

Gcom[®] SyncSockets[®] Tutorial

January, 2007

Gcom, Inc.

1800 Woodfield Drive
Savoy, IL 61874

217.351.4241
Fax: 217.351.4240

Email: support@gcom.com
<http://www.gcom.com>

© 2006-2007 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

Read Me First.....	5
SyncSockets Tutorial Intended Audience	5
SyncSockets Tutorial Purpose	5
SyncSockets Tutorial Program	6
SyncSockets Tutorial Program Limitations	6
Gcom SyncSockets Documentation.....	7
SyncSockets Overview.....	8
SyncSockets Protocol.....	8
Principle Components	8
SyncSockets Messages	9
SyncSockets Header Fields.....	9
How Everything Works Together	10
Tutorial Program Basics	11
Download and Installation	11
Include Files and Libraries	11
Compiling the Tutorial Program	11
Running the Tutorial Program	12
Running a Demonstration of the Tutorial Program	13
Tutorial Architecture	16
Tutorial Program Execution Flow	16
Data Structures and Poll List Handling	17
Poll List Handling During Data Transfer Phase	21
Basic SyncSockets API Function Calls	22
Most Useful Tutorial Program Routines	22
Selected Tutorial Program Routines	24
send_msg	24
recv_msg	24
open_ss_connection	25
issue_conn_req.....	26
Advanced Topics	28
Logging Options.....	28
Log File Handling	28
Connection Types	29
Extended Connect Requests	29
Special Operations	29
Option Field Values	30
Host Mode Operation	30
Encryption	31

Read Me First

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

SyncSockets 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[®] SyncSockets[®] API...
- To interface to a Gcom Protocol Appliance (GPA 2G) or a Gcom Protocol Kit (GPK)...
- To interact with another device that uses legacy protocols such as X.25, SNA, LU 6.2, or Bisync.

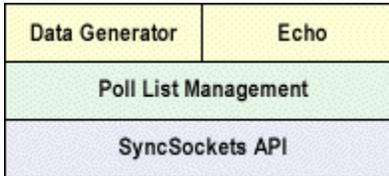
SyncSockets Tutorial Purpose

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

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

SyncSockets Tutorial Program

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



Tutorial Function	Description
Data Generator	The data generator function sends data messages over one or more SyncSockets 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 a SyncSockets 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.
SyncSockets API	The SyncSockets API is the standard Gcom API library used to open/close and send/receive SyncSockets 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.

SyncSockets Tutorial Program Limitations

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

- Generates data patterns and sends them to another device using Gcom's SyncSockets 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 SyncSockets API, are not generally useful for production applications.

Gcom SyncSockets Documentation

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

SyncSockets Overview

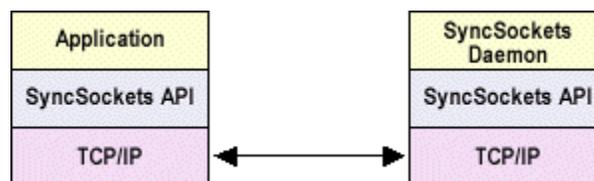
- [SyncSockets Protocol](#)
- [Principle Components](#)
- [SyncSockets Messages](#)
- [SyncSockets Header Fields](#)
- [How Everything Works Together](#)

SyncSockets Protocol

Gcom's SyncSockets protocol controls legacy protocol connections, which are usually synchronous serial connections, by exchanging messages over a TCP connection via the standard Unix socket mechanism.

Thus the term *SyncSockets*.

Principle Components



An application program uses the SyncSockets API, a Gcom-provided library of C functions (or their Java equivalents), to contact the SyncSockets Daemon (SSD), a Gcom-provided program, via a TCP/IP connection.

If you are using a Gcom Protocol Kit (GPK) product, the SyncSockets Daemon may reside on the same system as the application. If you are using a Gcom Protocol Appliance (GPA 2G) product, the SyncSockets Daemon resides on the appliance and the TCP/IP connection is probably across a high-speed Ethernet LAN segment.

Note: This document presumes the SyncSockets Daemon resides on a Gcom Protocol Appliance, but the TCP/IP connections apply equally to the Gcom Protocol Kit product.

SyncSockets Messages

Each TCP connection exchanges SyncSockets protocol messages and connects to one legacy protocol connection.

