

# **Gcom<sup>®</sup>**

# **Data Tunneling Tutorial**

**January, 2007**

## **Gcom, Inc.**

1800 Woodfield Drive  
Savoy, IL 61874

217.351.4241  
Fax: 217.351.4240

Email: [support@gcom.com](mailto:support@gcom.com)  
<http://www.gcom.com>

© 2007 Gcom, Inc. All Rights Reserved.

Non-proprietary—Provided that this notice of copyright is included, this document may be copied in its entirety without alteration. Permission to publish excerpts should be obtained from Gcom, Inc.

Gcom reserves the right to revise this publication and to make changes in content without obligation on the part of Gcom to provide notification of such revision or change. The information in this document is believed to be accurate and complete on the date printed on the title page. No responsibility is assumed for errors that may exist in this document.

Any provision of this product and its manual to the U.S Government is with “Restricted Rights”: Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at 252.227-7013 of the DoD FAR Supplement.

A partial list of registered trademarks includes Gcom, Rsys, Rsystem, and SyncSockets. All other product or company names may be trademarks of their respective owners.

Dave Grothe and Michael Lynch were the subject matter experts for this document.

## Contents

<b>Read Me First.....</b>	<b>5</b>
Data Tunneling Tutorial Intended Audience .....	5
Data Tunneling Tutorial Purpose.....	5
Data Tunneling Tutorial Program .....	6
Data Tunneling Tutorial Program Limitations .....	6
Gcom Data Tunneling Documentation .....	7
<b>Data Tunneling Overview .....</b>	<b>8</b>
Data Tunneling Protocol.....	8
Principle Components .....	8
Data Tunneling Messages.....	9
How Everything Works Together .....	9
<b>Tutorial Program Basics .....</b>	<b>10</b>
Download and Installation .....	10
Include Files and Libraries .....	10
Compiling the Tutorial Program .....	10
Running the Tutorial Program .....	11
Running a Demonstration of the Tutorial Program .....	12
<b>Tutorial Architecture .....</b>	<b>14</b>
Tutorial Program Execution Flow .....	14
Data Structures and Poll List Handling .....	15
Poll List Handling During Connecting Phase .....	18
Poll List Handling During Data Transfer Phase .....	19
Basic Data Tunneling API Function Calls .....	20
Most Useful Tutorial Program Routines .....	21
<b>Advanced Topics .....</b>	<b>22</b>
Logging Options.....	22
Log File Handling .....	22



## **Read Me First**

- [Data Tunneling Tutorial Intended Audience](#)
- [Data Tunneling Tutorial Purpose](#)
- [Data Tunneling Tutorial Program](#)
- [Data Tunneling Tutorial Program Limitations](#)
- [Gcom Data Tunneling Documentation](#)

### **Data Tunneling Tutorial Intended Audience**

This tutorial document and accompanying tutorial program are intended for developers who want to...

- Modify an existing communications application...
- To utilize the Gcom® Data Tunneling API...
- To interface to a Gcom Protocol Appliance (GPA 2G) ...
- To interact with another device that uses legacy protocols such as X.25, SNA, LU 6.2, or Bisync.

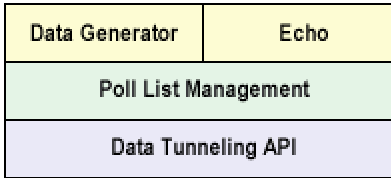
### **Data Tunneling Tutorial Purpose**

The purpose of this tutorial document and accompanying tutorial program is to illustrate:

- The use of Gcom's Data Tunneling API
- How to structure a program to handle multiple connections using the Unix system `poll()` routine

## Data Tunneling Tutorial Program

The tutorial program, a complete program that illustrates Gcom's Data Tunneling API constructs in context, is structured in the following manner:



Tutorial Function	Description
Data Generator	The data generator function sends encapsulated data messages over one or more TCP connections and expects to read them back. After sending an initial burst of messages, it enters a loop to read one message back and then send another message. You can configure the data generator to send fixed length or random length messages.
Echo	The echo function simply reads an encapsulated data message and writes it back on the same file descriptor.
Poll List Management	Poll list management is the portion of the tutorial program that uses the Unix <code>poll(2)</code> routine to manage data flow on multiple connections.
Data Tunneling API	The Data Tunneling API is the standard Gcom API library used to open/close and send/receive encapsulated data messages over TCP connections.

