# ActiveX TCP/IP Programming for the 700 RF Terminal

# How the system works...

The RF Terminal has a 6x24 LCD screen and up to 99 voice messages which can be activated by the host user program. Messages from the host user program are written to the serial port to which the applicable Base Station is attached. Up to 64 RF Terminals can be controlled by one base station, so the host user program must address the applicable RF Terminal by its ID character. When the host receives a message from the Base Station, it will receive data with the Terminal ID also included (not true for one-way mode).

There is no programming on the RF Terminal itself. All programming is on the host computer. Any language and/or platform that can read/write to a serial port can easily control a network of RF Terminals.

Some users will prefer sending the formatting sequences directly over the serial port using the Low Level Commands. Others will prefer the Windows ActiveX and TCP/IP controls.

**This is how the RF Terminal operates:**

Messages from the host user program are sent to the Base Station (via the serial port), then from the Base Station to the RF Terminal. The Terminal responds back to the Base with data and its Terminal ID. The data is then transmitted from the Base to the host computer where it is processed and the next command is sent out.  Each RF Terminal has a unique Terminal ID, allowing a single Base Station to handle up to 64 Terminals.

Dialog is established when a Terminal **SIGNS ON** to the RF network.  The host computer application waits until a Terminal **SIGNS ON**, then begins its processing by sending the first prompt out to the Terminal via the Base Station.  If the Terminal does not receive a prompt from the host, it goes into "sleep" mode, "waking up" and checking with the Base periodically to see if it has any messages waiting.  This conserves battery power and reduces radio traffic.

We have tried to make it easy for the programmer to communicate with the Base Station; no protocol or hand-shaking is required.  This type of communication is fine when the Base is located only a few feet from the serial port it is connected to. If you are locating your Base Station farther away, use shielded, grounded (bare wire Pin 1 touching shield) cable, lower baud rates and possibly, line drivers for very noisy environments.

# Before you begin...

Before you begin programming, there are some factors you should take into consideration during the planning process.

- **Plan for system failures**. This includes hardware failures, software failures and operator failures. In order to create an efficient application, you must put some thought into what you will do when different parts of the system fail.

- **Look for All Errors**. Be sure your program is trapping all possible error conditions that the Base Station may return to you. The list includes:

  **Sequence Errors detected**
  **Illegal Command detected**
  **Base Station Initialized**
  **Addressing a Terminal Not Signed In**

  Forgetting to program for these error conditions is a common mistake. Even though you think your code will never make a mistake, take advantage of the feedback that the Base Station provides.

- **Parse the Returned Strings thoroughly**. Don't assume anything about the next response from the Base to your program and look only for the partial string such as the ID only; parse the returned string completely and be sure you are examining every possibility. Failure to do so is a common mistake.

- **Plan for expansion**. You may start small (1 Base/1 Terminal) but try to create an application that will allow for easy expansion and addition - especially of Terminals.

- **Site Evaluation**. Site Testing does not require that you have an application up and running and can save you time when you do sit down to create your program if you already know what you will be dealing with in terms of Base Stations and Relays.

- **Use the Demo Programs**. The demo programs can at least allow you to see how the system functions and whether you can anticipate any system-wide problems. The demo programs should also be used as a response-time benchmark.

# Planning for failures...

## Hardware Failures

Let's assume that each part of the system has failed. How are you going to know what has happened and how are you going to recover?

- The most frequent failures are at the Terminal level. If a Terminal has a hardware failure, it will not be able to **SIGN OUT**. It is possible for the Terminal operator to press the ON/OFF key or the F1 key by accident, forcing the Terminal to **SIGN OUT** - sometimes in the middle of a transaction. This happens at battery-changing time also. You need to plan for partial transactions - do you trash the data you do have and start over, or pick up where you left off?

- Keep in mind that if a Terminal has **SIGN**ed **OUT** in mid-transaction, the Base Station clears any pending message for that Terminal before it will allow it to **SIGN ON** again. Make allowances to re-send messages or prompts that were cleared upon **SIGN ON** if necessary.

- Relay Station failures are often cable-related. If a Terminal puts out a "Who Can Hear Me?" message and a Relay that is for some reason not connected to the Base Station (bad cable, cut cable, broken connectors) hears it, it answers with the message:

  **Relay n Cannot Be**
  **Heard by the Base**
  **Notify Supervisor**
  **Press Any Key**

  At this point, it is up to the operator to notify someone that the Relay is not communicating with the Base and to check the cabling first. There is no message sent to the host, so it is very important that the operator that receives this message notify someone immediately.

