GCP++ Software Development Kit version 2.4.2

Developers Guide to GCP++

General

Introduction to Dart Communication's TCP/IP Tools (GCP++)
Overview of GCP Server operation
Installation of the software
Operating Environment
Glossary of common networking terms.
Overview of GCP Session Types describing UDP, TCP, TELNET, & TFTP sessions

API Definition

GCPopen() establishes a session
GCPdispatch() dispatches a command to a session
GCPquery() queries the GCP Server for information
GCPclose() closes a session

GCPalloc() makes a local copy of status information
GCPfree() frees the local status copy
GCPperror() converts a returned error code to an error description string

$#Introduction

Dart Communications is pleased to introduce the GCP++ family of TCP/IP Tools. Written by developers *for* developers, GCP++ is a toolkit of network middle-ware products that greatly simplifies the creation of IP-aware applications. The GCP Server Application, GCPxx.EXE, dynamically creates service sessions on behalf of client applications, encapsulating robust network and protocol processing code. These sessions encapsulate 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 other session types 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. TFTP sessions (both client and server) "put" and "get" files, eliminating any programming requirements for client applications.

4. Protocol functionality. The TELNET and TFTP sessions 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. Sessions are cleanly closed when an unrecoverable error occurs.

GCP is a Windows server application that multiple client applications may share. This is accomplished through the use of a small library (GCP.DLL) that handles all interprocess communications between the GCP Server and its clients. This product encapsulates hundreds of hours of development effort that is normally required within every TCP/IP-aware application.

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 Windows NT environment.

Dart Communications also offers contract programming, consulting and support to customers as needed.

$ Introduction

idxIntroduction

$#Overview

GCP++ encapsulates selected TCP/IP protocols, minimizing programming for multiple client applications. It does this by placing the Windows Sockets interface and all upper-layer protocol processing in the GCP++ TCP/IP Server application, and only communicating high-level information to the client application. The GCP.DLL provides a function-oriented interface that is much simpler to use than DDE, providing a mechanism for inter-process communication with the GCP server. All callbacks to the using application is accomplished via messaging to the user's window.

GCP.DLL starts GCPxx.EXE (if not running), translating function parameters into DDE-like messages that can be sent to the GCP server. In this manner each client program can take advantage of all the benefits of the client-agent architecture, without doing the necessary housework required for Window task-to-task communication.

IMPORTANT: The GCP Server can be started manually at any time, allowing customers to bind to the underlying WINSOCK implementation and read vendor-supplied information in the main GCP window. Only one instance of the GCP server can be running at any time; if hidden, starting GCPxx.EXE has the effect of making it visible.

Designed for ease of use, GCP++ is tolerant of and checks for programmer errors whenever possible. All memory is allocated under GCP++ control, so that the system is even tolerant of applications that allocate sessions and quit without closing them. The steps a programmer accomplishes to use the GCP Server are as follows:

Step 1: Open a Session

An application calls GCPopen() to create a session and establish a network resource: the parameters determine the connection type, the protocol to be used, the address of the remote process, and other information as required. A handle to a window and a callback message is also provided, so GCP can send a GCP_OPENED message to the application when the session is established. A handle to the session is provided as a reference for subsequent calls.

Step 2: Controlling the GCP Server

The application calls GCPdispatch() to task the session with a file transfer, a data transfer, or command other functionality of the server (such as hiding/showing itself).

Step 3: Querying the GCP Server

The application can call GCPquery() to get statistical and status information related to a session. GCPquery() is also used to access host addressing information from the network software.

$ Overview

idxOverview

Step 4: Close an Session

The application calls GCPclose() to free resources and release the session from further use.

$#Installation Instructions

The GCP++ Software Development Kit (SDK) distribution diskette provides all the software required to incorporate GCP into your applications. A user-friendly setup routine installs all the GCP++ files and modified the autoexec.bat to add a user-specified directory to the system path. In addition to the required development files, the SDK directory also includes all the files required to build the GCP_CLNT.EXE application**. GCP_CLNT is a complete test program, written using Visual C++, that the reader may refer to as an example client using GCP.** The SDK is comprised of the following files:

GCPxx.EXE	This is a standard Windows application that is copied into the gcp\bin directory during setup. This file must be included as run-time support for any developed application software (such as the included GCP_CLNT.EXE application). Any call to the API will automatically load GCPxx.EXE as a hidden application. When the last session is closed, GCPxx.EXE is automatically unloaded if it is hidden. If visible, the GCP server is not unloaded when the last session is closed. The GCP++ Unlimited Distribution License provides you with the right to distribute this file with all developed applications. The "xx" notation refers to the version number.
GCP.H	This is a C/C++ compatible include file that defines all the constructs that are utilized by the API (typedefs, structs, enums, defines, and unions). GCP.H is licensed for one development platform only, and may not be copied or distributed with your product.
GCP.DLL	This is the Dynamic Linked Library that provides the GCP++ Application Programming Interface (API) to client applications. Used to communicate with the GCP server, it translates all API function calls into inter-process messages and sends them to GCPxx.EXE. GCP is automatically loaded as a hidden application when one of the API functions is called. The GCP++ Unlimited Distribution License provides you with the right to distribute this file with all developed applications.
GCP.LIB	This file is created from the GCP.DLL file using the IMPLIBW.EXE utility, and can be linked into developed applications to simplify access to GCP.DLL.

