####,State="State",ResponseSent=Y/N
Call: call dump
Owner: line or phone dump

This is reported when the **CTSPIRequest::Dump** method is used to dump a request object. The *DRV_REQUESTID* is the TAPI request identifier for asynchronous requests. The *ResponseSent* field indicates if TAPI has been sent a result code for this request or not.

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*.

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.

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.

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).

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.

<########> 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.

Debug functions

The following functions may be used to control the debug facility within TSP++. All of the following functions are global so they may be called from within any object regardless of scope.

Function	Description
SetTraceLevel	Sets the current tracing level used by the TSP and by the JTTSPTRC module.
GetTraceLevel	Returns the current trace level being used.
ActivateTraceLog	This starts the lazy writer thread which can be used to store debug events into some other form of storage.

The following macros are defined in different system builds to help with the debugging process.

Macro	Defined in	Description
_TSP_TRACE	Retail/Debug	Outputs the text to the debug facility. The format is just like the CRT **printf** function.
_TSP_TRACEX	Retail/Debug	Outputs the text to the debug facility if the current tracing level is at a certain level or above. The format is just like the CRT **printf** function.
_TSP_DTRACE	Debug	Outputs the text to the debug facility. The format is just like the CRT **printf** function.

_TSP_DTRACEX	Debug	Outputs the text to the debug facility if the current tracing level is at a certain level or above. The format is just like the CRT **printf** function.
_TSP_ASSERT	Debug	Shows a MessageBox and trace event concerning a failed assertion. Gives the source and line number where the condition failed. The condition is not compiled in the retail build.
_TSP_ASSERTE	Debug	Shows the actual condition which failed along with filename and line number. The condition is not compiled in the retail build
_TSP_VERIFY	Debug	Similar to _TSP_ASSERT except the condition is compiled into the release build (although no runtime check is done).

The tracing macros take a variable argument list which is processed before the string is sent to the debug window. A second form of each macro takes a resource identifer to load from the resources in the TSP.

You must use the appropriate string format for all strings – if you have a Unicode build of the provider, all trace strings must be in Unicode.

> Warning: Do not use the string resource version of the tracing macros in the constructor of the **CServiceProvider** class or any other global constructor. **DLLMain** has not been called yet and the instance handle is not valid. If you do this, the string will fail to load.

TRACE Levels

The **_TSP_TRACEX** and **_TSP_DTRACEX** (see above) take a numeric value as the first parameter which indicates the desired trace log level. It can be one of the following values:

Trace Level	Description
TRC_NONE	No tracing performed
TRC_MIN	Minimum tracing – same as calling the non-X version of the trace macro
TRC_API	Trace all the TSPI apis entry and exit from the TSP. Requires that JTTSPTRC.DLL is in the same directory as the TSP.
TRC_STRUCT	Dump all the structures from the TSPI api entry and exit functions. Requires that the JTTSPTRC.DLL is in the same directory as the TSP.
TRC_DUMP	Dump all the strings and data buffers in the structures passed to and from the service provider.

TRC_FULL	All tracing events turned on
TRC_EXTRA	Dumps the call id map and other detailed information – slows down the TSP considerably. It is recommended that you do not use this option as it is mainly for debugging the TSP library itself.

The global functions **SetTraceLevel** and **GetTraceLevel** may be used to set and retrieve the current tracing level of the service provider. In addition, you can set the registry value **DebugLevel** in the main registry folder for the provider to set the current tracing level. The main registry folder is set to:

HKLM\SOFTWARE\Microsoft\Windows\Current Version\Telephony\Your Provider Name

Where the *"Your Provider Name"* is the value supplied to the **CServiceProvider** constructor.

Examples

_TSP_TRACE ("Hello");	This outputs the string "Hello" to the debug window in all builds
_TSP_TRACE ("Value=%d", iValue);	This outputs "Value=xx" with the 'xx' being replaced by the valie of the variable **iValue**.
_TSP_TRACEX (TRC_FULL, "Hello");	This outputs "Hello" if the trace level is at least TRC_FULL.
_TSP_TRACE (IDS_RES, iValue);	This outputs the string identified within the resource table as IDS_RES, replacing the first %% parameter with the contents of the variable **iValue**.

