DSSP Phone Emulator TAPI Service Provider Configuration and Programmer's Guide

TSP++ Version 3.00
Last Modified: 02/05/99 2:28 PM

Table of Contents

Introduction

This document provides an overview of the Sample TAPI Service Provider built for the DSSP Station emulator. It gives a description of the features of the TSP and in-depth design notes related to building the TSP with the TSP++ product.

Project Overview and History

The **JulMar DSSP Emulator** program was written to provide a stable test platform for a 1st party TAPI service provider sample that was provided with the TSP++ version 2.x product. The emulator is responsible for simulating different aspects of a PBX station.

In order to more fully test the 3rd party implementation of TSP++ (V3.0), this provider has been ported to the V3.0 library and is now included with V3.0 as a sample. This sample also demonstrates how to integrate the user-interface and TSP code into one executable file.

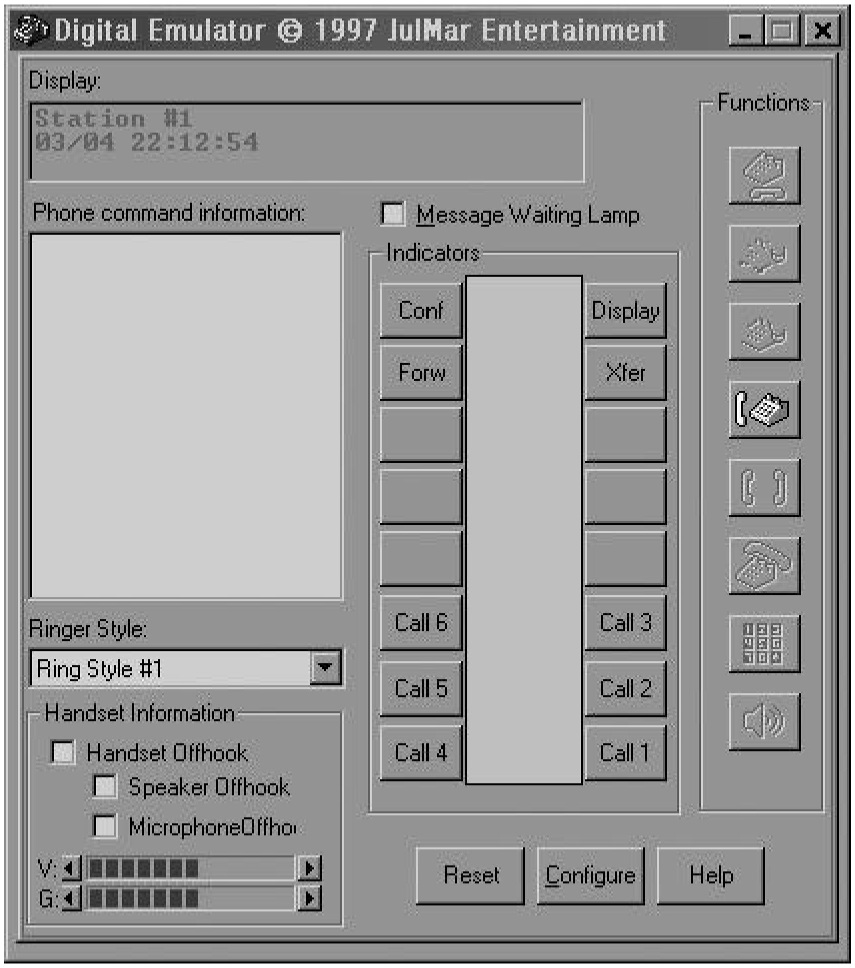
Digital Switch Emulator

The **EMULATOR** program is a component of the **DSSP32** samples. It acts like a telephone switch and station in a single program. The 32-bit version of the emulator which works with **DSSP32** may run on any network machine, but you must have TCP/IP connectivity to use the sample.

A user can interact with the emulator using it's interface – just as a user could interfere with the operation of the TSP by directly manipulating the station itself.

Running the Emulator

The first step in testing out the **DSSP32** sample is to start the emulator program. It must be running before any TAPI application is started or the service provider will return **LINEERR_NODEVICE** and cause TAPI to ignore and unload the sample provider. The time-out period is 1.5 seconds, so the system will seem to pause for about two seconds while the provider searches for the emulator. Once up, the emulator looks like:



Each of the buttons on the emulator causes some event to happen to our *simulated* PBX switch.