$ 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 15 Mar 94 are:

er Technologies SuperTCP for Windows
t TCP/IP
nage NEWT for Windows
Vorkplace for DOS (with included WINSOCK emulator)
s AIR for Windows (with included WINSOCK emulator)
oft WFW TCP/IP
 TCPOpen
oftware
ngong
ws NT
486/586 processor.

A 32-bit Windows NT version is under 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_SESSION provides client functionality, as it communicates with a matching UNIX host that provides the server portion. Not to confuse the issue, the GCP Server operates on the same workstation as the GCP client applications that are written for it. This document uses both meanings.
GCP	The GCP Server, responsible for interfacing to the network on behalf GCP++ client applications.
GCP++	The Dart Communications family of TCP/IP products for Windows and Windows NT.
Server	This term sometime refers to the server side of the client/server relationship, but may also refer to a generic communications server. The GCP Server is a TCP/IP server, serving communications to multiple client applications.
Socket	The socket is the abstaction used to describe the source and sink for data (much like a file handle).
TCP	Transmission 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.

[$] Glossary

[#] idxGlossary

$#Session Overview

GCP provides for the concurrent operation of numerous sessions. The number of sessions is limited only by the number of sockets supported by the TCP/IP stack vendor and the memory resources of the system. GCP operation makes maximum use of idle time processing, while leaving ample time for Windows to perform its clean-up tasks. Memory allocation errors will be reported to the application if resource limitations are impacting operations.

GCP currently supports 7 communication session types:

1. The TCP_SESSION provides basic TCP connection service to a remote UNIX/PC host.

2. The TCP_DAEMON accepts TCP connection requests and spawns a TCP_SESSION to control the new connection.

3. The UDP_SESSION creates a UDP socket for datagram communication.

4. The TELNET_SESSION provides filtered TELNET connection service to a remote UNIX/PC host.

5. The TELNET_DAEMON accepts TCP connection requests and spawns a TELNET_SESSION to control the new connection.

6. The TFTP_SESSION gets and puts files from/to remote TFTP server.

7. The TFTP_DAEMON services remote gets and puts, turning your PC into a TFTP server.

$ Session Overview

idxSessionOverview

$#KUDP_SESSION

The UDP_SESSION creates a UDP socket for full duplex datagram communication. Datagrams may be sent and received to any address.

Creation:

The UDP_SESSION is created using GCPopen(UDP_SESSION, hCallbackWnd, CallbackMsg, Instance, NULL, LocalPort). A socket is created and bound to the LocalPort specified.

Sending datagrams:

The GCPdispatch(hSession, GCP_SEND_DATAGRAM, pUdpCommandParams) function is be called to send datagrams.

Handling received messages:

The hCallbackWnd must be prepared to handle the CallbackMsg when sent by GCP. The wParam should be cast to the GCP_COMMAND type. The lParam should be cast to the LPGCP_STATUS type. The following GCP_COMMAND types may be sent to the client by UDP sessions: GCP_OPENED, GCP_SEND_DATAGRAM, GCP_RECV_DATAGRAM, GCP_CLOSE.

Destruction:

Use GCPclose() to gracefully close or abort a session.

$ UDP_SESSION

idxUDP_SESSION

K UDP_SESSION

$#K TCP_SESSION

A TCP session uses a TCP socket for full duplex stream communication. Characters/buffers may be sent and received. The application is responsible for segmenting the stream into meaningful records.

Creation:

The TCP_SESSION is created using GCPopen(TCP_SESSION, hCallbackWnd, CallbackMsg, Instance, RemoteAddress, NULL). A socket is created and bound to a free local port, the RemoteAddress is resolved, and a connection is made.

Alternatively, if a TCP_DAEMON accepts a connection, it spawns a new TCP_SESSION using data received when the daemon was created. The owner (client) of the TCP_DAEMON is then notifed with a GCP_OPENED message.

Sending buffers:

The GCPdispatch(hSession, GCP_SEND, pTcpCommandParams) function is used to send buffers.

Handling received messages:

The hCallbackWnd must be prepared to handle the CallbackMsg when sent by GCP. The wParam should be cast to the GCP_COMMAND type. The lParam should be cast to the LPGCP_STATUS type. The following GCP_COMMAND types may be sent to the client by a TCP session: GCP_OPENED, GCP_SEND, GCP_RECV, GCP_CLOSE.

Destruction:

Use GCPclose() to gracefully close or abort a session.

$ TCP_SESSION

idxTCP_SESSION

K TCP_SESSION

[$#K]TCP_DAEMON

The TCP_DAEMON listens on a specified port for a remote process to connect. When a connection occurs, a socket is created and used to initialize a TCP_SESSION. The new TCP_SESSION then notifies the owner of the TCP_DAEMON that he is active.

Creation:

The TCP_DAEMON is created using GCPopen(TCP_DAEMON, hCallbackWnd, CallbackMsg, Instance, NULL, LocalPort). A socket is created and bound to the local port specified, waiting for connections to occur. A GCP_OPENED message is sent to the client to signal the creation of the daemon (the client will need the hSession value to close the daemon session later).

Handling received messages:

The hCallbackWnd must be prepared to handle the CallbackMsg when sent by the daemon. The wParam should be cast to the GCP_COMMAND type. The lParam should be cast to the LPGCP_STATUS type. The following GCP_COMMAND types may be sent by TCP daemons: GCP_OPENED, GCP_CLOSE.

