NPI/SNA

Application Program Interface Guide

Document Version 2.1

February 2002

Supports 3270, 5250, and LU 0 under SCO, QNX, and Linux

Copyright © GCOM, Inc.
All rights reserved.

© 2002 GCOM, Inc. All rights reserved.

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

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

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

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

This manual was written, formatted, indexed, and produced by Senior Technical Writer Scott D. Smith and Technical Writer Geoff Gerrietts using Microsoft Word 5.1 and FrameMaker 4.04 on an Apple Macintosh platform. The source material was gathered by interviewing subject matter specialists Dave Healy, Carleton Herrick and Bob Crownover. Additional art work and binder art was produced by illustrator and publication specialist Charles Lipp using Adobe Illustrator and Adobe PageMaker.

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

FOR FURTHER INFORMATION

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

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

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

Table of Contents

About This Guide 11

Purpose of This Guide 11
Required Reading 11
Related Publications 12
General SNA 12
3270 and 5250 12
Organization of This Guide 13
Conventions Used in This Guide 14
Special Notices 14
Text Conventions 14

Overview 17

Product Description 19
Software Supported 19
Important Files 19

3270 SNA 20

Supported Features 20

3270 BIND Parameters 23
How NPI/SNA Applications Communicate With a Host or Host Program 24
3270 LU-SSCP Sessions 24
3270 LU-LU Sessions 25
3270 Data Transmission Modes 27
RU Chaining 27
NPI/SNA Response Modes 28
3270 Bracket Protocol 28

5250 SNA Supported Features 29

5250 BIND Parameters 32

How NPI/SNA Applications Communicate With a Host or Host Program 33
5250 LU-SSCP Sessions. 33
5250 LU-LU Sessions. 34
5250 Data Transmission Modes 35
RU Chaining 35
NPI/SNA Response Modes 36

NPI/SNA Software Architecture 37

Disadvantages of Blocking I/O. 39
Advantages of Non-Blocking I/O 39
Understanding the Event-Driven Model. 41
Gcom Remote API. 42
Architecture 42
Client Server Model 42
Running the RAPI Server. 43
Using the RAPI Library 43

NPI/SNA 45

Application 45

Requirements of an NPI/SNA Application 47
Header Files for **sim3270.c** and **sim5250.c 47**
Defines and Global Data 49
Opening an NPI/SNA Stream 53
Starting a Read 53
Connecting to a Local LU. 57
Connection Attempts Prior to ACTLU. 59
Connection Auto-Retry. 61
The *initialize()* Procedure 63
sim3270.c and **sim5250.c** *main()* Loop 67
Sending Data to the Host. 71
Begin Bracket (BB), End Bracket (EB), and Change Direction Indicator (CDI) Bits 73
State Transitions for 3270 and 5250 Writes 77
Example: Performing a 3270 Write 79
Example: Performing a 5250 Write 83
User-Supplied Callback Procedures For **sim3270.c** and **sim5250.c 87**
usr_bid() 87
usr_connected() 87
usr_disconnected() 87
usr_host_event() 91

usr_read_complete() . . . 93
usr_write_complete() . . . 97
User-Supplied Callback Procedures in **sim3270.c** Only . . . 99
State Changes Caused by Receipt of a Chain . . . 99
usr_bid() . . . 101
usr_host_event() . . . 105
User-Supplied Callback Procedures in **sim5250.c** Only . . . 107
State Changes Caused by Receipt of a Chain . . . 107
usr_host_event() . . . 107
usr_log_error_complete() . . . 109
usr_request_complete() . . . 109
usr_sslu_read_complete() . . . 111
usr_sslu_write_complete() . . . 113
Non-Callback Procedures In **sim3270.c** and **sim5250.c**. . . . **114**

API Overview . . . 115

NPI Streams Interface . . . 117
NPI/SNA API Routine Entry Points . . . 118
Common API Routines . . . 118
5250-Specific API Routines . . . 118
User-Supplied Callback Procedures . . . 119
Common Callback Routines . . . 119
5250-Specific Callback Routines . . . 119

