

NPI

Application Program Interface Guide

April 2003

Protocols: LAPB/X.25, SNA, and HDLC/SDLC

Copyright © 2003 GCOM, Inc.
All rights reserved.

© 1995-2003 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.

Rsystem is a registered trademark of GCOM, Inc. 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. 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, produced, and edited by Technical Writer Debra J. Schweiger using Microsoft Word 2000 and FrameMaker 6.0 on a Windows Millenium platform with the help of subject matter specialists Dave Grothe and Dave Healy, with illustrator and publication specialist Charles Lipp.

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>

C *ontents*

	1	List of Figures
	3	<i>About This Guide</i>
	3	Purpose of This Guide
	3	Knowledge Requirements
	4	Organization of This Guide
	4	Conventions Used In This Guide
	4	Special Notices
	4	Text Conventions
	6	Naming Conventions for NPI Routines
Chapter 1	7	<i>Introduction</i>
	9	Product Overview
	9	Important NPI API Files
	9	Gcom Remote API
	9	Architecture
	10	Running the RAPI Server
	11	Using the RAPI Library
	12	<i>Understanding a Sample NPI API Application</i>
Chapter 2	17	<i>Understanding the NPI API</i>
	18	Compiling a Program
	18	Linking the API Library
	19	NPI API Constants
	20	Signal Handling Prototype
	21	NPI API Fork Options
	22	NPI API Logging Options
	23	NPI Connect and Disconnect User Data
	24	NPI API Global Variables
	24	<i>npi_bind_ack</i>

24	<i>npi_conn_ind</i>
24	<i>npi_conn_con</i>
24	<i>npi_data_buf</i>
25	<i>npi_data_cnt</i>
25	<i>npi_ctl_buf[NPI_CTL_BUF_SIZE]</i>
25	<i>npi_ctl_cnt</i>
25	<i>npi_conn_ind_data_size</i> and <i>npi_conn_ind_data_skip</i>
25	<i>npi_disc_ind_data_size</i> and <i>npi_disc_ind_data_skip</i>
26	<i>npi_conn_con_data_size</i> and <i>npi_conn_con_data_skip</i>
26	<i>npi_discon_req_band</i> , <i>npi_reset_req_band</i> , <i>npi_flow_req_band</i> , <i>npi_data_req_band</i> , <i>npi_exdata_req_band</i> , <i>npi_dataack_req_band</i> , and <i>npi_other_req_band</i>

Chapter 3

27 NPI API Library Routine Reference

30	<i>npi_ascii_facil()</i>
31	<i>npi_bind_ascii_nsap()</i>
32	<i>npi_bind_nsap()</i>
33	<i>npi_close()</i>
34	<i>npi_conn_res()</i>
35	<i>npi_connect()</i>
36	<i>npi_connect_req()</i>
37	<i>npi_connect_wait()</i>
38	<i>npi_dataack_req()</i>
39	<i>npi_decode_ctl()</i>
40	<i>npi_decode_primitive()</i>
41	<i>npi_decode_reason()</i>
42	<i>npi_discon_req()</i>
43	<i>npi_discon_req_seq()</i>
44	<i>npi_drain_req()</i>
45	<i>npi_ext_bind_nsap()</i>
46	<i>npi_ext2_bind_nsap()</i>
48	<i>npi_ext_bind_ascii_nsap()</i>
49	<i>npi_ext2_bind_ascii_nsap()</i>
51	<i>npi_ext_conn_res()</i>
53	<i>npi_ext_conn_res_lstnr()</i>

54 *npi_ext_connect_req()*
55 *npi_ext_connect_wait()*
56 *npi_ext_listen()*
57 *npi_ext_nbio_complete_listen()*
58 *npi_fac_walk()*
59 *npi_flags_connect_wait()*
60 *npi_flags_listen()*
61 *npi_flow_req()*
62 *npi_get_a_msg()*
63 *npi_get_a_proto()*
64 *npi_get_and_log_facils()*
65 *npi_get_facils()*
66 *npi_get_stream_info()*
67 *npi_info_req()*
68 *npi_init()*
69 *npi_init_FILE()*
70 *npi_listen()*
72 *npi_max_sdu()*
73 *npi_nbio_complete_listen()*
74 *npi_open()*
75 *npi_perror()*
76 *npi_print_msg()*
77 *npi_printf()*
78 *npi_print_stream_info()*
79 *npi_put_data_buf()*
80 *npi_put_data_proto()*
81 *npi_put_exdata_proto()*
82 *npi_put_proto()*
83 *npi_rcv()*
83 Handling Resets
84 Delivery Confirmation
85 Handling Incoming Fragmented Data
90 *npi_read_data()*
91 *npi_reset_req()*
92 *npi_reset_res()*

93	<i>npi_send_connect_req()</i>
94	<i>npi_send_ext_conn_res()</i>
95	<i>npi_send_ext_connect_req()</i>
96	<i>npi_send_info_req()</i>
97	<i>npi_send_reset_req()</i>
98	<i>npi_send_reset_res()</i>
99	<i>npi_set_log_size()</i>
100	<i>npi_set_marks()</i>
101	<i>npi_set_pid()</i>
102	<i>npi_set_signal_handling()</i>
104	<i>npi_want_a_proto()</i>
105	<i>npi_write_data()</i>
106	<i>npi_x25_clear_cause()</i>
107	<i>npi_x25_diagnostic()</i>
108	<i>npi_x25_registration_cause()</i>
109	<i>npi_x25_reset_cause()</i>
110	<i>npi_x25_restart_cause()</i>
111	<i>put_npi_proto()</i>

Chapter 4 **113** *The PU Info API*

114	Introduction to the PU Info API
114	Using the PU Info API
117	<i>pu_decode_handle()</i>
118	<i>pu_dlpi_upa()</i>
119	<i>pu_get_pu_id()</i>
120	<i>pu_get_stats()</i>
121	<i>pu_get_board_info()</i>
122	<i>pu_get_npi_strm_stats()</i>
123	<i>pu_id_to_pu_handle()</i>
124	<i>pu_id_to_pu_number()</i>
125	<i>pu_map_npi_lpa_to_handle()</i>
126	<i>pu_strerror()</i>

Appendix A **127** *NPI Incoming Call Processing for X.25 Connections*

List of Figures

8	Figure 1	NPI Provider
10	Figure 2	GCOM API Libraries
10	Figure 3	Client Server Model
13	Figure 4	Passive Loopback Program for NPI API Library
15	Figure 5	Passive Loopback Program for NPI API Library (repeated)
85	Figure 6	Handling Fragmented NSDU's

PREFACE

About This Guide

Purpose of This Guide

This guide is written for C programmers who intend to transfer data between local and remote peers in a UNIX STREAMS environment using a network layer protocol. The data and other messages are passed to the STREAMS-based Network Provider Interface (NPI) by making calls to routines contained in the GCOM's NPI Application Program Interface (API) Library.

Knowledge Requirements

You should be familiar with the OSI Reference Model terminology, OSI Network Services and the implementation of UNIX STREAMS that your application is using. You must also be proficient with the C programming language and have specific knowledge of the network layer protocol that your application is using to transfer data.

Organization of This Guide

Table 1 shows the organization of this manual and tells you where to find specific information.

Table 1 Location of Important Information

<i>For information about:</i>	<i>Look at:</i>
Product description and related files	Section 1
Understanding a sample NPI API test program	Section 2
Understanding the NPI API header file, npiapi.h , including how to link header files, constants, the signal handling prototype, fork options, logging options, connect/disconnect user data messages and global variables	Section 3
NPI API Library routine reference	Section 4

Conventions Used In This Guide

This section discusses conventions used throughout this guide.

Special Notices

A special format indicates notes, cautions and warnings. The purpose of these notices is defined as follows:



Note: *Notes call attention to important features or instructions.*



Caution: Cautions contain directions that you must follow to avoid immediate system damage or loss of data.



Warning! Warnings contain directions that you must follow for your personal safety. Follow these instructions carefully.

Text Conventions

The use of italics, boldface and other text conventions are explained as follows:

Boldface terms Directories and file names appear in **boldface** typeface, such as the **hstpar.h** include file. Highlighted terms inside angle brackets refer to the global copy of the file. For instance, **<intsx25.h>** refers to **/rsys/include/intsx25.h**.

<i>Italic terms</i>	The following terms appear in <i>italics</i> : variables, arguments, parameters, fields, structures, glossary terms, routines, functions, programs, utilities, applications, flags, commands, and scripts. Examples include the <i>count</i> variable, <i>Command Type</i> field, <i>rteparam</i> structure, <i>Rsystem</i> defined term, <i>rsys_read()</i> routine, <i>avail</i> flag, <i>Add Route</i> command, and <i>gcomunld</i> script.
“Enter” vs. “Type”	When the word “enter” is used in this guide, it means type something and then press the Return key. Do not press Return when an instruction simply says “type.”
Screen Display	<p>This typeface is used to represent displays that appear on a terminal screen and in-line programming language statements such as <code>#ifdef</code>. Commands entered at the prompt use the same typeface only in boldface. For example:</p> <pre>C:> cd gcom % cd gcom # cd gcom</pre> <p>Each of these commands instructs you to enter “cd gcom” at the system prompt and press Return or Enter.</p>

Naming Conventions for NPI Routines

The general purpose of most routines can be inferred from the prefix and suffix naming conventions in Table 2.

Table 2 Naming Conventions for NPI Routines

<i>Prefix or Suffix?</i>	<i>String</i>	<i>Description</i>
Prefix	npi_	NPI API Library C routine
Suffix	_req	Request routines
Suffix	_res	Response routines

SECTION 1

Introduction

- 9 *Product Overview*
- 9 *Important NPI API Files*
- 15) *GCOM Remote API*

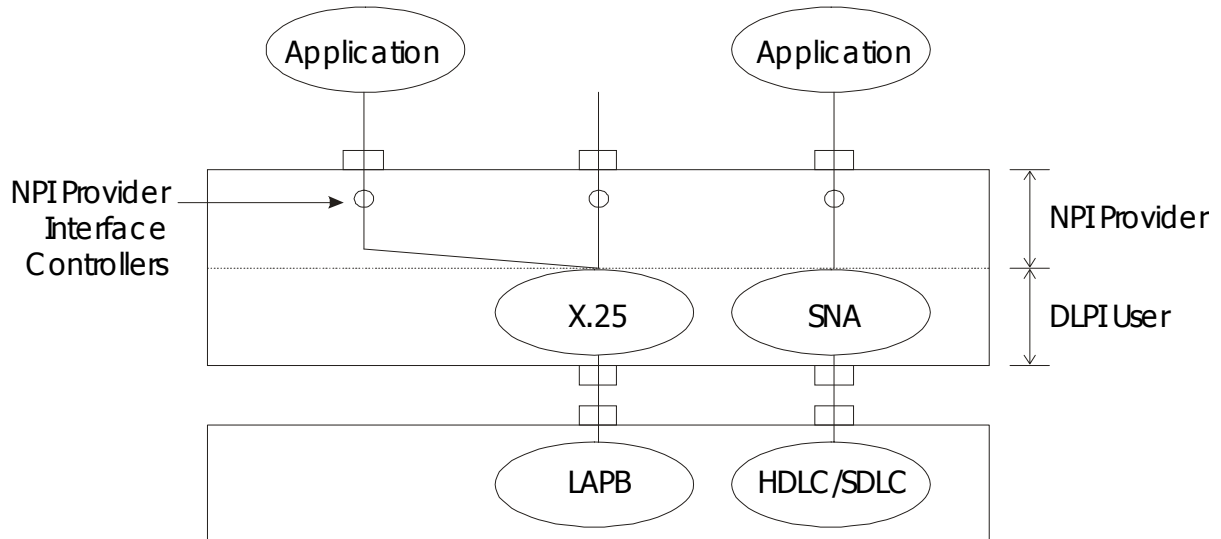


Figure 1 NPI Provider

Product Overview

Figure 1 - NPI Provider, above, shows how the SNA and X.25 packet level connect to an application through an application program interface (API). The API translates NPI protocol messages to your application by way of subroutine calls.

Important NPI API Files

The NPI device driver package includes a library of routines that you can use to interface your application program to the NPI driver. The two files that pertain to the API library are as follows:

/usr/lib/gcom/npia.a. The library file of NPI/X.25 interface routines to be linked in with your application program.

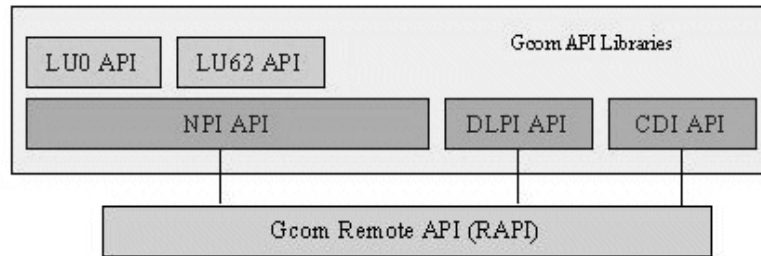
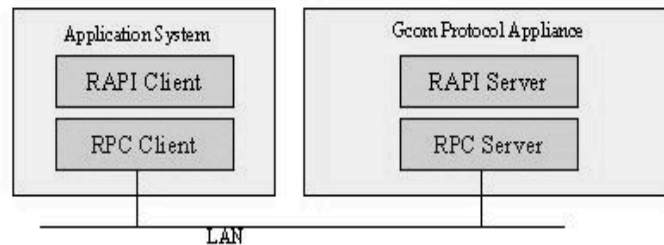
/usr/include/gcom/npia.h. Include this header file in your application program. It contains defines and prototype declarations useful to you application program. **npia.h** is compatible with both old-style C language and ANSI C. To use the ANSI style function prototype declarations, define the compile-time symbol **PROTOTYPE** prior to including **gcom/npia.h**.

GCOM Remote API

GCOM's Remote API (RAPI) is a library of functions that allows the standard GCOM APIs to operate on protocol stacks that are configured and running on a remote machine. It is especially useful in situations in which the application code resides on a server system and the protocol processing is performed on a GCOM Protocol appliance attached to the server via a LAN connection.

Architecture

Figure 2 - GCOM API Libraries, below, shows how the GCOM RAPI relates to all GCOM APIs. In the suite of GCOM API libraries, the NPI API interfaces to GCOM's NPI driver for X.25, SNA and Bisync protocols. The GCOM DLPI interfaces to GCOM's DLPI driver for link layer protocols such as LAPB, LAPD, HDLVC and Frame Relay. The GCOM CDI API library interfaces to GCOM's synchronous protocol drivers directly for raw frame access.

Figure 2 GCOM API Libraries**Figure 3 Client Server Model**

The Remote API is intended for use in a client/server environment as illustrated in Figure 3, above. The user's application program, linked with the GCOM RAPI library, runs on the client system. The server system is typically a GCOM Protocol Appliance. It contains the communication hardware, protocol software and the Remote API server.

Running the RAPI Server

The GCOM Remote API server is named *Gcom_rapisvr*. It is usually unnecessary to run this program with any arguments. By default the program runs in the background. It can be run from the command line or from a shell script.

It is common to run *Gcom_rapisvr* under root from an "rc" script. However, if permissions are set appropriately on the files that are to be accessed remotely, it is perfectly possible to run *Gcom_rapisvr* from a non-root user id.

Additional information on GCOM RAPI arguments, authentication, and other API routines can be found by accessing the GCOM RAPI white paper on the www.gcom.com web site.