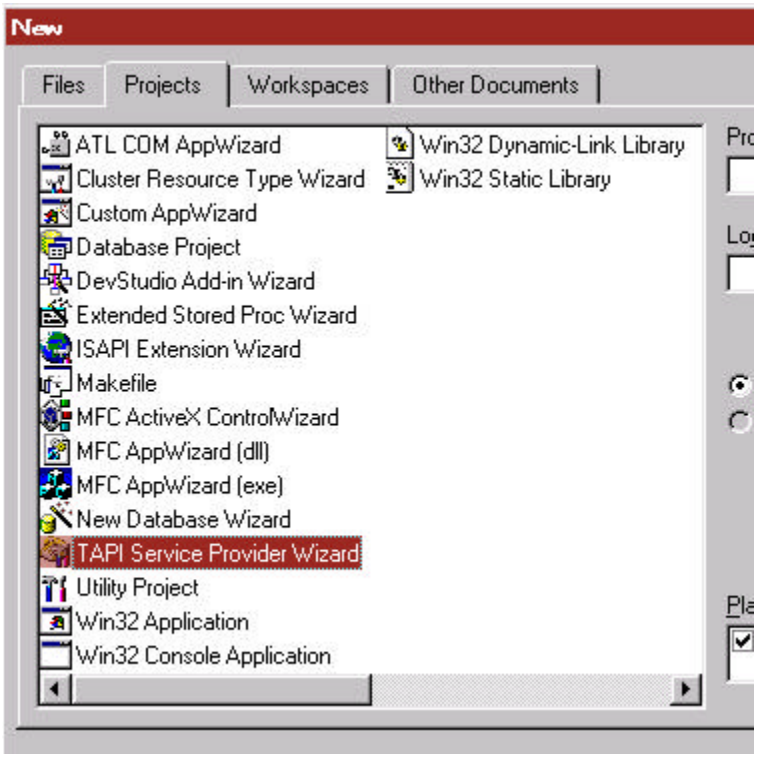
Note that in this case, the statements would generate code in all builds (release and debug). The macro could be replaced with _TSP_DTRACE if only debug builds should contain the statement.

Creating a new service provider

Using the TSP AppWizard

The easiest method of creating a new TAPI service provider is to use the provided TSP AppWizard. It is an optional component of install, but is selected by default.

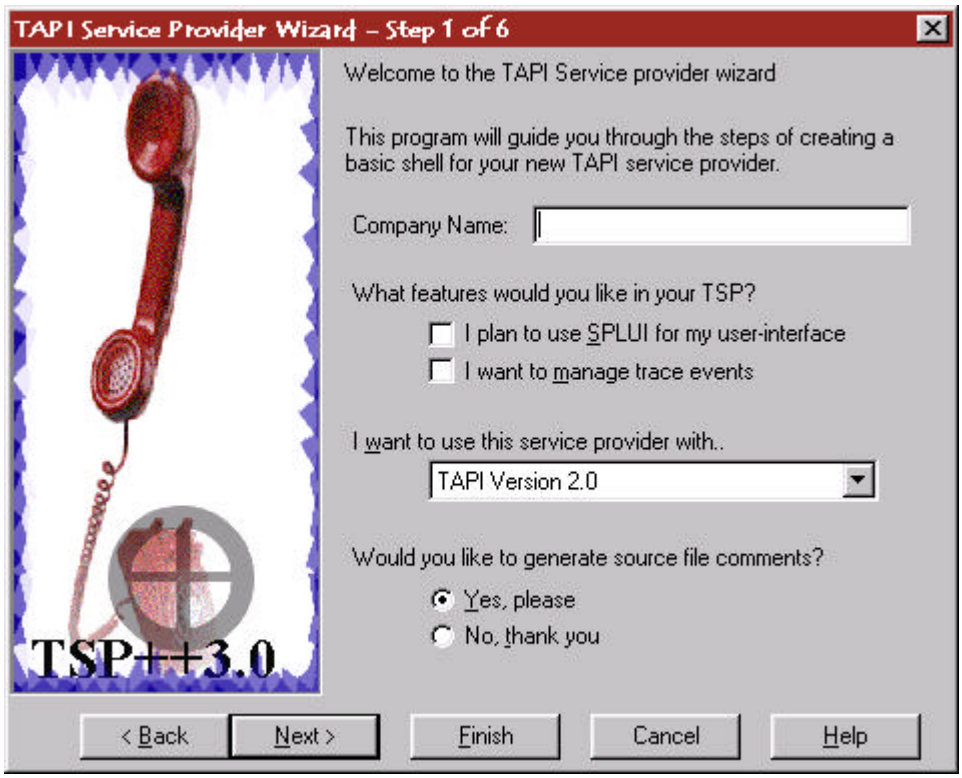
In order to use the AppWizard, open Microsoft Visual Studio 6.0 and click "File" and select "New" from the menu.