- The **Handset Information** group indicate what state our handset is in. The microphone and gain are controlled separately. If they are clicked upon, TAPI will get notified about a hookswitch change. If TAPI changes the state of the hookswitch device, then the checkboxes will also change.

  The volume and gain may be controlled through the buttons located and by the service provider using the **phoneSetVolume** and **phoneSetGain**.

- The **Message Waiting** button causes our Message Waiting lamp to turn on and off. This also is bi-directional in the service provider.

- Each of the function buttons (16 in all) running in the center of the phone may be setup as a different function. In the listed picture, 5 are call appearances, and 3 are features of the switch. The others are not set. When the **DSSP** sample first initializes, it will query the emulator about its settings and set itself up appropriately. The button information is retrievable from the TAPI function **phoneGetButtonInfo**. If any of the advanced line features are required (conference, transfer, forward), then a button *must* be defined in this set of 16. Otherwise, a **LINEERR_OPERATIONUNAVAIL** will be returned by the service provider.

- The buttons along the bottom of the emulator correspond to **Reset**, **Configure**, and **Exit**.. The emulator should be configured while no TAPI devices are running. This insures that the provider and the emulator stay in synch. Once the service provider starts, the emulator will not allow the configuration button to be pressed.

- The buttons along the right side of the emulator screen correspond to the following functions:

Hold	Place the active call (solid lamp) on hold.
Release	Hang up on the active call and reset the lamp.
Dial	Dial a series of digits on the LOCAL handset.
SimulateCall	Create an incoming call on one of the addresses.
Busy	Simulate a busy signal on outgoing call
Answer	Answer the incoming call from the LOCAL handset.
DialRemote	Simulate dialing events on REMOTE handset
Tones	Simulate tone generation from REMOTE handset

- The **Phone Command Information area** will contain a list of what the service provider is asking for, and how the emulator is responding. As each command is executed by the emulator program, detailed information about state changes being reported and commands received from the service provider are listed. The color schemes used are:

➢ **Green** for service provider commands

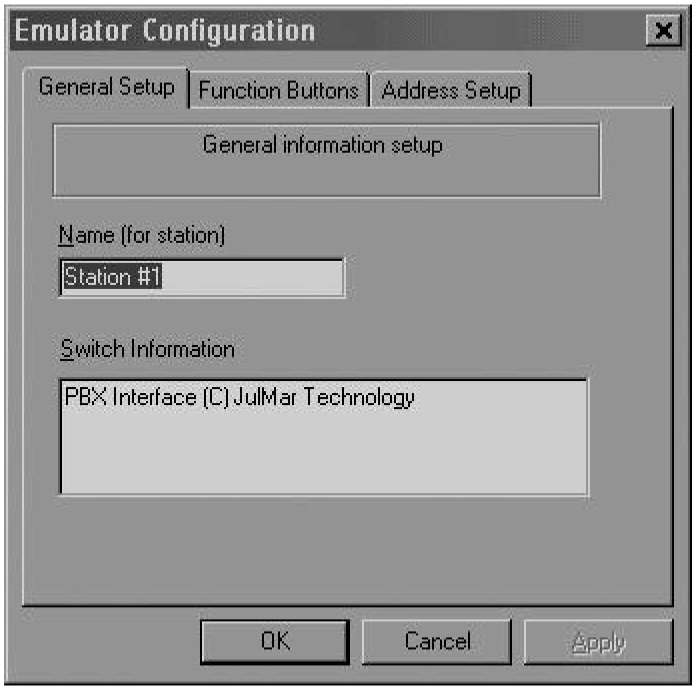
➢ **Red** for emulator commands

➢ **Blue** for status changes.

- The display will correspond to what the emulator is telling the service provider, and provides an emulation for the **phoneGetDisplay** API. It is currently read-only, although the **phoneSetDisplay** function will return a success result.

