GCP++ Visual Basic Custom Controls version 2.4.2

Developer's Guide to GCP++ TCP/IP Custom Controls

General

Introduction to the GCP++ family of TCP/IP controls
Overview of TCP/IP control operation
Installation of the software
Operating Environment
Glossary of networking terms used in this document

Custom Controls

VT220, UDP, TCP, TELNET, TFTP

[$#]Introduction

Dart Communications is pleased to introduce the GCP++ family of TCP/IP Custom Controls. Written by developers *for* developers, GCP++ is a toolkit of network middle-ware products that greatly simplifies the creation of robust IP-aware applications. This help file documents the complete line of GCP++ custom controls.

All custom controls utilize the GCP.EXE application as run-time support. GCP.EXE binds to the Windows Sockets library provided by your TCP/IP vendor, and encapsulates the following functionality:

1. Service initiation. To make a TCP connection, for example, a socket must be created, the destination address must be asynchronously resolved, the addess must be formatted, and the connection requested. The asynchronous notification of the connection must be caught, and error recover must be handled at each step. The level of complexity for creating daemons and UDP sockets is similar.

2. Buffering. The underlying protocol buffers may be full, making buffering a requirement for all embedded applications. GCP makes copies of all outbound buffers, providing them to the protocol process as they can be accepted.

3. File transfer. The TFTP control puts and gets files, eliminating any programming requirements for client applications.

4. Protocol functionality. The TELNET and TFTP controls implement those protocols so the programmer does not have to learn, code and test them.

5. Error recovery and cleanup. GCP++ defines a set of high-level error codes, recovering from and resolving low-level error whenever possible. Servers are closed cleanly when an unrecoverable error occurs.

GCP utilizes a Windows Sockets Interface (v. 1.1) to communicate with any Windows Sockets Compliant TCP/IP stack. Plans for GCP++ include porting it to the WIN32 environment.

Dart Communications also offers contract programming, consulting and support to customers as needed. If we don't have what you need, we will make it!

[$] Introduction

[#] idxIntroduction

[$#]Overview

Designed for ease of use, GCP++ Custom Controls are tolerant of and checks for programmer errors whenever possible. The steps a VB programmer accomplishes to use the GCP controls are as follows:

Step 1: Open the Comm Channel or Daemon

Each control has a few properties that must be set to establish the parameters for the open call. Once set, each control has an "OpenComm" property that may be set to TRUE to open the control. The VB application is informed that the control is operating via the "OnOpenComm()" event.

Daemons may be similarly opened by setting the "OpenDaemon" property to TRUE and waiting for the "OnOpenDaemon()" event to be fired.

Step 2: Making something happen

Each control is utilized by setting a property that accomplishes the desired action. For a TCP control, for example, buffers are transmitted by setting the "Output" property to the string that is to be transmitted over the network.

Step 3: Close the Comm Channel or Daemon

The "OpenComm" and "OpenDaemon" properties are set to FALSE when the application wished to close the connection. The "OnCloseComm()" and "OnCloseDaemon()" events are executed upon the completion of the operation.

[$] Overview

[#] idxOverview

[$#]Installation Instructions

The GCP++ Custom Controls distribution diskette provides all the software required to incorporate GCP++ Custom Controls into your VB applications. The following files are included:

GCPxx.EXE This is a standard Windows application that is installed into your target directory during setup. This file must be included as run-time support for any developed application software. Any call to a custom control will automatically load GCPxx.EXE as a hidden application. When the last control is closed, GCPxx.EXE is automatically unloaded. You must copy and distribute this file with all developed applications.

***.VBX** Visual Basic Custom Controls are provided as *.VBX files. These files are installed into your target directory.

VT220.EXE, TLNT.EXE, UDP.EXE, TCP.EXE, and TFTP.EXE are compiled VB forms that utilize these custom controls. VB version 3.0 source code for these applications are included.

GCP.GLB This is a Visual Basic file that defines the constants used by the GCP++ Custom Controls.

[$] Installation

[#] idxInstallation

[$#] Operating Environment and Requirements

Microsoft Windows 3.1 or Windows NT 3.1

Any Windows Sockets compliant version 1.1 TCP/IP stack. Please contact Dart Communications for the most current list of tested products. Vendors passing GCP++ interoperable testing as of 1 Jan 93 are:

Frontier Technologies SuperTCP for Windows
Distinct TCP/IP
NetManage NEWT for Windows
LAN Workplace for DOS (with included WINSOCK emulator)
SPRY's AIR for Windows (with included WINSOCK emulator)
Microsoft WFW TCP/IP
Lanera TCPOpen
FTP Software
Wollongong
80386/486/586 processor.

A 32-bit Windows NT version is planned for development.

[$] Operating Environment and Requirements

[#] idxOperatingEnvironment

[$#] Glossary

Client In the client/server relationship, the server provides a service to the client. For example, the encapsulated TELNET control provides client functionality, as it communicates with a matching UNIX host that provides the server portion.

GCP The GCP Server, responsible for interfacing to the network on behalf GCP++ client applications and custom controls.

GCP++ Dart Communication's family of TCP/IP products for Windows and Windows NT.

Peer When peers are connected, there is no distinction between a server side and a client side because they each encapsulate similar functionality and communicate as equals.

Server This term sometime refers to the server side of the client/agent relationship, but may also refer to a generic communications server.

Socket The socket is the abstaction used to describe the source and sink for data (much like a file handle).

TCP Transport Communications Protocol. Provides unformatted stream communications between two processes on a wide-area network.

TELNET This protocol builds upon TCP, defining a protocol for communications between two Network Virtual Terminals.

TFTP Trivial File Transport Protocol. A simple mechanism for sending files between workstations on a wide-area network.

UDP User Datagram Protocol. Provides record-level communications between two processes on a wide-area network.

VT-220 Emulation control for VT-220 functionality.

[$] Glossary

[#] idxGlossary

$#K VT-220 Control

Description The VT-220 Control uses TCP sockets for full duplex stream communication. All received characters are interpreted and presented on the control's screen. By setting control properties, the application can specify how the control will behave.

VT-220 offers exceptionally complete and accurate emulation of the DEC 'VT' series of terminals, including the VT220, VT102, and VT52. The graphical capabilities of the Windows environment are used to provide support for features such as 132 columns, multi-national characters, and double-size characters.

VT-220 has built-in support for the industry-standard 101-key PC keyboard, and for the DEC LK250 keyboard. Non-standard keyboards, such as those found on many laptop PC's, can also be used, with proper configuration.

Please refer to the Visual Basic VT220.MAK file that demonstrates how a form uses this control.

Keyboard Mapping VT-220 employs an application keypad mapping that is closely tied to the DEC VT keyboard:

DEC	IBM
PF1	NumLock
PF2	'/' on keypad
PF3	'*' on keypad
PF4	'-' on keypad
- (minus)	<Ctrl> + '+' on keypad
, (comma)	'+' (on keypad)
Enter	Enter (on keypad)
HoldScreen	Pause
PrintScreen	<Alt-F2>
Data/Talk	Not Supported
F6 - F10	F6 - F10
F11 - F20	Shift-F1 - F10
Help	F11
Do	F12

The numeric keypad is forced into 'NumLock' mode by Vt220. As a result, only the cursor keys on the separate inverted-T keypad will work as cursor-control keys.

The following keys are located on the separate 'Editing' keypad. Each key is mapped to its physical equivalent on the DEC keyboard (use the EditKeys property to set the DEC or IBM Mapping).

$ VT220

idxVT220

K VT220

DEC	IBM
Find	Insert
InsertHere	Home
Remove	PageUp
Select	Delete
PrevScreen	End
NextScreen	PageDown

The DEC user-defined keys (DECUDK) are accessed by pressing <Ctrl-F6> through <Ctrl-F10> for the first five, and <Ctrl-Shift-F1> through <Ctrl-Shift-F10> for the last ten.

File Name GCP_220.VBX

General Properties OpenComm (ensure RemoteAddress and RemotePort are initialized before setting to True), CommID, RemoteAddress, RemotePort, ShowServer

Auto Login Properties Username, Password, UsernamePrompt, PasswordPrompt

Presentation Properties ForeColor, BackColor, BoldColor, CharSet, ClearComm, ClearScreen, ResetTerminal, Columns, AutoWrap, LocalEcho

Control Properties LogFileName, KeyMap, TermType, Keypad, CursorKeys, Copy, Paste, SelectPrinter, PrintMode, PrintScreen, EditKeys, Emulate

Asynchronous Control Properties Protocol, CommSettings, FlowControl, Backspace

Events OnOpenComm(), OnCloseComm(), OnOutput(), OnInput(), Console()

[$#K]TCP Control

Description The TCP Control uses TCP sockets for full duplex stream communication. Characters/buffers are sent and received. The application is responsible for segmenting the stream into meaningful records.

Both client and daemon functionality is provided. When a daemon is established on a given port, clients are spawned dynamically when a remote TCP provides an inbound connection. There is no limit placed upon the number of connections a single control can support.

Please refer to the Visual Basic TCP.MAK file that demonstrates how a form uses this control.

File Name GCP_TCP.VBX

Properties OpenComm (ensure RemoteAddress and RemotePort are initialized before setting to True), CommID, OpenDaemon (ensure LocalPort is initialized before setting to True), DaemonID, Output, Instance, LocalPort, RemoteAddress, RemotePort, ShowServer

Events OnOpenComm(), OnOpenDaemon(), OnCloseComm(), OnCloseDaemon(), OnOutput(), OnInput()

[$] TCP

[#] idxTCP

[K] TCP

[$#K]UDP Control

Description The UDP Control uses UDP sockets for datagram communication. Characters/buffers may be sent and received on a per-datagram basis.

There is no limit placed upon the number of datagram ports a single control can provide. If a form creates more than one, it should closed them before exiting.

Please refer to the Visual Basic UDP.MAK file that demonstrates how a form uses this control.

File Name GCP_UDP.VBX

Properties OpenComm (ensure LocalPort is initialized before setting OpenComm), CommID, Output (ensure RemoteAddress and RemotePort are initialized before setting Output), Instance, LocalPort , RemoteAddress, RemotePort, ShowServer

Events OnOpenComm(), OnCloseComm(), OnOutput(), OnInput()

[$] UDP

[#] idxUDP

[K] UDP

[$#K]TELNET Control

Description The TELNET Control uses TCP sockets for full duplex stream communication. Characters/buffers may be sent and received. The application is responsible for segmenting the stream into meaningful records. Incoming TELNET commands are stripped out and provided to the Control as an event. Outgoing TELNET commands (including option and sub-option negotiation) can be specified using control properties.