Using the RAPI Library

In order to utilize the GCOM RAPI library it is necessary to link it into your program ahead of the “libgcom” library in order to link to the routines that perform the remote functions. A sample command line link for this is as follows:

```
cc -o foo foo.o /usr/lib/gcom/dlpiapi.a /usr/lib/gcom/rapi.a  
/usr/lib/gcom/libgcom.a
```



Note: *If RAPI library is omitted, then all file operations will be executed on the same machine on which the application program is running.*

In the application program, be sure to use the correct API routine to open data streams on a remote system. The open routing of each of these routines is passed a parameter which is a pointer to a string which names the remote host. Passing a NULL pointer, or a pointer to an empty string, indicates that the file is to be opened on the local machine.

When opening or closing NPI protocol data streams, use the functions:

Open routine: *npi_open*

Close routine: *npi_close*

Apart from using the specially provided open and close functions, there are no other programming interface considerations for making an application utilize remote protocol services.

Understanding a Sample NPI API Application

Figure 4 Passive Loopback Program for NPI API Library

```
1  #include <gcom/npiapi.h>
2  char          buf[1000] ;
3  int           fil ;
4  int           n ;
5  int           pkts ;
6  main()
7  {
8      npi_init(NPI_LOG_DEFAULT
9              | NPI_LOG_RX_PROTOS | NPI_LOG_TX_PROTOS
10             ,NPI_LOG_NAME
11             ) ;
12     fil = npi_listen("*", NPI_LISTEN_FORK) ;
13     if (fil < 0) exit(1) ;
14     while ((n = npi_read_data(fil, buf, sizeof(buf))) >= 0)
15     {
16         pkts++ ;
17         if (npi_write_data(fil, buf, n) < 0) exit(1) ;
18     }
19     npi_printf("End of test: packets received = %d\n", pkts) ;
20     exit(0) ;
21 }
```

Figure 2, above, shows how to use the NPI API Library. The program (which happens to be *Gcom_npilstn*) is a simple passive loopback program. It listens for any incoming connections, forks off a copy of itself to handle the incoming connection and loops any received data back to the sender. Refer to Section 3 starting on page 27 for a list of programs you can call from the NPI API.

The sample program contains the following features:

- Line 1 Include the **npiapi.h** file. A `-DPROTOTYPE` on the command line can cause ANSI style prototypes to be visible in the **npiapi.h** file.
- Lines 8–11 The program initializes the NPI API Library. The first parameter to the *npi_init()* function specifies logging options to be used by the library. The set of options on line 9 make the logging verbose. Therefore, they are not suitable for production code. The second parameter to *npi_init()* is the name of the log file that the library is to use. The default name is **/usr/spool/gcom/npi.log**.
- Line 12 The program calls the *npi_listen()* function to wait for incoming calls. The *npi_listen()* function returns the file handle of a Stream that has an open connection associated with it. The first parameter is an ASCII string representing the address on which the program is listening. The second parameter is an option that specifies that the *npi_listen()* function should fork a new copy of itself for each incoming connection. Thus, when *npi_listen()* returns, it will be on a child process of the original *npi_listen()* caller.

Figure 5 Passive Loopback Program for NPI API Library (*repeated*)

```
1 #include <gcom/npiaapi.h>
2 char          buf[1000] ;
3 int           fil ;
4 int           n ;
5 int           pkts ;
6 main()
7 {
8     napi_init(NPI_LOG_DEFAULT
9               | NPI_LOG_RX_PROTOS | NPI_LOG_TX_PROTOS
10              ,NPI_LOG_NAME
11              ) ;
12     fil = napi_listen("*", NPI_LISTEN_FORK) ;
13     if (fil < 0) exit(1) ;
14     while ((n = napi_read_data(fil, buf, sizeof(buf))) >= 0)
15     {
16         pkts++ ;
17         if (napi_write_data(fil, buf, n) < 0) exit(1) ;
18     }
19     napi_printf("End of test: packets received = %d\n", pkts) ;
20     exit(0) ;
21 }
```

Line 14 The program calls the *npi_read_data()* function to read a single data message from the Stream. The function returns a negative number if a protocol message arrives or if the Stream encounters an error.



Note: *The parameters for npi_read_data() are the same as for the standard C library read routine.*

Line 17 The program writes back any received data to the Stream; this is the loopback operation.



Note: *The parameters for npi_write_data() are the same as for the standard C library write routine.*

Line 19 The program uses the *npi_printf()* routine to write a message into the log file. This routine has the same calling sequence as the C library *printf()* routine.

Understanding the NPI API

18	<i>Linking the API Library</i>
19	<i>NPI API Constants</i>
20	<i>Signal Handling Prototype</i>
21	<i>NPI API Fork Options</i>
22	<i>NPI API Logging Options</i>
23	<i>NPI Connect and Disconnect User Data</i>
24	<i>NPI API Global Variables</i>
24	<i>npi_bind_ack</i>
24	<i>npi_conn_ind</i>
24	<i>npi_conn_con</i>
24	<i>npi_data_buf</i>
25	<i>npi_data_cnt</i>
25	<i>npi_ctl_buf[NPI_CTL_BUF_SIZE]</i>
25	<i>npi_ctl_cnt</i>
25	<i>npi_conn_ind_data_size and npi_conn_ind_data_skip</i>
25	<i>npi_disc_ind_data_size and npi_disc_ind_data_skip</i>
26	<i>npi_conn_con_data_size and npi_conn_con_data_skip</i>
26	<i>npi_discon_req_band, npi_reset_req_band, npi_flow_req_band, npi_data_req_band, npi_exdata_req_band, npi_dataack_req_band, and npi_other_req_band</i>

The **npia.h** file contains defines and prototypes used for the interface between your application and the GCOM NPI Streams driver. The **npia.a** file contains the routines that comprise the NPI API Library.

Compiling a Program

When compiling a program that is using the NPI API, be sure to include the option “-D_REENTRANT” on your compiler command line. The NPI API is a thread safe library and the *npia.h* file needs this option to be defined in order to be interpreted correctly when it is included.

Linking the API Library

To link the API Library with a program, issue the UNIX *cc* command using the following format:

```
cc -o program_name program_name.c /usr/lib/gcom/npia.a -lpthread
```

Example The following command links the API Library for the *npilstn()* program:

```
cc -o npilstn npilstn.c /usr/lib/gcom/npia.a -lpthread
```

NPI API Constants

The following constants parameterize the internal behavior of the NPI API Library. These were the constants used when the library was compiled. Changing these in the application program will not have the desired effect on the NPI API Library.

NPI_N_CONINDS	Number of incoming connection indications that can be queued on a listener stream. This constant is passed to NPI in an N_BIND_REQ for a listener stream. It is intended to be large enough so as not to lose incoming connections while forking a process to process a single incoming connection. At this writing, it is set to 16.
NPI_DATA_BUF_SIZE	The size of the character array <i>npi_data_buf</i> compiled into the NPI API Library. This array is available for use by the application (see below). Its size is larger than the largest possible STREAMS message buffer. At this writing, it is set to 5,000.
NPI_CTL_BUF_SIZE	The size of the character array <i>npi_ctl_buf</i> compiled into the NPI API Library. This array is available for use by the application (see below). Its size is larger than the largest possible STREAMS message buffer. At this writing, it is set to 5,000.
NPI_LOG_NAME	The default name of the log file into which the NPI API Library writes messages. At this writing, it is set to /usr/spool/gcom/npi.log .

Signal Handling Prototype

This is the *typedef* for the user-supplied function passed to *npi_set_signal_handling()*:

```
#ifdef PROTOTYPE
typedef int (*npi_sig_func_t) (int fid, char *ctl_ptr, int ctl_length,
                               char *data_ptr, int data_length);
#else
typedef int (*npi_sig_func_t) ();
#endif
```

When called, the value returned from the user-supplied function is currently ignored by the API. Returning zero is therefore recommended.

NPI API Fork Options

The fork options are passed to the *npi_listen()* routine. They control the process forking behavior when an incoming connection is completed. The forking styles are to fork a child process to handle the connection, or refrain from forking to another process and return to the user immediately because the user is using non-blocking I/O.

In the latter case, the return from *npi_listen()* is in the same, and only, process as its caller. When this occurs, the incoming connection has already been transferred to another Stream with the original listening Stream still listening. If the connection can be utilized and then disconnected before NPI_N_CONINDS connect indications get queued up within the NPI driver, then this technique can be effectively used to serially process incoming connections to the same address. Once you have processed a connection within your application program, you can await another by calling the *npi_listen()* routine again, just as in the first instance.

When using the other alternative, forking a process to handle the incoming connections, the *npi_listen()* routine forks a child process after transferring the incoming connection to a second Stream. The return from *npi_listen()* will be on the child process. The parent process continues execution within the *npi_listen()* routine in order to accept more connections.

The options are selected by using either of the following two defines in the call to *npi_listen()*.

NPI_LISTEN_FORK	Fork a child process after accepting an incoming connection.
NPI_LISTEN_NO_FORK	Return without forking after accepting an incoming connection.
NPI_NBIO	Return to user immediately. The user will use non-blocking I/O, such as <i>poll()</i> , to manage the listener Stream. See “ <i>npi_ext_nbio_complete_listen()</i> ” on page 57 for further details.

NPI API Logging Options

The following options are passed to the *npi_init()* routine. They control the kinds of messages that are written into the NPI API log file. These options represent individual bits of a mask. They are ORed together to form the set of options passed to *npi_init()*. A zero means no error reporting at all.

NPI_LOG_FILE	Write messages to the log file.
NPI_LOG_STDERR	Write messages to <i>stderr</i> .
NPI_LOG_RX_PROTOS	Log all received protocol messages in ASCII decoded form.
NPI_LOG_TX_PROTOS	Log all transmitted protocol messages in ASCII decoded form.
NPI_LOG_ERRORS	Log all UNIX error messages in a manner similar to <i>perror</i> .
NPI_LOG_RX_DATA	Log all received M_DATA messages (<i>npi_read_data</i>). Dump the data contents in hex into the log file. Used for debugging.
NPI_LOG_TX_DATA	Log all transmitted M_DATA messages (<i>npi_write_data</i>). Dump the data contents in hex into the log file. Used for debugging.
NPI_LOG_SIGNALS	Log signal handling.
NPI_LOG_CONINDS	Log NPI connection indications.
NPI_LOG_OPTIONS	Log initialization options.
NPI_LOG_FACILS	Log X.25 facilities pertaining to the connection at disconnect time. The <i>npi_get_and_log_facils()</i> API routine internally obtains the facilities and prints them into the log. The facilities will be logged to <i>stderr</i> , the log file or both depending upon other logging options.
NPI_LOG_DEFAULT	The default set of logging options consisting of NPI_LOG_FILE, NPI_LOG_STDERR and NPI_LOG_ERRORS.

NPI Connect and Disconnect User Data

The N_CONN_IND, N_CONN_CON and N_DISCON_IND messages are all received by the NPI API Library. Each of these messages may be accompanied by M_DATA. Depending upon the settings of some variables, the NPI API Library may save the data from these messages so that the next call to *npi_read_data()* transfers the data to the user's buffer area. These constants provide the defaults for the variables that control this activity:

NPI_CONN_IND_DATA_SIZE

Initializes *npi_conn_ind_data_size* variable

NPI_CONN_IND_DATA_SKIP

Initializes *npi_conn_ind_data_skip* variable

NPI_CONN_CON_DATA_SIZE

Initializes *npi_conn_con_data_size* variable

NPI_CONN_CON_DATA_SKIP

Initializes *npi_conn_con_data_skip* variable

NPI_DISC_IND_DATA_SIZE

Initializes *npi_disc_ind_data_size* variable

NPI_DISC_IND_DATA_SKIP

Initializes *npi_disc_ind_data_skip* variable

NPI API Global Variables

The NPI API package makes a certain set of its global variables available to the user. The variables and their intended uses are provided in this section.

The thread safe nature of the NPI API library means that the following variables, accessed as global variables, actually appear as different areas of memory for each thread that accesses them. For example, if thread A sets one of these variables to some value, thread B will perceive a different value because it has its own copy. Also, if thread A calls a routine that deposits information in one of these global locations (such as *npi_rcv()*), then only thread A can perceive the values returned by that call.

npi_bind_ack

```
unsigned char    npi_bind_ack[] ;
```

This array contains a copy of the N_BIND_ACK received by the NPI API routines. You may examine it for whatever information you may find useful.

npi_conn_ind

```
unsigned char    npi_conn_ind[] ;
```

This array contains a copy of the N_CONN_IND received by the NPI API routines. When forking child processes, each child has its own copy of this protocol message.

npi_conn_con

```
unsigned char    npi_conn_con[] ;
```

This array contains a copy of the N_CONN_CON (connect confirm) received by the NPI API Library in response to an N_CONN_REQ sent by the library (active connection).

npi_data_buf

```
unsigned char    npi_data_buf[NPI_DATA_BUF_SIZE] ;
```

This array may be used by the application program as a place to build data messages to send on a connection or as a place into which to receive data messages. It is an array that is made available for use to the application program.

npi_data_cnt

```
int                npi_data_cnt ;
```

This variable contains the count of the number of bytes contained in the most recently sent or received data message from the NPI driver.

npi_ctl_buf[NPI_CTL_BUF_SIZE]

```
unsigned char     npi_ctl_buf[NPI_CTL_BUF_SIZE] ;
```

This array is used by the NPI API driver to receive the M_PROTO portion of messages from the NPI driver. It contains the most recent M_PROTO received by the NPI API Library. A call to *npi_read_data()* overwrites this area with an invalid value; upon return, if the *PRIM_type* field does not contain all ones, then an M_PROTO was received while reading data.

npi_ctl_cnt

```
int                npi_ctl_cnt ;
```

This variable contains the length of the M_PROTO portion of the last message received from the NPI driver. If the last received message is a data-type message, this count will be negative. A call to *npi_read_data()* may fail because of a received M_PROTO. In such a case, the count in *npi_ctl_cnt* will be positive and the protocol message will reside in the array *npi_ctl_buf*.

npi_conn_ind_data_size and npi_conn_ind_data_skip

```
int                npi_conn_ind_data_size ;
```

```
int                npi_conn_ind_data_skip ;
```

These variables control the saving of data from an N_CONN_IND message received by the NPI API Library. If the message contains more bytes than *npi_conn_ind_data_size*, then all but the first *npi_conn_ind_data_skip* bytes are saved so that they can be transferred to the user's buffer at the next call to *npi_read_data()*. These variables may be changed by assigning into them if the default values are not useful.

npi_disc_ind_data_size and npi_disc_ind_data_skip

```
int                npi_disc_ind_data_size ;
```

```
int                npi_disc_ind_data_skip ;
```

These variables control the saving of data from an N_DISCON_IND

message received by the NPI API Library. If the message contains more bytes than *npi_disc_ind_data_size*, then all but the first *npi_disc_ind_data_skip* bytes are saved so that they can be transferred to the user's buffer at the next call to *npi_read_data()*. After the data has been read, *npi_read_data()* returns a negative value on the next following call to indicate prior receipt of the N_DISCON_IND message. These variables may be changed by assigning into them if the default values are not useful.

npi_conn_con_data_size* and *npi_conn_con_data_skip

```
int          npi_conn_con_data_size ;
int          npi_conn_con_data_skip ;
```

These variables control the saving of data from an N_CONN_CON message received by the NPI API Library. If the message contains more bytes than *npi_conn_con_data_size*, then all but the first *npi_conn_con_data_skip* bytes are saved so that they are transferred to the user's buffer at the next call to *npi_read_data()*. These variables may be changed by assigning into them if the default values are not useful.

npi_discon_req_band*, *npi_reset_req_band*, *npi_flow_req_band*, *npi_data_req_band*, *npi_exdata_req_band*, *npi_dataack_req_band*, and *npi_other_req_band

```
int          npi_discon_req_band
int          npi_reset_req_band
int          npi_flow_req_band
int          npi_data_req_band
int          npi_exdata_req_band
int          npi_dataack_req_band
int          npi_other_req_band
```

These variables control the priority band used by the corresponding requests. These priority bands allow privileged requests to be presented to NPI more rapidly. The default priority band is 0 for all requests; the fastest possible priority is 255. Priority levels are available in all versions of STREAMS other than the SVR3 version.