## Operator Errors

- Plan on your operator walking out of range and going to lunch in the middle of a transaction. What do you do with the data you do have, and where are you going to start up again?

- Let's say your operator is **SIGNED ON** and decides it's time to take a break. Instead of pressing the F1 key to **SIGN OUT**, he presses the OFF key. Pressing the OFF key is OK (it will **SIGN** him **OUT**) but there is a delay until the **SIGN OUT** is acknowledged. Because of the delay, the operator might think he didn't press the key hard enough and press it again - this time actually powering down the Terminal before the **SIGN OUT** was complete. If this happens, you need to plan to resend the last prompt to the Terminal when he **SIGN**s **ON** again.

# PromptNET TCP/IP Active X Control

**PromptNET/ActiveX** is a drop in COM component that allow programmers to easily add the ability to send prompts to and receive data from their R/F Terminal via an RF Base Station across a TCP/IP network connection.

**PromptNET** requires a "Client" computer on a TCP/IP network (to which up to 4 serial Base Stations can be attached) and a "Server" computer visible on the network to the Client.

The client computer runs the **PromptNET Client Utility** program as a background task. The server computer runs your application which uses the **PromptNET** ActiveX component to communicate with the Client.

The ActiveX component is compatible with Visual Basic, Visual C++, Delphi, and most other 32-bit development platforms. The Client Utility program requires Windows 95 or later, the Server ActiveX control requires Windows 98 or later. See the help file for installation instructions.

## Programming Considerations

### Network Setup

- The network settings on both client and server must support TCP/IP communications.
- It is critical that the client and server computers are "visible" to each other across your network. Both computers must have an IP address in the same subnet. The server requires a static IP address while the Client can either have a static address or use an assigned IP address via a DHCP server or equivalent. Refer to your Windows networking administration utility in the Control Panel to configure computer IP address settings.
- PromptNET uses ports 54123 (server) and 54124 (client).
- You can link server and client through a dial-up or DSL internet link as long as the server has a static IP address and your router passes the above ports.
- If you are unsure of how to set up your IP configuration properly, refer to your network administrator for help.

### Client Utility

- Make sure the Client Utility is properly installed on the client computer and communicating with at least one Base Station. Test the Client by cycling power on the Base Station. You should see a "Base SignOn" message in the monitor window.

## Server Communications

- Run the ***Server Test Utility*** on the server computer. Go to the client computer, set the IP address for the server computer and a unique "**Base Name**" for the ***Client Utility*** and attempt to connect to the ***Server Test Utility***. If the ***Client Utility*** connects, you are configured properly. Go to the server computer, shut down the ***Server Test Utility*** and begin work on your **PromptNET** server application.

- For Client/Server communications, the ***Client Utility*** is required to be running on the PC that the serial Base Stations are attached to.

- Before making any **WDIPterm** method calls in your application, make sure to set the **ServerOn** property to "true".

## Test For Good Communication

- Implement an ***event handler*** for **OnTermBaseRegister** that causes a beep or displays a message when called. If communication between the host PC and the  base station is good, your ***event handler*** will fire when your program is running and you power up an attached base station.

## Multiple Base Stations

- For installations using multiple base stations attached to a single client PC, simply use the four "channels" provided by the ***Client Utility*** program.

## Multiple Client Computers

- The number of client computers is limited only by the processing capability of your server.

## Terminal Tracking

- Since you get only one set of ***event handlers***, you will need some scheme for keeping track of where each terminal (up to 64 per base station, up to 4 base stations per client) is in its transaction sequence. One possible solution is to use  a "state" variable for each terminal (perhaps stored in an array). Test the state variable to determine the next prompt for any given terminal. See the samples on the RF Terminal Utilities CD for more ideas (You can also download them from our website: http://www.barcodehq.com/WDIPterminal.exe)

- It is very important to keep track of "login status" for each terminal. Every **SignOut** ***event*** should have an associated **SignIn** ***event*** and a given terminal should not be allowed to **SignIn** twice without and an intervening **SignOut**. Multiple **SignIns** from one terminal without appropriate **SignOut**s indicate either:

  1) A terminal going out of range and having its power cycled before returning within range OR

  2) Two (or more) terminals using the same *ID* (*terminal ID* conflict).