Both client and daemon functionality is provided. When a daemon is established on a given port, TELNET agents are spawned dynamically when a remote TELNET client provides an inbound connection. There is no limit placed upon the number of connections a single control can support. If a form creates more than one, it should closed them before exiting.

Please refer to the Visual Basic TLNT.MAK file that demonstrates how a form uses this control.

File Name GCP_TLNT.VBX

Properties OpenComm (ensure RemoteAddress and RemotePort are initialized before setting to True), CommID, OpenDaemon (ensure LocalPort is initialized before setting to True), DaemonID, Output, Instance, LocalPort, RemoteAddress, RemotePort, ShowServer, Cmd, DoOption, DontOption, WillOption, WontOption, DoSubOption, SubOption

Events OnOpenComm(), OnOpenDaemon(), OnCloseComm(), OnCloseDaemon(), OnOutput(), OnInput(),OnCmd()

[$] TELNET

[#] idxTELNET

[K] TELNET

[$#K]TFTP Control

Description The TFTP Control uses UDP sockets for the tranmission of files to/from other TFTP-capable hosts.

Both client and daemon functionality is provided. When a daemon is established it responds to remote requests for file transfer. There is no limit placed upon the number of clients a single control can support.

Please refer to the Visual Basic TFTP.MAK file that demonstrates how a form uses this control.

File Name GCP_TFTP.VBX

Properties OpenComm, CommID, OpenDaemon, DaemonID, Instance, RemoteAddress, RemoteFileSpec, Mode, InputFileSpec (ensure RemoteAddress, Mode, and RemoteFileSpec are initialized before setting InputFileSpec), OutputFileSpec (ensure RemoteAddress, Mode and RemoteFileSpec are initialized before setting OutputFileSpec), ShowServer

Events OnOpenComm(), OnOpenDaemon(), OnCloseComm(), OnCloseDaemon(), OnOutputFile(), OnInputFile()

[$] TFTP

[#] idxTFTP

[K] TFTP

[$#K]OpenComm Property

Description Commands that a communication channel be opened or closed. This property is write-only at run-time and is not shown on the property list.

Usage [*form*.]*Control*.OpenComm[= { True | False }]

Remarks The OpenComm property settings are:

Setting	Description
True	Communication channel is initiated
False	Communication channel is closed

Set to True to initiate the establishment of a communication channel. All required parameters must initialized first! The OnOpenComm() event will be called when the process completes.

Set to False to close the communication channel identified by the value of CommID. The OnCloseComm() event will be called when the process completes.

Protocol for connection must be set before opening a VT220 channel.

Example
```
VT1.RemoteAddress = "dart.com"
VT1.RemotePort = 23
VT1.Protocol = 1
VT1.OpenComm = True   ' the connection process begins
```

Data Type Integer (Boolean)

[$] OpenComm

[#] idxOpenComm

[K] OpenComm

[$#K]Copy, Paste Properties

Description When set to true, copies or pastes between the highlighted area of the screen and the clipboard.

Usage [*form*.]*Control*.Copy[= True]

Data Type Integer (Boolean)

[$] Copy

[#] idxCopy

[K] Copy

[$#K]SelectPrinter Property

Description When set to true, a dialog box is displayed to set the printer

Usage [*form*.]*Control*.SelectPrinter[= True]

Data Type Integer (Boolean)

[$] SelectPrinter

[#] idxSelectPrinter

[K] SelectPrinter

[$#K]AutoWrap Property

Description When the VT220 control is narrower than 80 or 132 columns, the value of AutoWrap determines if the additional characters will wrap to the next line (or be clipped by the right side of the window).

Usage [*form*.]*Control*.AutoWrap[= { True | False }]

Remarks The AutoWrap property settings are:

Setting	Description
True	To wrap all characters to the next line.
False	Clip characters on right side of control.

Example VT1.AutoWrap = True"

Data Type Integer (Boolean)

[$] AutoWrap

[#] idxAutoWrap

[K] AutoWrap

[$#K]LocalEcho Property

Description Echos typed characters to the screen. Mostly useful for debugging.

Usage [*form*.]*Control*.LocalEcho[= { True | False }]

Remarks The Local property settings are:

Setting	Description
True	To send all local keystrokes to screen (as well as to the host)
False	Default. No local keystrokes to screen.

Example VT1.LocalEcho = True

Data Type Integer (Boolean)

[$] LocalEcho

[#] idxLocalEcho

[K] LocalEcho

$#K OpenDaemon Property

Description Commands that a daemon be opened or closed. A daemon will accept incoming protocol requests. TCP and TELNET daemons accept incoming connections, and will create a new CommID to handle each new connection. The TFTP daemon handles remote TFTP requests without creating a new CommID. This property is write-only at run-time.

Usage [form.]Control.OpenDaemon[= { True | False }]

Remarks The OpenComm property settings are:

Setting	Description
True	Daemon is opened
False	Daemon is closed

Set to True to initiate the establishment of a daemon. Ensure all the required parameters are initialized first. The OnOpenDaemon() event will be called when the process completes.

Set to False to close the daemon identified by the value of DaemonID. The OnCloseDaemon() event will be called when the process completes.

Example
```
TCP1.LocalPort = 23
TCP1.OpenDaemon = True   ' we have made a TELNET server
```

Data Type Integer (Boolean)

$ OpenDaemon

idxOpenDaemon

K OpenDaemon

$#K Output Property

Description When set, the string is sent out over the communication channel specified in CommID. This property is write-only at run-time and is not shown on the property list.

Usage [*form*.]*Control*.Output[= *outString$*]

Remarks Output can be of any length up to 32K bytes.

Example
```
Sub Form_KeyPress (KeyAscii As Integer)
    TCP1.Output = Chr$(KeyAscii)
End Sub
```

Data Type String

[$] Output

[#] idxOutput

[K] Output

[$][#][K]LogFileName Property

Description Specifies a filename for the control to log all input (from the host) to. May be set to the null string ("") to turn off log.

Usage [*form*.]*Control*.LogFileName[= *LogFileSpec$*]

Example VT1.LogFileName = "c:\temp\session2.log"

Data Type String

[$] LogFileName

[#] idxLogFileName

[K] LogFileName

[$#K]PrintScreen Property

Description Specifies a filename for dumping the current screen contents.

Usage [*form*.]*Control*.PrintScreen[= *ScreenFileSpec$*]

Example VT1.PrintScreen = "c:\temp\screen.25"

Data Type String

[$] PrintScreen

[#] idxPrintScreen

[K] PrintScreen

[$#K]KeyMap Property Array

Description This property allows the user to redefine practically any key on the PC keyboard to produce a special character string. <shift>, <control> and <shift-control> modifiers are also supported. The index into the array is a value between 0 and 255 that represents the Windows Virtual Key Code (refer to constant.txt of the VB Development System).

Usage [*form*.]*Control*.KeyMap(keycode)[= *MapString$*]

Remarks VT220 allows you to arbitrarily redefine almost any key on the keyboard. The exceptions are the modifier keys: <Shift>, <Ctrl>, <Alt>, and <CapsLock>.

Key-redefinition strings may be up to 80 characters in length.

Each key can have up to four separate redefinitions:
* one for when the key is pressed in its unmodified, or 'base' state
* one for when the key is pressed together with the <Shift> key
* one for when the key is pressed together with the <Ctrl> key.
* one for when the key is pressed together with both <Shift> and <Ctrl>
These strings should be entered into to the input field as one continuous string, with each of the substrings separated by the '|' (pipe) character. For example:

test|TEST|Test|tEST

is a key-redefinition string containing four parts, one each for the base state, the <Shift> state, the <Ctrl> state, and the <Ctrl+Shift> state. If you omit one or more of the four states, the default behavior for the key will be used (if one exists).

The caret ('^') character can be used to specify a control character; this notation for the first 32 control characters is supported by VT220. Use printable characters starting at SP and ending at ? as the digit after the ^.

Another, more versatile notation is also available: this is the 'C' style 'backslash' notation. For example, '\033' could be used to represent the <Escape> character. If you use this type of notation, you must supply exactly three octal digits, including leading zeros if required.

The tilde ('~') character can be used to specify a substitution key. For example, a '~62' sequence says that virtual key code 62 (decimal) will be substituted when the KeyCode is referenced.

The SS3 (\217) and CSI (\233) 8-bit control characters are special, in that they will be

---

[$] KeyMap

[#] idxKeyMap

[K] KeyMap

converted to their 7-bit counterpart (when in 7-bit control mode) for tranmission to the host.

The sample VT220.EXE application saves these definitions to disk.

Example frmMDI.ActiveForm.VT1.KeyMap(Val(VirtualKey)) = Mapping$

Data Type String

$#K OutputFileSpec Property

Description When set, the file is sent out over the communication channel specified in CommID. This property is write-only at run-time.

Usage [*form*.]*Control*.OutputFileSpec[= *outString$*]

Remarks The RemoteFileSpec property must be set before OutputFileSpec is set, so the communication channel knows where to put it on the remote system.

The RemoteAddressand Mode (NET_ASCII or OCTET) properties must also be set for TFTP controls.

Example
```
TFTP1.RemoteAddress = "Dart.com"
TFTP1.RemoteFileSpec = "d:\server\your.img"
TFTP1.Mode = OCTET
TFTP1.OutputFileSpec = "c:\output\my.img"
```

Data Type String

$ OutputFileSpec

idxOutputFileSpec

K OutputFileSpec

[$#K]InputFileSpec Property

Description When set, the file is retrieved using the communication channel specified in CommID. This property is write-only at run-time.

Usage [*form*.]*Control*.InputFileSpec[= *outString$*]

Remarks The RemoteFileSpec property must be set before InputFileSpecis set, so the communication channel knows where to get it on the remote system.

The RemoteAddress and Mode (NET_ASCII or OCTET) properties must also be set for TFTP controls.

Example
```
TFTP1.RemoteAddress = "Dart.com"
TFTP1.RemoteFileSpec = "d:\server\your.img"
TFTP1.Mode = NETASCII
TFTP1.InputFileSpec = "c:\input\my.img"
```

Data Type String

[$] InputFileSpec

[#] idxInputFileSpec

[K] InputFileSpec

[$#K]Instance Property

Description Sets a value that is saved when the Output, InputFileSpec, and OutputFileSpec properties are set. Upon the completion of the file or buffer transfer, Instance is provided as a parameter to the OnOutput(), OnOutputFile() and OnInputFile() events.