Priority levels can be set “on the fly” in a program to ensure that certain requests get processed as quickly as possible instead of being queued behind outstanding data requests. An emergency disconnect request is an example of when this might be useful, as is the NPI flow control request.

SECTION 3

NPI API Library Routine Reference

30	<i>npi_ascii_facil()</i>
31	<i>npi_bind_ascii_nsap()</i>
32	<i>npi_bind_nsap()</i>
33	<i>npi_close()</i>
34	<i>npi_conn_res()</i>
35	<i>npi_connect()</i>
36	<i>npi_connect_req()</i>
37	<i>npi_connect_wait()</i>
38	<i>npi_dataack_req()</i>
39	<i>npi_decode_ctl()</i>
40	<i>npi_decode_primitive()</i>
41	<i>npi_decode_reason()</i>
42	<i>npi_discon_req()</i>
43	<i>npi_discon_req_seq()</i>
44	<i>npi_drain_req()</i>
45	<i>npi_ext_bind_nsap()</i>
46	<i>npi_ext2_bind_nsap()</i>
48	<i>npi_ext_bind_ascii_nsap()</i>
51	<i>npi_ext_conn_res()</i>
53	<i>npi_ext_conn_res_lstnr()</i>
54	<i>npi_ext_connect_req()</i>
55	<i>npi_ext_connect_wait()</i>
56	<i>npi_ext_listen()</i>
57	<i>npi_ext_nbio_complete_listen()</i>
58	<i>npi_fac_walk()</i>
59	<i>npi_flags_connect_wait()</i>
60	<i>npi_flags_listen()</i>
61	<i>npi_flow_req()</i>
62	<i>npi_get_a_msg()</i>

63	<i>npi_get_a_proto()</i>
64	<i>npi_get_and_log_facils()</i>
65	<i>npi_get_facils()</i>
66	<i>npi_get_stream_info()</i>
67	<i>npi_info_req()</i>
68	<i>npi_init()</i>
69	<i>npi_init_FILE()</i>
70	<i>npi_listen()</i>
72	<i>npi_max_sdu()</i>
73	<i>npi_nbio_complete_listen()</i>
74	<i>npi_open()</i>
75	<i>npi_perror()</i>
76	<i>npi_print_msg()</i>
77	<i>npi_printf()</i>
78	<i>npi_print_stream_info()</i>
79	<i>npi_put_data_buf()</i>
80	<i>npi_put_data_proto()</i>
81	<i>npi_put_exdata_proto()</i>
82	<i>npi_put_proto()</i>
83	<i>npi_rcv()</i>
90	<i>npi_read_data()</i>
91	<i>npi_reset_req()</i>
92	<i>npi_reset_res()</i>
93	<i>npi_send_connect_req()</i>
94	<i>npi_send_ext_conn_res()</i>
95	<i>npi_send_ext_connect_req()</i>
96	<i>npi_send_info_req()</i>
97	<i>npi_send_reset_req()</i>
98	<i>npi_send_reset_res()</i>
99	<i>npi_set_log_size()</i>
100	<i>npi_set_marks()</i>
101	<i>npi_set_pid()</i>
102	<i>npi_set_signal_handling()</i>
104	<i>npi_want_a_proto()</i>
105	<i>npi_write_data()</i>
106	<i>npi_x25_clear_cause()</i>
107	<i>npi_x25_diagnostic()</i>
108	<i>npi_x25_registration_cause()</i>

109 *npi_x25_reset_cause()*
110 *npi_x25_restart_cause()*
111 *put_npi_proto()*

npi_bind_nsap()

Prototype `int npi_bind_nsap(int npi_data,
 char *bind_sap,
 int nsap_lgth,
 int conind_nr,
 unsigned flags) ;`

Include File(s) **<gcom/npiapi.h>**, **<gcom/npi.h>**, **<gcom/npiext.h>** (for *flags*)

Description The *npi_bind_nsap()* routine issues an N_BIND_REQ to NPI and waits for an N_BIND_ACK response. The bind acknowledgement is copied to the global array *npi_bind_ack* so that it can be viewed directly by the application program.

Parameters *npi_data* Stream used to send the N_BIND_REQ

bind_nsap NSAP address to be bound to the stream. This address is either a sequence of BCD (4-bit) digits or an ASCII string. The ASCII string form is represented by the first byte containing the value 0xF0. The ASCII string form can contain wildcards as with *npi_bind_ascii_nsap()*.

nsap_lgth Length of the NSAP in bytes. Note that this implies an even number of digits if the BCD form is used.

conind_nr Number of connection indications that can be queued for this stream. If this number is non-zero, then the stream becomes a listener stream; that is, it becomes eligible as a target of an incoming connection. If this number is zero, then this stream cannot be the target for an incoming connection but is to be used for an outgoing connection.

flags These are the BIND_flags to supply with the N_BIND_REQ.

Return < 0 An error occurred (message written to log file).

 > 0 Size of N_BIND_ACK if success.

npi_close()

Prototype `int npi_close(int fd)`

Include Files **<gcom/npiapi.h>**

Description Closes the file opened by *npi_open*. This routine should be used to close the file rather than by calling the system “close” routine directly. The *npi_close* routine takes into account the possibility that the file may be opened to a remote server via the Remote API.

npi_close returns **0** for success and **-1** for failure. Upon failure, “errno” is set to indicate the reason for failure just as with the system “close” routine.

npi_connect()

<i>Prototype</i>	<code>int npi_connect(char *remote_sap, unsigned bind_flags) ;</code>
<i>Include File(s)</i>	<code><gcom/npiapi.h>, <gcom/npi.h>, <gcom/npiext.h></code> (for <i>bind_flags</i>)
<i>Description</i>	The <i>npi_connect()</i> routine opens an NPI stream, binds it using an empty address and the <i>bind_flags</i> supplied with the call to <i>npi_connect()</i> , and then uses <i>npi_connect_req()</i> (see next page) to issue an N_CONN_REQ to NPI and wait for an N_CONN_CON response. The connect confirm response is copied to the global array <i>npi_conn_con</i> so that it can be viewed directly by the application program.
<i>Parameters</i>	<p><i>remote_sap</i> The NSAP of the remote system as an ASCII string</p> <p><i>bind_flags</i> Flags to pass into the N_BIND_REQ that will be issued. The <i>bind_flags</i> are as defined in npi.h and may be used to request receipt confirmation service for the connection.</p>
<i>Return</i>	<p>< 0 An error occurred (message written to log file).</p> <p>> 0 File descriptor of an NPI stream that can be used to send data.</p>

npi_connect_req()

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

Include File(s) **<gcom/npiapi.h>**

Description The *npi_connect_req()* routine issues an N_CONN_REQ to NPI and waits for an N_CONN_CON response. If the *buf* parameter is non-NULL, *cnt* bytes of data accompanies the N_CONN_REQ. This is useful in X.25 “fast select” situations. The connect confirm response is copied to the global array *npi_conn_con* so that it can be viewed directly by the application program.

If, in awaiting the N_CONN_CON message, an N_DISCON_IND with data is received, the *npi_connect_req()* routine returns normally as if the connection completed allowing the user to read the data from the N_DISCON_IND by calling the *npi_read_data()* routine.

If any data accompanies the connect confirm, it is saved and returned to the user at the first use of *npi_read_data()*.

Parameters

<i>npi_data</i>	Stream used to send the N_CONN_REQ.
<i>peer_sap</i>	Address of the peer to which you wish to be connected. This address is represented as an ASCII string. No wildcard characters are allowed in this address.
<i>buf</i>	Pointer to a user buffer containing data to be sent with the N_CONN_REQ (may be NULL).
<i>cnt</i>	The number of bytes of data to send with the N_CONN_REQ.

Return

< 0	An error occurred (message written to log file).
> 0	Size of N_CONN_CON if success.

npi_connect_wait()

Prototype `int npi_connect_wait(int npi_data) ;`

Include File(s) **<gcom/npiapi.h>**

Description This routine waits on a listening stream for an incoming N_CONN_IND. When one is received, it returns a file descriptor for a new data stream opened to that connection.

This routine will block. To prevent blocking, the user should use poll() or select() to check the listener fid for incoming messages before calling this routine.

Parameters *npi_data* The fid of the NPI listener data stream. This file descriptor will continue to be a listener after the connection has been transferred to a new file descriptor.

Return < 0 An error occurred (message written to log file).
 > 0 The new file descriptor opened to the incoming connection.

npi_dataack_req()

Prototype `int npi_dataack_req(int npi_data) ;`

Include File(s) **<gcom/npiapi.h>**

Description The *npi_dataack_req()* routine sends an N_DATAACK_REQ on the given stream. This protocol message needs to be sent whenever an N_DATA_IND is received that has the receipt confirmation bit set. The *npi_read_data()* routine does this automatically for you. You need only to concern yourself with this if you are using the *npi_get_a_msg()* routine to read from the stream; this is a lower level routine.

Parameters *npi_data* The stream used to send the N_CONN_REQ

Return < 0 An error occurred (message written to log file).

 > 0 Size of N_DATAACK_REQ if success.

npi_decode_ctl()

- Prototype* `void npi_decode_ctl(char *p) ;`
- Include File(s)* **<gcom/npiapi.h>**
- Description* The *npi_decode_ctl()* routine is mainly used internally to log errors and message traffic. The M_PROTO presently in the global buffer *npi_ctl_buf* is decoded into ASCII and written to the log file.
- Parameters* *p* A pointer to a string to be printed as a label for the decoded message.
- Prototype* `*npi_decode_npi_error(np_uns32 err_ack_code) ;`
- Include File(s)* **<gcom/npiapi.h>,<gcom/npi.h>**
- Description* This routine converts the NPI_error field of an N_ERROR_ACK to ASCII.
- Parameters* A value contained in the NPI_error field of an N_ERROR_ACK.
- Return* Returns a pointer to a string containing the ASCII decoding of the error code.

npi_decode_primitive()

Prototype `char *npi_decode_primitive(long primitive) ;`

Include File(s) **<gcom/npiapi.h>**

Description This routine can be used to translate an NPI primitive code into an ASCII string.

Parameters *primitive* The numerical designation of the primitive.

Return Returns a pointer to a string containing either the ASCII designation of the string or its hexadecimal designation if the primitive is unknown.

npi_decode_reason()

Prototype `char *npi_decode_reason(long reason) ;`

Include File(s) **<gcom/npiapi.h>**

Description Converts a numerical reason code into an ASCII string.

Parameters *reason* The numerical reason code to decipher.

Return Pointer to a static buffer containing a descriptive string describing the reason code.

npi_discon_req()

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

Include File(s) **<gcom/npiapi.h>**, **<gcom/npi.h>** (for *reason*)

Description The *npi_discon_req()* routine issues an N_DISCON_REQ to NPI and waits for an N_OK_ACK response. If the *buf* parameter is non-NULL, *cnt* bytes of data accompanies the N_DISCON_REQ. This is useful in X.25 “fast select” situations.

Parameters

<i>npi_data</i>	Stream used to send the N_DISCON_REQ
<i>reason</i>	The code to go into the DISCON_reason field of the N_DISCON_REQ
<i>buf</i>	Pointer to a user buffer containing data to be sent with the N_DISCON_REQ (may be NULL)
<i>cnt</i>	The number of bytes of data to send with the N_DISCON_REQ

Return

< 0	An error occurred (message written to log file).
> 0	Size of N_OK_ACK if success.

npi_discon_req_seq()

Prototype `int npi_discon_req_seq(int npi_data,
 int reason,
 long seq,
 char *buf,
 int cnt) ;`

Include File(s) **<gcom/npiapi.h>**, **<gcom/npi.h>** (for *reason* and *N_conn_ind_t*)

Description The *npi_discon_req_seq()* routine disconnects the stream associated with the specified sequence number.

Parameters

<i>npi_data</i>	Stream file descriptor.
<i>reason</i>	The code to go into the DISCON_reason field of the N_DISCON_REQ.
<i>seq</i>	This parameter is used to distinguish between multiple incoming connect indications on a single stream. When it is necessary to disconnect only one of them without affecting the others the <i>seq</i> parameter tells the NPI driver which connection is being refused. A value of zero can be used when there is only a single connect indication outstanding on the stream. A non-zero value comes from the SEQ_number field of the <i>N_conn_ind_t</i> structure in <i>npi_ctl_buf</i> when a connect indication is received from the NPI driver. It is generally a good idea to use the received <i>SEQ_number</i> field unconditionally whenever calling the <i>npi_discon_req_seq()</i> routine since the only difference between this routine and the <i>npi_discon_req()</i> routine is the presence of the <i>seq</i> parameter.
<i>buf</i>	Pointer to a user buffer containing data to be sent with the N_DISCON_REQ (may be NULL).
<i>cnt</i>	The number of bytes of data to send with the N_DISCON_REQ.
<i>Return</i> < 0	An error occurred (message written to log file).
> 0	Success. Return message size.

mpi_drain_req()

Prototype `mpi_drain_req(int strm,
 int option);`

Include File(s) **<gcom/mpiapi.h>, <gcom/mpiext.h>**

Description Send an N_DRAIN_REQ to the NPI Provider. Do not await the N_OK_ACK. The N_DRAIN_REQ causes the NPI Provider to return an N_OK_ACK at some later point in time when the drain conditions have been met. Generally speaking the idea is to obtain notification when data packets have been “drained.” The option parameter specifies more precisely when the N_OK_ACK is sent back upstream. The following table explains the options in terms of mnemonics that are defined in the file *<gcom/mpiext.h>*. Only one mnemonic is to be used as the option parameter value.

Parameters

Mnemonic	Description
N_DRAIN_IMMED	Send the N_OK_ACK immediately.
N_DRAIN_SENT	Send the N_OK_ACK when the NPI Provider has sent all outgoing data packets to the S.25 protocol stack. Data packets could still be queued by X.25 awaiting flow control authorization at this point in time.
N_DRAIN_ACKD	Send the N_OK_ACK when all data packets have been sent and acknowledged. The acknowledgement has either local or end to end significance depending on the internal characteristics of the X.25 network over which the data packet(s) were sent.

Return <0 An error occurred.
 >=0 Success. An N_DRAIN_REQ was sent to the NPI Provider.

<i>rem_nsap</i>		The remote NSAP address. Incoming calls must originate from and outgoing calls must be directed to a matching address. This address uses the same form as <i>nsap</i> .
<i>rem_lgth</i>		Length of the remote NSAP address.
<i>lpa</i>		NPI LPA. Calls must match this LPA number.
<i>conind_nr</i>		Number of connection indications which can be queued before refusing further connections.
<i>flags</i>		Bind flags.
<i>data_mask</i>		A pointer to an array of 16 bytes. See Appendix A for a description of how this parameter is used.
<i>data_val</i>		A pointer to an array of 16 bytes. See Appendix A for a description of how this parameter is used.
<i>Return</i>	<0	Error condition.
	>=0	Success. Return value is the size of the bind acknowledgement, in bytes.

npi_ext2_bind_ascii_nsap()

```

Prototype   int npi_ext2_bind_ascii_nsap (int      npi_data,
                                             char      *ascii_nsap,
                                             char      *rem_ascii_nsap
                                             np_int32   lpa,
                                             np_uns8   *data_val
                                             np_uns8   *data_mask,
                                             int      conind_nr
                                             unsigned  flags) ;

```

Include File(s) <gcom/npiapi.h>

Description Similar to the routine *npi_ext_bind_nsap*. This routine accepts two additional parameters, *data_val* and *data_mask*. These are arrays of 16 bytes. If the parameter *conind_nr* is zero these two arrays are not used. If *conind_nr* is positive then the bytes of these two arrays are used to match against the user data field of an X.25 incoming call packet to determine whether or not the call is connected to the stream over which the *npi_ext2_bind_ascii_nsap* is issued.