Configuring the Emulator

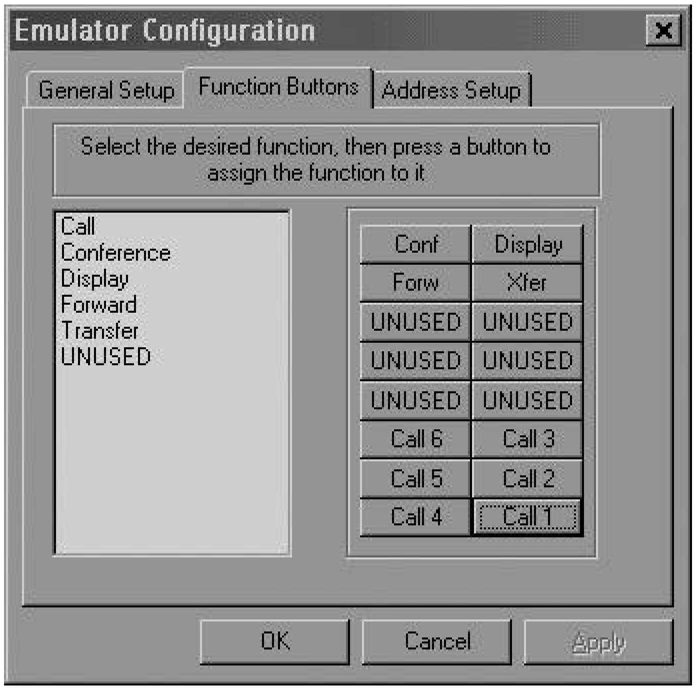
The first step in using the emulator is to configure it. On initial installation, the emulator will have no call appearances available to it. This means that no addresses are available for the line it is attached to, which is generally not a useful phone. The configuration button is the last button on the right of the dialog and will bring up a dialog which looks like:



Name (for station): Station #1

Switch Information: PBX Interface (C) JulMar Technology

The general tab allows the name for the station to be entered. This will be the caller ID information reported for any call placed on this address (not received). Also, the switch information is listed here. This information will be returned by the service provider in the **LINEDEVCAPS** and **PHONEDEVCAPS** structures.



The function button tab allows each of the buttons to be configured to a function. To configure a button simply select the function in the listbox on the left, and click the appropriate button on the right. Any button assigned to a call must then be setup with address information using the next tab. The current available functions are:

Call - This is a call appearance. There is no limit (except in buttons) to the number of available call appearances within the emulator. Each is considered a separate address, and only one address may be in an active state at any given time (the others must be inactive or in some hold state).

Display - This is used to update the display. It is available as a feature to the emulator, and is not currently used within the service provider (plans for use are forthcoming).

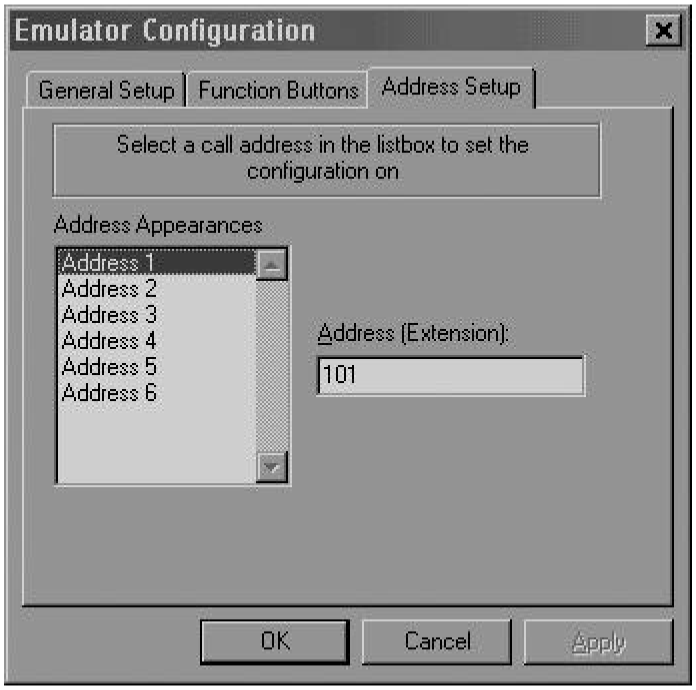
Conference - This implements a simply 3-way conference call. It is not usable within the emulator, but the service provider will require that the button is defined to support conferencing. The conference features of the emulator are currently under work for cross-addressing and more than three parties.

Forward - This implements a station-wide forwarding system. If the button is pressed within the emulator, it will forward to a bogus number (listed in the display). The service provider allows the forwarding to be to any dialable number. Specific call forwarding, internal/external forwarding, and busy forwarding are not supported.