Usage [form.]Control.Instance[= objectpointer&]

Remarks The Instance property may be used for any purpose. For example, it may be used to reference a structure or C++ object.

Data Type Long

[$] Instance

[#] idxInstance

[K] Instance

[$#K]Mode Property

Description For the TFTP Control only, set the Mode to either NETASCII or OCTET before setting InputFileSpec or OutputFileSpec.

Usage [*form*.]*TFTP1*.Mode[= NETASCII | OCTET]

Remarks NETASCII and OCTET are defined in the GCP.GLB file.

Data Type Integer

[$] Mode

[#] idxMode

[K] Mode

$#K Cmd Property

Description For the TELNET Control only, instructs the control to send the specified Cmd to the host. A two-byte string is sent comprised of an IAC character (255) and the specified character. This property is write-only at run-time, and is not shown on the property list.

Usage [*form*.]*TELNET1*.Cmd[= *CmdNumber*]

Remarks Cmd must be an integer between 1 and 254. Please refer to RFC854 for a description of TELNET commands.

Data Type Integer

[$] Cmd

[#] idxCmd

[K] Cmd

$#K DoOption Property

Description For the TELNET Control only, instructs the control to send a "DO OPTION" three-character sequence. This property is write-only at run-time, and is not shown on the property list.

Usage [*form*.]*TELNET1*.DoOption[= *OptionNumber*]

Remarks The application will be notified of the resulting negotiation via the OnCmd event. Option numbers are integers between 0 and 254. Please refer to RFC854 for a description of TELNET commands and option negotiation.

Data Type Integer

$ DoOption

idxDoOption

K DoOption

[$#K]DontOption Property

Description For the TELNET Control only, instructs the control to send a "DONT OPTION" three-character sequence. This property is write-only at run-time, and is not shown on the property list.

Usage [*form*.]*TELNET1*.DontOption[= *OptionNumber*]

Remarks The application will be notified of the resulting negotiation via the OnCmd event. Option numbers are integers between 0 and 254. Please refer to RFC854 for a description of TELNET commands and option negotiation.

Data Type Integer

[$] DontOption

[#] idxDontOption

[K] DontOption

$#K WillOption Property

Description For the TELNET Control only, instructs the control to send a "WILL OPTION" three-character sequence. This property is write-only at run-time, and is not shown on the property list.

Usage [*form*.]*TELNET1*.WillOption[= *OptionNumber*]

Remarks The application will be notified of the resulting negotiation via the OnCmd event. Option numbers are integers between 0 and 254. Please refer to RFC854 for a description of TELNET commands and option negotiation.

Data Type Integer

$ WillOption

idxWillOption

K WillOption

$#K WontOption Property

Description For the TELNET Control only, instructs the control to send a "WONT OPTION" three-character sequence. This property is write-only at run-time, and is not shown on the property list.

Usage [*form*.]*TELNET1*.WontOption[= *OptionNumber*]

Remarks The application will be notified of the resulting negotiation via the OnCmd event. Option numbers are integers between 0 and 254. Please refer to RFC854 for a description of TELNET commands and option negotiation.

Data Type Integer

$ WontOption

idxWontOption

K WontOption

$#K DoSubOption Property

Description For the TELNET Control only, instructs the control to send a sub-option negotiation sequence. This property is write-only at run-time, and is not shown on the property list.

Usage [*form*.]*TELNET1*.DoSubOption[= *OptionNumber*]

Remarks The application will be notified of the resulting negotiation via the OnCmd event. Option numbers are integers between 0 and 254. Please refer to RFC854 for a description of TELNET commands and option negotiation. Please refer to RFC855 for a description of sub-option negotiation. Ensure the SubOption string is set prior to setting this property.

Data Type Integer

$ DoSubOption

idxDoSubOption

K DoSubOption

[$#K]SubOption Property

Description For the TELNET Control only, sets the sub-option negotiation sequence in preparation for the DoSubOption command statement.

Usage [*form*.]*TELNET1*.SubOption[= *SubOptionCommandSequence*]

Remarks The application will be notified of the resulting negotiation via the OnCmd event.
Please refer to RFC854 for a description of TELNET commands and option negotiation.
Please refer to RFC855 for a description of sub-option negotiation.

Data Type String

[$] SubOption

[#] idxSubOption

[K] SubOption

[$#K]RemoteAddress Property

Description A RemoteAddress is required to specify the remote end of a TCP connection or the destination for a TFTP file or UDP datagram. For incoming datagrams, this property may be read to get the datagram's origination address.

Usage [*form*.]*Control*.RemoteAddress[= *AddressString$*]

Remarks This can be a logical name (ie. "Dart.com") or an internet address in dot notation (ie. "192.55.55.55").

Use "255.255.255.255" to broadcast datagrams.

Example
```
TCP1.RemoteAddress = "Dart.com"
TCP1.RemotePort = "4106"
TCP1.OpenComm = True
```

Data Type String

[$] RemoteAddress

[#] idxRemoteAddress

[K] RemoteAddress

[$#K]RemotePort Property

Description A RemotePort is required to specify the remote end of a TCP connection or the destination for a UDP datagram. For incoming datagrams, this property may be read to get the datagrams origination port.

Usage [*form*.]*Control*.RemotePort[= *WellKnownPort%*]

Remarks The remote port number one wishes to connect to is normally specified by the protocol to be used.

Example
```
TELNET1.RemoteAddress = "Dart.com"
TELNET1.RemotePort = "23"    ' well-known port used for TELNET
TELNET1.OpenComm = True
```

Data Type Integer

[$] RemotePort

[#] idxRemotePort

[K] RemotePort

[$#K]LocalPort Property

Description A LocalPort is required to specify the local end of a TCP daemon or the source for a UDP datagram.

Usage [*form*.]*Control*.LocalPort[= *WellKnownPort%*]

Remarks The local port number one wishes to use to is normally specified by the protocol.

Example
```
TELNET1.LocalPort = "23"    ' well-known port used for TELNET
TELNET1.OpenComm = True     ' we have made a TELNET server
```

Data Type Integer

[$] LocalPort

[#] idxLocalPort

[K] LocalPort

[$#K]RemoteFileSpec Property

Description The RemoteFileSpec property establishes what remote file is to be copied (to or from) by the TFTP Control. This property is write-only at run-time.

Usage [*form*.]*Control*.RemoteFileSpec[= *outString$*]

Remarks The RemoteFileSpec property must be set before InputFileSpec or OutputFileSpec is set, so the communication channel knows where to locate the file on the remote system.

Example
```
TFTP1.RemoteAddress = "Dart.com"
TFTP1.RemoteFileSpec = "d:\server\your.img"
TFTP1.Mode = NETASCII
TFTP1.InputFileSpec = "c:\input\my.img"    ' this starts the tranfer
```

Data Type String

[$] RemoteFileSpec

[#] idxRemoteFileSpec

[K] RemoteFileSpec

[$#K]CommID Property

Description The CommID identifies a communication channel. Since it is possible to establish multiple communication channels using a single control, the application references each channel by setting the CommID before setting another property that utilizes a communication channel.

Usage [*form*.]*Control*.CommID[= *SavedID%*]

Remarks Most developers will probably use a single channel per control, which means that CommID should never need to be modified. If multiple channels are to be used, read on.

If a daemon is opened, it can accept multiple connections which will generate multiple communication channels. In this case, the application is responsible for maintaining a list of CommIDs and setting CommID to one of those values before setting any property that operates upon the channel (ie: Output=string, OutputFileSpec=string, OpenComm=FALSE).

Example

```
Sub Telnet1_OnOpenComm (ErrorCode As Integer)
    If (Not ErrorCode) Then
        CommIDs[index] = TFTP1.CommID   ' save CommID
End Sub
```

Data Type Integer

[$] CommID

[#] idxCommID

[K] CommID

[$#K]DaemonID Property

Description The DaemonID identifies a daemon object. Since it is often possible to establish multiple daemons using a single control, the application references each by setting the DeamonID before setting a property that used the daemon. This property is both read and write at run-time.

Usage [*form*.]*Control*.DaemonID[= *SavedID%*]

Remarks Most developers will probably use a single daemon, which means that DaemonID should never need to be modified.

If multiple daemons are to be used, the application is responsible for maintaining a list of active DaemonIDs and setting DaemonID to one of those values before setting OpenComm=False (to close the daemon).

Example

```
Sub Telnet1_OnOpenDaemon (ErrorCode as Integer)
    If (Not ErrorCode) Then
        DaemonIDs[index] = TFTP1.DaemonID   ' save it
End Sub
```

Data Type Integer

[$] DaemonID

[#] idxDaemonID

[K] DaemonID

$#K ShowServer Property

Description Controls the visual display of the GCP++ server. This property is write-only at run-time.

Usage [*form*.]*Control*.ShowServer[= { True | False }]

Remarks The ShowServer property settings are:

Setting	Description
True	Show the GCP++ icon or window
False	Hide the GCP++ icon or window

Normal operation is hidden.

Example
```
TFTP1.ShowServer = True     ' shows the GCP server to the user
```

Data Type Integer (Boolean)

$ ShowServer

idxShowServer

K ShowServer

[$#K]ClearComm, ClearScreen, ResetTerminal Properties

Description ClearComm aborts any print operation in progress, aborts any escape sequence processing, clears buffers, takes the terminal out of printer controller mode.

ClearScreen clears the monitor screen.

ResetTerminal resets many terminal operating features to a default setting used by most application programs.

Usage [*form*.]*Control*.ClearScreen[= { True }]

Data Type Integer (Boolean)

[$] Clear

[#] idxClear

[K] Clear

[$#K]Username, Password, UsernamePrompt, PasswordPrompt Properties

Description Used only by the VT220 control to automatically login to the host.

Usage [*form*.]*VT1*.UsernamePrompt[= *prompt$*]

Remarks These properties must all be set before VT1.OpenComm is set to True. If not used, user logs on normally.

Example

```
VT1.UsernamePrompt = "ogin:"     ' any substring of the prompt will do
VT1.PasswordPrompt = "word:"
VT1.UserName = "baldwin"
VT1.Password = "MyPassword"
VT1.RemoteAddress = "dart.com"
VT1.OpenComm = True    '  initiate the connection
```

Data Type String

[$] AutoLogin

[#] idxAutoLogin

[K] Login

[$#K]ForeColor, BackColor, BoldColor Properties

Description Used only by the VT220 control to set screen colors.