Destruction:

Use GCPclose() to gracefully close or abort a session.

[$] TCP_DAEMON

[#] idxTCP_DAEMON

[K] TCP_DAEMON

$#K TELNET_SESSION

The TELNET_SESSION uses a TCP socket for full duplex stream communication. Characters/buffers may be sent and received. The application is responsible for segmenting the stream into meaningful records. TELNET option negotiation characters are stripped out and converted into GCP_TELNET messages. The client application may specify commands, options and sub-options to be negotiated on/off. Refer to RFC854 for a description of the TELNET protocol.

Creation:

The TELNET_SESSION is created using GCPopen(TELNET_SESSION, hCallbackWnd, CallbackMsg, Instance, RemoteAddress and RemotePort). A socket is created and bound to a free local port, the RemoteAddress is resolved, and a connection is made.

Alternatively, if a TELNET_DAEMON accepts a connection, it spawns a TELNET_SESSION using data received when the daemon was created. The owner of the TELNET_DAEMON is then notifed with a GCP_OPENED message.

Sending buffers:

The GCPdispatch(hSession, GCP_SEND, pCommandParams) function may be called to send buffers. They are sent exactly as provided.

Negotiating TELNET options:

The GCPdispatch(hSession, GCP_TELNET, pCommandParams) function is used to command option negotiation. An option is a number between 1 and 255 that the remote host will agree to or disagree to. The GCP_TELNET message received back will inform the client if the option was agreed to or not (by the remote host). Sub-option negotiation is supported via use of a sub-option command string. Refer to RFC855 for a complete description of TELNET option negotiation.

Handling received messages:

The hCallbackWnd must be prepared to handle the CallbackMsg when sent by the GCP server. The wParam should be cast to the GCP_COMMAND type. The lParam should be cast to the LPGCP_STATUS type. The following GCP_COMMAND types apply to a TELNET_SERVER: GCP_OPENED, GCP_SEND, GCP_RECV, GCP_TELNET, GCP_CLOSE.

$ TELNET_SESSION

idxTELNET_SESSION

K TELNET_SESSION

Destruction:

Use GCPclose() to gracefully close or abort a session.

[$#K]TELNET_DAEMON

The TELNET_DAEMON listens on a particular port for a remote process to connect. When a connection occurs, a socket is created and used to initialize a TELNET_SESSION. The new TELNET_SESSION then notifies the owner of the TELNET_DAEMON that he is active.

Creation:

The TELNET_DAEMON is created using GCPopen(TELNET_DAEMON, hCallbackWnd, CallbackMsg, Instance, NULL, LocalPort). A socket is created and bound to the local port specified, waiting for connections to occur. A GCP_OPENED message is sent to the client (the client will need the hSession value to close the daemon session later).

Handling received messages:

The hCallbackWnd must be prepared to handle the CallbackMsg when sent by the daemon. The wParam should be cast to the GCP_COMMAND type. The lParam should be cast to the LPGCP_STATUS type. The following GCP_COMMAND types apply to the TELNET_DAEMON: GCP_OPENED, GCP_CLOSE.

Destruction:

Use GCPclose() to gracefully close or abort a session.

[$] TELNET_DAEMON

[#] idxTELNET_DAEMON

[K] TELNET_DAEMON

$#KTFTP_SESSION

The TFTP_SESSION uses a UDP socket for file transfers. Once created, the session may be used to get and put multiple files between hosts in a serial fashion.

Creation:

The TFTP_SESSION is created using GCPopen(TFTP_SESSION, hCallbackWnd, CallbackMsg, Instance, NULL, NULL). A UDP socket is created and a GCP_OPENED message is sent to the client application.

Getting and putting files:

The GCPdispatch(hSession, GCP_GET_TFTP_FILE, pCommandParams) and GCPdispatch(hSession, GCP_PUT_TFTP_FILE, pCommandParams) functions may be called to get and put files. All file transfers are processed serially be each session.

Handling received messages:

The hCallbackWnd must be prepared to handle the CallbackMsg when sent by the session. The wParam should be cast to the GCP_COMMAND type. The lParam should be cast to the LPGCP_STATUS type. The following GCP_COMMAND types apply to the TFTP_SESSION: GCP_OPENED, GCP_PUT_TFTP_FILE, GCP_GET_TFTP_FILE, GCP_CLOSE.

Destruction:

Use GCPclose() to gracefully close or abort a session.

$ TFTP_SESSION

idxTFTP_SESSION

K TFTP_SESSION

$[$] #[#] K[K] TFTP_DAEMON

The TFTP_DAEMON responds to any IP host (UNIX, PC, etc.) requesting to get or put files using the Trivial File Transport Protocol (TFTP).

Creation:

The TFTP_DAEMON is created using GCPopen(TFTP_DAEMON, hCallbackWnd, CallbackMsg, Instance, NULL, NULL).   A UDP socket is created and a GCP_OPENED message is sent to the client application. The client is notified with GCP_PUT_TFTP_FILE and GCP_GET_TFTP_FILE messages when the daemon responds to a remote request. The TFTP protocol does not provide security, so files may be accessed on any peripheral and directory.