Transfer - This implements a simple transfer facility which can be driven by the service provider. A consultant call is provided in the emulator in the **Dialtone** state, which can then be dialed and transferred. This form of transfer (called a *consultation transfer*) is only available to the service provider. Blind transfer is the default and is used by both the emulator and service provider if the button is pressed directly.

Unused - this is the default state and should be used for any button not defined to a function.

The **Forward**, **Conference**, and **Transfer** buttons must be defined if these features will be available to the service provider.



Select a call address in the listbox to set the configuration on

Address Appearances

Address 1
Address 2
Address 3
Address 4
Address 5
Address 6

Address (Extension): 101

The address setup tab lists each address which was created by assigning a button to a call appearance. Each address needs an extension provided for it. The extensions should be unique phone numbers which identify the address and will be used in caller ID information.

32-bit Digital Switch Sample Service Provider (DSSP32)

The **DSSP32** sample is a service provider written for a telephone network with more advanced features than a modem. It works in conjunction with an emulator program that acts like a telephone switch and phone in one. (For more information on this, see the preceding section on the *DSSP Emulator*).

The 32-bit version of the emulator uses Windows Sockets to communicate with the service provider. For this reason, you *must* have TCP/IP installed on the machine that the emulator runs on and the machine that the service provider runs on. They can be the same machine if desired, if this is the case, configure the service provider to use **127.0.0.1** as the IP address of the server.

NOTE:

The provider is a sample, the configuration of the hardware for which a service provider is written really drives the implementation of the provider. For example, in our sample, we assume that if a conference call is dropped, the whole conference is dropped. This may not be the case on the hardware another provider is driving. The code presented here is an example of an implementation - not a guideline for any implementation for a switch.

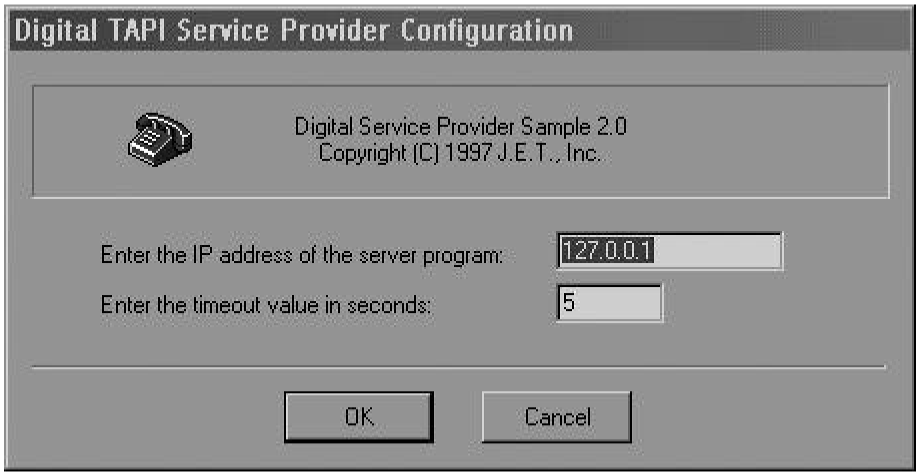
Installation

Before any TAPI applications are started, make sure to start the emulator program. Installation of the **DSSP32** sample is accomplished by copying the **DSSP32.TSP** file into the WINNT\SYSTEM32 (for Windows NT) or WIN95\SYSTEM (for Windows 95 with TAPI 2.1) directory and using the Telephony Control panel applet to add the provider to the system.

When the service provider is first installed, it will prompt for configuration information. See the section on *Configuring the DSSP Sample* for information on what each field means.

Configuring the DSSP32 sample

Since the 32-bit sample is designed to take advantage of client/server technology, it has a configuration screen (unlike its 16-bit predecessor). When you configure the **DSSP32** sample, the following screen will appear:


Digital TAPI Service Provider Configuration

Digital Service Provider Sample 2.0
Copyright (C) 1997 J.E.T., Inc.

Enter the IP address of the server program: 127.0.0.1

Enter the timeout value in seconds: 5

The **IP Address** indicates where the server **EMULATOR** program is running. If it is running on the same machine as the server provider (local), then enter **127.0.0.1** as the IP address. Otherwise, enter the dotted-decimal number indicating where the server is running.

The timeout value indicates how long the service provider will wait for the server to accept a pending connection during the initialization process. The longer this timeout, the longer the provider waits for the server. This means that if the server isn't running, or the network is down, the service provider will wait *x* seconds and pause the telephony subsystem (which will effectively pause the calling application during the **lineInitialize** function).