Usage [*form*.]*VT1*.ForeColor[= *color%*]

Remarks These properties may be set at any time. Possible colors are: 0 - BLACK, 1 - BLUE, 2 - GREEN, 3 - CYAN, 4 - RED, 5 - MAGENTA, 6 - BROWN, 7 - WHITE

Example
```
VT1.BackColor = 0
VT1.ForeColor = 7
VT1.BoldColor = 4
```

Data Type Integer

[$] Color

[#] idxColor

[K] Color

$#KCommSettings

Description Used only by the VT220 control to set asynchronous port parameters.

Usage [*form*.]*VT1*.CommSettings[= *commsettingsstring$*]

Remarks This property must be set before opening an asynchronous connection. Comm settings string consists of: "port:baud rate,parity,data bits,stop bits". This property should not be changed once the port is opened.

Example VT1.CommSettings = "COM1:9600,N,8,1"

Data Type String

$CommSettings

idxCommSettings

K CommSettings

$#K Protocol

Description Used only by the VT220 control to set protocol type to asynchronous or TELNET.

Usage [*form*.]*VT1*.Protocol[= *protocol%*]

Remarks This property must be set before opening a VT220 connection. Setting can either be 0 - asynchronous or 1 - TELNET. It is possible to close a VT1 session of one protocol, change the protocol and open the session using the new protocol, although this has not been extensively tested.

Example VT1.Protocol = 1

VT1.Protocol = 0

Data Type Integer

[$] Protocol

[#] idxProtocol

[K] Protocol

[$#K]Backspace

Description Used only by the VT220 control to set the function of the backspace button to backspace or delete.

Usage [*form*.]*VT1*.Backspace[= *backspace%*]

Remarks This property it used to set what action is performed when the backspace key is pressed. Possible values are 0 - DEL and 1 - BS

Example VT1.Backspace = 1

VT1.Backspace = 0

Data Type Integer

[$] Backspace

[#] idxBackspace

[K] Backspace

[$#K]FlowControl

Description Used only by the VT220 control to set the flow control options for an asynchronous session.

Usage [*form*.]*VT1*.FlowControl[= *flowcontrol%*]

Remarks This property it used to set the flow control options for an asynchronous VT220 control session. This property must be set before opening the port, ie before setting OpenComm to true. Flow control cannot be changed once a connection is established. Different values for flow control are 0 - NoFlowControl, 1 - SoftwareFlowControl or 2 - HardwareFlowControl.

Example VT1.FlowControl = 0

VT1.FlowControl = 2

Data Type Integer

[$] FlowControl

[#] idxFlowControl

[K] FlowControl

[$#K]CharSet Property

Description Used only by the VT220 control to set the character set.

Usage [*form*.]*VT1*.CharSet[= *charset%*]

Remarks These properties may be set at any time. Possible character sets are: 0 - DEC Large, 1 - DEC Small, 2 - IBM, 3 - ANSI, 4 - Terminal. The 'DEC' font is an exact emulation of the DEC Multinational Character Set, as found in the VT2xx terminals. The other three are standard Windows fixed-pitch fonts.

Example VT1.CharSet = 0

Data Type Integer

[$] CharSet

[#] idxCharSet

[K] CharSet

[$#K]TermType Property

Description Used to set both the terminal type (right after connecting during TELNET terminal type negotiation) and (if in VT-200 mode) if 7-bit or 8-bit controls will be used.

Usage [*form*.]*VT1*.TermType[= *type%*]

Remarks TermType must be set before connecting, as TELNET Terminal Type option negotiation can only happen once. If VT200 mode is negotiatied, however, the user can toggle between 7-bit and 8-bit controls (as can host applications). Possible values are:

0 - VT200 Mode, 8-Bit Controls. Sets the control to operate with a full range of capabilities in an 8-bit environment with 8-bit controls. Many applications designed for the VT100 terminal will run in this mode.

1 - VT200 Mode, 7-Bit Controls. Sets the terminal to operate with a full range of capabilities, using 8-bit graphic characers and 7-bit controls. This is the recommeded mode for most applications.

2 - VT100 Mode. Sets the terminal for use with application programs designed for a VT100 terminal and requireing strict VT100 compatibility. In general, use VT200 mode, 7-bit controls if possible.

3 - VT52 Mode. Sets the terminal for use with application programs designed for the VT52 terminal.

4 - No Emulation. Does not interpret any characters.

Example VT1.TermType = 1 ' recommended default

Data Type Integer

[$] TermType

[#] idxTermType

[K] Terminal Type

[$#K]KeyPad Property

Description Selects whether or not the keypad sends ASCII character codes or escape sequences.

Usage [*form*.]*VT1*.KeyPad[=choice%]

Setting	Description
0	Numeric keypad (default). Causes the auxiliary keypad to send ASCII character codes corresponding to the numeric characters on the keys.
1	Application keypad. Causes the auxiliary keypad to send escape sequences used by an application program.

Example VT1.KeyPad = 1

Data Type Integer

[$] KeyPad

[#] idxKeyPad

[K] KeyPad

[$#K]PrintMode Property

Description Selects the operating mode for the printer.

Usage [*form*.]*VT1*.PrintMode[=choice%]

Setting	Description
0	Normal Print Mode (default). Lets user invoke print functions from the keyboard.
1	Auto Print Mode . Prints the current line of text when the terminal receives a line feed, form feed, or vertical tab code from the host computer.
2	Controller Mode . Causes the printer port to treat the printer as a terminal, while the VT220 monitors traffic. (The host computer transfers data to the printer, without displaying the data on the monitor screen).

Example VT1.PrintMode = 1

Data Type Integer

[$] PrintMode

[#] idxPrintMode

[K] PrintMode

[$#K]EditKeys Property

Description Selects whether DEC or IBM mappings for the edit keypad (above the cursor keys on most keyboards) will be used.

Usage [*form*.]*VT1*.EditKeys[=choice%]

Setting	Description
0	DEC Mapping (default).
1	IBM Mapping.

Example VT1.EditKeys = 0

Data Type Integer

[$] EditKeys

[#] idxEditKeys

[K] EditKeys

[$#K]Emulate Property

Description Selects whether or not the control functions as an emulator or puts up characters as received.

Usage [*form*.]*VT1*.Emulate = [True | False]

Example VT1.Emulate = True

Data Type Bool (Integer)

[$] Emulate

[#] idxEmulate

[K] Emulate

[$#K]CursorKeys Property

Description Selects whether or not the cursor keys send ANSI cursor control sequences or application control functions.

Usage [*form*.]*VT1*.CursorKeys[=choice%]

Setting	Description
0	Normal cursor keys (default). Cursor keys send ANSI cursor control sequences (up, down, left, and right).
1	Application Cursor Keys. Cursor keys send application program control functions.

Example VT1.CursorKeys = 1

Data Type Integer

[$] CursorKeys

[#] idxCursorKeys

[K] CursorKeys

[$#K]Columns Property

Description Used by the VT220 control to set the width of the screen (number of columns)

Usage [*form*.]*VT1*.CharSet[= *columns%*]

Remarks This property may be set at any time. Possible values are: 0 - 80 Columns, 1 - 132 Columns

Example VT1.Columns = 1

Data Type Integer

[$] Columns

[#] idxColumns

[K] Columns

$#K OnOpenComm() Event

Description The OnOpenComm() event is generated each time the OpenComm property is set to True and whenever a daemon spawns a new communication channel.

Usage Sub Control_OnOpenComm (ErrorCode As Integer)

Remarks The ErrorCode **must** be checked. If (ErrorCode = GCP_OK), then the communication channel has opened successfully and CommID holds a valid reference. If (ErrorCode <> GCP_OK), then the communication channel failed, and ErrorCode can be used to translate the error.

If only a single channel is used, then CommID need not be changed as it will always reference the communication channel. If multiple channels are being used, the application **must** save the CommID property each time this event indicates another open channel. A unique value for CommID will be present each time this event is fired.

Example The following example shows how to handle communication errors and events. You can insert code to handle a particular error or event after its Case statement.

```
Sub Comm_OnOpenComm (ErrorCode As Integer)
    Select Case ErrorCode
        Case GCP_OK             ' OK return
        Case GCP_EUNKNOWN       ' An error of unknown origin
        Case GCP_EPARAM1,       ' parameter 1 is invalid
        Case GCP_EPARAM2,       ' parameter 2 is invalid
        Case GCP_EPARAM3,       ' parameter 3 is invalid
        Case GCP_EPARAM4,       ' parameter 4 is invalid
        Case GCP_EPARAM5,       ' parameter 5 is invalid
        Case GCP_EPARAM6,       ' parameter 6 is invalid
        Case GCP_EPARAM7,       ' parameter 7 is invalid
        Case GCP_ERDOS,         ' remote DOS error
        Case GCP_EDOS,          ' local DOS error
        Case GCP_ERNOENT,       ' remote file not found
        Case GCP_ENOENT,        ' local file not found
```

$ OnOpenComm

idxOnOpenComm

K OnOpenComm()

```
        Case GCP_ENOMEM,            ' Insufficient resources
        Case GCP_ENETWORK,          ' Unspecified network error
        Case GCP_EINVALIDSERVER,' Server handle invalid
        Case GCP_EALREADYOPEN,' Server was previously opened on specified socket
        Case GCP_ENOTOWNER,         ' Server is not owned by hWnd
        Case GCP_EBADSERVERTYPE,' Server type not supported
        Case GCP_EBADCALLBACK,' Callback window specified invalid
        Case GCP_EOF,               ' end of file reached
        Case GCP_EBADMSG,           ' Msg Not Supported by Server
        GCP_RCLOSE,                 ' Remote party has closed
        Case GCP_ENOGCP,            ' GCP++ server not present
        Case GCP_EBADADDRESS,       ' connection failed to given address
    End Select
End Sub
```

$#K OnCloseComm() Event

Description The OnCloseComm event is generated each time the OpenComm property is set to False.

Usage Sub Control_OnCloseComm (ErrorCode As Integer)

Remarks The ErrorCode should be checked. If (ErrorCode = GCP_OK), then the communication channel has closed normally. If (ErrorCode <> GCP_OK), then the communication channel failed, and ErrorCode can be used to translate the error.

This event may be generated in response to the application setting OpenComm to False, or it may reflect a remote close or a system failure.