To effectively run the tutorial program, you must run two copies – one in data generator mode and one in echo mode. A command line argument selects the mode.

The tutorial is packaged in a tar archive with make files for compiling the code on a number of selected platforms.

The code in the tutorial program is extensively commented; consequently, this document does not explain the tutorial program line by line, but explains instead the higher-level program structure with just a few detailed examples for illustrative purposes.

## Data Tunneling Tutorial Program Limitations

The tutorial program is not intended to be a useful Data Tunneling application in and of itself. It simply:

- Generates data patterns and sends them to another device using Gcom's Data Tunneling API.
- Or receives blocks of data and echoes them back to their originator.

These routines, although useful for building a test program and illustrating Gcom's Data Tunneling API, are not generally useful for production applications.

## **Gcom Data Tunneling Documentation**

All Gcom Data Tunneling API functions used in the tutorial program are fully documented in the *Gcom Data Tunneling User Guide* document at <http://www.gcom.com/support/documentation.html>.

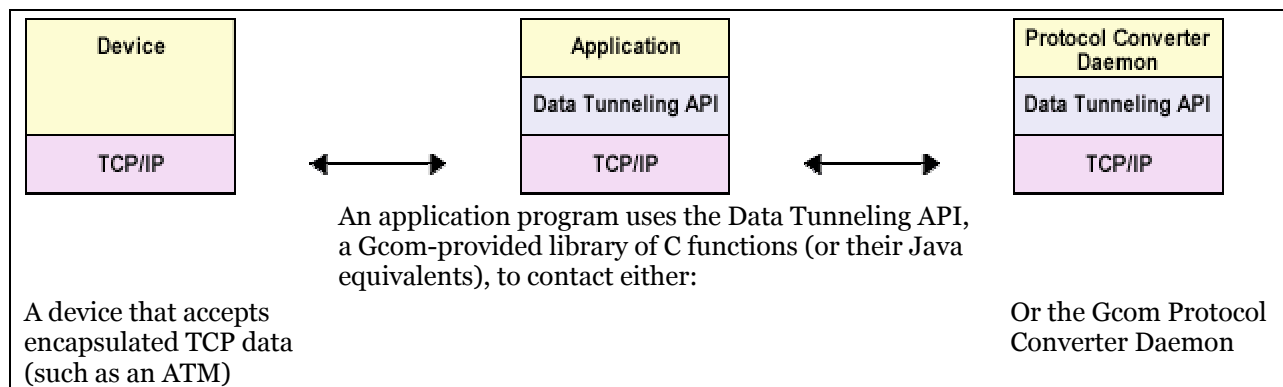
## Data Tunneling Overview

- [Data Tunneling Protocol](#)
- [Principle Components](#)
- [Data Tunneling Messages](#)
- [How Everything Works Together](#)

### Data Tunneling Protocol

Gcom's Data Tunneling protocol consists of encapsulated data messages sent and received over TCP/IP connections.

### Principle Components



If you are using a Gcom Protocol Appliance (GPA 2G) product, the Protocol Converter Daemon (Gcom\_pcd) resides on the appliance and the TCP/IP connection is probably across a high-speed Ethernet LAN segment.

**Note:**

- This document presumes the Protocol Converter Daemon resides on a Gcom Protocol Appliance, which converts between legacy protocols (X.25, SNA, Bisync, etc.) on one side of the connection and encapsulated TCP messages on the other side.
- Two peer applications may use Gcom's Data Tunneling API to communicate via encapsulated data messages sent over TCP.

## Data Tunneling Messages

Each TCP connection exchanges Data Tunneling messages with the remote peer. A remote peer can be Gcom Protocol Converter Daemon (`Gcom_pcd`), a device, or a peer application.

