JPBX System Emulator TAPI Service Provider Configuration and Programmer's Guide

TSP++ Version 3.00
Last Modified: 11/11/98 5:39 PM

Table of Contents

Revision Log

REV	DESCRIPTION	DATE	AUTHOR
1.00	Initial Draft	08/01/1998	MCS

Approval Sign-off

JulMar: _____ **Date:** _____

References

- JulMar JPBX Simulator Sockets Interface Programmer's Guide
- JulMar JPBX Simulator Configuration and User's Guide

Introduction

This document provides an overview of the Sample TAPI Service Provider built for the JPBX ACD Simulator. It gives a description of the features of the TSP and in-depth design notes related to building the TSP with the TSP++ product. For information on the features of JPBX or its companion programs, see the *JulMar PBX Simulator Configuration and User's Guide*.

This will describes the goals and design of the project. It is intended to ensure that all features and performance criteria for the software package have been identified. The descriptions contained within this document will be used to drive the detailed software design and to enable development of the project within a reasonable time frame.

Project Overview and History

The **JulMar PBX Emulator** program was written to provide a stable test platform for a client/server 3rd party TAPI service provider sample that will be provided with the TSP++ version 3.0 product. The emulator is responsible for simulating different aspects of a PBX/ACD switching system, and connecting call information. The PBX/ACD software emulates various features of a large-scale PBX system with multiple line devices including:

1) **Stations** – Physical phone devices which agents may interact with.

2) **VRUs** – Voice response unit, similar to station but automated voice only.

3) **Queues** – ACD queue based on agent groups.

4) **Route Points** – Incoming virtual trunks

5) **Predictive Dialer**s – Outgoing trunks with "state" detection abilities

TAPI

Microsoft Corp. has a telephony standard for call control under the Windows environment called TAPI. This software standard is designed for first-party control, where the PC is directly connected to the telephone hardware, and for third-party control where the clients use a local area network (LAN) to send their requests to a single TAPI controller (the server). TAPI allows software running on the computer to take advantage of telephony capabilities without being tied directly to a specific implementation of hardware. Several software manufacturers have provided support within their applications for TAPI.

System Design Requirements

1. The TAPI Service Provider should implement the full range of capabilities that would be present on a 3rd-party server TSP written for a commercial PBX or ACD system. Areas that are primarily for first party implementations should be left out of this sample for clarity.

2. In order to provide the best sample possible, the mechanics of the switch should stay within the JPBX simulator and the TSP should use a normal communication method to control and interact with it. In this case, the preferred mechanism is through Windows Sockets.

3. The Service provider should detect and be able to handle switch disconnect. It should attempt to reconnect every 30 seconds if the socket goes down. Because this is a sample, it should not attempt to manage incomplete requests if events are missed. Instead it should ignore such events. The source code should indicate what a production level provider would do in these instances.

4. The service provider should work with TAPI 2.1 and TAPI 3.0

Target Platform

Operating System	Windows NT 4 (SP4), Windows NT 5
Minimum Memory Configuration	32-64MB RAM
Hard Disk Space	2 MB image size
Development Tools	Microsoft Visual C++ 5.0 or 6.0 JulMar TSP++ 3.0

Service Provider Functionality

The JPBX TSP will be a standard 32-bit Windows Dynamic Link Library with specific named exports to communicate with the Windows Telephony sub-system.

The development will be done completely in Microsoft Visual C++. The TSP++ v3.0 class library will be used as the basis for the development. The Microsoft Foundation Classes will be used to implement the user-interface dialogs and data validation layers.

The TSP will communicate with the JPBX Server using standard Winsock (WSA) commands and APIs. This will require that TCP/IP is installed and configured on the NT machine that the TSP is expected to run on.

The Service Provider will be developed using Microsoft Visual C++ 6.0. It will run under Windows NT with the TAPI 2.1 upgrade. It will be self-contained DLL that will spawn and manipulate multiple threads to communicate with the client TAPI applications and the JPBX telephony server. The Winsock management will be performed within this DLL. The TSP will be reentrant and designed for portability to non-Intel platforms.