Example The following example shows how to handle communication errors and events. You can insert code to handle a particular error or event after its Case statement.

```
Sub Comm_OnCloseComm (ErrorCode As Integer)
    Select Case ErrorCode
        Case GCP_OK             ' OK return
        Case GCP_EUNKNOWN       ' An error of unknown origin
        Case GCP_EPARAM1,       ' parameter 1 is invalid
        Case GCP_EPARAM2,       ' parameter 2 is invalid
        Case GCP_EPARAM3,       ' parameter 3 is invalid
        Case GCP_EPARAM4,       ' parameter 4 is invalid
        Case GCP_EPARAM5,       ' parameter 5 is invalid
        Case GCP_EPARAM6,       ' parameter 6 is invalid
        Case GCP_EPARAM7,       ' parameter 7 is invalid
        Case GCP_ERDOS,         ' remote DOS error
        Case GCP_EDOS,          ' local DOS error
        Case GCP_ERNOENT,       ' remote file not found
        Case GCP_ENOENT,        ' local file not found
        Case GCP_ENOMEM,        ' Insufficient resources
        Case GCP_ENETWORK,      ' Unspecified network error
```

$ OnCloseComm

idxOnCloseComm

K OnCloseComm()

```
        Case GCP_EINVALIDSERVER,' Server handle invalid
        Case GCP_EALREADYOPEN,' Server was previously opened on specified socket
        Case GCP_ENOTOWNER,      ' Server is not owned by hWnd
        Case GCP_EBADSERVERTYPE,' Server type not supported
        Case GCP_EBADCALLBACK,' Callback window specified invalid
        Case GCP_EOF,            ' end of file reached
        Case GCP_EBADMSG,        ' Msg Not Supported by Server
        GCP_RCLOSE,              ' Remote party has closed
        Case GCP_ENOGCP,         ' GCP++ server not present
        Case GCP_EBADADDRESS,    ' connection failed to given address
    End Select
End Sub
```

$#K OnOpenDaemon() Event

Description The OnOpenDaemon event is generated each time the OpenDaemon property is set to True.

Usage Sub Comm_OnOpenDaemon (ErrorCode As Integer)

Remarks The ErrorCode **must** be checked. If (ErrorCode = GCP_OK), then the communication daemon has opened successfully and DaemonID holds a valid reference. If (ErrorCode <> GCP_OK), then the communication daemon failed during creation, and ErrorCode can be used to translate the error.

If only a single single daemon is used, then DaemonID need not be changed as it will always reference the single daemon. If multiple daemons are being used, however, the application **must** save the DaemonID property each time this event indicates another daemon has been created.

Example The following example shows how to handle communication errors. You can insert code to handle a particular error or event after its Case statement.

```
Sub Comm_OnOpenDaemon (ErrorCode As Integer)
        Select Case ErrorCode
    Case GCP_OK             ' OK return
    Case GCP_EUNKNOWN       ' An error of unknown origin
    Case GCP_EPARAM1,       ' parameter 1 is invalid
    Case GCP_EPARAM2,       ' parameter 2 is invalid
    Case GCP_EPARAM3,       ' parameter 3 is invalid
    Case GCP_EPARAM4,       ' parameter 4 is invalid
    Case GCP_EPARAM5,       ' parameter 5 is invalid
    Case GCP_EPARAM6,       ' parameter 6 is invalid
    Case GCP_EPARAM7,       ' parameter 7 is invalid
    Case GCP_ERDOS,         ' remote DOS error
    Case GCP_EDOS,          ' local DOS error
    Case GCP_ERNOENT,       ' remote file not found
    Case GCP_ENOENT,        ' local file not found
```

$ OnOpenDaemon

idxOnOpenDaemon

K OnOpenDaemon()

```
        Case GCP_ENOMEM,            ' Insufficient resources
        Case GCP_ENETWORK,          ' Unspecified network error
        Case GCP_EINVALIDSERVER,' Server handle invalid
        Case GCP_EALREADYOPEN,' Server was previously opened on specified socket
        Case GCP_ENOTOWNER,         ' Server is not owned by hWnd
        Case GCP_EBADSERVERTYPE,' Server type not supported
        Case GCP_EBADCALLBACK,' Callback window specified invalid
        Case GCP_EOF,               ' end of file reached
        Case GCP_EBADMSG,           ' Msg Not Supported by Server
        GCP_RCLOSE,                 ' Remote party has closed
        Case GCP_ENOGCP,            ' GCP++ server not present
        Case GCP_EBADADDRESS,       ' connection failed to given address
    End Select
End Sub
```

$#K OnCloseDaemon() Event

Description The OnCloseDaemon event is generated each time the OpenDaemon property is set to False.

Usage Sub Control_OnCloseDaemon (ErrorCode As Integer)

Remarks The ErrorCode should be checked. If (ErrorCode = GCP_OK), then the daemon has closed normally. If (ErrorCode <> GCP_OK), then the communication channel failed, and ErrorCode can be used to translate the error.

This event may be generated in response to the application setting OpenDaemon to False, or it may reflect a system failure.

Example The following example shows how to handle communication errors and events. You can insert code to handle a particular error or event after its Case statement.

```
Sub Comm_OnOpenComm (ErrorCode As Integer)
    Select Case ErrorCode
        Case GCP_OK             ' OK return
        Case GCP_EUNKNOWN       ' An error of unknown origin
        Case GCP_EPARAM1,       ' parameter 1 is invalid
        Case GCP_EPARAM2,       ' parameter 2 is invalid
        Case GCP_EPARAM3,       ' parameter 3 is invalid
        Case GCP_EPARAM4,       ' parameter 4 is invalid
        Case GCP_EPARAM5,       ' parameter 5 is invalid
        Case GCP_EPARAM6,       ' parameter 6 is invalid
        Case GCP_EPARAM7,       ' parameter 7 is invalid
        Case GCP_ERDOS,         ' remote DOS error
        Case GCP_EDOS,          ' local DOS error
        Case GCP_ERNOENT,       ' remote file not found
        Case GCP_ENOENT,        ' local file not found
        Case GCP_ENOMEM,        ' Insufficient resources
        Case GCP_ENETWORK,      ' Unspecified network error
```

$ OnCloseDaemon

idxOnCloseDaemon

K OnCloseDaemon()

```
        Case GCP_EINVALIDSERVER,' Server handle invalid
        Case GCP_EALREADYOPEN,' Server was previously opened on specified socket
        Case GCP_ENOTOWNER,      ' Server is not owned by hWnd
        Case GCP_EBADSERVERTYPE,' Server type not supported
        Case GCP_EBADCALLBACK,' Callback window specified invalid
        Case GCP_EOF,            ' end of file reached
        Case GCP_EBADMSG,        ' Msg Not Supported by Server
        GCP_RCLOSE,              ' Remote party has closed
        Case GCP_ENOGCP,         ' GCP++ server not present
        Case GCP_EBADADDRESS,    ' connection failed to given address
    End Select
End Sub
```

[$#K]OnInput() Event

Description The OnInput() event is generated each time incoming data is ready for reading.

Usage Sub Comm_OnInput (Buffer As String, ErrorCode As Integer)

Remarks The ErrorCode **must** be checked. If (ErrorCode = GCP_OK), then the buffer is full of valid data. If (ErrorCode <> GCP_OK), then a communication failure occurred, and ErrorCode can be used to translate the error. The user may check the InstanceData property of instance data. For datagram packets, the RemoteAddress and RemotePort properties may be checked for origination information.

Example The following example shows how to handle communication errors. You can insert code to handle a particular error or event after its Case statement.

```
Sub Comm_OnInput (Buffer As String, ErrorCode As Integer)
        if (Not ErrorCode)
                Text1.Text = Buffer
End Sub
```

[$] OnInput

[#] idxOnInput

[K] OnInput()

$#K OnInputFile() Event

Description The OnInputFile() event is generated each time a file has been received locally.

Usage Sub Comm_OnInputFile (LocalFileSpec As String, Instance As Long, ErrorCode As Integer)

Remarks The ErrorCode **must** be checked. If (ErrorCode = GCP_OK), then the file was successfully received. If (ErrorCode <> GCP_OK), then the transfer failed, and ErrorCode can be used to translate the error.

If the local workstation iniated the transfer, then Instance will contain the Instance property value that was current when the transfer was initiated.

The LocalFileSpec is a complete file specification of the local file.

Example The following example shows how a file is received.

```
Sub Comm_OnInputFile (LocalFileSpec As String, Instance As Long, ErrorCode As Integer)
        if (Not ErrorCode)
            Text1.Text = LocalFileSpec
End Sub
```

$ OnInputFile

idxOnInputFile

K OnInputFile()

$#K OnOutput() Event

Description The OnOutput() event is generated each time out-going data is successfully accepted by the underlying protocol.

Usage Sub Comm_OnOutput (Instance As Long, ErrorCode As Integer)

Remarks The ErrorCode **must** be checked. If (ErrorCode = GCP_OK), then the buffer is full of valid data. If (ErrorCode <> GCP_OK), then a communication failure occurred, and ErrorCode can be used to translate the error.

Instance will contain the Instance property value that was current when the transfer was initiated.

Example The following example shows how to handle communication errors. You can insert code to handle a particular error or event after its Case statement.

```
Sub Comm_OnOutput (Buffer As String, ErrorCode As Integer)
        if (Not ErrorCode)
            Text1.Text = "Buffer successfully sent"
End Sub
```

$ OnOutput

idxOnOutput

K OnOutput()

[$#K]OnOutputFile() Event

Description The OnOutputFile() event is generated each time a file has been sent from the local workstation.

Usage Sub Comm_OnOutputFile (LocalFileSpec As String, Instance As Long, ErrorCode As Integer)

Remarks The ErrorCode **must** be checked. If (ErrorCode = GCP_OK), then the file was successfully received. If (ErrorCode <> GCP_OK), then the transfer failed, and ErrorCode can be used to translate the error.

If the local workstation iniated the transfer, then Instance will contain the Instance property value that was current when the transfer was initiated.

The LocalFileSpec is a complete file specification of the local file that was sent.

Example The following example shows how the application knows a file was sent.

```
Sub Comm_OnOutputFile (LocalFileSpec As String, Instance As Long, ErrorCode As Integer)
        if (Not ErrorCode)
            Text1.Text = "File successfully sent"
End Sub
```

[$] OnOutputFile

[#] idxOnOutputFile

[K] OnOutputFile()

[$#K]Console() Event

Description The Console() event is used to provide information to a "console" window.