Data Tunneling messages consist of:

- Payload data
- A delimiter that preserves the logical message boundaries needed to send payload data over a byte stream connection such as TCP

A delimiter may be a header or a header/trailer combination.

Gcom's Data Tunneling API library offers a variety of delimiter formats, which Gcom calls *encapsulations*.

Please check the Gcom web site for the most up-to-date list of available encapsulations: [http://www.gcom.com/home/products/gpa\\_dt\\_encapsulation\\_summaries.html](http://www.gcom.com/home/products/gpa_dt_encapsulation_summaries.html)

## How Everything Works Together

To use a Gcom Data Tunneling connection, use the `dt_open` function from Gcom's Data Tunneling API library.

The application passes the `dt_open` function a host name, port number, encapsulation choice, and callback routine. A thread, generated by the `dt_open` function, calls the application-supplied callback routine when the connection completes.

To send or receive data, use the `dt_send` and `dt_recv` functions respectively. The `dt_send` function generates the encapsulation header; the `dt_recv` strips the encapsulation header. Thus, your application deals only with payload data.

Use the `dt_close` function to close the connection.

That really is just about all there is to it. Continue reading for:

- More detailed information about the tutorial program itself
- Information on how to use the Unix `poll(2)` routine

## Tutorial Program Basics

- [Download and Installation](#)
- [Include Files and Libraries](#)
- [Compiling the Tutorial Program](#)
- [Running the Tutorial Program](#)
- [Running a Demonstration of the Tutorial Program](#)

### Download and Installation

Installation Steps	
Unix/Linux Platform	Windows Platform
1. Create a directory for dt_tutorial.tar.	1. Create a directory for dt_tutorial.tar.
2. Copy dt_tutorial.tar to that directory.	2. Copy dt_tutorial.tar to that directory.
3. cd to that directory.	3. cd to that directory.
4. Type: tar xf dt_tutorial.tar	4. Use your favorite unzip program to unzip dt_tutorial.tar.
5. Type: cd dt_tutorial	5. Type: cd dt_tutorial

### Include Files and Libraries

The tutorial program uses standard include files that should be present on your system and the following Gcom-provided include files:

File Name and Location	Description
<gcom/dtapi.h>	Header file for DT API routines and constants, including encapsulation format type codes
/usr/lib/gcom/dtapi.a or /usr/lib/gcom/dtapi.so	DT API library file for Linux/Solaris platforms
c:\windows\system32\dtapi.dll	DT API library file for Windows platform

### Compiling the Tutorial Program

The tutorial directory contains subdirectories (Linux, Solaris 32-bit, Solaris 64-bit and Windows) used to compile the tutorial program. Each subdirectory contains a makefile.

Choose the subdirectory closest to your system; cd to that subdirectory, and proceed as follows:

<b>Linux/Solaris Platforms</b>	Type: make
<b>MinGW/MSys Compiler Windows Platform</b>	Type: make
<b>Visual C Compiler Windows Platform</b>	<ol style="list-style-type: none"> <li>1. Create an import library using the following command: lib /machine:i386 /def:/gcom/lib/dtapi.def</li> <li>2. Use the resulting import library to link against the tutorial.</li> </ol>

If you are building a Data Tunneling application of your own, consult the makefile to see the libraries to link with it.

## Running the Tutorial Program

### Command Syntax:

`dt_tutorial options`

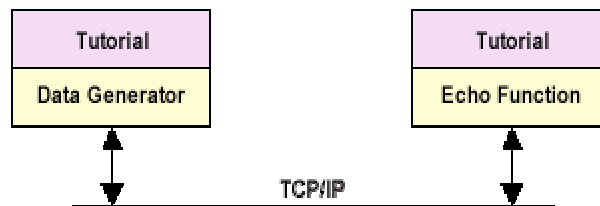
### Command Options:

Option	Argument	Default	Purpose
<code>-?</code>	None		Print help text.
<code>-b</code>	Burst Size	1	Set the initial burst of messages to send.
<code>-d</code>	Debug Level	0	Set the Data Tunneling API debug level. Use <code>-1</code> for maximum verbosity.
<code>-e</code>	Encapsulation	raw	Set the encapsulation format. <ul style="list-style-type: none"> <li>• Precede with <code>+</code> to include the byte count of the header itself in the header length.</li> <li>• Precede with <code>-</code> to exclude the byte count of the header itself from the header length.</li> <li>• Omit any prefix to include/exclude the byte count of the header itself in/from the header length depending upon the protocol specification.</li> </ul>
<code>-E</code>	None		List mnemonics for available encapsulations.
<code>-h</code>	Host Name or IP Address	localhost	Set the host name or IP address of the: <ul style="list-style-type: none"> <li>• Device</li> <li>• Or the machine on which the peer application resides</li> </ul>
<code>-i</code>	None		Use listen instead of connect.
<code>-I</code>	Iteration Count	0	Set the number of messages to send/receive per connection. Use <code>0</code> for continuous operation.
<code>-l</code> (lower case L)	Log File Name	dt_tutorial.log	Set the name of the log file into which the Data Tunneling API writes messages.

Option	Argument	Default	Purpose
-n	Number of Connections	1	Set the number of connections the tutorial program manages.
-p	Port Number	28000	Set the port number (to which connections are made) of the: <ul style="list-style-type: none"> <li>• Device</li> <li>• Or the machine on which the peer application resides</li> </ul>
-q	None		Enable quiet mode in which no connection setup messages are printed.
-s	Message Size	1024	Set the size of the message data field for sending messages. Use 0 for random size between 1 and 4096.
-t	Test Type	0	Set to: <ul style="list-style-type: none"> <li>• 0 for send/receive test</li> <li>• 1 for echo test</li> </ul>
-v	None		Enable verbose mode.

## Running a Demonstration of the Tutorial Program

Establish the following test setup:



### Procedure

On the machine on which you installed and built the tutorial program, run one copy of the tutorial program in data generator mode and another copy in echo mode using the following commands:

- o Echo:

```
dt_tutorial -h localhost -e gcom -n 4 -t 1 -i &
```
- o Data Generator:

```
dt_tutorial -h localhost -e gcom -n 4 -t 0 -I 500000 &
```

### Results

The two programs connect to each other via TCP, and start sending and receiving data. Note messages on your screen similar to the following:

```
dt_tutorial - Version: 1.4
1 connection(s) complete fd=8
2 connection(s) complete fd=9
3 connection(s) complete fd=10
```

```
4 connection(s) complete fd=11
```

```
WrRd: Running 500000 iterations per connection
```

```
1 connection(s) complete fd=7
```

```
2 connection(s) complete fd=8
```

```
3 connection(s) complete fd=9
```

```
4 connection(s) complete fd=10
```

At 10 second intervals, you see progress messages on your screen similar to the following:

```
Echo: send=119639 receive=119640 total=239279 rate=50719
```

```
WrRd: send=250688 receive=250669 total=501357 rate=50135
```

```
Echo: send=359290 receive=359292 total=718582 rate=48821
```

```
WrRd: send=479421 receive=479401 total=958822 rate=47940
```

When the test completes, the programs print a termination summary message similar to the following.

```
WrRd: Test complete
```

```
WrRd: send=2000016 receive=2000000 total=4000016 rate=45622
```

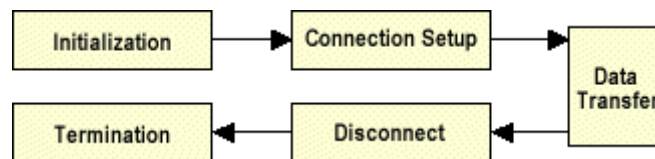
```
Echo: Test complete
```

```
Echo: send=2000016 receive=2000016 total=4000032 rate=45609
```

## Tutorial Architecture

- [Tutorial Program Execution Flow](#)
- [Data Structures and Poll List Handling](#)
- [Poll List Handling During Connecting Phase](#)
- [Poll List Handling During Data Transfer Phase](#)
- [Basic Data Tunneling API Function Calls](#)
- [Most Useful Tutorial Program Routines](#)