The data comparison is as follows, for *i* ranging from 0 to 15:

```

(user_data[i] & data_mask[i]) == (data_val[i] &
data_mask[i])

```

All 16 bytes, up to the number of bytes in the user data field of the incoming call packet, must match in this manner for the incoming call to be connected to this stream. If the incoming call has no user data field, then it is assumed to match any data field patterns.

As with the standard NPI N_BIND_REQ for incoming connections, the total patterns of all such bind request presented to the NPI Provider must be distinct from one another.

<i>Parameters</i>	<i>npi_data</i>	The file descriptor of the data file.
	<i>nsap</i>	The local NSAP address. Incoming calls must be presented to this address. This address is given as an ASCII string.
	<i>rem_nsap</i>	The remote NSAP address. Incoming calls must originate from an address that matches this address. This address is given as an ASCII string.
	<i>lpa</i>	NPI LPA. Calls must match this LPA number.
	<i>conind_nr</i>	Number of connection indications which can be queued before refusing further connections.
	<i>flags</i>	Bind flags.

	<i>data_mask</i>	A pointer to an array of 16 bytes. See Appendix A for a description of how this parameter is used.
	<i>data_val</i>	A pointer to an array of 16 bytes. See Appendix A for a description of how this parameter is used.
<i>Return</i>	<0	Error condition.
	>=0	Success. Return value is the size of the bind acknowledgement, in bytes.

npi_ext_conn_res()

```

Prototype   int npi_ext_conn_res(int          npi_data,
                                N_conn_ind_t *c,
                                long          tknval,
                                char          *fac_ptr,
                                int          fac_cnt) ;

```

Include File(s) <gcom/npiapi.h>, <gcom/npi.h>

Description The *npi_ext_conn_res()* routine sends a connect response on the NPI data stream and waits for an N_OK_ACK.

Special Case: After receiving a connect indication, a connect response must be returned, and an N_OK_ACK to the connect response is normally received. While the N_OK_ACK is queued at the stream head, a reset indication may be received from the apposite (peer). The processing and delivery of the N_RESET_IND requires that an M_FLUSH first be sent upstream, possibly flushing the N_OK_ACK to the connect response.

If a N_RESET_IND is received before the N_OK_ACK to the connect response, the reset exchange is completed, and this procedure returns success. The reset indication occurring under these conditions can be detected by the caller by examining the *CORRECT_prim* field of the *N_ok_ack_t* contained in *npi_ctl_buf*. The reset is otherwise transparent to the application.

Parameters

<i>npi_data</i>	Stream file descriptor.
*c	Pointer to the connect indication being responded to.
<i>tknval</i>	This value is used to transfer the connection to another stream. A value of zero indicates that the connection is to be accepted on the stream indicated by the <i>npi_data</i> parameter. A non-zero value indicates that a transfer to another stream is desired. The value of the <i>tknval</i> parameter comes from the N_BIND_ACK for the stream to which the connection is to be transferred. When using this mechanism, you must save the <i>TOKEN_value</i> field from the <i>N_bind_ack_t</i> structure that is returned in <i>npi_ctl_buf</i> upon successful return from the <i>npi_bind_nsap()</i> or <i>npi_bind_ascii_nsap()</i> routine.
*fac_ptr	A pointer to a user buffer which contains the X.25 facilities that are to be included in the N_EXT_CONN_RES sent to NPI to accept the incoming connection. If this pointer is NULL, then no facilities will be included and the primitive sent to NPI will be an N_CONN_RES rather than an N_EXT_CONN_RES.

fac_cnt The number of bytes in user provided facilities. If this parameter is less than or equal to zero, then no facilities will be included and the primitive sent to NPI will be an N_CONN_RES rather than an N_EXT_CONN_RES.

Return < 0 An error occurred (message written to log file).
 > 0 Success. Return a reset response.

npi_ext_conn_res_lstnr()

```

Prototype  npi_ext_conn_res_lstnr(int          lstnr,
                                int          npi_data,
                                N_conn_ind_t *c,
                                np_uns32   tknval,
                                char        *fac_ptr,
                                int         fac_cnt);

```

Include File(s) <gcom/npiapi.h>

Description This routine causes a connection to be accepted and transferred to another stream. It returns after receiving the N_OK_ACK for the connect response message that it sends on the listener stream. The N_OK_ACK is received on the new data stream. Upon successful return, the data stream is ready for sending data.

Parameters

<i>lstnr</i>	The listening stream on which the N_CONN_IND was received.
<i>npi_data</i>	A newly opened and bound data stream to which the connection is to be transferred.
<i>c</i>	A pointer to the N_CONN_IND that was received on the listener stream.
<i>tknval</i>	The value to use in the TOKEN_value field of the connect response. This is used to transfer the connection to a new stream represented by the argument <i>npi_data</i> . The <i>tknval</i> argument was obtained from the bind request on the <i>npi_data</i> stream. The connection will be accepted and transferred to the new stream.
<i>fac_ptr</i>	Pointer to facilities, in binary. NULL if no facilities. If this parameter is NULL then an N_CONN_RES will be sent to the NPI Provider. If it is non-NULL, then the N_EXT_CONN_RES form will be used in order to convey the facilities to the NPI Provider.
<i>fac_cnt</i>	Number of bytes of facilities. Set to zero if the <i>fac_ptr</i> argument is NULL.

Return

<0	An error occurred.
>0	Success. Connect response sent and N_OK_ACK received.

npi_ext_connect_req()

Prototype `int npi_ext_connect_req(int npi_data,
 char *peer_sap,
 char *buf,
 int cnt,
 char *fac_ptr,
 int fac_cnt) ;`

Include File(s) **<gcom/npiapi.h>**

Description The *npi_ext_connect_req()* routine functions just like the *npi_connect_req()* routine except that it allows the inclusion of X.25 facilities in the connect request. It sends an N_EXT_CONN_REQ to NPI rather than an N_CONN_REQ.

Parameters

<i>npi_data</i>	Stream used to send the N_CONN_REQ.
<i>peer_sap</i>	Address of the peer to which you wish to be connected. This address is represented as an ASCII string. No wildcard characters are allowed in this address.
<i>buf</i>	Pointer to a user buffer containing data to be sent with the N_CONN_REQ (may be NULL).
<i>cnt</i>	The number of bytes of data to send with the N_CONN_REQ.
<i>*fac_ptr</i>	A pointer to a user buffer which contains the X.25 facilities that are to be included in the N_EXT_CONN_REQ sent to NPI. If this pointer is NULL, then no facilities will be included and the primitive sent to NPI will be an N_CONN_REQ rather than an N_EXT_CONN_REQ.
<i>fac_cnt</i>	The number of bytes in user provided facilities. If this parameter is less than or equal to zero, then no facilities will be included and the primitive sent to NPI will be an N_CONN_REQ rather than an N_EXT_CONN_REQ.

Return

< 0	An error occurred (message written to log file).
> 0	Size of N_CONN_CON if success.

npi_ext_listen()

Prototype `int npi_ext_listen(char *nsap,
 unsigned fork_optns,
 char *fac_ptr,
 int fac_cnt) ;`

Include File(s) **<gcom/npiapi.h>**

Description The *npi_ext_listen()* routine functions just like the *npi_listen()* routine with the addition of the facilities parameters. The indicated facilities are used in the N_EXT_CONN_RES that is returned to NPI upon receipt of the N_CONN_IND. Note that an application using a permanent virtual circuit (PVC) should never use any form of a listen. An application should always connect to the PVC.

Parameters

nsap NSAP address in ASCII string form suitable for passing to the routine *npi_bind_ascii_nsap()*. See *npi_bind_ascii_nsap()* for a description of this parameter.

fork_optns Options that control the process forking behavior of *npi_listen()*. See “NPI API Fork Options” on page 21 for a description of the option values.

fac_ptr A pointer to a user buffer which contains the X.25 facilities that are to be included in the N_EXT_CONN_RES sent to NPI to accept the incoming connection. If this pointer is NULL, then no facilities will be included and the primitive sent to NPI will be an N_CONN_RES rather than an N_EXT_CONN_RES.

fac_cnt The number of bytes in user provided facilities. If this parameter is less than or equal to zero, then no facilities will be included and the primitive sent to NPI will be an N_CONN_RES rather than an N_EXT_CONN_RES.

Return < 0 Any kind of error that occurred in the binding and connection process. If the return is negative, typically some diagnostic message is logged by the NPI API software. In the case of negative returns with the forking option selected, you must call *npi_listen()* again to reinstate the listening function.

 > 0 Successful return. The return value is the UNIX file descriptor for the stream that has the open connection assigned to it. This descriptor is suitable for passing to any of the NPI API routines that have a stream parameter, or the UNIX *read*, *write*, *getmsg*, *putmsg* family of routines.

npi_ext_nbio_complete_listen()

Prototype `int npi_ext_nbio_complete_listen(int listen_fid,
 int options,
 char *fac_ptr,
 int fac_cnt) ;`

Include File(s) **<gcom/npiapi.h>**

Description The *npi_ext_nbio_complete_listen()* routine is called by your application in response to a *poll()* call that indicates “data available”. Your application must first call *npi_listen()* with the non-blocking I/O flag set.

Parameters *listen_fid* NPI listening stream file descriptor. See “*npi_listen()*” on page 70 for further details.

options Options that control the process forking behavior of *npi_listen()*. See “NPI API Fork Options” on page 21 for a description of the option values.

**fac_ptr* A pointer to a user buffer which contains the X.25 facilities that are to be included in the N_EXT_CONN_RES sent to NPI to accept the incoming connection. If this pointer is NULL, then no facilities will be included and the primitive sent to NPI will be an N_CONN_RES rather than an N_EXT_CONN_RES.

fac_cnt The number of bytes in user provided facilities. If this parameter is less than or equal to zero, then no facilities will be included and the primitive sent to NPI will be an N_CONN_RES rather than an N_EXT_CONN_RES.

Return = -1 An error has occurred.

 >= 0 Successful return. The return value is the UNIX file descriptor for the stream that has the open connection assigned to it. This descriptor is suitable for passing to any of the NPI API routines that have a stream parameter or the UNIX *read*, *write*, *getmsg*, *putmsg* family of routines.

npi_fac_walk()

Prototype `int npi_fac_walk(char *facp,
 unsigned facl,
 facil_proc_t *fcn) ;`

Include File(s) **<gcom/npiapi.h>**

Description The *npi_fac_walk()* routine walks a set of X.25 facilities as returned by the *npi_get_facils()* routine. For each facility found it calls a user-provided routine with parameters indicating the particular facility. The user routine can, in this manner, easily find certain facilities of interest to the user. For example, the user routine could pick out just the charging information related facilities.

The user-provided routine must correspond to the following prototype:

```
typedef int facil_proc_t(char fref, char *fval,  
                                          unsigned flgth, int marker) ;
```

The parameters to the *facil_proc_t* user routine are as follows.

fref The facility reference code as defined by CCITT Recommendation X.25.

fval A pointer to the facility value associated with the facility.

flgth The length, in bytes, of the facility value.

marker A count of the number of “facility markers” found thus far in the facilities. Standard CCITT defined facilities always precede the first facility marker.

The *flgth* parameter specifies only the length of the facility value. That is, for a class A facility this parameter will be equal to 1. For class C facilities, the *flgth* parameter will be equal to the “facility parameter field length” plus 1. That is, it will represent the length of the entire facility minus the facility code byte itself.

The user function must return the value zero to keep the “walking” procedure going. A non-zero return will terminate the facility walking procedure. Such a return value will be passed back to the caller of the *npi_fac_walk()* function.

Parameters *facp* Pointer to X.25 facilities as returned by the *npi_get_facils()* routine.

facl Length of facilities as returned by *npi_get_facils()*.

fcn User-provided function as described above.

Return < 0 An error occurred (ill-formed facility field).

 > 0 Successful return.

npi_flags_connect_wait()

Prototype `int npi_flags_connect_wait(int listen_fid,
 char *fac_ptr,
 int fac_cnt,
 int bind_flags) ;`

Include File(s) `<gcom/npia.h>, <gcom/npixt.h>, <gcom/npi.h>`

Description The *npi_flags_connect_wait* routine operates just like *npi_connect_wait* except that it uses the specified X.25 facilities when accepting the connection and also allows the user to specify additional flags to the bind request.

Parameters *npi_data* The file descriptor of the listener NPI data stream. This stream will continue listening for new connections once the routine returns.

fac_ptr A pointer to a user buffer which contains the X.25 facilities that are to be included in the N_EXT_CONN_RES sent to NPI to accept the incoming connection. If this pointer is NULL, then no facilities will be included and the primitive sent to NPI will be an N_CONN_RES rather than an N_EXT_CONN_RES.

fac_cnt The number of bytes in user provided facilities. If this parameter is less than or equal to zero, then no facilities will be included and the primitive sent to NPI will be an N_CONN_RES rather than an N_EXT_CONN_RES.

bind_flags The flags to use with the BIND.

Return `< 0` Any kind of error that occurred in the binding and connection process. If the return is negative, typically some diagnostic message is logged by the NPI API software. In the case of negative returns with the forking option selected, you must call *npi_listen()* again to reinstate the listening function.

`>=0` The new file descriptor opened to the incoming connection.

npi_get_a_msg()

Prototype `int npi_get_a_msg(int npi_data,
 char *buf,
 int cnt) ;`

Include File(s) **<gcom/npiapi.h>**

Description The *npi_get_a_msg()* routine issues a *getmsg* system call on the indicated stream. The M_PROTO portion of the received message is read into the global array *npi_ctl_buf* and *npi_ctl_cnt* gives its length, which is negative if no M_PROTO portion is received. The data portion is read into your buffer. The number of data bytes read is contained in the global *npi_data_cnt* upon return from the routine, negative if no data portion was read.

Parameters

<i>npi_data</i>	The stream from which to get a message
<i>*buf</i>	The buffer into which to receive the data portion of the message
<i>cnt</i>	The maximum amount of data to receive

Return

< 0	An error occurred (message written to log file).
> 0	Return value from <i>getmsg</i> .

npi_get_a_proto()

Prototype `int npi_get_a_proto(int npi_data) ;`

Include File(s) **<gcom/npiapi.h>**

Description The *npi_get_a_proto()* routine uses *npi_get_a_msg()* to read a message into *npi_ctl_buf* and *npi_data_buf*. If the message consists of an M_PROTO, the routine returns success. If the message contains a data block or if there is no M_PROTO, the routine writes messages to the log and fails.

Parameters *npi_data* The stream from which to get a message

Return < 0 Failure (message written to log file).

 > 0 Size of the M_PROTO, received.

npi_get_and_log_facils()

Prototype `void npi_get_and_log_facils(int npi_data) ;`

Include File(s) **<gcom/npiapi.h>**

Description The *npi_get_and_log_facils()* routine obtains a copy of the X.25 facilities that have been utilized for the NPI connection designated by *npi_data*. The facilities are decoded and printed to the screen and/or the NPI log file according to the log options established via the *npi_init()* function call.