Usage Sub VT1_Console (Msg As String)

Example The following example shows how the application knows a file was sent.

```
Sub VT1_Console (Msg As String)
        MsgBox Msg  ' inform user of exception event
End Sub
```

[$] Console

[#] idxConsole

[K] Console()

$#K OnCmd() Event

Description The OnCmd() event is generated each time the remote Network Virtual Terminal (NVT) sends a TELNET command sequence. Please refer to RFC854 and RFC855 for details.

Usage Sub Comm_OnCmd (Cmd As Integer, TelnetOption As Integer, SubOption as String, ErrorCode As Integer)

Remarks The ErrorCode **must** be checked. If (ErrorCode <> GCP_OK), then the receive failed, and ErrorCode can be used to translate the error.

The Cmd is typically a DO_CMD, DONT_CMD, WILL_CMD, WONT_CMD (defined in GCP.GLB)

The TelnetOption is a number from 0 to 254 that indicates a TELNET option.

The SubOption is a string that is only relevant when Cmd = SB_CMD.

Example The following example illustrates how Options are negotiated:

```
Sub TELNET1_OnCmd (Cmd As Integer, TelnetOption As Integer, SubOption As String, ErrorCode As Integer)
   Select Case Cmd
   Case SB_CMD
      ' negotiate suboption
      If TelnetOption = 24 Then
         TELNET1.SubOption = Str$(0) + "vt220"
         TELNET1.DoSubOption = 24
      End If
   Case DO_CMD
      If TelnetOption = 24 Then
         TELNET1.WontOption = 24
      Else
         TELNET1.WontOption = TelnetOption
      End If
```

$ OnCmd

idxOnCmd

K OnCmd()

```
    Case DONT_CMD
        TELNET1.WontOption = TelnetOption
    Case WILL_CMD
        If TelnetOption = 3 Then
            TELNET1.DoOption = 3
        Else
            TELNET1.DontOption = TelnetOption
        End If
    Case WONT_CMD
        TELNET1.DontOption = TelnetOption
    End Select
End Sub
```

[$#K]Error Codes

The file GCP.TXT has been included with the GCP++ Custom Controls, and provides the error constants described below:

GCP_OK OK return
GCP_EUNKNOWN An error of unknown origin has occurred
GCP_EPARAM1 parameter 1 is invalid (internal error condition)
GCP_EPARAM2 parameter 2 is invalid (internal error condition)
GCP_EPARAM3 parameter 3 is invalid (internal error condition)
GCP_EPARAM4 parameter 4 is invalid (internal error condition)
GCP_EPARAM5 parameter 5 is invalid (internal error condition)
GCP_EPARAM6 parameter 6 is invalid (internal error condition)
GCP_EPARAM7 parameter 7 is invalid (internal error condition)
GCP_ERDOS DOS error on the remote workstation
GCP_EDOS DOS error on the local workstation
GCP_ERNOENT remote file not found
GCP_ENOENT local file not found
GCP_ENOMEM Insufficient resources - object not created
GCP_ENETWORK Unspecified network error
GCP_EINVALIDSERVER Server handle invalid or could not create server
GCP_EALREADYOPEN Server was previously opened on specified socket
GCP_ENOTOWNER Server is not owned by hWnd provided
GCP_EBADSERVERTYPE Server type not supported
GCP_EBADCALLBACK Callback window specified does not exist
GCP_EOF end of file reached
GCP_EBADMSG Msg Request Not Supported by Function/Server
GCP_RCLOSE Remote party has closed connection
GCP_ENOGCP GCP++ server is not running and cannot be found
GCP_EBADADDRESS connection not possible to given address

[$] ErrorCode

[#] idxErrorCode

[K] ErrorCode

$#K RFC854

Network Working Group J. Postel
Request for Comments: 854 J. Reynolds
ISI
Obsoletes: NIC 18639 May 1983

TELNET PROTOCOL SPECIFICATION

This RFC specifies a standard for the ARPA Internet community. Hosts on the ARPA Internet are expected to adopt and implement this standard.

INTRODUCTION

The purpose of the TELNET Protocol is to provide a fairly general, bi-directional, eight-bit byte oriented communications facility. Its primary goal is to allow a standard method of interfacing terminal devices and terminal-oriented processes to each other. It is envisioned that the protocol may also be used for terminal-terminal communication ("linking") and process-process communication (distributed computation).

GENERAL CONSIDERATIONS

A TELNET connection is a Transmission Control Protocol (TCP) connection used to transmit data with interspersed TELNET control information.

The TELNET Protocol is built upon three main ideas: first, the concept of a "Network Virtual Terminal"; second, the principle of negotiated options; and third, a symmetric view of terminals and processes.

1. When a TELNET connection is first established, each end is assumed to originate and terminate at a "Network Virtual Terminal", or NVT. An NVT is an imaginary device which provides a standard, network-wide, intermediate representation of a canonical terminal. This eliminates the need for "server" and "user" hosts to keep information about the characteristics of each other's terminals and terminal handling conventions. All hosts, both user and server, map their local device characteristics and conventions so as to appear to be dealing with an NVT over the network, and each can assume a similar mapping by the other party. The NVT is intended to strike a balance between being overly restricted (not providing hosts a rich enough vocabulary for mapping into their local character sets), and being overly inclusive (penalizing users with modest terminals).

NOTE: The "user" host is the host to which the physical terminal is normally attached, and the "server" host is the host which is normally providing some service. As an alternate point of view, applicable even in terminal-to-terminal or process-to-process communications, the "user" host is the host which initiated the communication.

$ RFC854

idxRFC854

K RFC854

2. The principle of negotiated options takes cognizance of the fact that many hosts will wish to provide additional services over and above those available within an NVT, and many users will have sophisticated terminals and would like to have elegant, rather than minimal, services. Independent of, but structured within the TELNET Protocol are various "options" that will be sanctioned and may be used with the "DO, DON'T, WILL, WON'T" structure (discussed below) to allow a user and server to agree to use a more elaborate (or perhaps just different) set of conventions for their TELNET connection. Such options could include changing the character set, the echo mode, etc.

The basic strategy for setting up the use of options is to have either party (or both) initiate a request that some option take effect. The other party may then either accept or reject the request. If the request is accepted the option immediately takes effect; if it is rejected the associated aspect of the connection remains as specified for an NVT. Clearly, a party may always refuse a request to enable, and must never refuse a request to disable some option since all parties must be prepared to support the NVT.

The syntax of option negotiation has been set up so that if both parties request an option simultaneously, each will see the other's request as the positive acknowledgment of its own.

3. The symmetry of the negotiation syntax can potentially lead to nonterminating acknowledgment loops -- each party seeing the incoming commands not as acknowledgments but as new requests which must be acknowledged. To prevent such loops, the following rules prevail:

a. Parties may only request a change in option status; i.e., a party may not send out a "request" merely to announce what mode it is in.

b. If a party receives what appears to be a request to enter some mode it is already in, the request should not be acknowledged. This non-response is essential to prevent endless loops in the negotiation. It is required that a response be sent to requests for a change of mode -- even if the mode is not changed.

c. Whenever one party sends an option command to a second party, whether as a request or an acknowledgment, and use of the option will have any effect on the processing of the data being sent from the first party to the second, then the command must be inserted in the data stream at the point where it is desired that it take effect. (It should be noted that some time will elapse between the transmission of a request and the receipt of an acknowledgment, which may be negative. Thus, a host may wish to buffer data, after requesting an option, until it learns whether the request is accepted or rejected, in order to hide the "uncertainty period" from the user.)

Option requests are likely to flurry back and forth when a TELNET connection is first established, as each party attempts to get the best possible service from the other party. Beyond that, however, options can be used to dynamically modify the characteristics of the connection to suit changing local conditions. For example, the NVT, as will be explained later, uses a transmission discipline well suited to the many "line at a time" applications such as BASIC, but poorly suited to the many "character at a time" applications such as NLS. A server might elect to devote the extra processor overhead required for a "character at a time" discipline when it was suitable for the local process and would negotiate an appropriate option. However, rather than then being permanently burdened with the extra

processing overhead, it could switch (i.e., negotiate) back to NVT when the detailed control was no longer necessary.

It is possible for requests initiated by processes to stimulate a nonterminating request loop if the process responds to a rejection by merely re-requesting the option. To prevent such loops from occurring, rejected requests should not be repeated until something changes. Operationally, this can mean the process is running a different program, or the user has given another command, or whatever makes sense in the context of the given process and the given option. A good rule of thumb is that a re-request should only occur as a result of subsequent information from the other end of the connection or when demanded by local human intervention.

Option designers should not feel constrained by the somewhat limited syntax available for option negotiation. The intent of the simple syntax is to make it easy to have options -- since it is correspondingly easy to profess ignorance about them. If some particular option requires a richer negotiation structure than possible within "DO, DON'T, WILL, WON'T", the proper tack is to use "DO, DON'T, WILL, WON'T" to establish that both parties understand the option, and once this is accomplished a more exotic syntax can be used freely. For example, a party might send a request to alter (establish) line length. If it is accepted, then a different syntax can be used for actually negotiating the line length -- such a "sub-negotiation" might include fields for minimum allowable, maximum allowable and desired line lengths. The important concept is that such expanded negotiations should never begin until some prior (standard) negotiation has established that both parties are capable of parsing the expanded syntax.

In summary, WILL XXX is sent, by either party, to indicate that party's desire (offer) to begin performing option XXX, DO XXX and DON'T XXX being its positive and negative acknowledgments; similarly, DO XXX is sent to indicate a desire (request) that the other party (i.e., the recipient of the DO) begin performing option XXX, WILL XXX and WON'T XXX being the positive and negative acknowledgments. Since the NVT is what is left when no options are enabled, the DON'T and WON'T responses are guaranteed to leave the connection in a state which both ends can handle. Thus, all hosts may implement their TELNET processes to be totally unaware of options that are not supported, simply returning a rejection to (i.e., refusing) any option request that cannot be understood.

As much as possible, the TELNET protocol has been made server-user symmetrical so that it easily and naturally covers the user-user (linking) and server-server (cooperating processes) cases. It is hoped, but not absolutely required, that options will further this intent. In any case, it is explicitly acknowledged that symmetry is an operating principle rather than an ironclad rule.

A companion document, "TELNET Option Specifications," should be consulted for information about the procedure for establishing new options.