### Tutorial Program Execution Flow



Phase	Description
Initialization	Occurs in the <code>main</code> routine. Includes: <ul style="list-style-type: none"> <li>• Interpreting the command line arguments</li> <li>• Allocating the poll list and the connection list tables</li> <li>• Initializing the Data Tunneling API</li> <li>• Initializing the transmit data buffer with a counting pattern</li> </ul>
Connection Setup	Initiated from the <code>build_plist</code> routine. Includes: <ul style="list-style-type: none"> <li>• Filling in the information in the poll list and connection list</li> <li>• Opening TCP connections using <code>dt_open</code></li> </ul>
Data Transfer	Initiated from either the <code>write_read_test</code> or <code>echo_test</code> routine depending upon the value of the <code>-t</code> command line option. Includes: <ul style="list-style-type: none"> <li>• Setting up the connection list entries' callout functions according to the type of test being run</li> <li>• Calling the <code>run_plist</code> routine to operate the connections</li> <li>• Printing progress messages at regular intervals</li> </ul> A connection enters the Disconnect phase when it reaches its iteration count.
Disconnect	Initiated from the <code>read_data</code> routine when the iteration count is reached. Includes calling the <code>dt_close</code> function. The <code>run_plist</code> routine returns after all connections are closed.
Termination	Consists of a return from the <code>run_plist</code> routine when the poll list processing is complete and all connections are closed. The tutorial program prints one last progress message and then exits.

## Data Structures and Poll List Handling

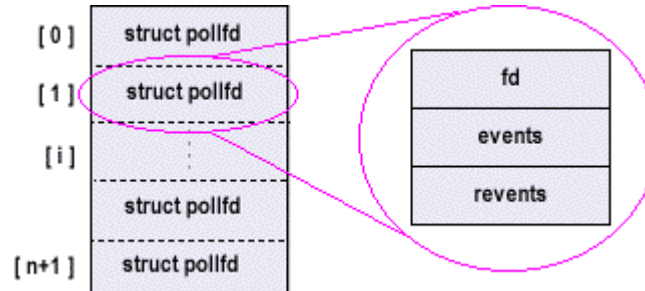
To handle multiple file descriptors in an application driven by spontaneously arriving input from any subset of the files, you must create some kind of enumerated list that identifies:

- File descriptors
- State of the file in which the application is interested — the two common cases:
  - o Is the file read-ready? That is, if the application reads from the file, are there data to be read or does the application block awaiting data?
  - o Is the file write-ready? That is, if the application writes to the file, are the data accepted or does the application block awaiting room for storing the data?

The `poll(2)` system call is the key to solving this problem.

`poll(2)` accepts an array of elements. Each element has a file descriptor and a set of conditions of interest. When one or more files meets the conditions specified for that file, `poll(2)` returns an updated array that reflects the actual conditions of the files. The `poll(2)` user then walks down the list looking for conditions of interest and handles files as it encounters them. Quite straightforward, really.

The array passed to `poll(2)` is called a *poll list*. Each array element consists of a *struct pollfd* structure.



Each structure contains three fields:

- File descriptor
- Events (or conditions) of interest – The events field is a bit-encoded field in which some bits correspond to read conditions and some to write conditions
- Returned events from the operating system – The revents field has the same format as the events field. It contains bits returned by the operating system, which performs a logical *and* of the events bits against the bits that represent the complete condition of the file. Thus, the settings in the revents field are restricted to what was requested in the events field. The revents field also contains some error bits set only by the operating system.

Bit definitions are provided in the `<sys/poll.h>` file and do not concern us here. In fact, different operating systems define different sets of bits. The tutorial program takes

this into account by defining its own values with the mnemonics `RD_EVENTS` and `WR_EVENTS`.

In the tutorial program, the poll list is held in a global array named *plist*. The `plist` variable is actually a pointer to the poll list and the poll list itself is allocated dynamically with the number of elements requested by the *-n* command line parameter.

In addition to the poll list, the `poll(2)` system call accepts a timeout parameter that specifies the number of milliseconds to wait until some file descriptor meets its requested conditions. A 0 value results in a quick check of all the descriptors and an immediate return. A -1 value indicates an infinite timeout.