Parameters *npi_data* NPI stream to use to access facilities. The stream must be open.

npi_get_facils()

Prototype `int npi_get_facils(int npi_data,
 char *fac_ptr,
 int fac_cnt) ;`

Include File(s) **<gcom/npiapi.h>**

Description The *npi_get_facils()* routine obtains a copy of the X.25 facilities that have been used for the NPI connection designated by *npi_data*. The facilities are returned to the user in binary form at the location specified by the parameter *fac_ptr*. At most *fac_cnt* bytes of facilities will be returned to the user.

This routine can be called at any time that the stream is open. If it is called just after a successful NPI connection has been established, then the facilities returned will represent those used in the call setup phase of the connection. If it is called after the completion of an NPI disconnect procedure, then the facilities will be the accumulated facilities from the call setup phase and the call clearing phase of the connection.

Calling this routine after NPI disconnection allows the user to obtain charging information associated with the X.25 call.

Consult *CCITT Recommendation X.25* and the GCOM document, *UNIX STREAMS Administrator's Guide*, for the format of the facilities.

<i>Parameters</i>	<i>npi_data</i>	NPI stream to use to access facilities. The stream must be open.
	<i>fac_ptr</i>	Pointer to user's memory area which is to receive the facilities. This area should be large enough to accommodate the largest possible set of facilities that can be returned. The X.25 standards limits the size of the facility field to 109 bytes. Since the facilities are accumulated they could possibly exceed this size. Allow 200 bytes to be safe.
	<i>fac_cnt</i>	Maximum size of the user's area.
<i>Return</i>	<code>< 0</code>	An error occurred.
	<code>> 0</code>	Size of the facilities returned.

npi_get_stream_info()

Prototype `int npi_get_stream_info(int fid,npi_stream_info_t
 *info_ptr);`

Include File(s) **<gcom/npiapi.h>**

Description The *fid* routine is an open file descriptor for an NPI data stream obtained by calling *npi_open*. The routine *info_ptr* is a pointer to an *npi_stream_info_t* structure.

This routine returns detailed information about the open stream including packet counts and lower level linkage information. See **npiapi.h** for additional details.

npi_info_req()

Prototype `int npi_info_req(int strm) ;`

Include File(s) `<gcom/npiapi.h>`, `<gcom/npi.h>` (for `N_info_ack_t`)

Description This routine sends an N_INFO_REQ to NPI and waits for an N_INFO_ACK. The contents of the N_INFO_ACK will be in the `npi_ctl_buf`, which the user can cast to an `N_info_ack_t` to retrieve its contents.

Parameters `npi_data` The stream on which to send the N_INFO_REQ.

Return `<0` Error condition.

`>0` Success.

npi_init()

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

Include File(s) **<gcom/npiapi.h>**

Description The *npi_init()* routine performs initialization functions for the NPI API Library. It should be called prior to using any other library function routine.

Parameters *log_optns* Options for controlling messages that the NPI API Library writes to its log file. This parameter consists of a number of single-bit values that are ORed together to form the parameter value. See “NPI API Logging Options” on page 22 for the options.

log_name Pointer to an ASCII string, or NULL. The string provides the name of the file that the NPI API will use for its log file. A NULL pointer results in the use of the default log file as defined in “NPI API Constants” on page 19.

Return < 0 Unsuccessful initialization.
 > 0 Successful initialization.

npi_listen()

Prototype `int npi_listen(char *nsap,
 unsigned fork_optns) ;`

Include File(s) **<gcom/npiapi.h>**

Description The *npi_listen()* routine listens for an incoming connection and returns the file descriptor of a data stream that is open for reading and writing data to NPI.

The routine issues an N_BIND_REQ using the *npi_bind_ascii_nsap()* routine. It then waits for an incoming N_CONN_IND message from NPI. It accepts the connection, transferring it to a new stream, and returns the file descriptor for the new stream to the caller.

The *fork_optns* control the process forking behavior of the function. Once the connection has been accepted and transferred to a new stream, the routine either returns directly to the caller or it forks a child process that returns the file descriptor to the caller.

In the case of direct return (no forking) the listening stream is left open so that additional incoming connection indications can be queued. The caller is given the file descriptor of a different stream for exchanging data. If you finish with the data stream and want to listen for more incoming connections, close the data stream and call *npi_listen()* again. It returns with the file descriptor for the next incoming connection.

In the case of forking a child process, the child returns the descriptor for the data stream and the parent continues to listen for incoming connections on the listener stream. Upon successful return, your program is running as the child process. When you are finished with the stream, simply exit the process.

When forking is taking place and children are exiting, the child processes will show up as *<defunct>* in a UNIX process status. This condition persists until the *npi_listen()* function of the parent receives another incoming connection. At that time, it receives notification of the exit status of its children and the *<defunct>* processes are completely terminated.

Subsequent calls to *npi_listen()* will never allow the user to create multiple listener streams. The *npi_listen()* routine will create and manage only a single listener stream per process. A user wishing to listen on several addresses can either wildcard the NSAP address, perform the binds directly and monitor those streams for incoming connect indications, or fork multiple processes and set up a listener stream on each.

Also note that connections to a permanent virtual circuit (PVC) never use any variant of a listen. An application should always connect to the PVC.

<i>Parameters</i>	<i>nsap</i>	NSAP address in ASCII string form suitable for passing to the routine <i>npi_bind_ascii_nsap()</i> . See <i>npi_bind_ascii_nsap()</i> for a description of this parameter.
	<i>fork_opts</i>	Options that control the process forking behavior of <i>npi_listen()</i> . If this parameter's NPI_NBIO bit is set, the value returned is the file descriptor for the listening stream. See "NPI API Fork Options" on page 21 for a description of the bit option values.
<i>Return</i>	< 0	Any kind of error that occurred in the binding and connection process. If the return is negative, typically some diagnostic message is logged by the NPI API software. In the case of negative returns with the forking option selected, you must call <i>npi_listen()</i> again to reinstate the listening function.
	> 0	Successful return. The return value is the UNIX file descriptor for the stream that has the open connection assigned to it. This descriptor is suitable for passing to any of the NPI API routines that have a stream parameter, or the UNIX <i>read</i> , <i>write</i> , <i>getmsg</i> , <i>putmsg</i> family of routines. If the NPI_NBIO bit is set in the <i>fork_opts</i> parameter, the value returned is the file descriptor for the listening stream. This value is suitable for use with the <i>npi_nbio_complete_listen()</i> and <i>npi_ext_nbio_complete_listen()</i> routines.

Return value is FID for listening stream if NPI_NBIO is set

npi_max_sdu()

Prototype `int npi_max_sdu(int npi_data) ;`

Include File(s) **<gcom/npiapi.h>**

Description This routine makes use of the *N_INFO_REQ* and *N_INFO_ACK* NPI protocol messages to determine what the maximum protocol size is on a given data stream.

Parameters *npi_data* The data stream for which the maximum packet size should be determined.

Return `< 0` Error condition.
 `> 0` Maximum packet size.

npi_open()

Prototype Old: int npi_open_data(void);
 New: int npi_open(char *hostname);

Include File(s) <gcom/npiapi.h>

Description This routine opens a stream to the NPI driver. It uses the clone open facility to do so. This routine is used internally by the NPI API Library to open files to the NPI driver. It is ordinarily not called by the user but is provided to allow you to perform low level functions with the NPI driver.

The npi_open_data(void) routine is equivalent to npi_open(NULL).

Return < 0 The file open failed (message written to log file).
 > 0 File descriptor for the open file.

Parameters *hostname* A pointer to an ASCII string name of the machine on which the operation is to be performed. If the pointer is NULL, or points to an empty string, the operation is performed on the machine on which the call to the API library is made (i.e., the local host).

npi_perror()

Prototype `void npi_perror(char *msg) ;`

Include File(s) **<gcom/npiapi.h>**

Description The *npi_perror()* routine is similar to the UNIX *perror* command, but it also prints to the log file.

Parameters *msg* Points to the message to be logged.

npi_printf()

Prototype `void npi_printf(char *fmt, ...) ;`

Include File(s) **<gcom/npiapi.h>**

Description The *npi_printf()* routine performs the same function as the UNIX *printf* routine. It performs its printing functions to the log file. The logging options specified in the *npi_init()* call determine whether the output is written to the log file, *stderr*, both or neither. This is the routine that the NPI API Library uses internally to write messages to the log.

The format of the output produced by *npi_printf()* consists of the process ID of the process that called the function, a time stamp and the message formatted according to the arguments passed to *npi_printf()*. For example,

```
npi_printf("Hello world!\n") ;
```

would produce output similar to the following:

```
609 08:14:33 Hello world
```

Parameters *fmt* Format string compatible with UNIX *printf* function
... Additional arguments to print (optional)

npi_print_stream_info()

Prototype void npi_print_stream_info(npi_stream_info_t*p);

Include File(s) <gcom/npiapi.h>

Description The *npi_print_stream_info* routine uses *npi_printf* to print out all the fields of the structure in ASCII. The *p* is a pointer to an *npi-stream_info_t* structure.

npi_put_data_buf()

Prototype `int npi_put_data_buf(int npi_data,
 int lgth) ;`

Include File(s) **<gcom/npiapi.h>**

Description The *npi_put_data_buf()* routine uses the *npi_write_data()* function to write *lgth* bytes from the global array *npi_data_buf* to the given stream.

Parameters *npi_data* Stream to which to write data
lgth Number of bytes to write

Return < 0 An error occurred (message written to log file).
 > 0 Return value from the UNIX *putmsg* call.

npi_put_proto()

Prototype `int npi_put_proto(int npi_data,
 int lgth) ;`

Include File(s) **<gcom/npiapi.h>**

Description The *npi_put_proto()* routine writes an M_PROTO to the NPI driver on the given stream. The M_PROTO is assumed to have been built in the global array *npi_ctl_buf*. This routine does not log the message even if the NPI_LOG_TX_PROTOS option is set.

Parameters *npi_data* Stream to which to write M_PROTO.
lgth Number of bytes to write.

Return < 0 An error occurred (message written to log file).
 > 0 Return value from the UNIX *putmsg* call.

npi_rcv()

Prototype

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

Include File(s) <gcom/npiapi.h>, <gcom/npi.h>, <gcom/npiext.h>

Description The *npi_rcv()* routine reads data and processes protocol messages from a stream. This routine is more sophisticated than the *npi_read_data()* routine, which simply reads a single data message without processing protocol messages. Unlike the *npi_get_a_proto()* routine, which just handles M_PROTO protocol messages, the *npi_rcv()* routine also handles normal and expedited data.

Normal data (N_DATA) and expedited data (N_EXDATA) are collectively called Network Service Data Units (NSDUs).

The *flags_in* parameter allows the *npi_rcv()* routine to partially support delivery confirmation and the returning of reset responses for a reset indication. See “Handling Resets” on page 83 and “Delivery Confirmation” on page 84 for details.

The *flags_out* parameter can indicate that the incoming data requested delivery confirmation and allows *npi_rcv()* to handle fragmented NSDUs while forwarding more data (M-bit) indications. An NSDU will be fragmented if the application’s input buffer is smaller than the incoming NSDU. See “Handling Incoming Fragmented Data” on page 85 for details.

Handling Resets

The reset service is defined in the *NPI Specification* as follows:

The *reset service* can be used by the network service user to resynchronize the use of the network connection; or by the network service provider to report detected loss of data unrecoverable within the network service.

If *flags_in* is set to NPIAPI_USER_RESET_RES when *npi_rcv()* is called, your application is responsible for returning the reset response for a reset indication. The *npi_reset_res()* NPI API Library routine generates reset responses. Otherwise, *npi_rcv()* returns the reset response. In either case, NPIAPI_RESET_INDICATION is returned to the called. When the application next calls *npi_rcv()* and the reset sequence completes, NPIAPI_RESET_COMPLETE is returned.

Furthermore, according to the *NPI Specification*:

The N_RESET_REQ primitive acts as a synchronization mark in the flow of N_DATA, N_EXDATA and N_DATAACK primitives transmitted by the issuing network service user; the N_RESET_IND acts as a synchronization mark in the flow of N_DATA, N_EXDATA, and N_DATAACK primitives received by the receiving network service user. Similarly, N_RESET_RES acts as a synchronization mark in the flow of N_DATA, N_EXDATA, and N_DATAACK primitives transmitted by the responding network service user, while the N_RESET_CON acts as a synchronization mark in the flow of N_DATA, N_EXDATA, and N_DATAACK primitives received by the network service user that originally issued the reset. The resynchronizing properties of the reset services are the following:

- All N_DATA, N_EXDATA, and N_DATAACK primitives issued before issuing the N_RESET_REQ/N_RESET_RES that have not been delivered to the other network service user before the N_RESET_IND/N_RESET_CON are issued by the network service provider, should be discarded by the network service provider.
- Any N_DATA, N_EXDATA, and N_DATAACK primitives issued after the synchronization mark will not be delivered to the other network service user before the synchronization mark is received.

Delivery Confirmation

Delivery confirmation occurs when the incoming NSDU's delivery confirmation bit is set. Then, if NPIAPI_USER_DATA_ACK is set, the application is responsible for sending the data acknowledgement when appropriate. Otherwise, the data acknowledgement is returned by *npi_rcv()*. Furthermore, according to the *NPI Specification*:

The *receipt confirmation service* is requested by the confirmation request parameter on the N_DATA_REQ primitive. For each and every NSDU with the confirmation request parameter set, the receiving network service user should return an N_DATAACK_REQ primitive. Such acknowledgements should be issued in the same sequence as the corresponding N_DATA_IND primitives are received, and are to be conveyed by the network service provider in such a way so as to preserve them distinct from any previous or subsequent acknowledgements. The network service user may thus correlate them with the original requests by counting. When an NSDU has been segmented into more than one Network Interface Data Unit (NIDU), Only the last NIDU is allowed to request receipt confirmation.¹

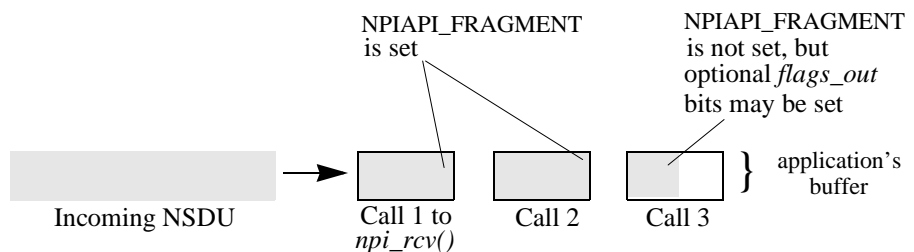
¹ UNIX International ISO Work Group, Revision 2.0.0, August 17, 1992. This applies to all future citations of the NPI specification.

Handling Incoming Fragmented Data

While it is possible for *npi_rcv()* to handle fragmented NSDUs, it might be easier to simply determine the largest incoming NSDU that you expect to receive and set your application's input buffer to that size. However, it is often difficult to make such a prediction. Therefore, your application might need to handle incoming fragmented normal and expedited data.

Figure 6, below, shows a situation where the incoming NSDU is more than twice the size of your application's buffer.

Figure 6 Handling Fragmented NSDU's



When *npi_rcv()* handles the first two buffers of the large NSDU shown above, the *flags_out* will contain the NPIAPI_FRAGMENT flag, indicating that another fragment of the same NSDU will follow. In the last buffer, the NPIAPI_FRAGMENT flag is not set. Delivery confirmation bits (such as the X.25 D-bit) and more flags (such as the X.25 M-bit) carried by the NSDU are reported to your application in the last fragment.