The user-interface and configuration will be developed using Microsoft Visual C++ 6.0 with the Microsoft Foundation Classes. It will be a self-contained DLL that will communicate with the TSP portion using the TSP++ registry initialization stream.

> The TSP will be built for UNICODE operation so it will *not* be compatible with Windows 95 or Windows 98. If you want to test the TSP in these environments you will need to rebuild it using the non-UNICODE version of the library. The produced binary may then be run in these non-UNICODE environments.

DLL Interfaces

Line Interfaces

The following line-oriented interface functions will be exported from the TSP. These will provide the functionality supported by our JPBX switch interface. For more information on these TAPI entry-points, consult the *Microsoft TSPI Programmers Reference* available on MSDN.

Note: The TSP++ class library automatically implements many of these functions internally and no additional code will be required to add it to our TSP implementation.

TSPI_lineAccept	**TSPI_lineBlindTransfer**
TSPI_lineAddToConference	**TSPI_lineClose**
TSPI_lineAnswer	**TSPI_lineCloseCall**

TSPI_lineCompleteTransfer

TSPI_lineConditionalMediaDetection

TSPI_lineConfigDialog

TSPI_lineDial

TSPI_lineDrop

TSPI_lineGatherDigits

TSPI_lineGenerateDigits

TSPI_lineGetAddressCaps

TSPI_lineGetAddressID

TSPI_lineGetAddressStatus

TSPI_lineGetCallInfo

TSPI_lineGetCallStatus

TSPI_lineGetDevCaps

TSPI_lineGetDevConfig

TSPI_lineGetID

TSPI_lineGetLineDevStatus

TSPI_lineGetNumAddressIDs

TSPI_lineHold

TSPI_lineMakeCall

TSPI_lineMonitorDigits

TSPI_lineMonitorMedia

TSPI_lineNegotiateTSPIVersion

TSPI_lineOpen

TSPI_linePrepareAddToConference

TSPI_lineRedirect

TSPI_lineRemoveFromConference

TSPI_lineSetAppSpecific

TSPI_lineSetCallParams

TSPI_lineSetCurrentLocation

TSPI_lineSetDefaultMediaDetection

TSPI_lineSetDevConfig

TSPI_lineSetMediaMode

TSPI_lineSetStatusMessages

TSPI_lineSetTerminal

TSPI_lineSetupConference

TSPI_lineSetupTransfer

TSPI_lineSwapHold

TSPI_lineUnhold

Agent Management Interfaces

The JPBX simulator supports agents and groups of agents for defined agent stations. The JTSP interface will use the JTAPROXY.EXE agent proxy to send the agent requests to the TSP where they will be managed.

The following agent interface functions will be managed in the TSP. For more information on these functions, consult the *Microsoft TSPI Programmers Reference* available on MSDN.

lineSetAgentGroup

lineSetAgentState

lineSetAgentActivity

lineGetAgentStatus

lineGetAgentActivityList

lineGetAgentCaps

lineGetAgentGroupList

Phone Interfaces

The JPBX simulator supports a minimal phone interface for each defined agent station. Most of the phone is read-only (i.e. you can only query the state not set it).

The following phone-oriented interface functions will be exported from the TSP. These will provide the functionality supported by our JPBX switch interface. For more information on these TAPI entry-points, consult the *Microsoft TSPI Programmers Reference* available on MSDN.

Note: The TSP++ class library automatically implements many of these functions internally and no additional code will be required to add it to our TSP implementation.

TSPI_phoneClose

TSPI_phoneGetStatus

TSPI_phoneConfigDialog

TSPI_phoneGetVolume

TSPI_phoneGetButtonInfo

TSPI_phoneNegotiateTSPIVersion

TSPI_phoneGetDevCaps

TSPI_phoneOpen

TSPI_phoneGetDisplay

TSPI_phoneSetGain

TSPI_phoneGetGain

TSPI_phoneSetHookswitch

TSPI_phoneGetHookswitch

TSPI_phoneSetStatusMessages

TSPI_phoneGetID

TSPI_phoneSetVolume

TSPI_phoneGetLamp

Required TAPI Service Provider Interfaces

The following interfaces will be implemented in order to support the insertion and removal of the service provider in the TAPI sub-system and to recognize the loading and unloading of the service provider by TAPI.