Handling received messages:

The hCallbackWnd must be prepared to handle the CallbackMsg when sent by the daemon. The wParam should be cast to the GCP_COMMAND type. The lParam should be cast to the LPGCP_STATUS type. The following GCP_COMMAND types apply to the TFTP_DAEMON: GCP_OPENED, GCP_GET_TFTP_FILE, GCP_PUT_TFTP_FILE, GCP_CLOSE.

Destruction:

Use GCPclose() to gracefully close or abort a session.

[$] TFTP_DAEMON

[#] idxTFTP_DAEMON

[K] TFTP_DAEMON

[$][#][K]HGCP_SESSION

HGCP_SESSION is a handle to the session. It is passed back to the client application with all callback messages. The handle is referred to by casting the lParam parameter of the callback window function to an LPGCP_STATUS type and referencing the hSession field. After being passed back with the GCP_OPENED message, it must be saved by the application for all future communication with the referenced session. A NULL value indicates a fatal error condition occurred during the opening process (check Status.Error for the error condition) and that the GCPopen() function failed.

[$] HSESSION

[#] idxHSESSION

[K] HSESSION, Server Handle

$#KGCP_ERROR

GCP_ERROR defines a limited number of error conditions that are valuable without introducing unnecessary detail. The GCP_ERROR enumerated type is used to report the success/failure of every API function call, and every GCP callback includes it as the Error field of the GCP_STATUS structure. It is recommended that the application programmer create a single error handling routine that checks for values not equal to GCP_OK.

For more detailed status information, visually inspect the GCP server's main window for status information.

```
/*  error codes returned by API functions or included in GCP_STATUS */
typedef enum {
    GCP_OK=0,                   /* OK return */
    GCP_UNKNOWN=600,                                        /* An error of unknown origin has occurred */
    GCP_RESERVED1,              /* reserved */
    GCP_ZEROCNT,                /* buffer of zero count has been presented */
    GCP_RESERVED3,              /* reserved  */
    GCP_RESERVED4,              /* reserved  */
    GCP_RESERVED5,              /* reserved  */
    GCP_RESERVED6,              /* reserved  */
    GCP_RESERVED7,              /* reserved  */
    GCP_NOGCPMEM,                       /* GCP Server cannot be started due to insufficient memory */
    GCP_RDOS,                   /*                  There was a DOS error on the remote workstation */
    GCP_DOS,                    /*                   There was a DOS error on the local workstation */
    GCP_RNOENT,                 /* remote file not found */
    GCP_NOENT,                  /* local file not found */
    GCP_NOMEM,                  /* Insufficient resources - object not created */
    GCP_NETWORK,                /* Unspecified network error */
    GCP_ALREADYOPEN,                          /* Daemon was previously opened on specified socket */
    GCP_BADSESSION,             /* Session handle is not known */
    GCP_BADSESSIONTYPE,         /* Session type not supported */
    GCP_BADCALLBACK,                                              /* Callback window does not exist */
    GCP_EOF,                    /* end of file reached */
    GCP_BADMSG,                 /* Msg Request Not Supported by Function/Session */
    GCP_RCLOSE,                             /* Remote party has closed connection or hung up phone */
    GCP_NOGCP,                  /* GCP Server is not running and cannot be found */
    GCP_BADNAME,                /* host name given is unknown */
    GCP_BADVERSION,             /* GCP Server and GCP.DLL versions do not match */
    GCP_NOLICENSE,              /* DLL interface is not licensed for distribution */
    GCP_BUSY,                                /* GCP Server cannot get processor to clear messages */
    GCP_CONNREFUSED,                                       /* host reached, but connection refused */
    GCP_ELAST
    }                           GCP_ERROR;
```

$ GCP_ERROR

idxGCP_ERROR

K GCP_ERROR, error condition, function returns

$#K GCP_COMMAND

The GCP_COMMAND enumerated type is used for messaging between the users application and GCP. It is passed to the API as a function parameter and is passed back as the wParam parameter to the callback window, to notify the application of a completed action. The developer should not assume that the callback will occur before or after the initiating call is returned.

```
typedef enum
	{
	GCP_OPENED=GCP_BASE,			/* session has been opened successfully or failed with error */
	GCP_CLOSE,		/* finish file/buffer transfers and close */
	GCP_ABORT,		/* abort any transfers in progress and close */
	GCP_SEND,		/* send a buffer or confirms a buffer was sent */
	GCP_RECV,		/* a buffer has been received */
	GCP_PUT_TFTP_FILE,	/* send a file or confirms that a file was sent */
	GCP_GET_TFTP_FILE,	/* get a file or confirms that a file was received */
	GCP_TELNET,		/* for negotiating Telnet host options and commands */
	GCP_SHOW,		/* show the GCP Server as an icon */
	GCP_HIDE,		/* hide the display of the GCP Server (default) */
	GCP_SEND_DATAGRAM,	/* parameters include destination address and port */
	GCP_RECV_DATAGRAM,	/* parameters include source address and port */
	GCP_LAST		/* place-holder */
	}			GCP_COMMAND;
```