SyncSockets messages contain a header section and data section. The header:

- Identifies the `operation` type, such as establish a connection, send data, and disconnect a connection.
- Specifies how much data follows.

There are two kinds of functions for sending and receiving SyncSockets messages:

- One kind passes an internal form of the SyncSockets header.
- The other kind passes the components of the SyncSockets header, and the SyncSockets API uses the components to build the actual SyncSockets header sent over the TCP connection. The tutorial program uses the latter kind. These functions have the suffix `_nh`, which stands for *No Header*.

SyncSockets Header Fields

SyncSockets header fields including the following:

Field Name	Description	
uint8_t op	Operation code that specifies the SyncSockets message type, such as:	
	SS_OP_CONN_REQ	Connect to a named connection configured in the SyncSockets Daemon.
	SS_OP_CONN_CONF	Confirm a connect request.
	SS_OP_LISTEN	Set up a listen for an incoming legacy connection via a named connection configured in the SyncSockets Daemon.
	SS_OP_DATA	Send data.
	SS_OP_DISC_REQ	Request that the legacy connection be disconnected.
	SS_OP_DISC_CONF	Confirm disconnection of the legacy connection.
uint32_t dlen	Number of bytes of data that follow the SyncSockets header	
uint32_t opt	Single bit options for the message that can be set to zero in simple case (such as those used in the tutorial program)	
uint32_t var	A variable use integer field that can be set to zero in simple cases (such as those used by the tutorial program)	
uint8_t proto	Protocol type code set by the SyncSockets Daemon and saved by the tutorial/application program for use in subsequent outgoing messages The SyncSockets Daemon knows the legacy protocol identity by way of the configured connection name, but an application, such as the tutorial program, can operate perfectly well with no explicit knowledge of the protocol type.	
void *data	Not a header field, but a pointer to the data buffer sent following the SyncSockets header	

How Everything Works Together

To use a SyncSockets connection, first open a TCP connection to the SyncSockets Daemon, which listens for incoming TCP connections on a port number (8000 by default).

The SyncSockets API makes this easy by providing a `ss_open` function that performs all the hard work for you. You don't even need to read the `socket(2)` man page in Unix to understand how to use this function.

The `ss_open` function returns a file descriptor that is subsequently passed to the `ss_send_nh` and `ss_recv_nh` functions. The send function accepts the parameters values described in [SyncSockets Header Fields](#). The receive function accepts pointers to variables described in [SyncSockets Header Fields](#) and returns values from the received SyncSockets header.

To	Do This
Establish a legacy connection.	<ol style="list-style-type: none"> 1. Send a <code>SS_OP_CONN_REQ</code>. 2. Receive a <code>SS_OP_CONN_CONF</code>.
Disconnect a legacy connection.	<ol style="list-style-type: none"> 1. Send a <code>SS_OP_DISC_REQ</code>. 2. Receive a <code>SS_OP_DISC_CONF</code>.
Send or receive data bytes.	Use <code>SS_OP_DATA</code> .

Either the SyncSockets application or the SyncSockets Daemon can send messages; however, the SyncSockets Daemon sends `SS_OP_CONN_REQ` only if the SyncSockets application first sends `SS_OP_LISTEN` to the SyncSockets Daemon.

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 <code>tutorial.tar</code> .	1. Create a directory for <code>tutorial.tar</code> .
2. Copy <code>tutorial.tar</code> to that directory.	2. Copy <code>tutorial.tar</code> to that directory.
3. <code>cd</code> to that directory.	3. <code>cd</code> to that directory.
4. Type: <code>tar xf tutorial.tar</code>	4. Use your favorite unzip program to unzip <code>tutorial.tar</code> .
5. Type: <code>cd tutorial</code>	5. Type: <code>cd tutorial</code>

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
<code><gcom/sshdr.h></code>	Header file for SyncSockets message types and other constants
<code><gcom/ssapi.h></code>	Header file for SyncSockets API function definitions
<code>/usr/lib/gcom/ssapi.a</code> or <code>/usr/lib/gcom/ssapi.so</code>	SyncSockets API library file for Linux/Solaris platforms
<code>C:\windows\system32\ssapi.dll</code>	SyncSockets API library 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:

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

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

Running the Tutorial Program

Command Syntax:

tutorial *options*

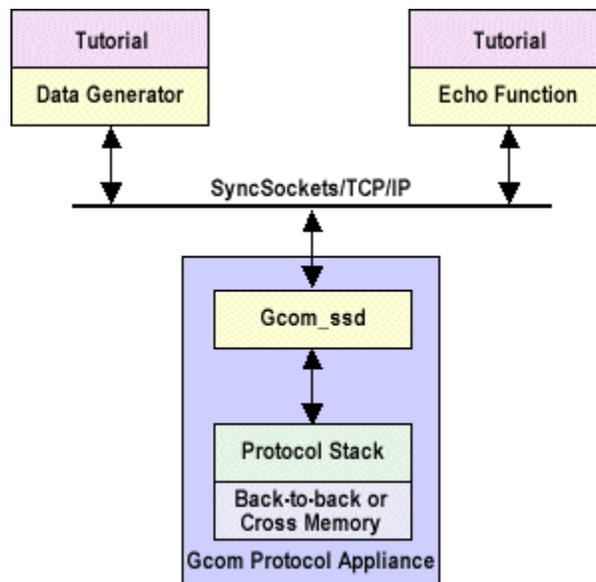
Command Options:

Option	Argument	Default	Purpose
-?	None		Print help text.
-b	Burst Size	1	Set the initial burst of messages to send.
-c	Connection Name	test_connection	Set the SyncSockets Daemon connection name to use in SS_OP_CONN_REQ or SS_OP_LISTEN.
-d	Debug Level	0	Set the SyncSockets API debug level. Use -1 for maximum verbosity.
-h	Host Name or IP Address	localhost	Set the host name or IP address of the machine on which the SyncSockets Daemon resides.
-i	None		Use listen instead of connect.
-I	Iteration Count	0	Set the number of messages to send/receive per connection. Use 0 for continuous operation.
-l (lower case L)	Log File Name	tutorial.log	Set the name of the log file into which the SyncSockets API writes messages.
-n	Number of Connections	1	Set the number of connections the tutorial program manages.
-p	Port Number	8000	Set the port number of the SyncSockets Daemon to which connections are made. (This parameter should not have to be set.)
-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.

Option	Argument	Default	Purpose
-T	Seconds	10	Set the timeout for connection setup to occur, and control the interval at which progress messages are printed.
-t	Test Type	0	Set to: <ul style="list-style-type: none"> • 0 for send/receive test • 1 for echo test
-v	None		Enable verbose mode.
-V	None		Enable verification of message data returned from the echo server (not yet implemented).

Running a Demonstration of the Tutorial Program

Establish the following test setup:



Procedure

1. On the Gcom Protocol Appliance, configure and run a legacy protocol stack (the simplest is an X.25 stack) in either cross memory loopback or in a two-port back to back test.
2. Note the:
SyncSockets Daemon configured connection names that can be used to establish a connection — for example, in the default cross-memory X.25 configuration, the two X.25 connection names are `x25_1` and `x25_2`.
IP address (or host name) of the Gcom Protocol Appliance on your local network
3. 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:

Data Generator:

```
tutorial -h IP-Addr-of-GPA -c x25_1 -n 5 -I 20000 -t 0 &
```

Echo:

```
tutorial -h IP-Addr-of-GPA -c x25_2 -n 5 -I 20000 -t 1 &
```

Results

The two programs connect to each other via the Gcom Protocol Appliance and protocol stack crossover connections, and start sending and receiving data.

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

```
Echo: send=6434 receive=6434 total=12868 rate=1440
WrRd: send=7169 receive=7163 total=14332 rate=1440
Echo: send=13543 receive=13543 total=27086 rate=1431
WrRd: send=14247 receive=14241 total=28488 rate=1428
```

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

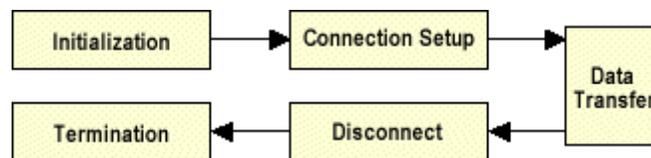
```
Echo: Test complete
Echo: send=600000 receive=600000 total=1200000 rate=1630

WrRd: Test complete
WrRd: send=600000 receive=600000 total=1200000 rate=1630
```