Note: The TSP++ class library automatically implements many of these functions internally and no additional code will be required to add it to our TSP implementation.

TSPI_providerConfig

TSPI_providerInit

TSPI_providerEnumDevices

TSPI_providerInstall

TSPI_providerGenericDialogData

TSPI_providerRemove

TSPI_providerShutdown **TSPI_providerUIIdentify**

User-Interface exports

The following exports will be made from the user-interface module of the JPBX service provider. This is a separate DLL module that provides the dialog support for configuring and installing/removing the service provider. In many cases, the name reflects a similar entry-point found in the service provider.

TUISPI_lineConfigDialog **TUISPI_providerGenericDialogData**

TUISPI_lineConfigDialogEdit **TUISPI_providerInstall**

TUISPI_providerConfig **TUISPI_providerRemove**

For more information on each of these functions, please consult the *Microsoft TAPI Service Provider reference*.

Supported Telephony Services

The JPBX Telephony Server supports many features found on a commercial PBX/ACD system. The following features will be managed and supported by this implementation of the JPBX service provider. For more information on the implementation for each of these, see the associated design section later in this document.

Call Notifications

This is primarily the management of unsolicited events from the JPBX server. Calls will be monitored and offered up to TAPI when there is a monitor or owner that is watching for calls. If there is no owner, the TSP will track the call itself and if the line is opened (through the **lineOpen** API) while the call is still in progress, it will be delivered to TAPI at that time.

This service provider will support the following call notifications:

LINE_CALLSTATE – The changing of a call's state on the line device
LINE_NEWCALL – The offering of a newly detected call created by the ACD itself.

Make Call and Dial

These two commands allow for the creation of a new call, and connection to a second destination party. The destination party can be a station on the ACD or an external party through a trunk device.

Accept and Answer a call

This command will allow the answering of an offering call appearance. When a call first appears on a station device, it is simply shown on the display. If the station accepts the call, then the call begins to have an alert tone associated with it (i.e. active ringing). Note that the call may be answered regardless of whether it was accepted at the station or not.

Drop a call

This command will allow an active call appearance to be dropped from the system. This includes both outgoing and incoming calls.

Hold support

This includes hold, release and the swapping of a held call with an active call. These functions allow the client to change the active call without dropping existing calls.

Transfer support

This includes both consultation and blind transfers.

Conference support

The JPBX simulator allows for up to three calls on a single station. Only one call may be active at any given time unless a conference is created. In this case, two or more calls at a single station are fused into a single call (although both maintain a separate call object within the TSP).

Note for TAPI 3.0: The latest draft of TAPI 3.0 indicates that call hubs (the new conference object in the COM model) will only be supported for calls with the same call-id. It is not yet certain how this will affect conferencing.

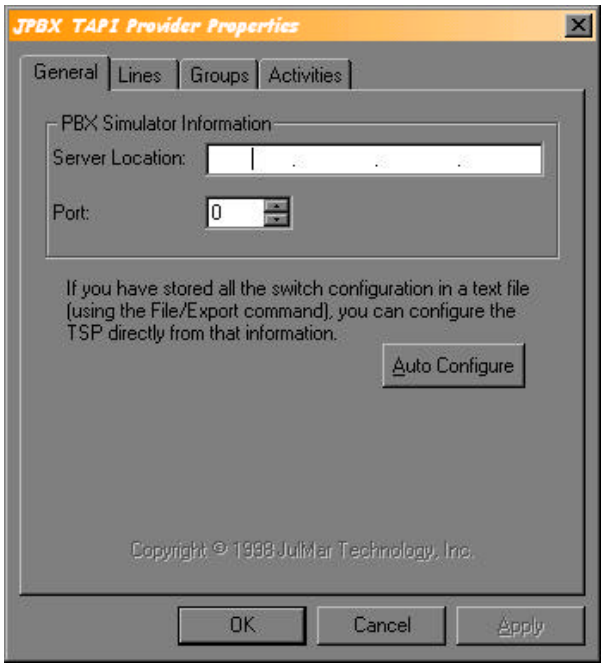
Phone support

When a change is detected in the associated phone station, the appropriate event will be forwarded to TAPI. This includes the display, lamps and detected digits.