NPI/SNA API Procedures . . . 120

sna_api_change_userid_passwd() . . . 122
sna_api_set_debug_level() . . . 123
sna_api_verify_userid() . . . 124
sna_close() . . . 125
sna_get_api_state() . . . 126
sna_init_log() . . . 127
sna_log_error() . . . 128
sna_open() . . . 129
sna_poll_retry() . . . 131
sna_send_request() . . . 132
sna_set_buffering_mode() . . . 134
sna_set_read_bfr() . . . 135
sna_sslu_start_write() . . . 136
sna_start_connect_lu() . . . 137
sna_start_write() . . . 139
usr_bid() . . . 144

usr_connected() 145
usr_disconnected() 146
usr_host_event() 147
BIND processing 147
Host Emulation. 148
Cryptography sessions 149
Compressed data sessions 149
usr_log_error_complete(). 151
usr_read_complete(). 152
usr_request_complete() 153
usr_sslu_read_complete(). 154
usr_sslu_write_complete() 156
usr_write_complete() 157

NPI/SNA User-Accessible Defines 159

SNA Disconnect Diagnostic Codes 160
Disconnects Resulting from a Badly Formatted Call Request 160
Calls Rejected Resulting From the State of the PU/LU 160
Disconnects Resulting From Administrative Actions 161
Disconnects Resulting From Host Actions 161
Disconnects Resulting From Data Link Events 161
Disconnects Resulting From the Receipt of Unsupported Rsystem Tokens . . . 161
Disconnects Resulting From State Errors 162
Disconnects Resulting From Client/Application/Stream Errors 162
Disconnect Diagnostics Interpreted from NPI Reason Codes 162
Disconnect Diagnostics Produced by Other Errors 163
Transmission Modes 163
SNA/API State Values 163
Host Events 164
Logging Options. 166
Data Transfer Flags 167

Appendix A 169

Normal Error Codes 169
Convergent Error Returns 172
Stream Problems 173
Shared Library Problems 174
Socket Errors 175
Non-Blocking And Interrupt I/O Errors 175
IPC/Network Software Argument Errors 175
Operational Errors 176
XENIX Error Numbers 177

Appendix B 178

3270 BIND Check Table 178
5250 BIND Check Table 180

Appendix C 183

Data structures for INITSELF requests 185
Format 0 INITSELF: 185
Format 1 INITSELF: 185

Appendix D 186

Appendix E 195

Appendix F. 203

Glossary. 205

Index. 207

PREFACE

About This Guide

Purpose of This Guide

This guide shows programmers how to interface their application to the GCOM NPI/SNA streams driver to access a Systems Network Architecture (SNA) host from a UNIX environment. The SNA Application Program Interface (API) can be used to access SNA 5250 3270, and LU 0 data streams.

Required Reading

To effectively write an NPI/SNA application, you should have experience using the C programming language, the UNIX operating system, and some familiarity with the SNA protocol. You should also be familiar with the material presented in Gcom, Inc.'s *UNIX STREAMS Administrator's Guide.*

GCOM recommends learning about the SNA protocol from the *Systems Network Architecture Concepts and Products* document, published by IBM (GC30-3072).

Related Publications

There are a number of related IBM publications that you may want to reference. To order IBM technical documents for SNA, FAX an inquiry to 1-800-253-3520.

General SNA

IBM's *Systems Network Architecture Technical Overview,* GC30-3073.

IBM's *Systems Network Architecture Format and Protocol Reference Manual: Architectural Logic,* SC30-3112.

IBM's *Systems Network Architecture — Sessions Between Logical Units,* GC20-1868.

3270 and 5250

IBM's *3270 Information Display System Data Stream Programmer's Reference,* GA23-0059-07.

IBM's *5394 Remote Control Unit Functions Reference Release 1 and Release 2,* SC30-3488-03. This book discusses 5250.

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

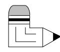
For information about:	*Look at:*
A description of the NPI/SNA product, the software it supports and the important files related to it	Section 1
3270 SNA supported features	Section 2
5250 SNA supported features	Section 3
NPI/SNA software architecture, including: • Blocking and non-blocking I/O • The event-driven model	Section 4
Understanding the **sim3270.c** and **sim5250.c** sample applications so you can write your own application	Section 5
API overview of the NPI/SNA API entry points and user-supplied callback procedures	Section 6
NPI/SNA API procedures that your application must call	Section 7
Callback procedures that your application must supply	Section 8
NPI/SNA user-accessible defines, including the symbolic names and values of various constants used by NPI/SNA procedures and callbacks	Section 9
Error return codes used by the NPI/SNA API routines	Appendix A
The bytes for LU type 2 and the BIND parameters and whether or not they are ignored for 3270, 5250 and LU 0	Appendix B
Source code for the **sim3270.c** sample NPI/SNA application for 3270 environments.	Appendix C
Source code for the **sim5250.c** sample NPI/SNA application for 3270 environments.	Appendix D
SNA-releated acronyms and other terms	Glossary

Conventions Used in This Guide

This section discusses conventions used throughout this guide.