## Concepts

Drop-in components are tools that are added to your programming environment "tool kit". Only the ActiveX variety are widely compatible with almost all development environments. When you use drop-in components in your program you will follow the standard object-oriented programming paradigm that uses ***properties***, ***methods***, and ***events*** to implement the functionality of the drop-in component.

***Properties*** are the various configuration variables used by the drop-in component. An example of a property is the **ServerOn** setting.

***Methods*** are function calls used to issue commands and access features of the drop-in component. An example of a method is sending an **Input** command to the terminal.

***Events*** are function definitions placed in your application's source code. The function definitions in your source code are called *Event Handlers*. The skeleton structure of the event handler's source code is automatically generated. The code in the *Event Handler* is called ("fired") by the drop-in component when a specific event occurs. An example of an event is when a terminal returns data and the **OnTermData** event is fired.

The details of how to access ***Properties/Methods/Events*** varies between development platforms. Details of how it works in some of the most popular platforms is illustrated in the samples included with the RF Utilities CD or available for download from our website at:

**http://www.barcodehq.com/utilities/PromptNETActiveX.exe**

## Properties

Properties are the various configuration variables used by the **WDIPterm** control. They are directly assignable in your application (eg. "**WDIPterm.ServerOn = true**") and can be set in your development environment's object browser.

*Note that your development environment may show more properties for the* **WDIPterm** *control than are listed here. This is normal. You may ignore properties you see listed in your development environment that are not listed here.*

### ServerOn

| | |
|---|---|
| *Valid values:* | *True, False.* |
| *Function:* | Set to *True* to enable the server. Set to *False* to turn the server off. You should leave this off unless your program is actually running. Setting it to *True* at design-time can cause problems. |

### Quiet

| | |
|---|---|
| *Valid values:* | *True, False.* |
| *Function:* | If **Quiet** is set to *True* then any status and error message generated by **WDIPterm** will be suppressed. |

### LogFile

| | |
|---|---|
| *Valid values:* | *blank* or a valid *file name.* |
| *Function:* | If the *file* does not exist it will be created. If it exists, it will be appended to. If **LogFile** is blank, no log file is maintained. |

### LogFileSize

| | |
|---|---|
| *Valid values:* | *1000 - 10000000* |
| *Function:* | Sets maximum log file size. When file size is exceeded, oldest entries are removed first. |

### ClientList

| | |
|---|---|
| *Valid Values:* | Read Only. |
| *Function:* | Returns a formatted string listing all attached client **BaseNames** and associated IP numbers.<br>Format is "**basename CR/LF ip address CR/LF basename...**". |

## Methods

*Methods* are commands that you issue to the **WDIPterm** control. All of the "**Inputxxx**" commands cause the terminal to wait for operator input.

*Note that your development environment may show more available methods for the* **WDterm** *control than are listed here. This is normal. You may ignore methods you see that are not listed here.*

### InputAny

| | |
|---|---|
| *Parameters:* | *basename, channel, terminal, line, position, prompt, shifted, timestamped* |
| *Function:* | This instructs the *terminal* attached to client *basename* on *channel* to display the *prompt* at *line* and *position* and wait for data to be entered from either terminal keypad or scanner. If *shifted* is set to "**true**", the terminal will start in shifted mode. *Timestamped* appends a (hhmmss) prefix to the returned data. |

### InputKeyBd

| | |
|---|---|
| *Parameters:* | *basename, channel, terminal, line, position, prompt, shifted, timestamped* |
| *Function:* | This instructs the *terminal* attached to client *basename* on *channel* to display the *prompt* at *line* and *position* and wait for data to be entered from the terminal keypad only. If *shifted* is set to "**true**", the terminal will start in shifted mode. *Timestamped* appends a (hhmmss) prefix to the returned data.. |

### InputScanner

| | |
|---|---|
| *Parameters:* | *basename, channel, terminal, line, position, prompt, allowbreakout, timestamped* |
| *Function:* | This instructs the *terminal* attached to client *basename* on *channel* to display the *prompt* at *line* and *position* and wait for data to be entered from the terminal scanner only. Setting allow breakout to true allow user to "breakout" of scanner only mode by pressing the end key on the terminal. A **termID+CR** will be sent to the host. |