User-Interface Dialogs

TSPI_providerConfig

The primary configuration interface will be invoked through **TSPI_providerConfig**. When it is invoked, it will pull up an interface that resembles:



If you do not see this tab as the first tab then you probably don't have the latest **COMCTL32.DLL** installed on your system. This tab uses the new IPADDRCONTROL which comes with IE4. Search on Microsoft's web site for information on obtaining a newer version of this control or install IE4.

The *Server Location* edit control should be filled in with the TCP/IP address of the machine running the PBX simulator.

The *port* edit control should be filled in with the numeric port number (defaulted to **4020**) that is configured in the PBX simulator software.

The *Auto Configure* button allows you to configure the TSP using the text file generated by the PBX Simulator's **File/Export** menu command. This allows easier entry of a large number of lines and groups. Note that the agent activities are not known in the

PBX simulator – they are tracked only by the TSP and therefore must be entered into this configuration using the last tab.

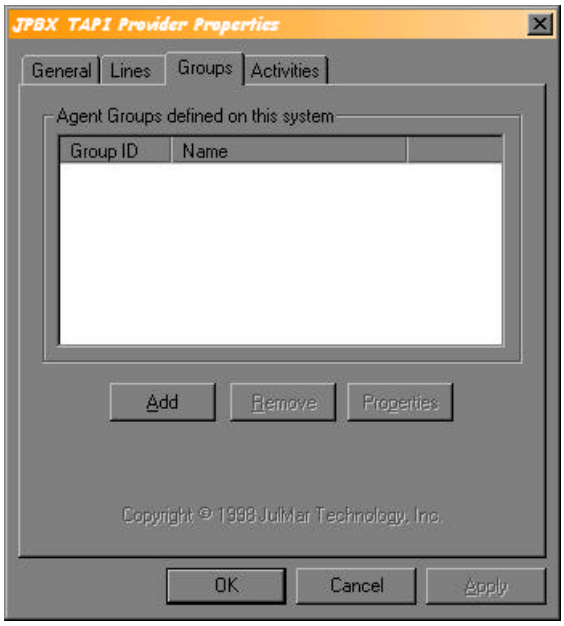
Lines

When you click the **Lines** tab, you are shown a screen for entering and editing line information.

To add a new line, click *Add* and enter the requested information. To remove a line, select the line you wish to remove and click the *Remove* button. To edit an existing line, select the line you wish to edit and click *Properties*.

Groups

When you click the **Groups** tab, you are shown a dialog to enter and edit agent groups.



To add a new group, click *Add* and enter the requested information. To remove a group, select the group you wish to remove and click the *Remove* button. To edit an existing group, select the group you wish to edit and click *Properties*.

Activities

The final tab is for entering agent activity information. This information is used in the **lineGetAgentActivityList** and **lineSetAgentActivity** features of the TSP.

To add a new activity, click *Add* and enter the requested information. To remove an activity, select the activity you wish to remove and click the *Remove* button. To edit an existing activity, select the activity you wish to edit and click *Properties*.

TSPI_lineConfigDialog

The **TSPI_lineConfigDialog** function pulls up a dialog about a specific line device configured in the TSP. This is the same dialog used for line properties when adding or editing lines in the provider configuration dialogs (see above).

TSPI_phoneConfigDialog

The **TSPI_phoneConfigDialog** function pulls up a dialog about a specific phone device configured in the TSP. The dialog is not editable (since the phones are based on station line device) and simply shown information about the line which the phone is part of.

Source Code Structure

The source code for the JTSP service provider utilizes all of the new features present in TSP++ version 3.0. It is divided into two main executable modules – both dynamic link libraries.

JTSPUI.DLL

The JTSPUI module is responsible for managing the user-interface for the JTSP project. It is where the dialogs and code that is related to configuring the service provider is stored.

Source Code Modules

Source File	Contents
JTspUI.cpp	This is the main TSPUI shell for the DLL. It handles the CServiceProvider overrides for TSPUI requests from TAPI and creates the appropriate dialogs when it receives those requests.
Config.cpp	This contains all the different MFC dialogs, which are used to display and configure information in the TSP.
Ddx.cpp	This s a helper file which adds some new dialog data exchange functions to MFC for list-boxes, combo-boxes and the new IPEDIT control.
Properties.cpp	This contains the code for the property sheet manager which is used as a base class for all the configuration dialogs.

