

S*TREAMS Bisync API*

© 1998 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.

Rsystem is a registered trademark of GCOM, Inc. Macintosh is a registered trademark of Apple Computer, Inc. FrameMaker is a trademark and registered trademark of Frame Technology Corporation. UNIX is a registered trademark of UNIX Systems Laboratories, Inc. in the U.S. and other countries. SCO is a trademark of the Santa Cruz Operation, Inc. MS-DOS, Visual C++, Microsoft and Windows are registered trademarks of Microsoft Corporation. IBM PC, IBM PC/AT and PC DOS are registered trademarks of International Business Machines Corporation. All other brand product names mentioned herein are the trademarks or registered trademarks of their respective owners.

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.

This manual was written, formatted, indexed, and produced by Technical Writer Geoff Gerriets using VIM 5.0a, FrameMaker 5.0, and CorelDRAW! 7 on Microsoft Windows95 and RedHat Linux 4.2 platforms. The source material was gathered by interviewing subject matter specialists Dave Grothe, Carleton Mills and Mikel Matthews.

This manual was printed in the U.S.A.

FOR FURTHER INFORMATION

If you want more information about GCOM products, contact us at:

GCOM, Inc.
1800 Woodfield
Savoy, IL 61874
(217) 351-4241
FAX: (217) 351-4240
e-mail: support@gcom.com
homepage: <http://gcom.com>



GCOM is committed to the conservation of America's natural resources.

CONTENTS

SECTION 1	7	<i>Introduction</i>
SECTION 2	9	<i>Preparing to Communicate: the Bind Phase</i>
	9	Initialize the API
	10	Opening the Data Stream
SECTION 3	11	<i>Establishing Communication: the Connect Phase</i>
	11	Address Format
	12	Establishing the Connection
SECTION 4	13	<i>Communications: the Data Phase</i>
	13	Implementation Model
	13	API Routines for Data Transfer
	16	Handling Normal Data
	17	Handling Qualified Data
SECTION 5	21	<i>Shutting Down: Disconnect and Close</i>
	21	Hanging Up: the Disconnect Phase
	21	Releasing the Address: the Unbind Phase

SECTION 1

Introduction

This manual documents Gcom's application programming interface for the Bisync protocol.

Gcom's implementation of the Bisync protocol interacts with the user through a Network Provider Interface (NPI) module. The Bisync protocol itself provides network and link-layer services, using Gcom's intsx25 line driver or a comparable CDI-level interface to communicate with a physical device.

Users of Gcom's Bisync develop to Gcom's NPI API. The NPI API is designed around the model of an X.25 virtual circuit. For Bisync, this virtual circuit concept is mapped onto a Bisync station. Calls are handled on an outgoing basis, and the data stream, once established, is formatted with Bisync control characters.

This API provides a wide variety of routines for managing and analyzing network connections. The remainder of this manual will explore how the NPI API is employed when communicating via Bisync.

SECTION 2

Preparing to Communicate: the Bind Phase

The process of call setup has a few distinct steps. The first of these steps is initializing the API.

Initialize the API

npi_init()

```
int npi_init(int log_optns, char *log_name);
```

A call to the ***npi_init()*** routine leads off all NPI API applications. This sets up the data structures within the API and configures the degree of logging.

The ***log_optns*** parameter is a bitwise OR of the various logging options.

The ***log_name*** parameter is a standard ASCII string to use as the logfile's name.

In the simplest possible case, this is the only real preparation that must be performed. Following the ***npi_init()***, a call to ***npi_connect()*** will return the file descriptor needed for the majority of additional communication. In cases where multiple lines will be used or in complex applications, it becomes more practical to open the data streams directly and manage them directly. In this event, the next step is to open the data stream.

Opening the Data Stream

npi_open_data()

```
int npi_open_data(void);
```

A call to the ***npi_open_data()*** routine is required for each unique Bisync station that the application will manage. The ***npi_open_data()*** call will return a file descriptor for the opened data stream. This file descriptor (sometimes called 'fid') will be used throughout the application to identify the station attached to it.