Select the **TAPI ServiceProvider Wizard** from the list. If that option is not available, then either the wizard was not installed, or the installation was unable to locate Visual Studio 6.0. Look in the **TSPLIB\BIN** directory (note that **TSPLIB** is wherever you selected) and check for the files **TSPWizard.awx** and **TSPWizard.hlp**. If the files are there, copy them to the **Common\MSDev98\Template** directory within your Visual Studio installation directory. Note that the **Template** directory will already exist.

Type in a new project name and a location for the project. Select the platform you are generating code for. Note that the wizard was designed for Win32 X86 platform but should work on other systems as well.

Click OK to begin the wizard.



Follow through the steps in the wizard answering each question as accurately as possible. If you have any trouble, or don't understand the question or prompt, click the **Help** button (available on each screen and dialog) for detailed information concerning the screens and questions.

The generated service provider is compilable and can be installed into the system immediately – it will load and inialize properly (although will obviously not be complete).

The rest of this section details methods that are overridden by the wizard and others that may be necessary depending on your implementation.

Typically Overridden methods

CServiceProvider

The TSP++ service provider implementation requires that a new class be 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.

CTSPIDevice

It is recommended that the device object is overriden and the derived object is used to store information related to the connection to/from the hardware. This is where any pool thread manager template would be instantiated. The derived device class will generally override:

Method	Description
Constructor	Zero out any new variables or structures
AllocStream	Called to allocate the **TStream** object to save and restore information for the service provider. This is overridden if you do not wish to store your configuration in the registry. If the standard registry storage method is sufficient, then this method does not need to be overridden.
Init	Initializes the device – connect to the device and create any event processing threads.
read	Called to serialize the line in from the registry. This is only called when the TSP++ SPUILIB library is used to store the objects there. This is called in the place of the below **Init** function when the objects are in the registry.
Init	Called to initialize the line – this should be overriden if the above **read** method is not.

CTSPILineConnection

In order to process events intended for line devices, the line object should be overriden in the derived provider implementation. A request map would be created using the **ON_TSPI_REQUEST** macros and worker functions would be defined in the derived class to handle each request. The following are the most common override points:

Method	Description
Constructor	Zero out any new variables or structures
OnAddressFeaturesChanged	Called when any address features change on this line. Allows you to adjust the reported features at the address level in line with what the device is capable of at that moment.
OnCallFeaturesChanged	Called when a call on this line is changing its call features – generally due to a call state change. Allows you to adjust the reported features at the call level in line with what the device is capable of at that moment.
OnLineFeaturesChanged	Called when the line is changing its line features. Allows you to adjust the reported features at the line level in line with what the device is capable of at that moment.

UnsolicitedEvent	Called when an event is not processed by a request handler (either because no request reported ownership through the return value or because there is no request running).
OnTimer	Called in response to the interval timer set through the **CTSPIDevice::SetIntervalTimer** function.
OpenDevice	Called when the first **lineOpen** request happens for this line.
read	Called to serialize the line in from the registry. This is only called when the TSP++ SPUILIB library is used to store the objects there. This is called in the place of the below **Init** function when the objects are in the registry.
Init	Called to initialize the line – this should be overriden if the above **read** method is not.

CTSPIPhoneConnection

In order to process events intended for phone devices, the phone object should be overriden in the derived provider implementation. A request map would be created using the **ON_TSPI_REQUEST** macros and worker functions would be defined in the derived class to handle each request. The following are the most common override points:

Method	Description
Constructor	Zero out any new variables or structures
UnsolicitedEvent	Called when an event is not processed by a request handler (either because no request reported ownership through the return value or because there is no request running).
OnTimer	Called in response to the interval timer set through the **CTSPIDevice::SetIntervalTimer** function.
OpenDevice	Called when the first **phoneOpen** request happens for this phone.
read	Called to serialize the phone in from the registry. This is only called when the TSP++ SPUILIB library is used to store the objects there. This is called in the place of the below **Init** function when the objects are in the registry.
Init	Called to initialize the phone – this should be overriden if the above **read** method is not.

CTSPICallAppearance

The call object can be overriden to provide additional data members or change the existing functionallity within the library (for example to change an incoming TAPI request before the request object is created).