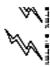
Special Notices

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**.

Italic terms — The following terms appear in *italics*: variables, arguments, parameters, fields, structures, glossary terms, routines, functions, programs, utilities, applications, flags, commands, and scripts. Examples include the *count* variable, *Command Type* field, *rteparam* structure, *Rsystem* defined term, *rsys_read()* routine, *avail* flag, *Add Route* command, and *gcomunld* 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

`This typeface` is used to represent displays that appear on a terminal screen and in-line programming language statements such as `#ifdef`. Commands entered at the prompt use the same typeface only in boldface. For example:

```
C:> cd gcom
% cd gcom
# cd gcom
```

Each of these commands instructs you to enter "cd gcom" at the system prompt and press Return or Enter.

Overview

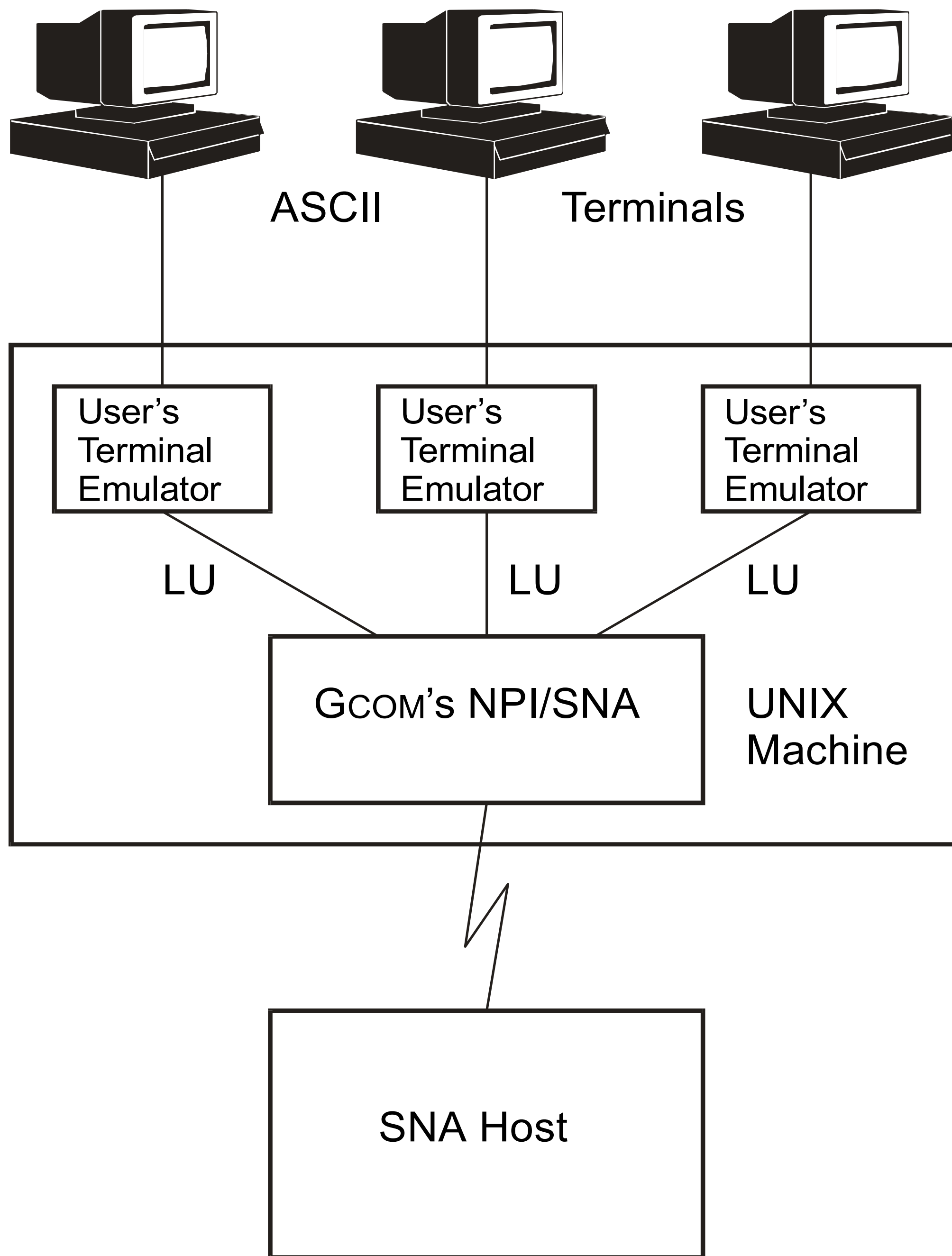


Figure 1 Typical NPI/SNA Application

Product Description

Figure 1 shows how the GCOM NPI/SNA API (Application Program Interface) software package allows programs running in a UNIX environment to communicate with IBM or compatible computers using a subset of the IBM Systems Network Architecture (SNA) communications protocol suite. Although the messages exchanged with the remote host typically conform to the 3270 and 5250 Data Stream protocol, NPI/SNA does not place any restrictions on the content of the messages.

Software Supported

NPI/SNA provides support for SNA 3270 LU-LU sessions using Logical Unit (LU) type 2 and Physical Unit (PU) type 2.

Important Files

The following important files are associated with the SNA API:

npi.h Network Provider Interface (NPI) protocol-related definitions.