In addition to opening the stream, the application will need to bind it to an empty address.

npi_bind_ascii_nsap()

```
int npi_bind_ascii_nsap(int strm, char *bind_ascii_nsap,
                       int conind_nr, unsigned flags);
```

This routine is used to bind a stream to a particular network address so that it will accept incoming calls. In Bisync applications, this really has no real effect on the operation, but it is required to advance the stream's state so that it will be ready to connect.

The ***strm*** parameter is the file descriptor of an open NPI data stream.

The ***bind_ascii_nsap*** string is an address pattern to bind to. Since Bisync calls are handled as outgoing connections, this parameter should be an empty string ("").

The ***conind_nr*** parameter indicates how many connections should be queued and awaiting the application's attention before rejecting further connection requests. Again, since Bisync connections are outbound, this parameter's meaning is limited. It must be a 0.

For the purposes of a Bisync connection, the ***flags*** parameter is not useful. Possible values are enumerated fully in ***npi.h***.

SECTION 3

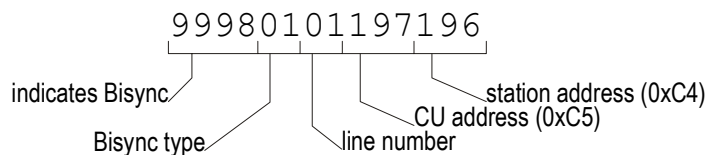
Establishing Communication: the Connect Phase

The actual connection can be made in any number of ways. The most trivial case will involve simply placing a call to *np_i_connect()*, while a more complicated scenario — one in which *np_i_open_data()* has been called — might employ either *np_i_connect_req()* or the non-blocking version, *np_i_send_connect_req()*.

Address Format

Regardless of which technique an application employs for establishing a connection, the address used to connect will be formatted the same way.

A Bisync address looks like the following::



The first four digits of the Bisync address will always be 9998. This identifies the address as a Bisync address. The next two digits are a code indicating the Bisync type: 01 is 3270 Bisync while 03 is 3780 Bisync. The next two digits (the seventh and eighth) are the line number. In 3270, this address is followed by a three-digit representation of the CU address (translated into base 10) and a three digit representation of the station address (also translated into base 10).

Establishing the Connection

npi_connect()

```
int npi_connect(char *remote_sap, unsigned bind_flags);
```

The simplest possible means of connecting, ***npi_connect()*** takes an address and a set of bind flags and attempts to establish a connection with the specified Bisync station. The routine will block until the connection is established then return a file descriptor for the open file.

The ***remote_sap*** parameter is an ASCII address to connect to.

The ***bind_flags*** parameter is a bitwise OR of the flags to use in the N_BIND_REQ. Bisync connections don't support any of the flags ordinarily used here.

npi_connect_req()

```
int npi_connect_req(int strm, char *peer_sap, char *buf,
                    int cnt);
```

A step more complicated. This routine blocks like ***npi_connect()***, but it requires a file descriptor already open and bound. It also provides the opportunity to send data with the connect request. This data-passing option is really only useful in X.25.

The ***strm*** parameter is the file descriptor of the open data stream.

The ***peer_sap*** parameter is a standard ASCII string containing an address to connect to.

The ***buf*** parameter points to a buffer of data to send with the request. The ***cnt*** parameter is the number of bytes in the data buffer. For Bisync, ***buf*** should be NULL and ***cnt*** should be -1.

npi_send_connect_req()

```
int npi_send_connect_req(int strm, char *peer_sap,
                         char *buf, int cnt);
```

The most convoluted of the routines, this initiates a connection but does not wait for the connection to be established before returning. The success or failure of the connection attempt must be determined and handled by the application. This routine also requires a file descriptor opened in advance.

Parameters for this routine are identical to the parameters for ***npi_connect_req()***.