Using the DSSP32 Sample Provider

Once the emulator is started and configured (the configuration will only need to be done the first time), the **DSSP** sample provider may be started by invoking any TAPI application. The **Debug Information** window will list any connections made by the **DSSP** sample provider.

Files

The DSSP project is located in the **DSSP** directory and consists of the following main files:

File	Description
DRV.CPP	This contains the communication methods for directing traffic to and from the emulator program. It utilizes the MFC **CSocket** class.
INIT.CPP	This contains the initialization code.
CONFIG.CPP	This contains the configuration and user-interface code.
DSSP.CPP	This contains the **CserviceProvider** overridden class.
LINE.CPP	This contains the **CTSPILineConnection** override code.
LINEREQ.CPP	This contains the request management code for the line.
PHONE.CPP	This contains the **CTSPIPhoneConnection** override code.

Basic class structure

This samples uses a series of classes and overrides to create the service provider.

CDSProvider

The derived service provider class is **CDSProvider**. It provides the basic shell for the service provider which sets up the classes to manage TAPI requests. It contains a constructor which uses the **CServiceProvider::SetRuntimeObjects** method to establish an override for the device, line and phone objects.

Method	Purpose
CDSProvider	Constructor for the service provider. It gives the CServiceProvider class its required parameters and sets the objects in place to override other TSP++ classes through the SetRuntimeObjects method.
providerEnumDevices	This returns the number of lines and phones supported (1 and 1).

CDSDevice

The sample also creates a **CDSDevice** class that manages communication with the actual device through the companion application. It contains driver methods that are called by the line and phone objects to perform work on the emulated switch.

Method	Purpose
Init	Overrides the initialization of the device to create it's socket object and connect to the device. If the connected isn't up within two seconds, the socket is closed and the provider refuses to load.
DRV_xxxx	Driver functions – these are called by the line and phone request management functions to send commands to the emulator. Each DRV_xxx function corresponds to a switch function. Note that this is specific to this provider implementation – it is not a standard part of the TSP++ library.

CDSLine

This overrides the **CTSPILineConnection** and manages all the initialization and state machines for the line device that is modeled by the TSP.

Method	Purpose
Init	Overrides the initialization of the line to adjust the capabilities presented by this provider.
GetDevConfig	Overrides the **TSPI_lineGetDevConfig** function to return some piece of information so that HyperTerminal will recognize the device.
SetDevConfig	Overrides the **TSPI_lineSetDevConfig** function to return some piece of information so that HyperTerminal will recognize the device.
OnCallFeaturesChanged	This adjusts call features based on what the emulator is capable of – specifically, this disallows conferences to be placed on hold, and doesn't allow conferenced calls to be dropped.
OnAddressFeaturesChanged	This adjusts the address features based on the emulator capabilities.

OnLineFeaturesChanged	This adjusts the line features based on the emulator capabilities.
UnsolicitedEvent	This is called when an event is not completely handled by a request function –or when there is no request pending for a received event.
processXXXX	These functions manage requests from TAPI and perform all call control functions against the emulator.

The restrictions of the line device are:

1. Only unconditional forwarding is supported. The device may not be forwarded unless there are *no* active calls on the device.
2. **linePickup** will return an error result since there is no way to see a call outside the emulator device.
3. **lineUnpark** is always marked as available on a line but may return an error if no call is waiting to be unparked on the emulator. Since there is no method to query for this information, the function cannot be masked out of the address features.

CDSPhone

This overrides the **CTSPIPhoneConnection** and manages all the initialization and state machines for the phone device that is modeled by the TSP. A few of the common phone functions are supported. Any changes made in the user interface of the emulator should be reflected in the service provider.

Method	Purpose
Init	Overrides the initialization of the phone to adjust the capabilities presented by this provider.
UnsolicitedEvent	This is called when an event is not completely handled by a request function –or when there is no request pending for a received event.
performXXXX	These functions manage requests from TAPI and perform phone-control functions against the emulator.

The restrictions of the phone device are:

1. **phoneSetDisplay**, **phoneSetButton**, **phoneSetLamp** are not supported as there is no capability in the emulator at this time..