$ GCP_COMMAND

idxGCP_COMMAND

K GCP_COMMAND, messages

$#KGCP_STATUS

A pointer to the GCP_STATUS structure is included with all callback messages as the WndProc's lParam parameter, and may be retreived at other times by using the GCPquery() function.

```
typedef GCP_STATUS far *LPGCP_STATUS;

typedef struct
      {
      HGCP_SESSION hSession;        /* reference to communication session/daemon */
      GCP_SESSION_TYPE SessionType;             /* type of session (TCP_SESSION, TELNET_DAEMON, etc) */
      unsigned long OpenInstance;/* instance data from GCPopen() */
      GCP_COMMAND_PARAMS Params;                      /* parameters specific to GCP_COMMAND received */
      GCP_STATISTICS Stats;          /* session statistics */
      GCP_ERROR Error;/* Error code, if any */
      }  GCP_STATUS;

typedef struct
      {
      unsigned long StartTime,           /* time when connection occurred in milliseconds */
      StopTime,                 /* current time in milliseconds */
      InCnt,                    /* bytes received at workstation over connection */
      OutCnt,                   /* bytes transmitted by workstation over connection */
      InRate,                   /* bytes/sec transfer rate in */
      OutRate;                  /* bytes/sec transfer rate out */
      }  GCP_STATISTICS;
```

$ GCP_STATUS

idxGCP_STATUS

K GCP_STATUS, status, server status

$#KGCPdispatch ()

```
GCP_ERROR far pascal GCPdispatch(
        HGCP_SESSION hSession,
        GCP_COMMAND Msg,
        GCP_COMMAND_PARAMS far *pParams)
```

This function dispatches a parameterized command message to the GCP server.

Parameters

HGCP_SESSION	hSession (NULL for GCP_HIDE and GCP_SHOW)
GCP_COMMAND Msg	The action being requested. Possible values are GCP_SEND, GCP_SEND_DATAGRAM, GCP_PUT_TFTP_FILE, GCP_GET_TFTP_FILE, GCP_TELNET, GCP_CLOSE, GCP_ABORT, GCP_SHOW, GCP_HIDE
GCP_COMMAND_PARAMS Params	According to the Msg specified, Params may be referenced as Params.Buffer, Params.Telnet, or Params.Tftp. GCP_SHOWand GCP_HIDE take no parameters and cause no callback message.

Return Value

The GCP_ERROR enumerated type is returned. Checking for legitimate parameter values is provided.

Callback Messages

Callback messages arrive at the callback window (hCallbackWnd) with the message (CallbackMsg) that was specified in the GCPopen() function. The wParam contains the callback message (type GCP_COMMAND) and lParam contains a pointer that should be cast to type LPGCP_STATUS. Possible callback messages are: GCP_SEND, GCP_SEND_DATAGRAM, GCP_PUT_TFTP_FILE, GCP_GET_TFTP_FILE, GCP_TELNET, GCP_CLOSE.

The GCP_RECV and GCP_RECV_DATAGRAM messages provide asynchronous notification of incoming data.

$ GCPdispatch

idxGCPdispatch

K GCPdispatch()

[$] [#] [K] GCPperror ()

```
LPSTR far pascal GCPperror(
      GCP_ERROR Error )
```

This function converts a GCP_ERROR type to a textual description.

Parameters

GCP_ERROR is from the LPGCP_STATUS structure returned in a callback message.

Return Value

A null-terminated strings that can be used as a textual description of the error. Typically less than 80 characters in length.

[$] GCPperror

[#] idxGCPperror

[K] GCPperror()

[$][#][K]GCPclose (), GCP_CLOSE, GCP_ABORT

```
GCP_ERROR far pascal GCPclose (
      HGCP_SERVER hServer,
      BOOL Abort)
```

This function closes the hSession specified.

Parameters

HGCP_SESSION	hSession
BOOL Abort	Set to TRUE to immediately close session. Set to FALSE to allow the session to compete buffers/file transmissions before closing.

Return Value

The GCP_ERROR enumerated type is returned. Checking for legitimate parameter values is provided.

Callback Messages

Only GCP_CLOSE is received as a callback message, confirming that the specified hSession has been closed or aborted and that system resources have been released. The hServer cannot be referenced again. This message may arrive without requesting a close, in response to the remote host closing a TCP connection or a fatal error occurring. Check Status.Error for an error condition that may have prompted an unrequested GCP_CLOSE.

[$] GCPclose

[#] idxGCPclose

[K] GCPclose()

$#K GCP_COMMAND_PARAMS

```
typedef union /* currently supported request types */
        {
        GCP_BUFFER_PARAMS Buffer;
        GCP_TFTP_FILE_PARAMS Tftp;
        GCP_TELNET_PARAMS Telnet;
        }  GCP_COMMAND_PARAMS;
```

This structure has two purposes. First, it is used as the 3th parameter of the GCPdispatch() function to parameterize the command being sent to the session. Second, the GCP_STATUS structure includes this type to parameterize the GCP_COMMAND being passed back as wParam. The structure is defined as a UNION, to simplify the correct reference.

For example, to prepare a buffer for sending, the Params.Buffer.Ptr, Params.Buffer.Cnt, and Params.Buffer.CommandInstance should be initialized before GCPsession() is called. When the transmission is verified through the receipt of a GCP_SEND message, the lParam parameter is cast to an LPGCP_STATUS type, and the instance may be referenced as pStatus->Buffer.CommandInstance.

$ GCP_COMMAND_PARAMS

idxGCP_COMMAND_PARAMS

K GCP_COMMAND_PARAMS

$#K GCPopen ()

```
GCP_ERROR far pascal GCPopen (
      GCP_SESSION_TYPE SessionType,
      HWND hCallbackWnd,
      WORD CallbackMsg,
      unsigned long OpenInstance,
      const char far *pAddress,
      unsigned short Port)
```

This function first creates a session object of the specified GCP_SESSION_TYPE type, and then opens a port, waits for a connection or initiates a connection as appropriate. The client application is informed of success or failure after all set-up is accomplished.