Class Organization

The following C++ classes are defined in the above modules:

Source File	Contents
JTspUI.cpp	**CJTspUIApp** – this is the **CServiceProvider** derived class, which doubles as a **CWinApp** for the DLL. See the documentation concerning the TSP++ UI library's object management features. It contains the following overrides: **providerConfig** **providerInstall** **lineConfigDialog** **phoneConfigDialog**
Config.cpp	**CGeneralPage** – this is the first tab in the **providerConfig** dialog. It is based on the MFC **CPropertyPage** class. **CLinePage** – this is the second tab in the **providerConfig** dialog. It provides edit support for line devices. It is based on the MFC **CPropertyPage** class. **CGroupPage** – this is the third tab in the **providerConfig** dialog. It provides edit support for agent groups. It is based on the MFC **CPropertyPage** class. **CActivityPage** – this is the final tab in the **providerConfig** dialog. It provides edit support for agent activities. It is based on the MFC **CPropertyPage** class. **CConfigSheet** – this is the **CPropertySheet** owner for all the above pages. It is a simple class which simply adds each of the above pages before displaying itself.
Properties.cpp	**CActivityPropPage** – this is the editable page which is displayed when a new activity is added or an existing one is changed. It is based on the MFC **CPropertyPage** class. **CLinePropPage** – this is the editable page which is displayed when a new line is added, an existing one is changed, or the **lineConfigDialog** function is called. It is based on the MFC **CPropertyPage** class. **CGroupPropPage** – this is the editable page which is displayed when a new group is added or an existing one is changed. It is based on the MFC **CPropertyPage** class. **CPhonePropPage** – this is the editable page which is displayed when the **phoneConfigDialog** function is called. It is based on the MFC **CPropertyPage** class.

CSingleSheet – this is a **CPropertySheet** class which takes a single property page and displays it.

Configuration keys

Configuration for the JTSP service provider is stored under the following registry key:

HKLM\SOFTWARE\Microsoft\Windows\Current Version\Telephony\JulMar Sample TAPI Server

This location is chosen due to that name being given in the constructor of the **CJTspUIApp** object (see JTspUI.cpp).

The following sub-keys are created either directly by the JTSPUI project or by the TSP++ library component to store the configuration information:

Registry Key	Description
Lines	This is where the line configuration information is stored. Each line device added to the user interface is serialized into this folder in the registry. Each line will have a single binary entry here. This is created and maintained by TSP++.
Phones	This is where the phone configuration information is stored. Each phone device added to the user interface is serialized into this folder in the registry. Each phone will have a single binary entry here. This is created and maintained by TSP++.
Agent Groups	This is where the agent group configuration information is stored. Each group added to the user interface is serialized into this folder in the registry. Each group will have a single binary entry here. This is created and maintained by TSP++.
Agent Activities	This is where the agent activity configuration information is stored. Each activity added to the user interface is serialized into this folder in the registry. Each activity will have a single binary entry here. This is created and maintained by TSP++.
IPAddress	This key is created by the JTSPUI project. The IP address of the JPBX simulator is stored here in string format
IPPort	This key is created by the JTSPUI project. The IP port of the JPBX simulator is stored here in DWORD format

JTSP

The JTSP module is responsible for managing the actual TSP interfaces. It conforms to the specifications presented in the Microsoft TSPI programmers reference.