Tutorial Architecture

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

Tutorial Program Execution Flow



Phase	Description
Initialization	Occurs in the <code>main</code> routine. Includes: <ul style="list-style-type: none"> • Interpreting the command line arguments • Allocating the poll list and the connection list tables • Initializing the SyncSockets API • Initializing the transmit data buffer with a counting pattern
Connection Setup	Initiated from the <code>make_connections</code> routine. Includes: <ul style="list-style-type: none"> • Establishing TCP connections with the SyncSockets Daemon • Sending out the <code>SS_OP_CONN_REQ</code> or <code>SS_OP_LISTEN</code> messages • Setting up the connection list entries' callout functions to process the expected response from the SyncSockets Daemon • Calling the <code>run_plist</code> routine to operate the connections in a poll-driven fashion <p>The <code>make_connections</code> returns and the tutorial program enters the Data Transfer phase after all connections are established.</p>
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"> • Setting up the connection list entries' callout functions according to the type of test being run • Calling the <code>run_plist</code> routine to operate the connections • Printing progress messages at regular intervals <p>A connection enters the Disconnect phase when it reaches its iteration count.</p>

Phase	Description
Disconnect	<p>Initiated from several places in the tutorial program that check to see if the iteration count is reached. For each connection that reaches its iteration count:</p> <ul style="list-style-type: none"> • Call the <code>issue_disc_req</code> routine to send a <code>SS_OP_DISC_REQ</code>. • Set the read callout function to a routine that expects to receive a <code>SS_OP_DISC_CONF</code>. <p>The tutorial program enters the Termination phase after all connections are disconnected.</p>
Termination	<p>Consists of a return from the <code>run_plist</code> routine when the poll list processing is complete and all connections are disconnected. The tutorial program prints one last progress message and then exits.</p>

Data Structures and Poll List Handling

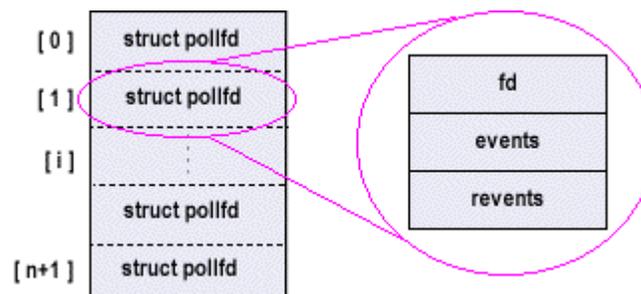
To handle multiple file descriptors in an application driven by spontaneously arriving input from any subset of the files, you must create some kind of enumerated list that identifies:

- File descriptors
- State of the file in which the application is interested – the two common cases:
 - 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?
 - 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:

`plist` is the poll list

`num_connections` is the number of elements in the poll list

`1000` (one second) is the timeout duration

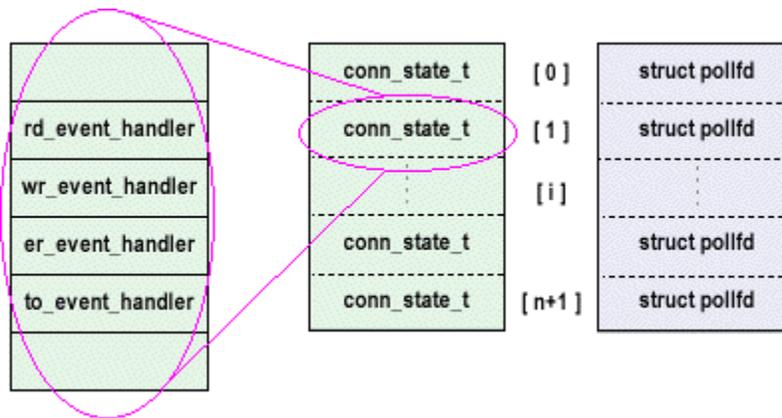
As you may surmise from the name of the variable, `poll(2)` returns the number of file descriptors that meet the requested conditions. If `nfds` is:

- 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 `nfd`s 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` function pointers to functions 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.

- 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`. Gcom's SyncSockets API library for Windows contains an implementation of `poll` that is written in terms of `select`. 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.
- The tutorial program is a single-threaded application. You have a tricky coordination problem if you have a multi-threaded application and one thread is operating a poll list while other threads are making changes to it. The problem involves the fact that the operating system copies the poll list from the application down into the OS, operates on it, and then copies it back just before returning to the application. So any changes to the poll list made by a thread running in parallel are overwritten. There are solutions to this problem but they are beyond the scope of this introduction to `poll(2)`.