THE NETWORK VIRTUAL TERMINAL

The Network Virtual Terminal (NVT) is a bi-directional character device. The NVT has a printer and a keyboard. The printer responds to incoming data and the keyboard produces outgoing data which is sent over the TELNET connection and, if "echoes" are desired, to the NVT's printer as well. "Echoes" will not be expected to traverse the network (although options exist to enable a "remote" echoing mode of operation, no host is required to implement this option). The code

set is seven-bit USASCII in an eight-bit field, except as modified herein. Any code conversion and timing considerations are local problems and do not affect the NVT.

TRANSMISSION OF DATA

Although a TELNET connection through the network is intrinsically full duplex, the NVT is to be viewed as a half-duplex device operating in a line-buffered mode. That is, unless and until options are negotiated to the contrary, the following default conditions pertain to the transmission of data over the TELNET connection:

1) Insofar as the availability of local buffer space permits, data should be accumulated in the host where it is generated until a complete line of data is ready for transmission, or until some locally-defined explicit signal to transmit occurs. This signal could be generated either by a process or by a human user.

The motivation for this rule is the high cost, to some hosts, of processing network input interrupts, coupled with the default NVT specification that "echoes" do not traverse the network. Thus, it is reasonable to buffer some amount of data at its source. Many systems take some processing action at the end of each input line (even line printers or card punches frequently tend to work this way), so the transmission should be triggered at the end of a line. On the other hand, a user or process may sometimes find it necessary or desirable to provide data which does not terminate at the end of a line; therefore implementers are cautioned to provide methods of locally signaling that all buffered data should be transmitted immediately.

2) When a process has completed sending data to an NVT printer and has no queued input from the NVT keyboard for further processing (i.e., when a process at one end of a TELNET connection cannot proceed without input from the other end), the process must transmit the TELNET Go Ahead (GA) command.

This rule is not intended to require that the TELNET GA command be sent from a terminal at the end of each line, since server hosts do not normally require a special signal (in addition to end-of-line or other locally-defined characters) in order to commence processing. Rather, the TELNET GA is designed to help a user's local host operate a physically half duplex terminal which has a "lockable" keyboard such as the IBM 2741. A description of this type of terminal may help to explain the proper use of the GA command.

The terminal-computer connection is always under control of either the user or the computer. Neither can unilaterally seize control from the other; rather the controlling end must relinguish its control explicitly. At the terminal end, the hardware is constructed so as to relinquish control each time that a "line" is terminated (i.e., when the "New Line" key is typed by the user). When this occurs, the attached (local) computer processes the input data, decides if output should be generated, and if not returns control to the terminal. If output should be generated, control is retained by the computer until all output has been transmitted.

The difficulties of using this type of terminal through the network should be obvious. The "local" computer is no longer

able to decide whether to retain control after seeing an end-of-line signal or not; this decision can only be made by the "remote" computer which is processing the data. Therefore, the TELNET GA command provides a mechanism whereby the "remote" (server) computer can signal the "local" (user) computer that it is time to pass control to the user of the terminal. It should be transmitted at those times, and only at those times, when the user should be given control of the terminal. Note that premature transmission of the GA command may result in the blocking of output, since the user is likely to assume that the transmitting system has paused, and therefore he will fail to turn the line around manually.

The foregoing, of course, does not apply to the user-to-server direction of communication. In this direction, GAs may be sent at any time, but need not ever be sent. Also, if the TELNET connection is being used for process-to-process communication, GAs need not be sent in either direction. Finally, for terminal-to-terminal communication, GAs may be required in neither, one, or both directions. If a host plans to support terminal-to-terminal communication it is suggested that the host provide the user with a means of manually signaling that it is time for a GA to be sent over the TELNET connection; this, however, is not a requirement on the implementer of a TELNET process.

Note that the symmetry of the TELNET model requires that there is an NVT at each end of the TELNET connection, at least conceptually.

STANDARD REPRESENTATION OF CONTROL FUNCTIONS

As stated in the Introduction to this document, the primary goal of the TELNET protocol is the provision of a standard interfacing of terminal devices and terminal-oriented processes through the network. Early experiences with this type of interconnection have shown that certain functions are implemented by most servers, but that the methods of invoking these functions differ widely. For a human user who interacts with several server systems, these differences are highly frustrating. TELNET, therefore, defines a standard representation for five of these functions, as described below. These standard representations have standard, but not required, meanings (with the exception that the Interrupt Process (IP) function may be required by other protocols which use TELNET); that is, a system which does not provide the function to local users need not provide it to network users and may treat the standard representation for the function as a No-operation. On the other hand, a system which does provide the function to a local user is obliged to provide the same function to a network user who transmits the standard representation for the function.

Interrupt Process (IP)

Many systems provide a function which suspends, interrupts, aborts, or terminates the operation of a user process. This function is frequently used when a user believes his process is in an unending loop, or when an unwanted process has been inadvertently activated. IP is the standard representation for invoking this function. It should be noted by implementers that IP may be required by other protocols which use TELNET, and therefore should be implemented if these other protocols are to be supported.

Abort Output (AO)

Many systems provide a function which allows a process, which is generating output, to run to completion (or to reach the same stopping point it would reach if running to completion) but without sending the output to the user's terminal. Further, this function typically clears any output already produced but not yet actually printed (or displayed) on the user's terminal. AO is the standard representation for invoking this function. For example, some subsystem might normally accept a user's command, send a long text string to the user's terminal in response, and finally signal readiness to accept the next command by sending a "prompt" character (preceded by <CR><LF>) to the user's terminal. If the AO were received during the transmission of the text string, a reasonable implementation would be to suppress the remainder of the text string, but transmit the prompt character and the preceding <CR><LF>. (This is possibly in distinction to the action which might be taken if an IP were received; the IP might cause suppression of the text string and an exit from the subsystem.)

It should be noted, by server systems which provide this function, that there may be buffers external to the system (in the network and the user's local host) which should be cleared; the appropriate way to do this is to transmit the "Synch" signal (described below) to the user system.

Are You There (AYT)

Many systems provide a function which provides the user with some visible (e.g., printable) evidence that the system is still up and running. This function may be invoked by the user when the system is unexpectedly "silent" for a long time, because of the unanticipated (by the user) length of a computation, an unusually heavy system load, etc. AYT is the standard representation for invoking this function.

Erase Character (EC)

Many systems provide a function which deletes the last preceding undeleted character or "print position"* from the stream of data being supplied by the user. This function is typically used to edit keyboard input when typing mistakes are made. EC is the standard representation for invoking this function.

*NOTE: A "print position" may contain several characters which are the result of overstrikes, or of sequences such as <char1> BS <char2>...

Erase Line (EL)

Many systems provide a function which deletes all the data in the current "line" of input. This function is typically used to edit keyboard input. EL is the standard representation for invoking this function.

THE TELNET "SYNCH" SIGNAL

Most time-sharing systems provide mechanisms which allow a terminal user to regain control of a "runaway" process; the IP and AO functions described above are examples of these mechanisms. Such systems, when used locally, have access to all of the signals supplied by the user, whether these are normal characters or

special "out of band" signals such as those supplied by the teletype "BREAK" key or the IBM 2741 "ATTN" key. This is not necessarily true when terminals are connected to the system through the network; the network's flow control mechanisms may cause such a signal to be buffered elsewhere, for example in the user's host.
To counter this problem, the TELNET "Synch" mechanism is introduced. A Synch signal consists of a TCP Urgent notification, coupled with the TELNET command DATA MARK. The Urgent notification, which is not subject to the flow control pertaining to the TELNET connection, is used to invoke special handling of the data stream by the process which receives it. In this mode, the data stream is immediately scanned for "interesting" signals as defined below, discarding intervening data. The TELNET command DATA MARK (DM) is the synchronizing mark in the data stream which indicates that any special signal has already occurred and the recipient can return to normal processing of the data stream.

The Synch is sent via the TCP send operation with the Urgent flag set and the DM as the last (or only) data octet.

When several Synchs are sent in rapid succession, the Urgent notifications may be merged. It is not possible to count Urgents since the number received will be less than or equal the number sent. When in normal mode, a DM is a no operation; when in urgent mode, it signals the end of the urgent processing.

If TCP indicates the end of Urgent data before the DM is found, TELNET should continue the special handling of the data stream until the DM is found.

If TCP indicates more Urgent data after the DM is found, it can only be because of a subsequent Synch. TELNET should continue the special handling of the data stream until another DM is found.

"Interesting" signals are defined to be: the TELNET standard representations of IP, AO, and AYT (but not EC or EL); the local analogs of these standard representations (if any); all other TELNET commands; other site-defined signals which can be acted on without delaying the scan of the data stream.

Since one effect of the SYNCH mechanism is the discarding of essentially all characters (except TELNET commands) between the sender of the Synch and its recipient, this mechanism is specified as the standard way to clear the data path when that is desired. For example, if a user at a terminal causes an AO to be transmitted, the server which receives the AO (if it provides that function at all) should return a Synch to the user.

Finally, just as the TCP Urgent notification is needed at the TELNET level as an out-of-band signal, so other protocols which make use of TELNET may require a TELNET command which can be viewed as an out-of-band signal at a different level.
By convention the sequence [IP, Synch] is to be used as such a signal. For example, suppose that some other protocol, which uses TELNET, defines the character string STOP analogously to the TELNET command AO. Imagine that a user of this protocol wishes a server to process the STOP string, but the connection is blocked because the server is processing other commands. The user should instruct his system to:

1. Send the TELNET IP character;

2. Send the TELNET SYNC sequence, that is:

Send the Data Mark (DM) as the only character in a TCP urgent mode send operation.

3. Send the character string STOP; and

4. Send the other protocol's analog of the TELNET DM, if any.

The user (or process acting on his behalf) must transmit the TELNET SYNCH sequence of step 2 above to ensure that the TELNET IP gets through to the server's TELNET interpreter.

The Urgent should wake up the TELNET process; the IP should wake up the next higher level process.

THE NVT PRINTER AND KEYBOARD

The NVT printer has an unspecified carriage width and page length and can produce representations of all 95 USASCII graphics (codes 32 through 126). Of the 33 USASCII control codes (0 through 31 and 127), and the 128 uncovered codes (128 through 255), the following have specified meaning to the NVT printer:

NAME	CODE	MEANING
NULL (NUL)	0	No Operation
Line Feed (LF)	10	Moves the printer to the next print line, keeping the same horizontal position.
Carriage Return (CR)	13	Moves the printer to the left margin of the current line.