The tutorial program calls `poll(2)` as follows:

```
nfds = poll(plist, num_connections, 1000) ;
```

where:

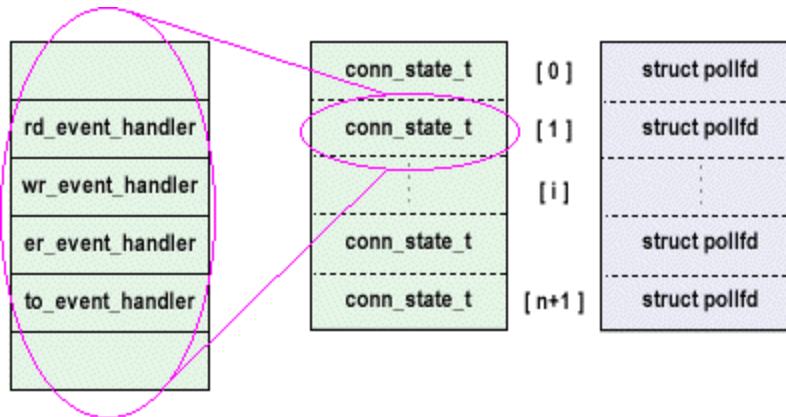
- o `plist` is the poll list
- o `num_connections` is the number of elements in the poll list
- o 1000 (one second) is the timeout duration

As you may surmise from the name of the variable, `poll(2)` returns the number of file descriptors that meet the requested conditions. If `nfds` is:

- Negative — `poll(2)` failed and the `errno` variable contains the error condition. A failure is not always serious. It could simply mean the application was signaled, as from an alarm signal. In such cases, the application simply calls `poll(2)` again.
- 0 — The timeout occurred. Thus, `poll(2)` can be used as a way to keep track of time at some intervals. This is not high precision because returns with a positive `nfds` consume time that is not accounted for by calls that result in timeouts. If high-precision time tracking is necessary, use `poll(2)` to wake up the application and then use `gettimeofday(2)` to obtain high-precision time.
- Positive — There are some elements of the poll list that meet the requested conditions. The application walks down the poll list and examines the `revents` field of each element. It can stop looking when it finds `nfds` of them with non-zero `revents`.

The tutorial program centralizes these operations into a single `run_plist` routine. The question is, what does it do when the conditions are met (non-zero `revents`) for a particular file?

There is another array of structures in the tutorial program called *clist*. It has the same number of structure elements as *plist* and a corresponding index. Picture the two arrays together as follows.



Each clist element corresponds to one plist element. So when `run_plist` examines `plist[i]` and finds either read, write, or error conditions, it refers to `clist[i]` to discover what to do.

The clist structure fields on which we are focusing are the `rd_`, `wr_`, `er_` and `to_event_handler` routine pointers to routines that handle, respectively, the read conditions, write conditions, error conditions, and timeout conditions of the file corresponding to index *i*.

Thus, you see in `run_plist` a piece of code similar to the following.

```
if (plist[i].revents & RD_EVENTS)
    rslt = clist[i].rd_event_handler(i) ;
```

There is comparable code for handling write conditions and error conditions. Timeout conditions are handled similarly, but there are no condition bits to test, so the timeout calls apply to all elements of the array.

So with only one routine that scans the poll list, an application can store pointers to different functions in the function pointer variables and treat conditions differently on a file-by-file basis.

**Important notes:**

- The tutorial program is a simple-minded application from the standpoint of using `poll(2)`. More sophisticated applications must grapple with an additional problem: What happens when files in the middle of the poll list get closed? (This happens all the time with network connections.)

The answer is quite simple – and you can see an example in the tutorial program. At the start, the tutorial program opens the `/dev/null` file and saves the file descriptor number in a global variable. When the file is closed, the tutorial program stores the `dev_null_fd` value in the `fd` field of the `plist` element and sets the `events` field to 0 (to not request any conditions). `dev_null_fd` is a valid file descriptor, so the operating system does not flag an error on that slot, and, in fact, doesn't do much of anything with it because the tutorial program is not requesting any conditions.