NPIAPI_RC_FLAG indicates that a delivery confirmation bit was set.

The NPIAPI_MORE_DATA flag indicates that a more flag, such as the X.25 M-bit, was set at the end of the NSDU. This probably means that the NSDU itself is part of a larger data unit.

Furthermore, according to the *NPI Specification*:

The network service user must send any integral number of octets of data greater than zero. In a case where the size of the NSDU exceeds the NIDU (as specified by the size of the *NICU_size* parameter of the N_INFO_ACK primitive), the NSDU may be broken up into more than one NIDU. When an NSDU is broken up into more than one NIDU, the N_MORE_DATA_FLAG will be set on each NIDU except the last one. The *RC_flag* may only be set on the last NIDU.

The size of the last data block can be determined by checking the global variable *npi_data_cnt*. Checking this variable on each transmission is usually considered good form, since it is possible for an M-bit packet to be less than full in some implementations.

<i>Parameters</i>	<i>npi_data</i>	Stream file descriptor.
	<i>*buf</i>	Buffer into which the data will be read.
	<i>cnt</i>	Size of the data buffer. If <i>cnt</i> > 0, then <i>*buf</i> is valid.
	<i>flags_in</i>	One of the following input flags: NPIAPI_USER_DATA_ACK. Indicates how <i>npi_rcv()</i> should respond to received data with the NPIAPI_RC_FLAG set. If the NPIAPI_USER_DATA_ACK <i>flags_in</i> bit is set, the application is responsible for returning a data acknowledgement (see “ <i>npi_dataack_req()</i> ” on page 38 for details). Otherwise, <i>npi_rcv()</i> returns the data acknowledgement. Independent of this bit value, NPIAPI_RC_FLAG is returned when data is received with the NPI_RC_FLAG bit. NPIAPI_USER_RESET_RES. Indicates how <i>npi_rcv()</i> should respond to a reset indication. If NPIAPI_USER_RESET_RES is set, the application is responsible for returning a reset response after taking appropriate recovery actions (see “ <i>npi_reset_res()</i> ” on page 92 for details). Otherwise, <i>npi_rcv()</i> returns a reset response. In either case, <i>npi_rcv()</i> returns NPIAPI_RESET_INDICATION to the caller.
		 Note: <i>If your application is using non-blocking I/O on the data stream, it should generate responses because a retry mechanism can be supplied by your application.</i>
	<i>*flags_out</i>	Used to return one of the following flags: NPIAPI_MORE_DATA. Data was received with the N_MORE_DATA_FLAG bit set. NPIAPI_MORE_DATA is set in <i>flags_out</i> only on the last fragment of a NSDU. That is, only if NPIAPI_FRAGMENT is not set in <i>flags_out</i> . NPIAPI_RC_FLAG. Data was received with the N_RC_FLAG set. If the NPIAPI_USER_DATA_ACK <i>flags_in</i> bit was not set, <i>npi_rcv()</i> sends a N_DATAACK to the NPI provider. NPIAPI_RC_FLAG is set in <i>flags_out</i> only on the last fragment of a NSDU. That is, only if NPIAPI_FRAGMENT is not set in <i>flags_out</i> .

NPIAPI_FRAGMENT.

This indicates that your application's buffer contains a fragment of the incoming NSDU, and more fragments are expected. This flag is returned on successive calls by *npi_rcv()* until the last fragment of the NSDU is returned. This flag is not set on the last fragment.

N_X25_Q_BIT.

Indicates that your application's buffer contains data marked with the Q-bit. Q-bit data is sometimes referred to as "qualified" data. The significance of the Q-bit varies from application to application, but it is often used to distinguish control messages from normal data.

<i>Return Values</i>	The return code is attempting to describe what was received by a single call to <i>npi_rcv()</i> . The valid return values are described as follows.
Errors	<p>NPIAPI_NO_NOTHING Contains neither control nor data. A hang-up has occurred.</p> <p>NPIAPI_PARAM_ERROR A parameter error occurred. Flags pointer is NULL or buffer pointer is NULL or buffer length is non-positive.</p> <p>NPIAPI_NOT_INIT NPI was not initialized with <i>npi_init()</i>.</p> <p>NPIAPI_GETMSG_ERROR The FID was not usable. Interrupted system calls are returned by <i>npi_rcv()</i> or some other error has occurred.</p> <p>NPIAPI_EAGAIN No data or control message available. This is returned only if non-blocking I/O was sent on the stream descriptor. This return code does not indicate a serious error, but demonstrates the lack of any messages to read from the stream.</p>
Non-errors	<p>NPIAPI_NORMAL_DATA Normal data. In this case, the NPIAPI_FRAGMENT flag may be set, and the NPIAPI_RC_FLAG or NPIAPI_MORE_DATA flags may be set (but not both) or no flags may be set.</p> <p>NPIAPI_EXPEDITED_DATA Expedited data can have only the NPIAPI_FRAGMENT flag set.</p>

- Return Values (cont.)* **NPIAPI_DATA_ACK**
 The application previously sent data with **N_RC_FLAG** set. The **N_DATAACK** acknowledges that previous transmission.
- NPIAPI_DISC_IND**
 The connection has been disconnected. Data might be present.
- NPIAPI_RESET_INDICATION**
 An **N-RESET_IND** message was received. See **NPIAPI_USER_RESET_RES** *flags_in* description on page 86 for details.
- NPIAPI_RESET_COMPLETE**
 A reset sequence has completed and the application can now resume data transfer.
- NPIAPI_CONNECT_COMPLETE**
 A connection has been established. Either a **N_CONN_CON** or an **N_OK_ACK** to a **N_CONN_RES** was received. Data might be present.
- NPIAPI_CONNECT_IND**
 An **NPIAPI_CONNECT_IND** has been received and copied to the global array *npi_conn_ind*. You can analyze this message by pointing a pointer of type *N_conn_ind_t** to that array. The structure definition is in the file <gcom/npi.h>.
- NPIAPI_EXT_CONNECT_IND**
 An **NPI_EXT_CONNECT_IND** has been received and copied to the global array *npi_conn_ind*. You can analyze this message by pointing a pointer of type *N_ext_conn_ind_t** to that array. The structure definition is in the file <gcom/npiext.h>.
- NPIAPI_EXT2_CONNECT_IND**
 An **NPI_EXT2_CONN_IND** has been received and copied to the global array *npi_conn_ind*. You can analyze this message by pointing a pointer of type *N_ext2_conn_ind_t** to that array. The structure definition is in the file <gcom/npiext.h>.
- NPIAPI_EXT_CONNECT_COMPLETE**
 Similar to **NPIAPI_CONNECT_COMPLETE** except that a **N_EXT_CONN_CON** was received. The structure definition for *N_ext_conn_con_t* can be found in <gcom/npiext.h>.

NPIAPI_EXT2_CONNECT_COMPLETE

Similar to NPIAPI_CONNECT_COMPLETE except that a N_EXT_CONN_CON was received. The structure definition for *N_ext2_conn_con_t* can be found in <gcom/npixt.h>.

NPIAPI_BIND_ACK

A previously sent N_BIND_REQ was accepted by the NPI provider.

NPIAPI_INFO_ACK

This is an information acknowledgement in response to an information request by the application.

NPIAPI_ERROR_ACK

Indicates that a previously transmitted message is being rejected.

NPIAPI_OTHER

An unsupported NPI message was received.

npi_read_data()

Prototype `int npi_read_data(int npi_data,
 char *buf,
 int cnt);`

Include File(s) **<gcom/npiapi.h>**

Description The *npi_read_data()* routine reads a single data message from the NPI driver on the indicated stream. The data is read into the buffer area pointed to by *buf* and consist of at most *cnt* bytes.

This routine does not expect to read an M_PROTO from the stream. Except for two cases, to be discussed, it is considered an error to receive a M_PROTO. Similarly, if the return from *getmsg* indicates that no data message has been read, an error is returned. Under these circumstances, messages are written to the log file.

Note: *To read data and process protocol messages, use the npi_rcv() procedure.*

The routine accepts two types of M_PROTO, an N_DATA_IND message and an N_DISC_IND. The protocol header is left in the *npi_ctl_buf* global array for your examination. The data portion is transferred to your buffer and the number of data bytes read into that buffer is returned by the function. If an N_DATA_IND message is read and if it specifies the receipt confirmation option, *npi_read_data()* sends an N_DATAACK_REQ back to NPI before returning to the user. Thus, you need not be concerned with the acknowledgement of data. If a N_DISC_IND is read, the next call to *npi_read_data()* results in a negative return indicating that the connection has been closed. The N_DISC_IND will still occupy the global array *npi_ctl_buf*. If the N_DISC_IND did not contain any data, then the negative return occurs immediately upon receipt of that message.

The ordinary case is that *npi_read_data()* receives a message with no protocol portion and valid data. In such a case it simply returns the length of the data.

<i>Parameters</i>	<i>npi_data</i>	Stream from which to read data
	<i>*buf</i>	Buffer to which to read the data
	<i>cnt</i>	Maximum number of bytes to read
<i>Return</i>	< 0	An error occurred (message written to log file).
	> 0	Number of bytes read into user buffer.

npi_reset_req()

Prototype `int npi_reset_req(int npi_data) ;`

Include File(s) **<gcom/npiapi.h>**

Description This routine sends an N_RESET_REQ primitive and waits for the corresponding N_RESET_CON. Most messages that cross in the mail with the reset request will be discarded.

Test1-A
Test2-A

Parameters *npi_data* Stream file ID. This is the file descriptor returned by an open call on the **/dev/npi_clone** file.

Return == -1 The routine failed because of *getmsg()* errors and disconnect indications received

 == 0 A connect message was received, suggesting that an error occurred in your application

 > 0 The reset has completed with an N_RESET_CON

npi_reset_res()

Prototype `int npi_reset_res(int npi_data) ;`

Include File(s) **<gcom/npiapi.h>**

Description The *npi_reset_res()* routine sends a reset response to NPI and awaits the receipt of an N_OK_ACK from NPI. It is appropriate to call this routine after receiving an N_RESET_IND on a data stream.

If you are using *npi_read_data()* and if it returns a negative result, you may want to check the *npi_ctl_buf* global to see if it contains an N_RESET_IND. If so, and if you wish to keep your connection going, you may then call *npi_reset_res()* to acknowledge the reset and put the connection back in the data state.



Caution: The receipt of an N_RESET_IND means that data messages may have been lost.

Parameters *npi_data* Stream to which to send the N_RESET_RES

Return < 0 An error occurred (message written to log file).

 > 0 Size of N_OK_ACK received.

npi_send_connect_req()

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

Description The *npi_send_connect_req()* routine formats and sends an NPI connect request in a connection-oriented environment.

Include File(s) **<gcom/npiapi.h>**

Parameters *npi_data* Stream file ID. This is the file descriptor returned by an open call on the **/dev/npi_clone** file.

**peer_sap* Address of the peer to which you wish to be connected.

**buf* Data accompanying the request.

cnt Length of data.

Return < 0 Failure

 >= 0 Success.

npi_send_ext_connect_req()

Prototype `int npi_send_ext_connect_req(int npi_data,
 char *peer_sap,
 char *buf,
 int cnt,
 char *fac_ptr,
 int fac_cnt) ;`

Description The *npi_send_ext_connect_req()* routine sends an extended connect request that includes facilities. Use this if facilities are present. Otherwise, use *npi_send_connect_req()*.

Include File(s) **<gcom/npiapi.h>**

Parameters *npi_data* Stream file ID. This is the file descriptor returned by an open call on the **/dev/npi_clone** file.

**peer_sap* Address of the peer to which you wish to be connected.

**buf* Data accompanying the request.

cnt Length of data.

**fac_ptr* A pointer to a user buffer which contains the X.25 facilities that are to be included in the N_EXT_CONN_RES sent to NPI to accept the incoming connection. If this pointer is NULL, then no facilities will be included and the primitive sent to NPI will be an N_CONN_RES rather than an N_EXT_CONN_RES.

fac_cnt The number of bytes in user provided facilities. If this parameter is less than or equal to zero, then no facilities will be included and the primitive sent to NPI will be an N_CONN_RES rather than an N_EXT_CONN_RES.

Return `< 0` Failure. The software may not be properly initialized.

`>= 0` Success.

npi_send_info_req()

Prototype int npi_send_info_req (int npi_data) ;

Description The *npi_send_info_req()* routine formats and sends an N_INFO_REQ primitive. There is no need to wait for the acknowledgement.

Include File(s) <**gcom/npiapi.h**>

Parameters *npi_data* Stream file ID. This is the file descriptor returned by an open call on the **/dev/npi_clone** file.

Return < 0 Failure. The software may not be properly initialized.
 >= 0 Request successfully sent.

npi_send_reset_req()

Prototype `int npi_send_reset_req (int npi_data) ;`

Description The *npi_send_reset_req()* routine formats and sends an N_RESET_REQ primitive on the specified data stream. There is no need to wait for the response.

Include File(s) **<gcom/npiapi.h>**

Parameters *npi_data* Stream file ID. This is the file descriptor returned by an open call on the **/dev/npi_clone** file.

Return `< 0` Failure. The software may not be properly initialized.
 `>= 0` Request successfully sent.

npi_send_reset_res()

Prototype `int npi_send_reset_res(int npi_data) ;`

Description The *npi_send_reset_res()* routine formats and sends a NPI reset response. A reset response is sent in response to a received reset indication. There is no need to wait for the response.

Include File(s) **<gcom/npiapi.h>**

Parameters *npi_data* Stream file ID. This is the file descriptor returned by an open call on the **/dev/npi_clone** file.

Return `< 0` Failure. The software may not be properly initialized.
 `>= 0` Message sent.

npi_set_log_size()

Prototype `int npi_set_log_size(long nbytes) ;`

Include File(s) **<gcom/npiapi.h>**

Description This routine sets the maximum length of the NPI API log, in bytes. If *nbytes* is 0, the log will be allowed to grow without bound. For any other size, the log will be wrapped when *nbytes* bytes have been written to the log.

Parameters *nbytes* Maximum length, in bytes, of the log.

Return < 0 Error condition.

0 Success.

npi_set_marks()

Prototype `void npi_set_marks(int fid,
 unsigned rd_lo_mark,
 unsigned rd_hi_mark,
 unsigned wr_lo_mark,
 unsigned wr_hi_mark) ;`

Include File(s) **<gcom/npiapi.h>**

Description This routine allows the user to set the STREAMS high and low watermarks for a given data stream. These marks are used to control the automatic flow control provided by NPI.

When a stream has more messages queued than the high watermark, NPI will exert backpressure on the stream by deferring acceptance of any additional messages. When the messages are processed until the stream's queue drops below the low watermark, NPI will resume accepting new messages. NPI maintains separate queues for inbound and outbound messages, so separate watermarks are also used.

Parameters *fid* The data stream on which to set the watermarks.
 rd_lo_mark The low watermark for inbound traffic.
 rd_hi_mark The high watermark for inbound traffic.
 wr_lo_mark The low watermark for outbound traffic.
 wr_hi_mark The high watermark for outbound traffic.

npi_set_pid()

Prototype `void npi_set_pid(int fid) ;`

Include File(s) **<gcom/npiapi.h>**

Description When an application first opens a data stream, the API automatically registers the calling program's process ID number (PID) with the STREAMS driver by invoking this routine. The user should almost never be required to invoke this directly.

Parameters *fid* The stream on which to register the PID.