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 SyncSockets application that receives data from some sort of network connection and you want to write it to another (say a SyncSockets) 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 SyncSockets API Function Calls

The tutorial program uses only five SyncSockets API function calls:

SyncSockets API Function	Purpose
<code>ss_init</code>	Set the name of the SyncSockets API log file and log option bits. This is the first call to the SyncSockets API and is best made early in the main program.
<code>ss_open</code>	Open a TCP connection to the SyncSockets Daemon. The tutorial program passes the hostname/IP-address of the Gcom Protocol Appliance running the SyncSockets Daemon. The tutorial program's use of this function is just about as sophisticated as any SyncSockets application will ever need.
<code>ss_close</code>	Close a connection – the opposite of <code>ss_open</code> .
<code>ss_send_nh</code>	Send a SyncSockets message over an open TCP connection.
<code>ss_recv_nh</code>	Receive a SyncSockets message from an open TCP connection.

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

You can write a sophisticated SyncSockets 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 SyncSockets API to construct a set of routines that send certain types of SyncSockets messages and manage the Connection Setup, Data Transfer and Disconnect phases.

Routine	Purpose
<code>disc_complete</code>	Called whenever a disconnect sequence completes In the tutorial program, this routine closes the SyncSockets TCP connection. In a production SyncSockets application, it is perfectly permissible to leave a connection open and re-use it for another legacy connection.
<code>echo_data</code>	Write-ready callout routine that sends the message residing in the receive buffer inside the clist slot on which it is operating Used during the Data Transfer phase of the echo test. This is how the message gets echoed.
<code>issue_conn_conf</code>	Send a <code>SS_OP_CONN_CONF</code> over an open SyncSockets TCP connection. Used after receiving a <code>SS_OP_CONN_REQ</code> . Sending the <code>SS_OP_CONN_CONF</code> completes the connection setup.

Routine	Purpose
issue_conn_req	Send a <code>SS_OP_CONN_REQ</code> over an open SyncSockets TCP connection. Used when your SyncSockets application anticipates initiating a legacy connection to the remote device.
<code>issue_disc_conf</code>	Send a <code>SS_OP_DISC_CONF</code> in response to receiving a <code>SS_OP_DISC_REQ</code> . This completes the disconnect sequence and leaves the SyncSockets TCP connection open.
<code>issue_disc_req</code>	Send a <code>SS_OP_DISC_REQ</code> over an open SyncSockets TCP connection. This disconnects the legacy connection while leaving the SyncSockets TCP connection open.
<code>issue_listen</code>	Send a <code>SS_OP_LISTEN</code> over an open SyncSockets TCP connection. Used when a SyncSockets application anticipates receiving a legacy connection from the remote device.
open_ss_connection	Open a SyncSockets TCP connection. Returns a file descriptor placed into the poll list and used in subsequent SyncSockets API function calls pertaining to the just-opened socket.
<code>process_disc_req</code>	Print any ASCII disconnect message text that may accompany the <code>SS_OP_DISC_REQ</code> . Called if a <code>SS_OP_DISC_REQ</code> is received.
<code>read_data</code>	Read-ready callout routine Used during the data transfer phase for both the data generator test and the echo test.
<code>recv_conn_conf</code>	Read-ready callout routine Used during the connection setup phase when a SyncSockets application anticipates receiving a <code>SS_OP_CONN_CONF</code> to complete the legacy connection setup.
<code>recv_conn_req</code>	Read-ready callout routine Used during the connection setup phase when a SyncSockets application anticipates receiving a <code>SS_OP_CONN_REQ</code> in response to a previously sent <code>SS_OP_LISTEN</code> .
<code>recv_disc_conf</code>	Read-ready callout routine Used during the disconnect phase to receive a <code>SS_OP_DISC_CONF</code> in response to a previously sent <code>SS_OP_DISC_REQ</code> .
recv_msg	Receive a SyncSockets message on a TCP socket. Called when the socket is known to be read-ready.
send_msg	Send a SyncSockets message on a TCP socket whose file descriptor was returned by <code>open_ss_connection</code> . Called when the socket is known to be write-ready.
<code>write_data</code>	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.