SECTION 4

Communications: the Data Phase

Implementation Model

Gcom's implementation of Bisync separates normal data from control messages. Control information gets handled using X.25's qualified data (Q-bit) mechanism, while normal data is in regular (non-Q-bit) packets. This provides a simple mechanism for handling data transfer, and simultaneously provides control information to the user.

API Routines for Data Transfer

Receiving Data

The NPI API provides two major routines for reading data from a stream. The simplest, ***npi_read_data()***, expects to return a buffer of regular data. The other routine, ***npi_rcv()***, provides a greater wealth of calling options and also a greater presents a more complex interface.

npi_read_data()

```
int npi_read_data(int strm, char *buf, int cnt);
```

The standard read routine is ***npi_read_data()***. This routine will read data, N_DATA_IND messages, and N_DISC_IND messages from a data stream.

The ***strm*** parameter is the file descriptor for the open and connected stream. The ***buf*** parameter points to the data buffer, and ***cnt*** is how many bytes the data occupy.

npi_rcv()

```
int npi_rcv(int strm, char *buf, int cnt,
           long flags_in, long *flags_out);
```

npi_rcv() is the more complex means of reading data. It handles both data and protocol messages for an application, and takes a wide variety of options.

The ***strm*** parameter is the file descriptor of the open and connected stream. Once again, ***buf*** points to a data buffer and ***cnt*** indicates the amount of data the buffer can contain.

The ***flags_in*** parameter is a bitwise OR of the following options: NPIAPI_USER_DATA_ACK indicates that the application will handle sending its own ACKs, and NPIAPI_USER_RESET_RES indicates that the application will handle returning a reset response when a reset indication is received.

The ***flags_out*** parameter can be a bitwise OR of any of the following: NPIAPI_MORE_DATA indicates that data was received with the N_MORE_DATA_FLAG set, NPIAPI_RC_FLAG indicates that data was received with the N_RC_FLAG set, N_X25_Q_BIT indicates that qualified data was received, and NPIAPI_FRAGMENT indicates that the buffer contains a fragment of the incoming NSDU and more fragments are expected.

The return values from ***npi_rcv()*** are also rich in vocabulary. The following table summarizes the error returns and the success returns presented by ***npi_rcv()***

<i>Return Value</i>	<i>Significance</i>
<i>NPIAPI_NORMAL_DATA</i>	normal data received
<i>NPIAPI_EXPEDITED_DATA</i>	expedited data was received
<i>NPIAPI_DATA_ACK</i>	acknowledgement from a previous send with the N_RC_FLAG set was received
<i>NPIAPI_DISC_IND</i>	the connection was disconnected and data may be present
<i>NPIAPI_RESET_INDICATION</i>	N_RESET_IND was received
<i>NPIAPI_RESET_COMPLETE</i>	reset sequence has completed and the application can resume data transfer
<i>NPIAPI_CONNECT_COMPLETE</i>	a connection has been established and data may be present
<i>NPIAPI_CONNECT_IND</i>	a connection has been established and data may be present
<i>NPIAPI_BIND_ACK</i>	previously sent N_BIND_REQ was accepted by the NPI provider

Table 1 ***npi_rcv()*** return values

<i>Return Value</i>	<i>Significance</i>
<i>NPIAPI_INFO_ACK</i>	an information acknowledgement in response to an information request from the application
<i>NPIAPI_ERROR_ACK</i>	a previously transmitted message is being rejected
<i>NPIAPI_OTHER</i>	an unsupported NPI message received
<i>NPIAPI_NO_NOTHING</i>	(error) contains neither control nor data
<i>NPIAPI_PARAM_ERROR</i>	(error) a parameter error occurred; flags pointer is NULL or buffer pointer is NULL or buffer length is non-positive
<i>NPIAPI_NOT_INIT</i>	(error) NPI was not initialized with <code>np_i_init()</code>
<i>NPIAPI_GETMSG_ERROR</i>	(error) the file descriptor (<i>strm</i>) was not usable
<i>NPIAPI_EAGAIN</i>	(error) no data or control message available; used with nonblocking I/O

Table 1 *np_i_rcv()* return values

Transmitting Data

The NPI API provides two major routines for data transmission. The first, *np_i_write_data()*, is the simplest of the two. *np_i_put_data_proto()* is slightly more complicated in that it allows for flags to accompany the data.

np_i_write_data()

```
int np_i_write_data(int strm, char *buf, int cnt);
```

The *np_i_write_data()* routine is the standard routine employed when transmitting data. After formatting a block of data for transmission, passing a pointer to it to this routine will send it on its way. This routine cannot be used to send control information.