> If this object is overriden, remember to always check to make sure it truly *is* a call object before you use the new methods or data. In many cases, a **CTSPIConferenceCall** can be passed from the library instead of a call object. Since they are two different objects, any derivation changes made to the call object are *not* carried over to the conference call.

General steps for creating a provider

Step 1: Create the CServiceProvider object

The first step in building a new TSP is to construct the service provider object. This should be a global object in one of the project source files.

If any of the basic objects needed overriding, the service provider constructor would be the place to issue a **SetRuntimeObjects** call.

Step 2: Override the device object

Create an object derived from the **CTSPIDevice**object and add all the device-connection and manipulation code to it. All the created line and phone objects will have access to this object so it is convenient to place device-specific management here. This would include event retrieval from the hardware and some piece of code to identify the line or phone that an event is intended for.

If the device has many devices which will generate a large amount of event traffic, it is recommended that you utilizie the thread pool manager template to generate a thread pool manager for worker threads.

Step 3: Override the line device object

The next override is to the **CTSPILineConnection** object. If the SPLUI library is not used to store the objects in the registry, then the **Init** method should be used to add all the addresses necessary for the line to operate. This is not necessary if SPLUI is used to store configuration information in the object format.

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 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.

Step 4: Create the request handler map

Using the **DECLARE_TSPI_REQUEST**, **BEGIN_TSPI_REQUEST**, **ON_TSPI_REQUEST**, and **END_TSPI_REQUEST** (see the section on *CTSPILineConnection* for more information on this topic) generate the request handler map. This maps TSPI requests to worker functions in the line object.

Persistent object information

Note: to use the persistent object streams you must include **spbstrm.h** in your source files – this is not done automatically by TSP++ unless your code was generated by the AppWizard.

In each provider, there is persistent information that needs to be loaded to make the provider operable. This might include information about how to connect to the hardware (a COM port, IP address, etc.), station information, devices controlled, etc.

TSP++ provides a mechanism to store and retrieve information using *binary streams*. These streams are similar in nature to the C++ iostreams implementation except that they work with binary information rather than text.

The user-interface library SPLUI contains the mechanism to write the information into the registry, and the TSP++ provider library contains the code necessary to restore the information. By default, each object created in the user-interface code is loaded into the provider when it is initialized by TAPI. Since the information stored and retrieved is generic TAPI information, the library also provides a mechanism for the developer to include device-specific information necessary for the implementation.

This is accomplished by overriding the **read** and **write** methods of each object which needs to store information.

The input parameter to these functions is a **TStream** object (see the *Reference Guide* for more detailed information on this object). The **TStream** object has insertion and extraction operators assigned for each basic C data type.

```
CmyLine::read(TStream& istm)
{
        // Read the basic information
        CTSPILineConnection::read(istm);
        // Read my specific information
        istm >> m_iMyInteger;
        istm >> m_strMyString;
}
```

Changing the stream destination

The TSP++ library also allows the developer to change the desintation for the configuration information stored. The default location is the registry, but if the configuration is too large to store in the registry, or there is information that must be encrypted or secured in some fashion, the stream object can be overridden.

To create a new stream object, override the basic **TStream** class and the **CTSPIDevice::AllocStream** and **CTSPUIDevice::AllocStream** functions. For example, consider an implementation which stores the data into a file:

```
#include <spbstrm.h>
class TFilestream : public TStream
{
// Class data
protected:
	TString m_strFile;
	FILE* m_fpFile;

// Constructor
public:
	// custom constructor
	TFilestream(LPCTSTR pszFilename) :
		m_fpFile(0), m_strFile(pszFilename) {
	}
	virtual bool open() {
		m_fpFile = _tfopen(m_strFile.c_str(), _T("r+b"));
		if (m_fpFile == NULL) m_fpFile = _tfopen(m_strFile.c_str(), _T("w+b"));
		if (m_fpFile != NULL) fseek(m_fpFile, 0, SEEK_SET);
		return (m_fpFile != NULL);
	}
	virtual void close() {
		if (m_fpFile) fclose(m_fpFile);
		m_fpFile = NULL;
	}
	virtual bool read(void* pBuff, unsigned int size) {
		if (m_fpFile == NULL || feof(m_fpFile)) return false;
		return (fread(pBuff, 1, size, m_fpFile) == size);
	}
	virtual bool write(const void* pBuff, unsigned int size) {
		if (m_fpFile == NULL) return false;
		return (fwrite(pBuff, 1, size, m_fpFile) == size);
	}
};
```

To use the newly created class, override the **AllocStream** methods in the UI library and the TSP++ library:

```
#include “filstream.h”
TStream* CMyDevice::AllocStream()
{
	return new TFilestream(_T("C:\\JPBX.CFG"));
}
```

Asynchronous request handing

When a request is generated by a calling application, TAPI will call the appropriate **TSPI_xxxx** handler in the TSP. The following actions are taken by the TSP++ library:

1. The exported function in *spdll.cpp* will receive the function and either pass it to the debug **JTTSPTRC.DLL** or pass it to the handler in *tsplayer.cpp*.