<i>Parameters</i>	<i>fid</i>	NPI data stream file descriptor.
	<i>func</i>	User application's signal message handler function to be invoked when the specified signal is caught by the NPI API.
	<i>sig_num</i>	Signal to be generated by the NPI provider as it sends to the user a message specified in the <i>primitive_mask</i> argument. If <i>sig_num</i> is zero, no further signals are generated by the NPI provider.
	<i>primitive_mask</i>	The set of NPI protocol messages that cause the NPI provider to generate a signal as it sends the message to the user. The NPI provider can generate the signal when sending a N_DISCON_IND message. That is, the only bit in the <i>primitive_mask</i> that is examined is (1 << N_DISCON_IND). If <i>primitive_mask</i> is zero, no further signals are generated by the NPI provider.
<i>Implementation Restrictions</i>		<ul style="list-style-type: none"> • Only one signal is supported, SIGPOLL, also known as SIGIO on some UNIX systems. This restriction is caused by an incomplete implementation of the STREAMS mechanism that associates a process group with a stream head. • Only one file descriptor (<i>fid</i>) may be used at a time. That is, when a signal is caught, the <i>fid</i> from the most recent call on <i>npi_set_signal_handling()</i> is used to read messages.
<i>Return</i>	<i>< 0</i>	Success.
	<i>> 0</i>	Failure.

npi_want_a_proto()

Prototype `int npi_want_a_proto(int npi_data,
 int proto_type) ;`

Include File(s) `<gcom/npiapi.h>, <gcom/npi.h>, <gcom/npiext.h>` (for *proto_type*)

Description The *npi_want_a_proto()* routine uses *npi_get_a_proto()* to read a protocol message from the indicated stream. It checks to make sure that the protocol message read is of the desired type. It returns the size of the protocol message if it succeeds. The protocol message is left in the *npi_ctl_buf* global array so that the user can examine it.



Caution: This routine is used internally by the NPI API Library to receive expected protocol responses during bind, connect and reset phases of the connection. You may wish to use this routine if you are programming the NPI driver at a low level.

Parameters *npi_data* The stream from which to receive a protocol message.
 proto_type The type of protocol message to be received.

Return `< 0` An error occurred (message written to log file). The kinds of errors that can occur are: message read was not a protocol message; message was too short; protocol message type was invalid; protocol message was not of the desired type.
 `> 0` Size of the protocol message received.

npi_write_data()

Prototype `int npi_write_data(int npi_data,
 char *buf,
 int cnt) ;`

Include File(s) **<gcom/npiapi.h>**

Description The *npi_write_data()* routine writes a data message to the NPI driver on the indicated stream. The data are located in the buffer area pointed to by *buf* and consist of *cnt* bytes.

If the data message exceeds the size of a packet, the NPI driver will automatically segment the message into packets, using X.25's M-bit mechanism to indicate that they should be reassembled at the other end.

Parameters *npi_data* The stream to which to write data.
buf The buffer from which to write the data.
cnt The number of bytes to write.

Return < 0 An error occurred (message written to log file).
 >= 0 Return value from the UNIX *putmsg* call.

npi_x25_clear_cause()

Prototype `char *npi_x25_clear_cause(int cause) ;`

Include File(s) **<gcom/npiapi.h>**

Description This routine translates an X.25 clear cause into a descriptive ASCII string.

Parameters *cause* The X.25 clear cause which should be converted to a descriptive phrase.

Return Static ASCII string describing the clear cause.

npi_x25_diagnostic()

Prototype `char *npi_x25_diagnostic(int diagnostic) ;`

Include File(s) **<gcom/npiapi.h>**

Description This routine converts an X.25 diagnostic code into an ASCII string describing the diagnostic.

Parameters *diagnostic* The X.25 diagnostic code which should be converted into an ASCII string.

Return A pointer to a static ASCII string describing the diagnostic code.

npi_x25_registration_cause()

- Prototype* `char *npi_x25_registration_cause(int cause) ;`
- Include File(s)* **<gcom/npiapi.h>**
- Description* This routine converts an X.25 registration cause code into an ASCII string describing the registration cause.
- Parameters* *cause* The registration cause which should be converted to an ASCII string.
- Return* A pointer to a static ASCII string describing the registration cause.

npi_x25_reset_cause()

Prototype `char *npi_x25_reset_cause(int cause) ;`

Include File(s) **<gcom/npiapi.h>**

Description This routine converts an X.25 reset cause code into an ASCII string describing the reset cause.

Parameters *cause* The reset cause which should be converted into an ASCII string.

Return Pointer to a static ASCII string describing the reset cause.

npi_x25_restart_cause()

Prototype `char *npi_x25_restart_cause(int cause) ;`

Include File(s) **<gcom/npiapi.h>**

Description This routine converts an X.25 restart cause code into an ASCII string which describes the restart cause.

Parameters *cause* The restart cause code which should be translated into an ASCII string.

Return A pointer to a static ASCII string describing the restart cause.

SECTION 4

The PU Info API

114	<i>Introduction to the PU Info API</i>
117	<i>pu_decode_handle()</i>
118	<i>pu_dlp_i_upa()</i>
119	<i>pu_get_pu_id()</i>
120	<i>pu_get_stats()</i>
121	<i>pu_get_board_info()</i>
122	<i>pu_get_npi_strm_stats()</i>
123	<i>pu_id_to_pu_handle()</i>
124	<i>pu_id_to_pu_number()</i>
125	<i>pu_map_npi_lpa_to_handle()</i>
126	<i>pu_strerror()</i>

Introduction to the PU Info API

The PU Info API was originally designed to support users building diagnostic tools for monitoring SNA. The API has been generalized beyond this initial concept to support X.25 and Bisync in addition to SNA. The current incarnation of the PU Info API provides tools for monitoring the status of downstream traffic for these protocols.

Using the PU Info API

The PU Info API is designed around a PU handle and a PU ID. The PU ID is assigned to a given PU by the configuration process. For SNA, the PU ID is the PU ID number used by SNA. For X.25 and Bisync, *Gcom_config* and *Gcom_monitor* have configuration parameters in their vocabulary which allow explicitly setting the PU ID number for a given NPI lower.

A PU handle is derived from a combination of different components, encoded into a single unsigned long integer. The PU handle and PU ID number are used to confirm one another, to assure that configuration commands have not changed the underlying configuration between creation of the handle and its use.

Table 3 Errorcode Defines

Mnemonic #define name	Num	Description	Likely causes
PUE_SUCCESS	0	Success	
PUE_OPEN_ERROR	-1	Control stream open failure	Dev.gcom not running or the user does not have read and write access to the /dev/gcom/ control stream special files.
PUE_MOD_STATS	-2	Module stats request failure	Should not occur.
PUE_LWR_STATS	-3	<i>lwr_stats</i> management request failure	A user specified PU handle (NPI LPA) is no longer valid. The PU has likely been de-configured.
PUE_LWR_CONFIG	-4	<i>lwr_config</i> management request failure	Returned by <i>pu_get_board_info()</i> and <i>pu_get_board_info()</i> when the NPI LPA specified by the PU handle is no longer configured.
PUE_UPR_STATS	-5	<i>upr_stats</i> management request failure	Returned by <i>pu_get_board_info()</i> to indicate a likely configuration error. (Should not occur.)
PUE_INVALID_HANDLE	-6	<i>pu_handle</i> non-positive	A user specified PU handle (NPI LPA) is non-positive.
PUE_EXPECTED_SNA_MODULE	-7	NPI LPA not SNA server	The NPI LPA specified by a PU handle is not an SNA server instance.
PUE_EXPECTED_DLPI_USER	-8	NPI LPA not DLPI user interface	The SNA PU (NPI LPA) is not configured with a DLPI USER interface to a DLPI provider supported link level. This condition may be caused by improperly specified configuration.
PUE_EXPECTED_CDI_USER	-9	DLPI LPA not CDI user interface	A DLPI LPA is not configured with a CDI USER interface to a CDI provider supported board/port driver. This condition may be caused by improperly specified configuration.
PUE_PU_LPA_ERROR	-10	<i>pu_id</i> assertion failure	The PU specified by the PU handle is now invalid or is a different PU. That is, administrative actions have likely affected the configuration.

Table 3 Errorcode Defines

Mnemonic #define name	Num	Description	Likely causes
PUE_PORT_NOT_CONFIGURED	-11	Port/board (CDI UPA) not configured	The CDI UPA corresponding to board/port is not configured. See <i>pu_get_pu_id()</i> .
PUE_PORT_NOT_IN_USE	-12	Port/board not referenced	A SNA server based PU is not configured over the specified board/port. See <i>pu_get_pu_id()</i> .
PUE_PROVIDER_NOT_INIT	-13	No UPAs and/or LPAs	While processing a application request, a CDI, DLPI or NPI provider was encountered that had no UPAs or LPAs. That is, the <i>Gcom_monitor</i> was likely started without the -n argument.
PUE_PU_NOT_CONFIGURED	-14	Specified PU ID not configured	Returned by <i>pu_id_to_lpa()</i> to indicate that the specified PU has not been configured.
PUE_PU_UNKNOWN	-15	Unable to map from board/port	A variant of PUE_PU_NOT_CONFIGURED.
PUE_BOARD_UNKNOWN	-16	Not returned from driver	Driver did not return board number.
PUE_LINKAGE_ERROR	-17	PU not attached/bound to DLPI	The NPI LPA is not properly attached/bound to the DLPI Provider.
PUE_CDI_OPEN_ERROR	-18	CDI control stream open failure	
PUE_CDI_UPR_STATS	-19	CDI <i>upr_stats</i> management request failure	
PUE_NO_DLPI_UPA	-20	The PU ID does not have a dlpi upa	
PUE_PU_MISMATCH	-21	PU ID does not match NPI's <i>pu_id</i>	
PUE_PU_AMBIGUOUS	-22	Multiple PUs possible	
PUE_UNSUPPORTED_MODULE	-23	Unsupported module encountered	

pu_decode_handle()

Prototype `char *pu_decode_handle(ulong_t pu_handle);`

Include File(s) **<gcom/pu_info.h>**

Description This routine translates a PU handle into an ASCII string of the form:
<Type=t, CDI-UPA=u, NPI-LPA=l>.

Parameters *pu_handle* The PU handle to translate into a string.

Return Values A string describing the PU handle.

pu_dlpi_upa()

Prototype `int pu_dlpi_upa (int board,
 int port,
 int station,
 int *upa_ptr);`

Include File(s) **<gcom/pu_info.h>**

Description This fills in the DLPI UPA number given the board, port, and station. This routine will not work with Bisync, since Bisync stacks do not have a DLPI UPA.

Parameters

<i>board</i>	The board number for the station in question.
<i>port</i>	The port number for the station in question.
<i>station</i>	The station number for the station in question.
<i>upa_ptr</i>	A pointer to an integer's worth of storage. Will be filled in with the DLPI UPA number.

Return Values

0	PUE_SUCCESS: Success.
<0	Error condition. See "" on page 115.

pu_get_pu_id()

Prototype `int pu_get_pu_id (int board,
 int port,
 int station,
 int *pu_id_ptr,
 ulong_t *pu_hndl_ptr) ;`

Include File(s) **<gcom/pu_info.h>**

Description This routine uses the board number, port number, and station number to determine the PU number and prepare a PU handle for use in retrieving further statistics.

Parameters

<i>board</i>	The board number for the station in question.
<i>port</i>	The port number for the station in question.
<i>station</i>	The station number for the station in question.
<i>pu_id_ptr</i>	A pointer to an integer's worth of storage. Will be filled in with a PU ID number.
<i>pu_hndl_ptr</i>	A pointer to an unsigned long's worth of storage. Will be filled in with a handle to the PU (used for retrieving other statistics).

Return Values

0	PUE_SUCCESS: Success.
<0	Error condition. See "" on page 115.

pu_get_npi_strm_stats()

Prototype extern int pu_get_npi_strm_stats(
 int npi_data_fd,
 int *pu_up_down_ptr,
 int *link_state_ptr,
 int *modem_state_ptr
);

Include Files <gcom/pu_info.h>

Description This function is the same as *pu_get_stats*, except that it operates on an open data stream.

The *npi_data_fd* file descriptor is for an open NPI data stream.

The *pu_up_down_ptr* file descriptor points to a variable of type *int* that will receive a **1** or a **0** depending upon whether the associated PU is “up” or “down”, respectively.

The *link_state_ptr* file descriptor points to a variable of type *int* that will receive a **1** or a **0** depending upon whether the link layer below the associated PU is “up” or “down”, respectively.

The *modem_state_ptr* file descriptor points to a variable of type *int* that will receive the state of the modem signals at the underlying physical interface for the link layer below the associated PU is “up” or “down”, respectively.

pu_id_to_pu_number()

Prototype `int pu_id_to_pu_number (ulong_t pu_id);`

Include File(s) `<gcom/pu_info.h>`

Description This routine takes a PU ID number and translates it into a PU number.

Parameters `pu_id` The PU ID number to translate into a PU number.

Return Values `>=0` The PU number.
 `<0` Error condition. See “” on page 115.

pu_strerror()

Prototype `char *pu_strerror (int pu_errnum) ;`

Include File(s) **<gcom/pu_info.h>**

Description This routine translates a PU Info error return into an English-language error description.

Parameters *pu_errnum* The error code number to be converted into a string.

Return Values Returns a string describing the errorcode.

NPI Incoming Call Processing for X.25 Connections

When an incoming call packet is received by the NPI Provider from the X.25 protocol stack, NPI attempts to find a “listening” stream to associate with the X.25 virtual circuit. It does so by performing a pattern matching operation based upon information contained in the incoming call packet and information provided in the NPI bind requests of the various listening streams.

It can happen that a particular incoming call can match more than one listening pattern. In this case, the NPI Provider calculates a measure of “best fit” based upon the lengths of the various parameters that participated in the pattern match for each listening stream. The stream with the largest measure is the one to which the incoming call is assigned. The lengths involved are the number of bytes in the ASCII representation of the *nsap* and *rem_nsap* parameters, and the number of bytes of user data from the incoming call packet that are used to compare to the listening pattern.

Table 4 - NPI API Routines, below, summarizes the various components of the pattern matching.