The are two basic type of sessions. An "active" session is a TELNET_SESSION, TCP_SESSION, UDP_SESSION or TFTP_SESSION. They are used to establish outbound connections, are spawned by a daemon when a connection is accepted, or utilize a UDP port. A daemon session is a TELNET_DAEMON, TCP_DAEMON, or TFTP_DAEMON session. They respond to remote connection requests or other service request. When a daemon is created, a GCP_OPENED message is received by the client application. When connected to, daemons will spawn active sessions, which also notify the client application with a GCP_OPENED message. The Status.SessionType field must be checked to differentiate between the two conditions.

Parameters

GCP_SESSION_TYPE SessionType Specifies type of session desired.

GCP currently supports 7 communication session types:

1. The TCP_SESSION provides basic TCP connection service to a remote UNIX/PC host.

2. The TCP_DAEMON accepts TCP connection requests and spawns a TCP_SESSION to control the new connection.

3. The UDP_SESSION creates a UDP socket for datagram communication.

4. The TELNET_SESSION provides filtered TELNET connection service to a remote UNIX/PC host.

5. The TELNET_DAEMON accepts TCP connection requests and spawns a TELNET_SESSION to control the new connection.

6. The TFTP_SESSION gets and puts files from/to remote TFTP server.

$ GCPopen

idxGCPopen

K GCPopen()

7. The TFTP_DAEMON services remote gets and puts, turning your PC into a TFTP server.

```
typedef enum                    /* currently supported sessions and daemons */
    {
    UDP_SESSION=GCP_BASE+100,
    TCP_SESSION,
    TCP_DAEMON,
    TELNET_SESSION,
    TELNET_DAEMON,
    TFTP_SESSION,
    TFTP_DAEMON
    }                           GCP_SESSION_TYPE;
```

HWND hCallbackWnd	Set to window handle to be called for all notifications. Used in conjunction with CallbackMsg to define the callback handling code.
WORD CallbackMsg	Set to user callback message for all notifications.
unsigned long OpenInstance	Value to be passed back in GCP_STATUS structure. Suitable for holding a far pointer, or other instance information.
char far * pAddress	Pointer to address (name or dot notation) for connection. Only applicable to TCP_SESSION, TELNET_SESSION (set to NULL for others). Must be a NULL terminated string.
unsigned short Port	Port on remote host (TCP_SESSION, TELNET_SESSION) or on local host (TCP_DAEMON, TELNET_DAEMON, UDP_SESSION). NULL for TFTP_SESSION and TFTP_DAEMON.

Return Value

The GCP_ERROR enumerated type is returned. Checking for legitimate parameter values is provided.

Callback Messages

Callback messages arrive at the callback window (hCallbackWnd) with message (CallbackMsg) specified in GCPopen(). The wParam contains the GCP_OPENED callback message and lParam contains a pointer that should be cast to type LPGCP_STATUS.

$#K GCPquery ()

```
GCP_ERROR far pascal GCPquery(
      HGCP_SESSION hSession,
      GCP_QUERY Msg,
      GCP_QUERY_RESULTS far *pResults)
```

Use GCPquery() to access utilty functions and session status information.

Parameters

HGCP_SESSION hSession

GCP_QUERY Msg GCP_GET_LOCAL_HOST, GCP_GET_FIRST_HOST, GCP_GET_NEXT_HOST, and GCP_GET_STATUS.

T_LOCAL_HOST Gets name and address information about the host workstation.

T_FIRST_HOST Retrieves the first record in the host table. Use this and GCP_GET_NEXT_HOST to pre-fill listboxes for user selection. NOTE: The HOSTS. file can only be accessed if it is in the Windows directory, the current directory, or within the default PATH environment variable.

T_NEXT_HOST Retrieves each subsequent record (a return value of GCP_OK indicates a record was present; GCP_EOF indicates there are no more records). After GCP_GET_FIRST_HOST is called, GCP_GET_NEXT_HOST should be called until a GCP_EOF error is returned; this will ensure that the HOSTS. file is properly closed.

T_STATUS Allows the application to synchronously query the referenced hSession for Status information. This function is also used to check the validity of an hSession handle. A return value of GCP_OK indicates a healthy session and a return value of GCP_BADSESSION indicates that no such session exists.
NOTE: calling this function with GCP_GET_STATUS will also check for data to read/write over the network. Use this feature to provide a "heartbeat" to the GCP Server (if necessary).

GCP_QUERY_RESULTS Results Location where GCP will copy data pertaining to the query type.

Associated Structures

```
typedef enum                /* possible query messages */
      {
      GCP_GET_LOCAL_HOST,       /* gets local host from host table */
      GCP_GET_FIRST_HOST,       /* gets the first host from the host table */
      GCP_GET_NEXT_HOST,        /* gets the next host from the host table */
      GCP_GET_STATUS,    /* gets status information from the hSession */
      }  GCP_QUERY;
```

$ GCPquery

idxGCPquery

K GCPquery()

```
typedef union                    /* currently supported utility types */
      {
      HOST Host;
      GCP_STATUS Status;
      }  GCP_QUERY_RESULTS;

typedef struct                   /* definition of network host structure */
      {
      char Name[64];             /* official name of host */
      char Addr[16];             /* net address in dot notation */
      }  HOST;
```

Return Value

The GCP_ERROR enumerated type is returned. Checking for legitimate parameter values is provided.