### InputYesNo

| | |
|---|---|
| *Parameters:* | *basename, channel, terminal, line, position, prompt* |
| *Function:* | This instructs the *terminal* attached to client *basename* on *channel* to display the *prompt* at *line* and *position* and wait for a **Yes** (Enter key or C key) or a **No** (0 key or B key) from the terminal keypad. |

*Note: C and B keys are used to facilitate keypad entry while scanning with the integrated laser.*

**InputPassword**

*Parameters:* *basename, channel, terminal, line, position, prompt, shifted*

*Function:* This instructs the *terminal* attached to client *basename* on *channel* to display the *prompt* at *line* and *position* and wait for data to be entered from the terminal keypad only. The entered data is not displayed on the terminal.

**InputSerial**

*Parameters:* *basename, channel, terminal, line, position, prompt*

*Function:* This instructs the *terminal* attached to client *basename* on *channel* to display the *prompt* at *line* and *position* and wait for data to be received through the terminal serial port. Waiting for serial input can be bypassed by pressing the **enter key** on the terminal which will send an empty data string to the host (fires the **OnTermData** event handler).

**OutputSerial**

*Parameters:* *basename, channel, terminal, data*

*Function:* This instructs the *terminal* attached to client *basename* on *channel* to send *data* to the terminal's serial port. Data must be less than 232 characters in length for each call to **OutputSerial**. If you are sending data to a printer attached to the terminal, make sure to set the **Protocol** parameter in the RF Terminal to XON/XOFF. See the RF Terminal Manual for details.

**SendDisplay**

*Parameters:* *basename, channel, terminal, line, position, prompt*

*Function:* This instructs the *terminal* attached to client *basename* on *channel* to display the *prompt* at *line* and *position*. Must be followed by an "**Input**" ***method*** call to take effect.

**ClearScreen**

*Parameters:* *basename, channel, terminal*

*Function:* This instructs the *terminal* attached to *client basename* on *channel* to clear its display. Must be followed by an "**Input**" ***method*** call to take effect.

**ClearLine**

*Parameters:* *basename, channel, terminal, line*

*Function:* This instructs the *terminal* attached to client *basename* on *channel* to clear the specified *line* on its display. Must be followed by an "**Input**" ***method*** call to take effect.

**SendDate**

*Parameters:* *basename, channel, terminal, line*

*Function:* This instructs the *terminal* attached to client *basename* on *channel* to display date and time on the specified *line* number. Must be followed by an "**Input**" ***method*** call to take effect.

## Beep

| | |
|---|---|
| *Parameters:* | *basename, channel, terminal, count* |
| *Function:* | This instructs the *terminal* attached to client *basename* on *channel* to beep *count* times. *Count* may be a value from 1 to 9. Must be followed by an "**Input**" method call to take effect. |

## PlayVoice

| | |
|---|---|
| *Parameters:* | *basename, channel, terminal, msgnum* |
| *Function:* | This instructs the *terminal* attached to client *basename* on *channel* to play voice message number *msgnum*. *Msgnum* may be a value from 1 to 99. Must be followed by an "**Input**" ***method*** call to take effect. |

## ReInit

| | |
|---|---|
| *Parameters:* | *basename, channel, terminal* |
| *Function:* | This instructs the *terminal* attached to client *basename* on *channel* to re-initialize. Must be followed by an "**Input**" ***method*** call to take effect. <br><br> *NOTE: Base Stations use the message "**Buffer Reinitialized...**" to indicate a single terminal re-initialization.* |

## ReInitAll

| | |
|---|---|
| *Parameters:* | *basename, channel* |
| *Function:* | This instructs all *terminals* attached to client *basename* on *channel* to re-initialize. |

## TestClient

| | |
|---|---|
| *Parameters:* | none |
| *Function:* | This instructs the Server to "ping" all attached clients. Results are recorded in the log. |

## PageClient

| | |
|---|---|
| *Parameters:* | *basename, message, beeps* |
| *Function:* | Sends a *message* and/or *beeps* to the client computer *basename*. |

## Events

**WDIPterm** ***events*** occur when a specific condition is met. When an ***event*** is "fired", an *event handler* function in your application is called.

Though the details of exactly how it is done varies from one programming environment to the next, the source code skeletons for the various *event handlers* are automatically generated and inserted into your source code for you. See the samples for more specific information.