Basic Processing Flow

In this sample provider, each request in the line or phone queue is given the opportunity to examine device responses and decide if it is applicable to that request. As a device response is processed, the **CTSPIDevice::ReceiveData** method enumerates through each line and phone and gives it the chance to see the packet. As soon as one of the **processXXX** functions returns an indicator saying that it processed the response, the enumeration stops. Each request is sub-divided into a state machine by a **processXXX** function that is applicable for the device and function being invoked.

Request routing

The requests are routed using the TSP++ request map architecture as in the JTSP sample and all are coded in the **linereq.cpp** and **phone.cpp** files.

As an example, look at the **processAnswer** function which is mapped to the **REQUEST_ANSWER** (**TSPI_lineAnswer**) in **line.cpp**

```
bool CDSLine::processAnswer(RTAnswer* pReq, LPCVOID lpBuff)
{
        EVENTBUFF* pevBuff = (EVENTBUFF*) lpBuff;
        CTSPICallAppearance* pCall = pReq->GetCallInfo();

        switch (pReq->GetState()) {
                // Step 1:
                // Ask the switch emulator to answer the call - this would be to press
                // the button associated with the offering address.
                case STATE_INITIAL:
                        pReq->SetState(STATE_WAITFORCONNECT);
                        GetDeviceInfo()->DRV_AnswerCall(
                                        pReq->GetAddressInfo()->GetAddressID());
                        break;
                // Step 2:
                // Address should indicate a connected end-party.
                case STATE_WAITFORCONNECT:
                        if (pevBuff->dwResponse == EMRESULT_ADDRESSCHANGED) {
                                const LPEMADDRESSCHANGE lpAddrChange =
                                        (const LPEMADDRESSCHANGE) pevBuff->lpBuff;
                                if (lpAddrChange->wAddressID ==
                                        pReq->GetAddressInfo()->GetAddressID() &&
                                        lpAddrChange->wNewState ==
                                                ADDRESSSTATE_CONNECT) {
                                        CompleteRequest(pReq, 0);
                                        pCall->SetCallState(LINECALLSTATE_CONNECTED);
                                        return true;
                                }
                                else CompleteRequest(pReq, LINEERR_OPERATIONFAILED);
                        }
                }
                break;
        }
        // If we failed, then complete the request with an error - note the
        // emulator doesn't tell us WHICH request failed, we have to rely on the
        // knowledge that the error had to do with our address and therefore was
        // probably our request.
        if (pevBuff && pevBuff->dwResponse == EMRESULT_ERROR &&
                (pevBuff->dwError == EMERROR_INVALADDRESSID ||
                 pevBuff->dwAddress == pReq->GetAddressInfo()->GetAddressID())) {
                CompleteRequest(pReq,
                        (pevBuff->dwError == EMERROR_INVALADDRESSSTATE) ?
                                LINEERR_INVALCALLSTATE :
                        (pevBuff->dwError == EMERROR_INVALADPARAM) ?
                                LINEERR_INVALPARAM :
                                LINEERR_OPERATIONFAILED);
                return true;
        }
        // Let the request flow through the unsolicited handler
        return false;
}// CDSLine::processAnswer
```

In this request, there are two defined states – **STATE_INITIAL** and **STATE_WAITFORCONNECT**. When the request is first started, it is in the **STATE_INITIAL** state, and the TSP sends a command to answer the given call object on the emulator.

On the next entry, the state has been changed to **STATE_WAITFORCONNECT** (by the **SetState** command issued during the **INITIAL** state). The TSP then watches for the response from the emulator. Once the response is seen, the request is either completed successfully, or the request is completed with an error. Note that very little error-recovery or detected is attempted – this is to keep this sample simple and easy to understand. For a more complete sample with error recovery, see the *JTSP* sample.

Unsolicited event processing

If no request handled or expected the response, then it is processed by an *unsolicited* event processor. This function handles events that come from the device which were unexpected and typically happened as a result of some outside influence on the device (i.e. the user pressing a button on the emulator). This is handled in the **line.cpp** and the **phone.cpp** files.

Implementation Notes

The **DSSP32** sample uses a secondary thread to communicate with the emulator. The implementation relies on the MFC architecture and creates a *user-interface* thread which basically means it has a **CWnd** handle map associated with this secondary thread. This is simply for the **CSocket** support in MFC. The device thread manages the socket interface to the emulator and passes all responses to the **CTSPIDevice::ReceiveData** function which will then pass it through the line and phone objects.