Callback Messages

N/A for this function.

$#K GCPalloc ()

```
LPARAM far pascal GCPalloc(
      WPARAM Msg,
      LPARAM pStatus)
```

Use GCPalloc() to make a local copy of the GCP_STATUS structure passed back through the callback window. Since the GCP server uses SendMessage() to notify the client of a completed action, there are some documented system restrictions as to what the client application can do (so you may or may not need this function). Use GCPalloc() to make a local copy of the GCP_STATUS structure and use PostMessage to send it to yourself. This isolates your application completely from the GCP server.

GCPalloc() allocates a global fixed buffer for the GCP_STATUS information, copies it in, and makes a second buffer for received data (if needed). Remember to use GCPfree() in your PostMessage handler to free the memory (this takes care of the second buffer too)!

Look in the GCP_CLNT.CPP file for an example of using this feature.

Parameters

WPARAM Msg	wParam from callback window (the GCP_COMMAND message) is used directly
LPARAM pStatus	lParam from callback window (the GCP_STATUS pointer) is used directly

Return Value

An LPARAM value is returned that may be used directly in the PostMessage() call. If memory cannot be allocated, NULL is returned. For example:

```
// your GCP callback message

case M_GCP_SENDMESSAGE_CALLBACK:

        LPARAM lNewParam;

        if (!PostMessage (hWnd, M_GCP_POSTMESSAGE_CALLBACK,

                        wParam, lNewParam=GCPalloc(wParam, lParam)))
```

$ GCPalloc

idxGCPalloc

K GCPalloc()

```
          {
          MessageBox(hWnd,"PostMessage error","",MB_OK); // could not post message
          GCPfree(lNewParam); // clean up after failure
          }
      break;
// your second callback message
case M_GCP_POSTMESSAGE_CALLBACK:
      LPGCP_STATUS pStatus=(LPGCP_STATUS)lParam;
      GCP_COMMAND Msg=(GCP_COMMAND)wParam;
      // your processing goes here
      // remember for free your copy of GCP_STATUS
      BOOL success=GCPfree(lParam);  // fails if lParam isn't a memory block
      break;
```

$#KGCPfree ()

BOOL far pascal GCPfree(LPARAM pStatus)

Use GCPfree() to free memory allocated by GCPalloc(). The must be done for every memory allocation performed by GCPalloc(). The secondary data buffer is likewise freed.

Look in the GCP_CLNT.CPP file for an example of using this feature.

Parameters

LPARAM pStatus	lParam from callback window (the GCP_STATUS pointer) is used directly

Return Value

If pStatus is a valid pointer to memory, it is freed and TRUE is returned. If pStatus is not a valid pointer, FALSE is returned. See GCPalloc() for an example of how this is used.

$ GCPfree

idxGCPfree

K GCPfree()

[$][#][K]GCP_OPENED

The GCPopen() function initiates a sequence of actions that are ultimately successful or unsuccessful. The GCP_OPENED message, when received as the wParam parameter by the client application, indicates that the process has completed.

Upon receipt of this message the application should cast lParam to a LPGCP_STATUS type and reference the hSession field. If hSession==NULL, then the GCPopen() failed and there is no session to reference. If hSession!=NULL, then the action succeeded and the hSession value is provided for future references to the session created.

[$] GCP_OPENED

[#] idxGCP_OPENED

[K] GCP_OPENED

[$#K]GCP_SEND, GCP_SEND_DATAGRAM and GCP_BUFFER_PARAMS

GCP_SEND is used to command a buffer be sent, and to acknowledge that the buffer was sent (always check pStatus->Error for GCP_OK). For effeciency reasons, the actual data sent is not part of the confirming GCP_SEND message.

GCP_SEND_DATAGRAM is used to command a datagram be sent, and to acknowledge that the buffer was sent. For efficiency reasons, the actual data sent is not part of the confirming GCP_SEND_DATAGRAM message.

The GCP_BUFFER_PARAMS structure is used in both cases:

```
typedef struct
    {
    const byte far *Ptr;        /* pointer to buffer */
    size_t Cnt;                         /* size of buffer to send */
    unsigned long CommandInstance;/* instance for client use...returned with GCP_SEND */
    char RemoteAddress[24];                     /* remote address (dot notation) for UDP sessions only */
    unsigned short RemotePort;/* remote port for UDP sessions only */
    }  GCP_BUFFER_PARAMS;
```

	Location of buffer to send (not passed back with GCP_SEND message).
	Size of buffer to send or was sent.
dInstance	Instance that will be reported back with the GCP_SEND message that acknowledges the transmission.
ldress	Address for datagram only.
rt	Port for datagram only.

[$] GCP_SEND

[#] idxGCP_SEND

[K] GCP_SEND

[$#K]GCP_RECV, GCP_RECV_DATAGRAM and GCP_BUFFER_PARAMS

GCP_RECV is used to signal that a buffer has been received (always check pStatus->Error for GCP_OK).

GCP_RECV_DATAGRAM is used to signal that a datagram is received.