Source Code Modules

Source File	Contents
Agent.cpp	This file contains the line device handler for the agent features of the TSP. It manages the unsolicited events for agent-related events and sends the appropriate PBX commands to log agents in and out and change their state and activity.
Conference.cpp	This file contains the line device handlers for the conferencing features of the TSP. It manages the unsolicited events for conference-related events and sends the appropriate PBX commands to create and destroy conferences.
Answer.cpp	This file contains the line device handler for the **TSPI_lineAnswer** function.
Device.cpp	This file handles the Winsock connection to the PBX and the reception and parsing of each event received. It contains an override of the **CTSPIDevice** class object.
DropCall.cpp	This file contains the line device handler for the **TSPI_lineDropCall** function.
Hold.cpp	This file contains the line device handlers for the hold/unhold/swaphold features of the TSP.
Line.cpp	This file contains the miscellaneous line overrides from the TSP++ **CTSPILineConnection** class. It handles special device requirements related to line initialization and changing features based on call activity and device status.
MakeCall.cpp	This file contains the line device handlers for the dialing features of the TSP. It manages the unsolicited events for dial-related events and sends the appropriate PBX commands to place calls.
Route.cpp	This file contains the line device handler for the **TSPI_lineRedirect** and **TSPI_lineBlindTransfer** event which are identical on this PBX.
Transfer.cpp	This file contains the line device handlers for the transfer features of the TSP. It manages the unsolicited events for transfer-related events and sends the appropriate PBX commands to blind and consultation transfer a call.
unsolicited.cpp	This file contains the event handler for the line and phone overrides. All events received from the PBX which are not acknowledgements

	form sent commands are processed by the code in this file.
jtsp.cpp	This is the **CServiceProvider** override for the service provider. It contains the constructor for the TSP.
Interface.cpp	This file contains the Winsock connection object which is used to connect, send, and receive events from the PBX. It is used by the **CTSPIDevice** override in device.cpp.
EventTypes.cpp	This file contains code related to parsing the events from the PBX.
Generate.cpp	This file contains the line device handler for the **TSPI_lineGenerateDigits** function.
Phone.cpp	This file contains our **CTSPIPhoneConnection** override and all the phone device handlers.
UUI.cpp	This file contains our line device handler for the **TSPI_lineSendUserUserInfo** function.

Class Organization

The following C++ classes are defined by the JTSP project. Each are defined in the **JTSP.H** header file:

Class	Description
CJTProvider	**CServiceProvider** derived class, which provides the required constructor for the TSP++ class library. This constructor also calls the **SetRuntimeObjects** method, which allows TSP++ library objects to be overridden.
CJTDevice	**CTSPIDevice** derived class, which handles the connection and event processing to the PBX. It also owns a worker thread object based on the **CThreadPoolMgr** template class provided with TSP++
CJTLine	**CTSPILineConnection** derived class, which handles the initialization of each line device configured and then provides the request handlers for each **TSPI_lineXXX** request given by TAPI.
CJTPhone	**CTSPIPhoneConnection** derived class, which handles the initialization of each phone device configured and then provides the request handlers for each **TSPI_phoneXXX** request given by TAPI.

Basic processing flow of JTSP

Thread management

Event Thread

JTSP uses a single thread to pull events from the Winsock connection to the PBX. That thread waits for data on the socket, verifies the integrity of the received event and then queues it for the device input thread. It then goes back to sleep waiting for the next event. This approach ensures that the provider will not lose data due to socket overrun since the processing done on the request is very minimal (simply queuing it to a linked-list).

Device Input Thread

This thread parses each received event string from the PBX and creates an event block from it. It then examines the created event block and determines which line or phone device should process the event based on information contained in the packet. It then queues the event block into the thread pool manager queue. If the event could be intended for multiple devices then it is queued for each device.

Worker Threads

These threads are created and managed by the new Thread Pool template class. Events are queued using the line/phone extension as the serialize key. This means that:

1. Events intended for a specific line/phone device are always handled in the order they are received

2. Multiple events for different devices may be handled simultaneously.

3. If an event is received for a device that is already processing an event by an existing worker thread, that event is queued until the existing (and all previous events in the queue) are handled first.

Note that the worker threads are not guaranteed to always be used for the same line or phone device. Any worker thread may pass an event to any line or phone device.

TAPI Request Threads

TAPISRV manages a thread pool internally and uses this list of threads to pass incoming requests from TAPI clients either local to the machine or on network clients. TAPI requests are generally intended to be handled sequentially for each line or phone device. Each incoming TAPI request can have one of two things happen to it.

1. The request can be started on the incoming TAPI thread – this means that the thread was *not* created by the service provider, and potentially was created *before*

the TSP was loaded. This means TLS data (if you use it) might not be associated with these threads.