Table 4 NPI API Routines

Listen Parameter	NPI API Routine	Description
<i>nsap</i>	npi_bind_nsap, npi_bind_ascii_nsa, npi_ext_bind_nsap, npi_ext_bind_ascii_nsap, npi_ext2_bind_nsap, npi_ext2_bind_ascii_nsap	<p>The NSAP provided in the bind request contains decimal digits. Wildcard characters “*” and “?” are options. The NSAP pattern is matched against the called address from the incoming call packet. The character “?” matches a single digit from the address. The character “*” matches zero or more characters.</p> <p>NSAP “123” matches any address that begins with “123”. The NSAP “*” matches any address.</p> <p>NSAP “” (empty string) matches a zero length address.</p> <p>NSAP “*” only matches addresses with non-zero lengths.</p>
<i>rem_nsap</i>	npi_ext_bind_nsap, npi_ext_bind_ascii_nsap, npi_ext2_bind_nsap, npi_ext2_bind_ascii_nsap	<p>Contains an NSAP pattern with the same wildcard characters. Pattern matches against the calling address from the incoming call packet.</p> <p>For <i>rem_nsap</i>, the pattern “*” matches any calling address, including that of zero length.</p> <p>Bind routines that do not have this parameter behave as though it is set to “*”.</p>
<i>lpa</i>	npi_ext_bind_nsap, npi_ext_bind_nsap, npi_ext2_bind_ascii_nsap	<p>The line number of the physical interface over which the incoming call arrived.</p> <p>Non-zero The incoming call must be from the specified physical interface in order to match up with this listen.</p> <p>Zero The interface over which the incoming call arrived is ignored in the matching process.</p> <p>Bind routines that do not have this parameter behave as though it is set to zero.</p>

<p><i>data_val,</i> <i>data_mask</i></p>	<p><code>np_i_ext2_bind_nsap,</code> <code>np_i_ext2_bind_ascending</code></p>	<p>These are arrays of 16 bytes used to match against the user data field of the incoming call packet. The number of bytes matched is the smaller of 16 or the number of bytes in the user data field of the packet. A user data field of zero length matches any data pattern. Bind routines that do not specify these parameters behave as though both consisted of 16 bytes of zeros.</p> <p>Byte by byte comparison is as follows:</p> $(user_data[i] \& data_mask[i]) = (data_val[i] \& data_mask[i])$
--	--	---

INDEX

Symbols

- * wildcard character 31
- ? wildcard character 31

a

- angle bracket conventions 4
- ASCII log protocol messages 22
- asterisk wildcard character 31

b

- bind
 - acknowledgement 32
 - NPI stream 35
- bind_ascii_nsap parameter 31
- bind_flags parameter 35
- BIND_flags to supply with N_BIND_REQ 31
- bind_nsap parameter 32
- boldface conventions 4
- buf parameter 36

c

- c pointer 34
- call, obtaining charging X.25 information 65
- cautions, purpose of 4
- cc command, UNIX 18
- charging information, obtain 65
- class A facility 58
- class C facilities 58
- clone open facility 74
- cnt parameter 36
- conind_nr parameter 31, 32

- connect
 - listen for incoming 70
 - request contains X.25 facilities 54
 - request sent 93
 - request, extended 95
 - response sent 34
 - response sent on NPI stream 51
- connection
 - has been disconnected 88
 - has been established 88
 - indications 22
- connection indications 31
- Constants 19
- conventions
 - names for routines 6
 - notes, cautions and warnings 4
 - text 4

d

- data
 - available indication 73
 - handling fragmented 85
 - message read 90
 - message write 105
 - read 83
 - stream file descriptor 103
- data_lgth parameter 81
- data_ptr parameter 81
- defunct processes 70
- delivery confirmation
 - defined 84
 - npi_rcv() 83
- disconnect
 - return value, npi_rcv() 88
 - stream sequence number

w/np_i_discon_req_seq() 43
 time, log X.25 facilities at 22
 dlpi_close 11
 dlpi_open 11
 DPROTOTYPE 14
 dump memory 76

e

EINTR 102
 enter vs. type 5
 error messages log 22
 expedited data
 handling w/np_i_rcv() 83
 return value for np_i_rcv() 87
 written via np_i_put_exdata_proto() 81

f

fac_cnt parameter 54
 fac_ptr pointer 54
 facil_proc_t 58
 facilities
 X.25 logged or printed 64
 X.25, obtain a copy 65
 facility markers 58
 facI parameter 58
 facp parameter 58
 fast select connect request 36
 fcN parameter 58
 fid parameter 103
 FID was not usable 87
 FILE data structure 69
 file descriptor for NPI data stream 103
 file descriptor, NPI data stream 103
 flags_in parameter 83, 86
 flags_out parameter 83, 86
 flgth parameter 58
 fmt parameter 77
 fork

a child process 70
 behavior of np_i_listen() 56
 options 21
 fork_optns parameter 56
 fragmented NSDUs, handling 85
 fref parameter 58
 func parameter 103
 fval parameter 58

g

Gcom Protocol Appliance 9
 Gcom Remote API
 architecture 9
 client server model 10
 running the RAPI server 10
 Gcom_npilstn 14
 getmsg system call issued 62
 global variables 24
 Guide organization 4

h

hang-up has occurred 87
 header files, linking 18
 highlighted term conventions 4

i

incoming connection indications 19
 indent parameter 76
 info_ptr 66
 information acknowledgement 89
 initialization
 code example 14
 failed 87
 via np_i_init() 68
 via np_i_init_FILE() 69
 italic text conventions 5

l

Library Routine Reference 27

linking the API Library 18

listen

code example 14

for an incoming connection 70

listen_fid parameter, defined 57

log

at disconnect time 22

dump memory in hex format 76

errors, np_i_decode_ctl() 39

file default name 19

file name 68

file specified 69

file write 22

initialization options 22

options 22

printf to 77

verbose mode 14

X.25 facilities sent to 64

log_FILE parameter 69

log_name parameter 68

log_optns parameter 68

Loopback Program 13

m

M_DATA 22

M_DATA messages log 22

M_FLUSH first be sent upstream 51

M_PROTO length 25

marker parameter 58

M-bit indications, forwarding w/np_i_rcv() 83

memory dump 76

message

read 90

read from stream 104

rejected 89

unsupported one was received 89

write 105

n

n parameter 76

N_BIND_ACK 24

N_BIND_REQ

issue using np_i_bind_ascii_nsap() 70

issued 31, 32

passed NPI_N_CONINDS 19

N_CONN_CON 23, 24

N_CONN_IND 23, 24

N_CONN_REQ

issued 35, 36

np_i_conn_con 24

N_DATA, handling w/np_i_rcv() 83

N_DATAACK_REQ sent 38

N_DISCON_IND 23

N_DISCON_REQ issued to NPI 42

N_EXDATA, handling w/np_i_rcv() 83

N_EXDATA_REQ used to write expedited data 81

N_EXT_CONN_REQ sent to NPI 54

N_EXT_CONN_RES returned to NPI 56

N_INFO_REQ, format and send 96

N_ok_ack_t in np_i_ctl_buf 51

N_RESET_CON primitive, described 84

N_RESET_IND

primitive, described 84

receipt indicates loss of data 92

N_RESET_REQ

format and send 97

primitive, defined 84

sent 91

N_RESET_RES primitive, described 84

N_X25_Q_BIT

described 87

NIDU (Network Interface Data Unit) 84

non-blocking I/O

if using 86

non-blocking I/O flag

- set, calling `npi_listen()` 73
- notes, purpose of 4
- NPI
 - API Library, how to use 14
 - data stream file descriptor 103
 - parameterize internal behavior 19
 - Provider, pictured 8
 - stream opened 35
- `npi.h` 35
- `npi.log` 14
- `npi_ascii_facil` 30
- `npi_ascii_facil()` 30
- `npi_bind_ack`
 - bind ack copied into 32
 - defined 24
- `npi_bind_ascii_nsap()` 31
 - called from `npi_listen()` 70
- `npi_bind_nsap()` 32
- `npi_close()` 33
- `npi_conn_con` 24
- `npi_conn_con` global array 35
- `NPI_CONN_CON_DATA_SIZE` 23
- `npi_conn_con_data_size` 23, 26
- `NPI_CONN_CON_DATA_SKIP` 23
- `npi_conn_con_data_skip` 23, 26
- `npi_conn_ind` 24
- `NPI_CONN_IND_DATA_SIZE` 23
- `npi_conn_ind_data_size` 23, 25
- `NPI_CONN_IND_DATA_SKIP` 23
- `npi_conn_ind_data_skip` 23, 25
- `npi_conn_res()` 34
- `npi_connect()` 35
- `npi_connect_req()` 36
- `npi_connect_wait()` 37
- `npi_ctl_buf` 25
 - `N-ok_ack_t` 51
 - receives `M_PROTO` message 62
 - receives message 63
- `npi_ctl_buf` size 19
- `NPI_CTL_BUF_SIZE` 19, 25
- `npi_ctl_cnt` 25
- `npi_ctl_cnt` global variable 62
- `npi_data` parameter 31
- `npi_data_buf` 24
 - size of 19
 - write bytes from 79
- `NPI_DATA_BUF_SIZE` 19, 24
- `npi_data_cnt` 25
- `npi_data_req_band` 26
- `npi_dataack_req()` 38
- `npi_dataack_req_band` 26
- `npi_decode_ctl()` 39
- `npi_decode_primitive()` 40
- `npi_decode_reason()` 41
- `NPI_DISC_IND_DATA_SIZE` 23
- `npi_disc_ind_data_size` 23, 25
- `NPI_DISC_IND_DATA_SKIP` 23
- `npi_disc_ind_data_skip` 23, 25
- `npi_discon_req` 42
- `npi_discon_req_band` 26
- `npi_discon_req_seq()` 43
- `npi_drain_req()` 44
- `npi_exdata_req_band` 26
- `npi_ext_bind_nsap()` 45
- `npi_ext_conn_res_lstnr()` 51
- `npi_ext_connect_req()` 54
- `npi_ext_connect_wait()` 55
- `npi_ext_listen()` 56
- `npi_ext_nbio_complete_listen()` 57
 - return value useful for 71
- `npi_ext2_bind_ascii_nsap()` 49
- `npi_ext2_bind_nsap()` 46
- `npi_fac_walk()` 58
- `npi_flags_connect_wait()` 59
- `npi_flags_listen()` 60
- `npi_flow_req()` 61
- `npi_flow_req_band` 26
- `npi_get_a_msg()` 62
- `npi_get_a_proto()` 63
- `npi_get_and_log_facils()` 22, 64

- npi_get_facils() 65
 - returns X.25 facilities 58
- npi_get_stream_info() 66
- npi_info_req() 67
- npi_init() 14, 22, 68
- npi_init_FILE() 69
- npi_listen forking behavior 56
- npi_listen() 14, 21, 70
- npi_listen() forking behavior 56
- NPI_LISTEN_FORK 21
- NPI_LISTEN_NO_FORK 21
- NPI_LOG_CONINDS 22
- NPI_LOG_DEFAULT 22
- NPI_LOG_ERRORS 22
- NPI_LOG_FACILS 22
- NPI_LOG_FILE 22
- NPI_LOG_NAME 19
- NPI_LOG_OPTIONS 22
- NPI_LOG_RX_DATA 22
- NPI_LOG_RX_PROTOS 22
- NPI_LOG_SIGNALS 22
- NPI_LOG_STDERR 22
- NPI_LOG_TX_DATA 22
- NPI_LOG_TX_PROTOS 22
- npi_max_sdu() 72
- NPI_N_CONINDS 19, 21
- NPI_NBIO
 - defined 21
 - npi_listen() return value if set 71
- npi_nbio_complete_listen() 73
 - return value useful for 71
- npi_open 33, 66
- npi_open() 74
- npi_other_req_band 26
- npi_perror() 75
- npi_print_msg() 76
- npi_print_stream_info 78
- npi_printf 78
- npi_printf() 16, 77
- npi_put_data_buf() 79
- npi_put_data_proto() 80
- npi_put_exdata_proto() 81
- npi_put_proto() 82
- npi_rcv() 83
- npi_read_data() 16, 22, 90
- npi_reset_req_band 26
- npi_reset_res() 91, 92
- npi_send_connect_req() 93
- npi_send_ext_conn_res() 94
- npi_send_ext_connect_req() 95
- npi_send_info_req() 96
- npi_send_reset_req() 97
- npi_send_reset_res() 98
- npi_set_log_size() 99
- npi_set_marks() 100
- npi_set_pid() 101
- npi_set_signal_handling() 102
- npi_sig_func_t 20
- npi_sig_func_t typedef 102
- npi_want_a_proto() 104
- npi_write_data() 16, 22, 105
- npi_x25_clear_cause() 106
- npi_x25_diagnostic() 107
- npi_x25_registration_cause() 108
- npi_x25_reset_cause() 109
- npi_x25_restart_cause 110
- npiapi.a
 - defined 9
- npiapi.h
 - contains defines and prototypes 18
 - defined 9
 - include code example 14
- NPIAPI_BIND_ACK return value, described 89
- NPIAPI_CONNECT_COMPLETE return value, described 88
- NPIAPI_CONNECT_IND return value, described 88
- NPIAPI_DATA_ACK return value, described 88
- NPIAPI_DISC_IND return value, described 88
- NPIAPI_EAGAIN return value, described 87
- NPIAPI_ERROR_ACK return value, described 89

NPIAPI_EXPEDITED_DATA return value, described 87

NPIAPI_FRAGMENT
described 87
if set 85

NPIAPI_GETMSG_ERROR return value, described 87

NPIAPI_INFO_ACK return value, described 89

NPIAPI_MORE_DATA
described 86
purpose 85

NPIAPI_NO_NOTHING return value, described 87

NPIAPI_NORMAL_DATA return value, described 87

NPIAPI_NOT_INIT return value, described 87

NPIAPI_OTHER return value, described 89

NPIAPI_PARAM_ERROR return value, described 87

NPIAPI_RC_FLAG
described 86
how to respond if set 86
in handling fragmented data 85

NPIAPI_RESET_COMPLETE return value, described 88

NPIAPI_RESET_COMPLETE, value returned 83

NPIAPI_RESET_INDICATION
flags_in value 83
returned to caller 86

NPIAPI_RESET_INDICATION return value, described 88

NPIAPI_USER_DATA_ACK
if set 84

NPIAPI_USER_DATA_ACK, described 86

NPIAPI_USER_RESET_RES
described 86
flags_in value 83
if set 86

npi-stream_info_t 78

NSAP
address to be bound 31
address to be bound to the stream 32

nsap parameter 56

nsap_lgth parameter 32

NSDU
defined 83
handling fragmented 85

p

p parameter 39, 76

peer address to be connected to 36

peer_sap parameter 36, 54

perror command, UNIX 75

poll()
call response indicating data available 57
npi_nbio_complete_listen() called in responded to 73

PRIM_type field 25

primitive_mask parameter 103

print
code example 16
facilities, npi_get_and_log_facils() 64
indentation 76
to a log file, npi_printf 77
to log file, npi_perror() 75
X.25 facilities to log, NPI_LOG_FACILS 22

printf routine, UNIX 77

process
defunct 70
forking a 70
ID, part of npi_printf() output 77

proto_type parameter 104

protocol
messages, log 22
read messages 83

PROTOTYPE 9

put_npi_proto() 111

q

question mark wildcard character 31

r

RAPI Library 11

read

- a message into `npi_ctl_buf` 63
- data and process protocol messages 83
- data message 90
- data message, code example 16
- M_PROTO message into `npi_ctl_buf` 62
- protocol message 104

reason parameter 42

receipt confirmation

- bit set 38
- service, defined 84

Remote API 9

`remote_sap` parameter 35

requirements, knowledge 3

reset

- indication, handling w/`npi_rcv()` 83
- response sent 92
- response, format and send 98
- responses, returning w/`npi_rcv()` 83
- sequence has completed 88
- service
 - defined 83
 - resynchronizing properties of 84
- restrictions to `npi_set_signal_handling()` 103
- resynchronizing properties of the reset services 84
- return values for `npi_rcv()` 87
- routine naming conventions 6

s

screen display 5

`seq` parameter 43

`sig_num` parameter 103

SIGIO signal supported 103

signal

- generation request 102
- handling for log 22

handling prototype 20

UNIX 102

`signal()` routine used to become signal handler 102

SIGPOLL signal supported 103

`sigset()` routine used to become signal handler 102

`stderr` write 22

stream file descriptor 103

t

terminology conventions 4

text conventions 4

`tknval` parameter 34

type vs. enter 5

u

UNIX

process status 70

`cc` command 18

error messages, logged 22

International ISO Work Group 84

`perror` command 75

`printf` routine 77

`putmsg` call 79

`read`, `write`, `getmsg`, `putmsg` family of routines 56

signals 102

SIGPOLL or SIGIO 103

v

variables, global 24

verbose logging 14

w

warnings, purpose of 4

wildcard characters 31

`write`

bytes from `npi_data_buf` 79
code example 16
data message 105
expedited data 81
prototype message 82

x

X.25

facilities allowed in connect request 54
facilities obtained by `npi_get_and_log_facils()`
64
facilities, get a copy 65
facilities, `npi_fac_walk()` 58
fast select connect request 36