Each ***event*** passes relevant information to your *event handler* function. **OnTermData** passes the data that was keyed or scanned into the terminal.

**Terminal ID** is always passed as 0-63. A terminal ID value of 99 is used as a placeholder for logging purposes.

Once you have the *event handler* skeletons, you can proceed to add whatever functionality you desire to each ***event***.

You must set the **ServerOn** property to **true** before any ***events*** can be fired.

### OnTermBaseRegister

| | |
|---|---|
| *Data passed:* | *basename, channel* |
| *Event:* | An attached base station on client *basename* has successfully powered up on *channel* and communicated with the host computer via the serial connection. |

### OnTermSignIn6

| | |
|---|---|
| *Data passed:* | *basename, channel, terminal* |
| *Event:* | A six-line *terminal* has signed in on *channel* at client *basename*. Terminal ID is passed in *terminal*. |

### OnTermSignIn4

| | |
|---|---|
| *Data passed:* | *basename, channel, terminal* |
| *Event:* | A four-line *terminal* has signed in on *channel* at client *basename*. Terminal ID is passed in *terminal*. |

### OnTermSignOut

| | |
|---|---|
| *Data passed:* | *basename, channel, terminal* |
| *Event:* | A *terminal* has signed out on *channel* at client *basename*. Terminal ID is passed in *terminal*. |

### OnTermData

| | |
|---|---|
| *Data passed:* | *basename, channel, terminal, data* |
| *Event:* | A *terminal* on *channel* at client *basename* has sent *data* in response to an **Input** ***method*** call. |

**OnTermNotSignedIn**

| | |
|---|---|
| *Data passed:* | *basename, channel, terminal* |
| *Event:* | A command has been sent to a terminal that is not signed in. |

**OnTermSequenceError**

| | |
|---|---|
| *Data passed:* | *basename, channel, terminal* |
| *Event:* | The one-for-one host prompt/terminal response protocol has been violated. The host cannot send a second **Input** command until it has received a response from the first **Input** command. If a base station receives 5 sequence errors in a row, a Host Logic error is generated and the base shuts itself down. |

*While PromptNET/ActiveX will intercept and prevent most logic errors, they are still possible so you should implement this event handler!*

**OnTermIllegalCommand**

| | |
|---|---|
| *Data passed:* | *basename, channel, terminal* |
| *Event:* | An illegal command has been sent to a terminal. |

*PromptNET/ActiveX is designed to prevent illegal commands but software is not always perfect and we may not have imagined all the ways in which our customers will want to use it!*

**OnTermUpArrow**

| | |
|---|---|
| *Data passed:* | *basename, channel, terminal* |
| *Event:* | The up-arrow button has been pressed on a *terminal*. You must issue another **Input** method call before **WDIPterm** can respond to another keypress on this terminal. |

**OnTermDownArrow**

| | |
|---|---|
| *Data passed:* | *basename, channel, terminal* |
| *Event:* | The down-arrow button has been pressed on a *terminal*. You must issue another **Input** method call before **WDIPterm** can respond to another keypress on this *terminal*. |

**OnTermLeftArrow**

| | |
|---|---|
| *Data passed:* | *basename, channel, terminal* |
| *Event:* | The left-arrow button has been pressed on a *terminal*. You must issue another **Input** method call before **WDIPterm** can respond to another keypress on this *terminal*. |

**OnTermRightArrow**

| | |
|---|---|
| *Data passed:* | *basename, channel, terminal* |
| *Event:* | The right-arrow button has been pressed on a *terminal*. You must issue another **Input** method call before **WDIPterm** can respond to another keypress on this *terminal*. |

## OnTermBeginKey

| | |
|---|---|
| *Data passed:* | *basename, channel, terminal* |
| *Event:* | The BEGIN button has been pressed on a *terminal*. You must issue another **Input** method call before **WDIPterm** can respond to another keypress on this *terminal*. |

## OnTermEndKey

| | |
|---|---|
| *Data passed:* | *basename, channel, terminal* |
| *Event:* | The END button has been pressed on a *terminal*. You must issue another **Input** method call before **WDIPterm** can respond to another keypress on this *terminal*. |

## OnTermSearchKey

| | |
|---|---|
| *Data passed:* | *basename, channel, terminal* |
| *Event:* | The SEARCH button has been pressed on a *terminal*. You must issue another **Input** method call before **WDIPterm** can respond to another keypress on this *terminal*. |