In addition, the following codes shall have defined, but not required, effects on the NVT printer. Neither end of a TELNET connection may assume that the other party will take, or will have taken, any particular action upon receipt or transmission of these:

BELL (BEL)	7	Produces an audible or visible signal (which does NOT move the print head).
Back Space (BS)	8	Moves the print head one character position towards the left margin.
Horizontal Tab (HT)	9	Moves the printer to the next horizontal tab stop. It remains unspecified how either party determines or establishes where such tab stops are located.
Vertical Tab (VT)	11	Moves the printer to the next vertical tab stop. It remains unspecified how either party determines or establishes where such tab stops are located.
Form Feed (FF)	12	Moves the printer to the top of the next page, keeping the same horizontal position.

All remaining codes do not cause the NVT printer to take any action.

The sequence "CR LF", as defined, will cause the NVT to be positioned at the left margin of the next print line (as would, for example, the sequence "LF CR"). However, many systems and terminals do not treat CR and LF independently, and will have to go to some effort to simulate their effect. (For example, some terminals do not have a CR independent of the LF, but on such terminals it may be possible to simulate a CR by backspacing.) Therefore, the sequence "CR LF" must be treated as a single "new line" character and used whenever their combined action is intended; the sequence "CR NUL" must be used where a carriage return alone is actually desired; and the CR character must be avoided in other contexts. This rule gives assurance to systems which must decide whether to perform a "new line" function or a multiple-backspace that the TELNET stream contains a character following a CR that will allow a rational decision.

Note that "CR LF" or "CR NUL" is required in both directions (in the default ASCII mode), to preserve the symmetry of the NVT model. Even though it may be known in some situations (e.g., with remote echo and suppress go ahead options in effect) that characters are not being sent to an actual printer, nonetheless, for the sake of consistency, the protocol requires that a NUL be inserted following a CR not followed by a LF in the data stream. The converse of this is that a NUL received in the data stream after a CR (in the absence of options negotiations which explicitly specify otherwise) should be stripped out prior to applying the NVT to local character set mapping.

The NVT keyboard has keys, or key combinations, or key sequences, for generating all 128 USASCII codes. Note that although many have no effect on the NVT printer, the NVT keyboard is capable of generating them.

In addition to these codes, the NVT keyboard shall be capable of generating the following additional codes which, except as noted, have defined, but not reguired, meanings. The actual code assignments for these "characters" are in the TELNET Command section, because they are viewed as being, in some sense, generic and should be available even when the data stream is interpreted as being some other character set.

Synch

This key allows the user to clear his data path to the other party. The activation of this key causes a DM (see command section) to be sent in the data stream and a TCP Urgent notification is associated with it. The pair DM-Urgent is to have required meaning as defined previously.

Break (BRK)

This code is provided because it is a signal outside the USASCII set which is currently given local meaning within many systems. It is intended to indicate that the Break Key or the Attention Key was hit. Note, however, that this is intended to provide a 129th code for systems which require it, not as a synonym for the IP standard representation.

Interrupt Process (IP)

Suspend, interrupt, abort or terminate the process to which the NVT is connected. Also, part of the out-of-band signal for other protocols which use TELNET.

Abort Output (AO)

Allow the current process to (appear to) run to completion, but do not send its output to the user. Also, send a Synch to the user.

Are You There (AYT)

Send back to the NVT some visible (i.e., printable) evidence that the AYT was received.

Erase Character (EC)

The recipient should delete the last preceding undeleted character or "print position" from the data stream.

Erase Line (EL)

The recipient should delete characters from the data stream back to, but not including, the last "CR LF" sequence sent over the TELNET connection.

The spirit of these "extra" keys, and also the printer format effectors, is that they should represent a natural extension of the mapping that already must be done from "NVT" into "local". Just as the NVT data byte 68 (104 octal) should be mapped into whatever the local code for "uppercase D" is, so the EC character should be mapped into whatever the local "Erase Character" function is. Further, just as the mapping for 124 (174 octal) is somewhat arbitrary in an environment that has no "vertical bar" character, the EL character may have a somewhat arbitrary mapping (or none at all) if there is no local "Erase Line" facility. Similarly for format effectors: if the terminal actually does have a "Vertical Tab", then the mapping for VT is obvious, and only when the terminal does not have a vertical tab should the effect of VT be unpredictable.

TELNET COMMAND STRUCTURE

All TELNET commands consist of at least a two byte sequence: the "Interpret as Command" (IAC) escape character followed by the code for the command. The commands dealing with option negotiation are three byte sequences, the third byte being the code for the option referenced. This format was chosen so that as more comprehensive use of the "data space" is made -- by negotiations from the basic NVT, of course -- collisions of data bytes with reserved command values will be minimized, all such collisions requiring the inconvenience, and inefficiency, of "escaping" the data bytes into the stream. With the current set-up, only the IAC need be doubled to be sent as data, and the other 255 codes may be passed transparently.

The following are the defined TELNET commands. Note that these codes and code sequences have the indicated meaning only when immediately preceded by an IAC.

NAME	CODE	MEANING
SE	240	End of subnegotiation parameters.
NOP	241	No operation.
Data Mark	242	The data stream portion of a Synch. This should always be accompanied by a TCP Urgent notification.
Break	243	NVT character BRK.

Interrupt Process	244	The function IP.
Abort output	245	The function AO.
Are You There	246	The function AYT.
Erase character	247	The function EC.
Erase Line	248	The function EL.
Go ahead	249	The GA signal.
SB	250	Indicates that what follows is subnegotiation of the indicated option.
WILL (option code)	251	Indicates the desire to begin performing, or confirmation that you are now performing, the indicated option.
WON'T (option code)	252	Indicates the refusal to perform, or continue performing, the indicated option.
DO (option code)	253	Indicates the request that the other party perform, or confirmation that you are expecting the other party to perform, the indicated option.
DON'T (option code)	254	Indicates the demand that the other party stop performing, or confirmation that you are no longer expecting the other party to perform, the indicated option.
IAC	255	Data Byte 255.

CONNECTION ESTABLISHMENT

The TELNET TCP connection is established between the user's port U and the server's port L. The server listens on its well known port L for such connections. Since a TCP connection is full duplex and identified by the pair of ports, the server can engage in many simultaneous connections involving its port L and different user ports U.

Port Assignment

When used for remote user access to service hosts (i.e., remote terminal access) this protocol is assigned server port 23 (27 octal). That is L=23.

$#K RFC855

Network Working Group J. Postel
Request for Comments: 855 J. Reynolds
ISI
Obsoletes: NIC 18640 May 1983

TELNET OPTION SPECIFICATIONS

This RFC specifies a standard for the ARPA Internet community. Hosts on the ARPA Internet are expected to adopt and implement this standard.

The intent of providing for options in the TELNET Protocol is to permit hosts to obtain more elegant solutions to the problems of communication between dissimilar devices than is possible within the framework provided by the Network Virtual Terminal (NVT). It should be possible for hosts to invent, test, or discard options at will. Nevertheless, it is envisioned that options which prove to be generally useful will eventually be supported by many hosts; therefore it is desirable that options should be carefully documented and well publicized. In addition, it is necessary to insure that a single option code is not used for several different options.

This document specifies a method of option code assignment and standards for documentation of options. The individual responsible for assignment of option codes may waive the requirement for complete documentation for some cases of experimentation, but in general documentation will be required prior to code assignment. Options will be publicized by publishing their documentation as RFCs; inventors of options may, of course, publicize them in other ways as well.

Option codes will be assigned by:

Jonathan B. Postel
University of Southern California
Information Sciences Institute (USC-ISI)
4676 Admiralty Way
Marina Del Rey, California 90291
(213) 822-1511

Mailbox = POSTEL@USC-ISIF

Documentation of options should contain at least the following sections:

Section 1 - Command Name and Option Code

Section 2 - Command Meanings

The meaning of each possible TELNET command relevant to this option should be described. Note that for complex options, where

"subnegotiation" is required, there may be a larger number of possible commands. The concept of "subnegotiation" is described

---

$ RFC855

idxRFC855

K RFC855

in more detail below.

Section 3 - Default Specification

The default assumptions for hosts which do not implement, or use, the option must be described.

Section 4 - Motivation

A detailed explanation of the motivation for inventing a particular option, or for choosing a particular form for the option, is extremely helpful to those who are not faced (or don't realize that they are faced) by the problem that the option is designed to solve.

Section 5 - Description (or Implementation Rules)

Merely defining the command meanings and providing a statement of motivation are not always sufficient to insure that two implementations of an option will be able to communicate. Therefore, a more complete description should be furnished in most cases. This description might take the form of text, a sample implementation, hints to implementers, etc.

A Note on "Subnegotiation"

Some options will require more information to be passed between hosts than a single option code. For example, any option which requires a parameter is such a case. The strategy to be used consists of two steps: first, both parties agree to "discuss" the parameter(s) and, second, the "discussion" takes place.

The first step, agreeing to discuss the parameters, takes place in the normal manner; one party proposes use of the option by sending a DO (or WILL) followed by the option code, and the other party accepts by returning a WILL (or DO) followed by the option code. Once both parties have agreed to use the option, subnegotiation takes place by using the command SB, followed by the option code, followed by the parameter(s), followed by the command SE. Each party is presumed to be able to parse the parameter(s), since each has indicated that the option is supported (via the initial exchange of WILL and DO). On the other hand, the receiver may locate the end of a parameter string by searching for the SE command (i.e., the string IAC SE), even if the receiver is unable to parse the parameters. Of course, either party may refuse to pursue further subnegotiation at any time by sending a WON'T or DON'T to the other party.

Thus, for option "ABC", which requires subnegotiation, the formats of the TELNET commands are:

IAC WILL ABC

Offer to use option ABC (or favorable acknowledgment of other party's request)

IAC DO ABC

Request for other party to use option ABC (or favorable acknowledgment of other party's offer)

IAC SB ABC <parameters> IAC SE

One step of subnegotiation, used by either party.

Designers of options requiring "subnegotiation" must take great care to avoid unending loops in the subnegotiation process. For example, if each party can accept any value of a parameter, and both parties suggest parameters with different values, then one is likely to have an infinite oscillation of "acknowledgments" (where each receiver believes it is only acknowledging the new proposals of the other). Finally, if parameters in an option "subnegotiation" include a byte with a value of 255, it is necessary to double this byte in accordance the general TELNET rules.