2. A handler function in *tsplayer.cpp* will get called to process the command (either from **JTTSPTRC.DLL** or *spdll.cpp*) and will:

   a) Validate the parameters passed in the function- all handles are checked to ensure that they are real objects and memory buffers are validated if the TAPI specification says they must exist.

   b) Copy passed buffers into local memory

   c) Convert strings from Unicode to Ansi if necessary

   d) Identify a target object based on the TSPI request. This object will always be the **CServiceProvider** object**,** a specific **CTSPILineConnection** object, a specific **CTSPIPhoneConnection** object, a specific **CTSPIConferenceCall** object, or a specific **CTSPICallAppearance** object).

   e) Call a virtual method exposed by the object to handle the request. This is always the TSPI name without the **TSPI_** prefix.

3. The target object will perform more validations specific to the request and it's own state. It will then do one of the following:

   a) Reject the request with an error.

   b) Process the request (if it is a **Get** request) and return a zero return code

   c) Generate a **CTSPIRequest** object representing the request, queue it to the line or phone object and return the passed **DRV_REQUESTID** to TAPI indicating the request has been started. If no request is pending on the line/phone device then the request is started immediately on the current TAPISRV worker thread.

An example of an asynchronous request

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 passes control to *tsplayer.cpp*

?? The *tsplayer.cpp* module validates the parameters, converts the dialstring to Ansi if this is a non-Unicode driver, determines which line device is being requested, locates the object and calls the virtual **MakeCall** method parameters for this request

?? The line object will receive the request and validate the request for this particular line:

1. It will check the **LINEDEVSTATUS.dwLineFeatures** to make sure the **LINEFEATURE_MAKECALL** bit is set. This is done to make sure the function is available at this moment.

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. It then validates the address specified in the **LINECALLPARAMS** (if any), 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.

6. A **RTMakecall** object is created and filled with all the information about the call to place.

7. A new call appearance is created on the selected/found address using the **CTSPIAddressInfo::CreateCallAppearance** method.

8. The **CTSPICallAppearance::MakeCall** method is called and passed the **RTMakeCall**object.

?? 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 based on the **RTMakeCall** object.

?? When the line connection adds the request packet, if there is no pending request in the list, the connection will call the **ReceiveData** method to initiate this request. If there is already a running request, 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, and finally back to TAPI to indicate a successful start.

?? When data is received by the device input thread (provided in the derived provider code), it will send the event to the appropriate line device for processing through the **ReceiveData** function. Or, if it uses the thread pool template, the input thread will queue the received event for a worker thread to pickup and call the line's **ReceiveData** function.

?? The **ReceiveData** function will (unless overriden) pull the first request from the request list using the **GetCurrentRequest** method (which also may be overriden). If a request exists, the function will scan the request handler map and call the worker function identified for the request.

?? The worker function will then process the event using the data in the request object. If the event is not relevant to the request, it would return a zero return code to **ReceiveData**.

?? If the request returns a zero return code, the **UnsolicitedEvent** method is invoked on the line to let it handle an *unhandled* event.

?? Once the call has been placed on the device -or- has failed, the **CompleteCurrentRequest** method should be called by the line owner. This will perform the callback notification to TAPI indicating success/failure, and the request will be freed. If there are any other pending requests in our queue, they will then be started and the whole process starts over.

Since TAPI can have several outstanding requests for a connection, one request may affect others later in the queue. To allow for this, the **CTSPIConnection** 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. In this case, the **WaitForRequest** may be called to wait (with optional time-out) for the request to finish. *Note that the thread is completely blocked during this time!* Special care should be taken to make sure that the execution thread that is servicing the hardware is not paused in this function.

For a list of request types and the appropriate TSPI functions to export for each operation, see the section on the *CTSPIRequest* object.

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.

If you have suggestions concerning the TSP++ product or documentation, we want to hear them! Send your notes to staff@julmar.com.

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)

---- *TAPI 3.0 Development Guide*, MSDN

---- *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