Here the *strm* parameter is the file descriptor of a connected stream, *buf* points to a buffer of data, and *cnt* is the number of bytes of data the buffer can hold.

np_i_put_data_proto()

```
int np_i_put_data_proto(int strm, char *data_ptr,
                       int data_lgth, long flags);
```

This routine is similar to *np_i_write_data()* with the addition of a *flags* parameter. A 0 value in the *flags* parameter will produce the same result as an *np_i_write_data()*. To set the Q-bit, the `N_X25_Q_BIT` define can be used in the *flags* parameter.

Handling Normal Data

Buffer Formats

Buffers of normal data prepared by the application will use special Bisync control characters to indicate whether the buffer should be transmitted using non-transparent mode (in which case the buffer will contain no embedded control characters) or if the Bisync engine must escape the data (transparent mode). Similarly, the application can also choose to indicate that a block of data is part of a larger unit or whether it's the last or only block in the series.

Buffers that the application receives from the Bisync engine will be formatted just as if they were properly prepared according to these conventions. In other words, a received buffer could be echoed with no modification to the data.

The control characters are used in a bracketing fashion, forming the first and last one or two bytes in a buffer. The following example illustrates four buffer styles. might be how one would prepare a buffer of data of length *cnt* that is pointed to by *buf*

	buf[0]		intermediary bytes		buf[cnt-1]
for initial or middle blocks	STX		data that doesn't need escaping		ETB
for final blocks	STX		data that doesn't need escaping		ETX
	buf[0]	buf[1]	intermediary bytes	buf[cnt-2]	buf[cnt-1]
for initial or middle blocks	DLE	STX	data that needs escaping	DLE	ETB
for final blocks	DLE	STX	data that needs escaping	DLE	ETX

In the first two examples, the first byte (*buf[0]*) is an STX. The second and subsequent bytes are data that don't require escaping. The final byte in each buffer (*buf[cnt-1]*) is either an ETB or an ETX, depending on whether the block is at the end of a message (signified with an ETX) or not. These two examples illustrate a buffer prepared for non-transparent mode. The data contained in either of these buffers must not contain embedded control code.

The next two examples illustrate transparent mode. A buffer prepared for transparent mode begins with a DLE followed by STX. The buffer ends with another DLE followed by either ETB or ETX, depending on whether the block is at the end of a message (signified by ETX) or not.

EBCDIC Characters in Normal Data

The following list of defines has been prepared especially for Bisync. These defines stand in for the EBCDIC characters that may occur in a standard data block.

NPI_E_DLE	Escape character
NPI_E_STX	Start text
NPI_E_ETX	End text
NPI_E_ETB	End block

Handling Qualified Data

Buffer Formats

Qualified data can come in one of two buffer formats. The first format begins with 0x07 and contains an additional character of protocol information. The second begins with 0x05 and contains an additional character indicating an error type. The errors may also contain additional meaningful data beyond those two bytes.

Defines for Protocol Characters

These defines can be matched against the second byte of a qualified-data buffer that begins with 0x07.

NPI_DQ_rti	Request to initiate
NPI_DQ_pti	Permission to initiate
NPI_DQ_ack	Acknowledgement
NPI_DQ_eot	End of transmission
NPI_DQ_rvi	RVI
NPI_DQ_eotc	EOT confirmation
NPI_DQ_rvi_ack	RVI and ACK
NPI_DQ_deot	Disconnect

Defines for Error Characters

These defines can be matched against the second byte of a qualified-data buffer that begins with 0x05. Additional information may be available beyond the initial two bytes for some errors.