Selected Tutorial Program Routines

- [send_msg](#)
- [recv_msg](#)
- [open_ss_connection](#)
- [issue_conn_req](#)

send_msg

Purpose: Send a SyncSockets message of a given type on the indicated file.

The routine itself supplies some default values for some header fields in the SyncSockets message and, thus, is tailored to the needs of the tutorial program (which has no need to control the values of these other fields).

Parameters

Parameter	Description
int inx	Index into plist/clist
int opcode	SyncSockets operation code to send
char *message	Data portion of the message
int msglen	Number of data bytes

Code Listing

Code	Purpose
<code>conn_state_t *c = &clist[inx] ;</code>	Set up a pointer to the clist slot for ease of access.
<code>c->state = ss_send_nh(plist[inx].fd, opcode, msglen, 0, 0, c->proto, message);</code>	Call the <code>ss_send_nh</code> function to send the message. Note: The file descriptor comes from the plist. The <code>proto</code> field comes from a value stored in the clist by the <code>recv_msg</code> routine.
<code>return(c->state);</code>	Return the value returned <code>ss_send_nh</code> . This is the state of the SyncSockets connection.

recv_msg

Purpose: Receive a SyncSockets message from the indicated file.

The SyncSockets header field values are captured in the clist entry for the file.

Parameters

Parameter	Description
int inx	Index into plist/clist

Code Listing

Code	Purpose
<code>conn_state_t *c = &clist[inx] ;</code>	Set up a pointer to the clist slot for ease of access.

Code	Purpose
<code>c->rlen = sizeof(c->buf) ;</code>	Set up to receive a SyncSockets message. The <code>rlen</code> field is set to the size of the receive buffer inside the <code>clist</code> entry. The <code>ss_recv_nh</code> routine uses this value to limit the amount of returned data.
<code>c->rop = 0xFF ;</code>	Set the <code>rop</code> field to an invalid value for a SyncSockets operation code so there is no mistake about what was read if the call to <code>ss_recv_nh</code> fails.
<code>errno = 0 ;</code>	Ensure that if <code>errno</code> gets set, it is because of <code>ss_recv_nh</code> and not some residual value.
<code>c->state = ss_recv_nh(plist[inx].fd, &c->rop, &c->rlen, &c->ropt, &c->rvar, &c->proto, c->buf) ;</code>	Call <code>ss_recv_nh</code> to receive the SyncSockets message. The header components are captured in the <code>clist</code> entry. Any data field value is stored in the receive buffer within the <code>clist</code> entry. The caller of <code>recv_msg</code> can examine these fields.
<pre>if (c->state < 0) { if (errno != ECONNRESET) { print_perror(inx, "ss_recv error") ; } return(-1) ; }</pre>	Check for error return. This code simply provides the ability to print a message.
<code>return(c->state) ;</code>	Return the state of the SyncSockets connection or -1 if an error occurred. Note: Recovery from signals is the responsibility of the caller.

open_ss_connection

Purpose: Open a TCP/IP connection to the SyncSockets Daemon running on a Gcom Protocol Appliance.

Global parameters obtained from the command line are used to construct the host address and port number.

Parameters

None

Code Listing

Code	Purpose
<pre>int cfd = -1 ; char openloc[256];</pre>	The <code>cfd</code> variable will be the file descriptor for the connection. <code>openloc</code> is where the hostname and port number are constructed.

Code	Purpose
<code>sprintf(openloc, "%s:%s", host, port);</code>	The format of the host name and port calls for the two to be coded in ASCII and separated by a semicolon (:).
<pre>If ((cfd = ss_open(openloc, SS_O_CLIENT, NULL, NULL)) > 0) { return(cfd); }</pre>	<p>Call <code>ss_open</code> to establish the TCP/IP connection. If the call succeeds, return the file descriptor.</p> <p>The first parameter is the host:port. The second parameter says the tutorial program wants to be a client. The last two parameters are only used when opening in host mode and thus can be <code>NULL</code> for our purposes.</p> <p>Note: <code>ss_open</code> blocks until the TCP/IP connection is established.</p>
<code>return(-1);</code>	If control reaches this point, the call to <code>ss_open</code> failed.

issue_conn_req

Purpose: Send a `SS_OP_CONN_REQ` on the SyncSockets TCP connection.