Later, when a more sophisticated application must find an available slot in the `plist/clist` to use for a new connection, it scans the `plist` array looking for a slot whose `fd` is equal to `dev_null_fd` – that's an available slot.

The `connection_complete()` routine contains code that illustrates how to do this.

- We have seen application samples on other web sites that use the term *poll* in a completely different sense. These other sites use the term when the application visits each file and ascertains if it is read-ready. It usually does so by calling the operating system and querying the state of the file in a non-blocking fashion. This, of course, leads to the application completely consuming the CPU – it never blocks so that other applications can run. This is an ancient technique that was often used for MS-DOS applications but is not appropriate for modern Unix or Windows NT classes of application code.

The Unix `poll(2)` system call blocks the caller until conditions are met with one or more files in the poll list.

- Windows does not implement the `poll(2)` routine; however, it does implement a similar routine called `select`. The tutorial program contains a poll routine that is written in terms of `select`. It is only compiled when the compilation platform is Windows. That's why the tutorial program, which is written entirely in terms of `poll`, functions in a Windows environment just as it does in a Unix or Linux environment.

**Poll List Handling During Connecting Phase**

The `start_dt_connection` routine uses the `dt_open` function to initiate the TCP connection, whether outgoing or incoming. In either case, it passes the `connection_complete` routine as the callback routine.

When either an outgoing or incoming connection completes, this callback routine is called from a separate thread within the Data Tunneling API. The `connection_complete` routine is passed the file descriptor for the open connection.

This raises a subtle problem in handling poll lists. Suppose the main program is blocked calling the `poll` routine. Now the `connection_complete` routine is invoked from a different thread and wants to do the following.

1. Set the new file descriptor into the `fd` field of the `plist` slot for this connection.
2. Get the main program to wake up from the `poll` system call.

It cannot accomplish #1 by simply assigning the file descriptor into the `fd` field in the `plist`. Why is that? Some Unix systems overwrite the entire poll list when they return from the `poll` system call. The theory is that the user passes the list down to the operating system; the operating system modifies it; and the operating system writes it back to the user's list. If a separate thread stores into any field of the poll list, that field is overwritten when `poll` returns to the user.

To accomplish #2, the tutorial program creates a pipe pair of file descriptors and inserts the read end of the pipe into slot 0 of the `plist`. When a callback routine wants to wake up the main poll loop, it simply writes something into the pipe. This makes the read end of the pipe read-ready and triggers the `POLLIN` bit for that poll list element.

To accomplish #1, the tutorial program allocates a third list called `ulist`, which is an array of `pollfds` like that in `plist`. It is the *update list*. The callback routine:

- Stores the new file descriptor into the `fd` slot in `ulist`.
- Sets the `events` field to the desired value.
- Writes the integer index value of the slot into the pipe.

When the callback routine writes into the pipe, the call to `poll` returns with the read end of the pipe marked as read ready. The read-ready routine for the pipe then reads the index from the pipe and transfers the information from the `ulist` to the `plist`.

The routines involved in this operation are `update_plist`, which writes into the pipe, and `pipe_read`, which is the read-ready routine for the pipe slot in the poll list.

## Poll List Handling During Data Transfer Phase

It is fairly clear that it is a good idea to wait until a file is read-ready before receiving from it, so the application does not block. But it is a more subtle point to wait until a file is write-ready before sending to it.

The data generator test uses the read and write events bits in the following manner to accomplish this:

1. Send the initial burst of messages without regard to flow control. This is a simplifying assumption, namely that the initial burst will not encounter flow control back pressure.
2. Set the `RD_EVENTS` in the `plist` slot.
3. Call `run_plist` to operate the polling list.

4. When the file becomes read-ready, call the `read_data` callout routine to read in the message. It clears the `RD_EVENTS` bits and sets the `WR_EVENTS` bits in the plist slot.
5. When the file is write-ready, call the `write_data` callout routine. It sends the next test message.

The echo test works essentially the same way except there is no initial burst sent. The `echo_data` routine occupies the `wr_event_handler` callout pointer and sends a message from the receive buffer rather than generating a new message.