2. The request can be queued if there is existing TAPI requests associated with the intended line or phone device owner. If this happens, the derived TSP code will not see the incoming TAPI thread at all – the request will be created and then the thread is released without being passed through the derived provider.

TAPI Request Management

Each incoming TAPI request causes the creation of a **CTSPIRequest** object which manages the information for that specific request. In this sample provider, each request is associated with a handler function in the line or phone using the new **ON_TSPI_REQUEST** macros (see the TSP++ documentation for more information on this topic).

The following is a section of the code from **LINE.CPP** in the JTSP source code which shows how each TSPI request is routed by the TSP++ library.

```
/*-----------------------------------------------------------------------------*/
// TSPI Request map
/*-----------------------------------------------------------------------------*/

BEGIN_TSPI_REQUEST(CJTLine)
        ON_AUTO_TSPI_REQUEST(REQUEST_ACCEPT)
        ON_AUTO_TSPI_REQUEST(REQUEST_SECURECALL)
        ON_AUTO_TSPI_REQUEST(REQUEST_RELEASEUSERINFO)
        ON_TSPI_REQUEST_SENDUSERINFO(OnSendUUI)
        ON_TSPI_REQUEST_ANSWER(OnAnswer)
        ON_TSPI_REQUEST_MAKECALL(OnMakeCall)
        ON_TSPI_REQUEST_DIAL(OnDial)
        ON_TSPI_REQUEST_DROPCALL, OnDropCall)
        ON_TSPI_REQUEST_HOLD(OnHoldCall)
        ON_TSPI_REQUEST_UNHOLD(OnRetrieveCall)
        ON_TSPI_REQUEST_SWAPHOLD(OnSwapHold)
        ON_TSPI_REQUEST_REDIRECT(OnRedirectOrBlindTransfer)
        ON_TSPI_REQUEST_SETUPXFER(OnSetupTransfer)
        ON_TSPI_REQUEST_COMPLETEXFER(OnCompleteTransfer)
        ON_TSPI_REQUEST_BLINDXFER(OnRedirectOrBlindTransfer)
        ON_TSPI_REQUEST_SETUPCONF(OnSetupConference)
        ON_TSPI_REQUEST_PREPAREADDCONF(OnPrepareAddToConference)
        ON_TSPI_REQUEST_ADDCONF(OnAddToConference)
        ON_TSPI_REQUEST_REMOVEFROMCONF(OnRemoveFromConference)
        ON_TSPI_REQUEST_SETAGENTGROUP(OnSetAgentGroup)
        ON_TSPI_REQUEST_SETAGENTSTATE(OnSetAgentState)
        ON_TSPI_REQUEST_SETAGENTACTIVITY(OnSetAgentActivity)
        ON_TSPI_REQUEST_GENERATEDIGITS(OnGenerateDigits)
END_TSPI_REQUEST()
```

When the indicted request is received or an event is being processed with the request being the *current* request, TSP++ routes the request to the indicated function.

In JTSP, the request functions send the appropriate PBX command to the switch and then wait for the acknowledgement from the switch. based on the response code

indicated in the acknowledgement event, the request is considered successful or failed and TAPI is notified using the **CompleteRequest** member function.

The processing of the request is always left to the unsolicited handler since the events could be received due to another program interacting with the PBX (and therefore may not be a request of ours).