This routine illustrates the simple method for constructing and sending connect requests.

Parameters

Parameter	Description
<code>int inx</code>	Index into <code>plist/clist</code>

Code Listing

Code	Purpose
<code>conn_state_t *c = &clist[index] ;</code>	Set up a pointer to the <code>clist</code> slot for ease of access.
<code>int result = 0;</code>	Local variable to hold return code from calling the <code>send_msg</code> routine.
<code>sprintf(c->name, conn_name, index+1) ;</code>	<p>Derive the name of the connection and save it in the <code>clist</code> entry.</p> <p>The <code>conn_name</code> variable is set from the command line and is the name of a configured connection the targeted SyncSockets Daemon has in its tables.</p> <p>The idea of passing the <code>index+1</code> value is that the connection name could be of the form <code>link_%d</code> and lead to a series of connection names - <code>link_1</code>, <code>link_2</code>, etc. - for each separate connection. This let you use the tutorial program to test non-cloned connection types such as link layer connections or SNA LU sessions.</p>
<pre>result = send_msg(index, SS_OP_CONN_REQ, c->name, strlen(c->name) + 1)</pre>	<p>Call the internal <code>send_msg</code> routine to send the <code>SS_OP_CONN_REQ</code>.</p> <p>The connection name goes in the data portion of the message. Although it is not necessary to do so, the tutorial program sizes the data field to include the ASCII NUL on the end of the string.</p>

Code	Purpose
Return	The actual code tests the result to see if an error occurred and, if so, returns -1. Otherwise it returns 0.

Advanced Topics

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

Note: The [SyncSockets Guide](#) is the definitive reference for using these constructs. Please consult it before using anything defined in this section.

- [Logging Options](#)
- [Log File Handling](#)
- [Connection Types](#)
- [Extended Connect Requests](#)
- [Special Operations](#)
- [Option Field Values](#)
- [Host Mode Operation](#)
- [Encryption](#)

Logging Options

When you call the `ss_init` function, one of the passed parameters is a set of logging options. This quantity is actually a bit-encode word, the bits of which are defined in the `<ssapi.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 SyncSockets API contains several routines for managing the log file produced by the API:

Routine	Purpose
<code>ss_printf</code>	A printf-like routine that writes a message into the log
<code>ss_lock_print</code>	Lock the log so the SyncSockets application can call <code>ss_printf</code> multiple times without any other thread writing to the log in the interim.
<code>ss_unlock_print</code>	Unlock the log.
<code>ss_strstate</code>	Convert a SyncSockets state value to ASCII for printing.
<code>ss_strop</code>	Convert a SyncSockets operation field value to ASCII for printing.
<code>ss_strsop</code>	Convert a SyncSockets special operation code to ASCII for printing.

Routine	Purpose
<code>ss_set_log_size</code>	Set the size of the log and the wrap point. Used to enable the circular log-handling mechanism within the SyncSockets API. Typical use: Use <code>ss_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.

Connection Types

The SyncSockets API can communicate using connection types other than TCP/IP. For example: It can communicate using Unix domain sockets.

The use of any connection type other than TCP is strongly discouraged. The SyncSockets Daemon uses only TCP connections.

Extended Connect Requests

The tutorial program uses the simple form of the `SS_OP_CONN_REQ` message. There is an extended form that provides more control over the parameters of the legacy connection.

The extended forms all contain a data structure in the `data` field of the `SS_OP_CONN_REQ` message. The first field of the structure is always the connection name - for compatibility with the simple form. The remaining fields set parameters for the legacy connection.

To use the extended form, a SyncSockets application must know the legacy protocol type ahead of time - because each legacy protocol has a different structure for its extended form. This loss of generality is the price paid for increased control.

The following is a list of extended form data structures and their associated protocols. Please refer to the [SyncSockets Guide](#) for usage details.

Note: All of the fields of these structures are specified as ASCII strings. This avoids byte order and word length problems that may occur when dissimilar types of systems run a SyncSockets application and the SyncSockets Daemon.