The GCP_BUFFER_PARAMS structure is used:

```
typedef struct
    {
    const byte far *Ptr;       /* pointer to buffer for incoming data */
    size_t Cnt;                          /* size of buffer to read */
    unsigned long CommandInstance;/* N/A */
    char RemoteAddress[24];                        /* remote address (dot notation) for UDP sessions only */
    unsigned short RemotePort;/* remote (sending) port for UDP sessions only */
    }  GCP_BUFFER_PARAMS;
```

	Location of buffer to received.
	Size of buffer received.
dInstance	N/A.
ddress	Address for datagram source only.
rt	Port for datagram source only.

[$] GCP_RECV

[#] idxGCP_RECV

[K] GCP_RECV

$#KGCP_GET_TFTP_FILE, GCP_PUT_TFTP_FILE and GCP_TFTP_PARAMS

GCP_GET_TFTP_FILE is used to command a file be retrieved, and to acknowledge that the action is complete (always check pStatus->Error for GCP_OK). It is also received when the TFTP_DAEMON has captured a local file from a remote TFTP process. The GCP_TFTP_PARAMS structure is used in both cases:

```
typedef struct
       {
       char LocalFileSpec[64];/* NULL terminated local file spec */
       char RemoteFileSpec[64]; /* NULL terminated remote file spec */
       unsigned long CommandInstance;/*  */
       char RemoteAddress[24];            /* name or address for remote TFTP or NULL */
       TFTP_MODE Mode;     /* enumerated type either NETASCII or OCTET */
       }  GCP_TFTP_FILE_PARAMS;
```

Spec	Null-terminated specification of file on local workstation.
leSpec	Null-terminated specification of file on remote workstation. N/A for TFTP_DAEMON.
dInstance	Instance that will be reported back with the GCP_GET_TFTP_FILE message that acknowledges the transfer. N/A for TFTP_DAEMON.
ldress	Remote address (name or dot notation) for file transfer. Null-terminated.
	Either NETASCII (0) or OCTET (1) may be specified (character data or image data).

$ GCP_GET_TFTP_FILE

idxGCP_TFTP

K GCP_GET_TFTP_FILE;GCP_PUT_TFTP_FILE;GCP_TFTP_PARAMS

[$#K]GCP_TELNET and GCP_TELNET_PARAMS

A thorough understanding of RFC854 and RFC855 is required by the programmer to use GCP_TELNET.

Use GCP_TELNET to cause the TELNET_SESSION to construct an option sequence. An incoming GCP_TELNET message is used to signal the client application what the remote host has sent.

```
#define GO_AHEAD_CMD 249
#define WILL_CMD 251
#define WONT_CMD 252
#define DO_CMD 253
#define DONT_CMD 254
#define SB_CMD 250 /* suboption negotiation */

typedef struct
      {
      unsigned char Option;               /* TELNET option to be negotiated */
      unsigned char Cmd;      /* option command WILL_CMD, WONT_CMD, DO_CMD, DONT_CMD */
      unsigned int SubOptionLen; /* length of suboption control sequence */
      char SubOption[256];                                                  /* suboption control sequence */
      }                                   GCP_TELNET_PARAMS;
```

Specifies the TELNET option being negotiated . The RFCs describe what options are possible. This value will be utilized whenever the Cmd parameter contains WILL_CMD, WONT_CMD, DO_CMD, DONT_CMD, or SB_CMD.

Specifies the TELENT option command to assert to the host, or was asserted by the host. Any valid TELNET command can be used, with some common ones defined (WILL_CMD, etc.)

nLen Specifies the length of the SubOption character array. Only used when Cmd==SB_CMD.

n After option negotiation proceeds into the sub-option negotiation, this string is used to communicate sub-options with the remote host. Only used when Cmd==SB_CMD.

Example:

Suppose one wishes to establish a TELNET session with a remote host using a particular terminal type. The programmer would proceed as follows:

1. Call GCPopen() with the appropriate parameters set.

2. After receiving the GCP_OPENED message, several GCP_TELNET messages will probably arrive.

3. If a GCP_TELNET message arrives with Option==24 and Cmd==DO_CMD, respond with a GCPdispatch() function call using GCP_TELNET, Option=24, Cmd=WILL_CMD. This means you agree to send terminal type information in a subsequent sub-option command.

4. When a sub-option command arrives with Option==24, inspect the SubOption array for a '\001'. That is your queue to send terminal type information. Make sure the first character of the SubOption

[$] GCP_TELNET

[#] idxGCP_TELNET

[K] GCP_TELNET

string is a '\000'. Refer to RFC1091.

5. If a GCP_TELNET message arrives with Option==3 and Cmd==WILL_CMD, respond with a GCPdispatch() function call using GCP_TELNET, Option=3, Cmd=DO_CMD. This means you agree that the host will suppress GO_AHEADS.

6. If any unwanted Options arrive with Cmd==DO_CMD, always respond with a WONT_CMD. IMPORTANT: The Option must match the one received.

[$#K]GCP_HIDE, GCP_SHOW

GCP_HIDE commands GCPxx.EXE to hide from view. If no sessions are active, this command will cause the GCP server to terminate. The GCP server starts off as hidden when started using an API call like GCPopen().

GCP_SHOW commands GCPxx.EXE to show itself as an icon. If not active, this command will cause the GCP server to load. The GCP server starts off as a normal window when started via the "run" command or other interactive means.

[$] GCP_HIDE

[#] idxGCP_HIDE

[K] GCP_HIDE, GCP_SHOW

$#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.