Sample handler implementation

An example from the JTSP source code is provided below. This is the handler for the **TSPI_lineAnswer** request which is used by TAPI to answer an incoming call:

```
/*****************************************************************************
** Procedure:  CJTLine::OnAnswer
**
** Arguments: 'pReq' - Request object representing this ANSWER event
**            'lpBuff' - Our CEventBlock* pointer
**
** Returns:    void
**
** Description:  This function manages the TSPI_lineAnswer processing
**               for this service provider.
**
*****************************************************************************/

bool CJTLine::OnAnswer(RTAnswer* pRequest, LPCVOID lpBuff)
{
       // Cast our pointer back to an event block
       const CEventBlock* pBlock = static_cast<const CEventBlock*>(lpBuff);
       CTSPICallAppearance* pCall = pRequest->GetCallInfo();


       // If we are in the initial state (i.e. this request has not been
       // processed before by any other thread). Then move the packet to the
       // waiting state so other threads will not interfere with other events
       // timers.  This is guaranteed to be thread-safe and atomic.
       if (pRequest->EnterState(STATE_INITIAL, STATE_WAITING))
       {
              // Send the "AN" request to the PBX for this call. This
              // inline will generate the proper command using the given
              // call's call-id.
              GetDeviceInfo()->DRV_Answer(this, pCall);
       }

       // If we are in the waiting stage (2) then see if we received an event
       // from the switch (vs. an interval timer) and if that event was an
       //ACK/NAK in response to the command we issued.
       else if (pRequest->GetState() == STATE_WAITING && pBlock != NULL)
       {
              // If this is a command response for our RELEASECALL, then
              // manage it and don't pass it through to the unsolicited
              // handler code by returning true.
              const CPECommand* peCommand = dynamic_cast
                     <const CPECommand*>
                     (pBlock->GetElement(CPBXElement::Command));
              const CPEErrorCode* pidError = dynamic_cast
                     <const CPEErrorCode*>
                     (pBlock->GetElement(CPBXElement::ErrorCode));

              if (pBlock->GetEventType() == CEventBlock::CommandResponse &&
                     peCommand->GetCommand() == CPECommand::AnswerCall &&
                     pidError != NULL)
```

```
                {
                        // Complete the request with the appropriate error code.
                        TranslateErrorCode(pRequest, pidError->GetError());
                        return true;
                }
        }

        // Check to see if our request has exceeded the limit for processing.  If
        // so, tell TAPI that the request failed and delete the request.
        if (pRequest->GetState() == STATE_WAITING &&
                (pRequest->GetStateTime()+REQUEST_TIMEOUT) < GetTickCount())
                CompleteRequest(pRequest, LINEERR_OPERATIONFAILED);

        // Let the request fall through to the unsolicited handler – it will
        // change the call state when the PBX reports that it has been answered.
        return false;

}// CJTLine::OnAnswer
```

The first step in the request is to cast the passed event pointer to the internal event object. This pointer is always passed as a void* pointer since it is the event block created by the derived provider which could be anything.

The second step is to ensure that other threads will not interfere with this request while it is being processed. Specifically, this means the interval timer thread which comes through every second (in JTSP – see **device.cpp**). This is done through the **SetState** or **EnterState** methods of the request. This moves the current state of the request to something other than **STATE_INITIAL** which is where it always starts.

Next, the TSP sends the answer request to the PBX.

When an event is received or an interval timer fires, this request will be re-entered and one of the two code paths will be taken:

1. On an event, since the state is **STATE_WAITING** as per the second step above, the event type is checked to see if this is an acknowledgement to the answer command we sent to the PBX. If so, the error code is checked and the request is completed.

2. On a timer, the time elapsed since we changed the state of the request to **STATE_WAITING** is checked. It we have exceeded 30 seconds of time then the TSP assumes that the switch is not going to respond for whatever reason and the request is canceled. This is important because no other TAPI requests will be started until this one completes.

The phone request handlers are located in **phone.cpp** and the line handlers are located in the file with the same name as the request (i.e. **conference.cpp**, **hold.cpp**, **transfer.cpp**, etc.)

For more information on request routing and processing, see the TSP++ Programmers Reference.

Event processing

In the JTSP sample, the unsolicited event handler always handles event processing in JTSP. When an event is routed by TSP++, it looks up the current request object for the line or phone. It then passes the request and the received event to the handler functions as defined in the line/phone request map (see above).

If the handler function returns a non-zero value, then no more processing is done (i.e. the handler is indicating that it *processed* the request).

If, however, the handler returns a zero return code *or* there is no active request in the line/phone list then the event is passed to the virtual function:

virtual bool UnsolicitedEvent(LPCVOID *lpBuff*);

This function may then provide some event management that is independent of TAPI requests.

In the JTSP sample, the unsolicited event handler is where *all* the work is done for the TSP. The request handlers always return zero unless the received event is an acknowledgement for a specific command that the request sent.

Both the phone and line handlers are contained in **unsolicited.cpp**.