Structure	Protocol
<code>ss_op_x25_conn_req_data_t</code>	X.25
<code>ss_op_raw_dial_conn_req_data_t</code>	Raw mode dial control
<code>ss_op_pcap_conn_req_data_t</code>	Libpcap connection

Special Operations

The `SS_OP_SPECIAL` operation code, which permits extended protocol-specific vocabulary, is not discussed in the tutorial at all. When a SyncSockets application knows the protocol ahead of time, it can use special operations to increase control over the legacy protocol.

Please consult the [SyncSockets Guide](#) for the complete list of special operations. We will call attention to a few of them here.

The `SS_SOP_TEST_REQ` and `SS_SOP_TEST_CONF` operations can often be useful. When the SyncSockets Daemon receives a `SS_SOP_TEST_REQ`, it passes it on to the legacy protocol module, which then returns the same data in a `SS_SOP_TEST_CONF`. Thus, you can use this message to create a *heart beat* between a SyncSockets application and the SyncSockets Daemon. In redundant systems, you can use this message to verify the communication pathway to the Gcom Protocol Appliance is still functional.

The `TEST` operations can even be used with no SyncSockets connection active. In this case, the SyncSockets Daemon itself answers the message because there is no legacy protocol to which it can be passed. This can serve as a heart beat mechanism.

You can send the `SS_SOP_LINE_STATUS` operation code on a legacy connection to obtain the state of the modem signals on the serial line over which the legacy connection is operating. It also returns the state (up/down) of any link layer entity that may be running on the line.

Option Field Values

We have not discussed the single bit options that can be set into the `opt` field in the SyncSockets header. These bits are dependent upon the legacy protocol. Some of the options are specified without regard to protocol type and different legacy protocols interpret them differently.

For example, `SS_OPT_DELIVERY_CONFIRMATION` is used to tag a `SS_OP_DATA` the SyncSockets application wants acknowledged via the return of a `SS_SOP_DATA_ACK` message. Different legacy protocols participate in this mechanism in different manners.

The `SS_OPT_MORE_DATA` bit is used to indicate that this `SS_OP_DATA` is part of a longer logical message. Once again, different legacy protocols interpret this bit differently (or not at all).

A particularly interesting option is `SS_OPT_X25_SLOW_CHANNEL`. This is an X.25 only option. Setting it in any `SS_OP_DATA` message has the effect of converting the X.25 virtual circuit into what is called a *slow channel*. Data sent on a slow channel are only sent one packet at a time and then only to utilize what would otherwise be idle line time. This mechanism is quite useful for *trickle-feed* file transfer that is sandwiched between higher-priority interactive data traffic.

Please consult the [SyncSockets Guide](#) for the full list of option field values.

Host Mode Operation

The tutorial program illustrates how a SyncSockets client interfaces to the SyncSockets Daemon, which is acting as a SyncSockets host; however, it is not too difficult to write a

SyncSockets application that acts as a SyncSockets host itself. Such an application could use the SyncSockets protocol as a convenient client/server protocol.

For example, the Gcom Remote Line Monitor contains a SyncSockets host application that acts as a configuration server. Ethereal clients connect to this server to configure the line monitor ports.

Encryption

Use the `ss_crypt` function to set the encryption mode and encryption key on a SyncSocket. Once encryption mode has been established, all `SS_OP_DATA` message data fields are automatically encrypted when sent and decrypted when received.

The `ss_crypt` function is called after the call to the `ss_open` function returns the file descriptor. The file descriptor is passed to the `ss_crypt` function with a mnemonic specifying the encryption and number of encryption keys.

At the moment, the only supported encryption mode is DES with either 1, 2, or 3 keys.

There are three ways to use encryption:

- Encryption between the SyncSockets application and the SyncSockets Daemon – The SyncSockets Daemon decrypts the data and sends it on the legacy connection as clear text. To accomplish this, set the encryption parameters for the configured connection in the SyncSockets Daemon configuration file so that it knows how to decrypt the data.
- Encryption between two SyncSockets applications – In this case, you have two SyncSockets applications communicating directly with each other, with no SyncSockets Daemon involvement. One SyncSockets application acts as a host (listening on a TCP socket) and the other as a client. Each SyncSockets application must know the key ahead of time.
- Encryption between a SyncSockets application and a host computer – The SyncSockets Daemon connection definition does NOT include encryption parameters, so the encrypted data is placed into the legacy protocol. The host application at the other end of the legacy connection decrypts the data. The host and the SyncSockets application must agree on the encryption method and keys.