NPI_DQ_err_no_info	No additional info
NPI_DQ_err_bid_retry	Retry exhausted bid
NPI_DQ_err_ENQ_sent	Too many ENQ's sent
NPI_DQ_err_ENQ_rcvd	Too many ENQs received
NPI_DQ_err_NAK_rcvd	Too many NAKs received
NPI_DQ_err_tymo_rcv	Time out rcv
NPI_DQ_err_ttd_retry	Too many TTD's
NPI_DQ_err_wack_retry	Too many WACK's
NPI_DQ_err_data_wrong_st	data in wrong state
NPI_DQ_err_dq_wrong_st	Q-bit in wrong state

Receiving Qualified Data

The NPI API provides two standard methods for receiving qualified data. The first and most direct employs ***npi_read_data()*** to retrieve a buffer of data, then checks the API's global variables to see if anything special accompanied the received data. The second method is slightly more involved, and uses ***npi_rcv()*** to retrieve the data buffer, analyzing the return value and the flags passed back to determine whether the data is normal or qualified.

npi_read_data()

The following code fragment illustrates distinguishing qualified (Q-bit) data from normal data using ***npi_read_data()***. The API's global variable ***npi_ctl_cnt*** is used to check the size of the protocol messages accompanying the data. If the data is qualified data, an N_DATA_IND will have been received, and the N_X25_Q_BIT flag will be set:

```
#include <sys/npi.h>
#include <gcom/npiext.h>

nbytes = npi_read_data(fid, mybuf, sizeof(mybuf));
if (npi_ctl_cnt == sizeof(N_DATA_IND))
{
    /* data indication received. Q-bit? */
    N_data_ind_t *p = (N_data_ind_t *) npi_ctl_buf;
    if (p->DATA_xfer_flags & N_X25_Q_BIT)
    {
        /* Q-bit is set */
    }
    else {
        /* Q-bit is not set */
    }
}
}
```

npi_rcv()

The following code fragment illustrates distinguishing qualified (Q-bit) data from normal data using ***npi_rcv()***. The ***flags*** parameter is examined after the call to see if the N_X25_Q_BIT flag has been set.

```
#include <gcom/mpiext.h>
rslt=npi_rcv(fid, mybuf, sizeof(mybuf), 0, &flags);
if (rslt == NPIAPI_NORMAL_DATA)
{
    if (flags & N_X25_Q_BIT)
    {
        /* Q-bit set */
    } else {
        /* Q-bit not set */
    }
}
```

Sending Qualified Data

The NPI API really only provides a single option for sending qualified data. The ***npi_put_data_proto()*** routine handles sending qualified data through the simple mechanism of allowing flags to accompany the data.

npi_put_data_proto()

The following code fragment illustrates using ***npi_put_data_proto()*** to send qualified (Q-bit) data. The process is a simple call with the N_X25_Q_BIT define used for flags.

```
#include <sys/mpi.h>
#include <gcom/mpiext.h>

rslt = npi_put_data_proto(fid, buf, sizeof(buf),
N_X25_Q_BIT);
```


SECTION 5

Shutting Down: Disconnect and Close

Hanging Up: the Disconnect Phase

npi_discon_req()

```
int npi_discon_req(int strm, int reason, char *buf,
                  int cnt);
```

This routine returns immediately and will handle disconnecting the Bisync call gracefully.

strm is the file descriptor of the connected stream. The *reason* parameter is a reason code to transmit with the N_DISCON_REQ. Again, *buf* points to a buffer of data to accompany the request and *cnt* indicates the amount of data in the buffer.

Releasing the Address: the Unbind Phase

close()

```
int close(int strm);
```

Calling the system's standard *close()* routine on the data stream's file descriptor (the *strm* parameter) offers the most direct and simplest method for releasing the resources consumed by a disconnected Bisync call.

Bisync communication does not present any circumstances under which this would not be the most appropriate way to handle cleaning up after a call.