If you are writing a Data Tunneling application that receives data from some sort of network connection and you want to write it to another (say a Data Tunneling) connection, follow a similar pattern:

1. Read the data into a buffer associated with the data source when it becomes read-ready.
2. Turn off that file's `RD_EVENTS` and turn on the `WR_EVENTS` for the destination file.
3. Write the buffer of saved data when the destination becomes write-ready.
4. Turn off the `WR_EVENTS` for the destination file and turn on the `RD_EVENTS` for the source.

(For full-duplex data flow, also perform the same algorithm for the reverse direction.)

When you do this, you couple the flow control on the destination to the source. So if the destination goes into a flow control stop condition, you do not read any more data from the source, which lets flow control back pressure develop for the source connection.

If you don't do this, you could end up with the entire data-handling application blocking while trying to send to the destination. In that case, if relieving flow control blockage requires some human intervention (such as pressing **Ctrl-Q** on the keyboard), you can bring the entire data communication application to a halt.

## Basic Data Tunneling API Function Calls

The tutorial program uses only five DT API function calls:

Data Tunneling API Function	Purpose
dt_init	Set the name of the Data Tunneling API log file and log option bits. This is the first call to the Data Tunneling API and is best made early in the main program.

<b>Data Tunneling API Function</b>	<b>Purpose</b>
dt_open	Open a TCP connection to the remote peer. A remote peer can be Gcom Protocol Converter Daemon (Gcom_pcd), a device, or a peer application. The tutorial program passes the hostname name/IP address of the remote peer, port number, encapsulation choice and callback routine. The tutorial program's use of this function is just about as sophisticated as any Data Tunneling application will ever need.
dt_close	Close a connection – the opposite of dt_open.
dt_send	Send an encapsulated payload data message over an open TCP connection.
dt_recv	Receive a payload data message – with encapsulation header removed – from an open TCP connection.

There are more functions in the complete Data Tunneling API, many of which relate to more sophisticated log file handling. For example: Functions exist that let the Data Tunneling application write to the Data Tunneling API's log file or set the length and wrap point of the file.

You can write a sophisticated Data Tunneling program using only these five functions plus, perhaps, a few more of the log-handling functions.

## Most Useful Tutorial Program Routines

The following table summarizes tutorial program routines that illustrate how to use the Data Tunneling API.

<b>Routine</b>	<b>Purpose</b>
connection_complete	Illustrates a connection completion callback routine.
start_dt_connection	Illustrates the use of dt_open .
read_data	Read-ready callout routine Used during the data transfer phase for both the data generator test and the echo test.
write_data	Write-ready callout routine that sends the next data message from the test data buffer Used during the Data Transfer phase of the data generator test.

## Advanced Topics

This section discusses some Data Tunneling API constructs not used by the tutorial program.

**Note:** The *Gcom Data Tunneling User Guide* document at <http://www.gcom.com/support/documentation.html> is the definitive reference for using these constructs. Please consult it before using anything defined in this section.

- [Logging Options](#)
- [Log File Handling](#)

### Logging Options

When you call the `dt_init` function, one of the passed parameters is a set of logging options. This quantity is actually a bit-encoded word, the bits of which are defined in the `<dtapi.h>` file. Thus, when setting these options, it is possible to be more selective than simply turning everything on or off.

### Log File Handling

The Data Tunneling API contains several functions for managing the log file produced by the API:

Function	Purpose
<code>dt_printf</code>	A printf-like function that writes a message into the log
<code>dt_lock_print</code>	Lock the log so the Data Tunneling application can call <code>dt_printf</code> multiple times without any other thread writing to the log in the interim.
<code>dt_unlock_print</code>	Unlock the log.
<code>dt_set_log_size</code>	Set the size of the log and the wrap point. Used to enable the circular log-handling mechanism within the Data Tunneling API. Typical use: Use <code>dt_printf</code> to write some startup messages into the log; set the size of the log to some value; set the wrap point to the present location in the log. When the log wraps, the initial messages in the log are preserved.