npiext.h SNA-specific flag definitions.

sim3270.c A sample 3270 NPI/SNA application to help you write your own application.

sim5250.c A sample 5250 NPI/SNA application to help you write your own application.

snaapi.c Contains the API for SNA routines.

snaapi.h Contains defines that you may need to use, including SNA disconnect diagnostic codes, callback procedure typedefs, and UNIX errno codes returned by the SNA API procedures.

3270 SNA Supported Features

Figure 2 shows how NPI/SNA and the serial hardware implement the lower six layers of the SNA protocol suite, presenting an SNA Function Management Layer interface to your application software.

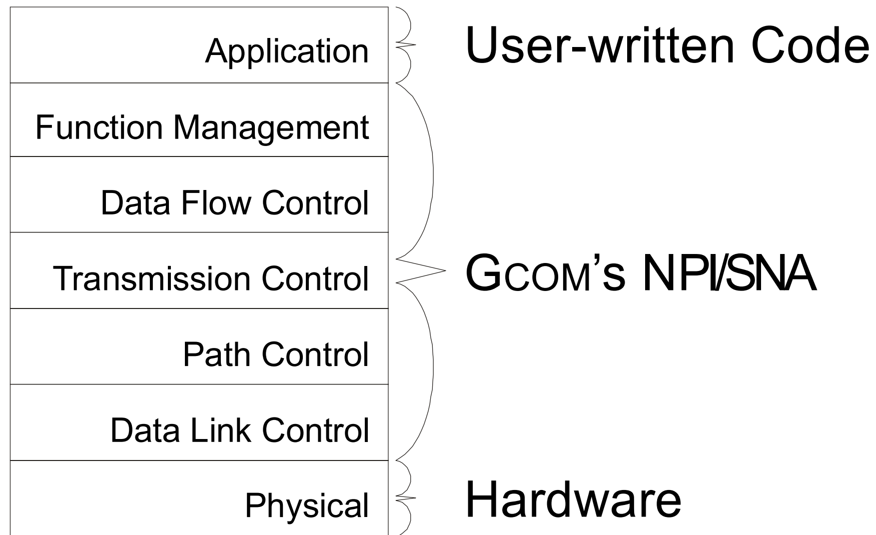


Figure 2 SNA Protocol Stack

3270 NPI/SNA enables your application to:

- connect to a previously activated local Logical Unit (LU)
- disconnect from the LU
- accept or reject a host BID or Begin Bracket (BB) request
- transfer chains over the LU
- access BB, End Bracket (EB) and Change Direction Indicator (CDI) in received chains
- specify BB and CDI in transmitted chains
- access and specify code selection indicator bit
- accept or reject host data
- be informed of processed host commands that affect the exchange of RUs (Request/Response Units)

3270 BIND Parameters

The SNA BIND command carries options governing the exchange of data between the remote SNA host and the local system. Table 2 lists the specific parameters that NPI/SNA enforces for BIND operations.

Table 2 3270 BIND Parameters Enforced

Primary Attributes	*Secondary Attributes*	*Other Attributes*
Must use immediate request mode	Must use immediate request mode	BIND command not negotiable
Cannot use compression	Cannot use compression	Function Management (FM) profile 3
Cannot exchange Function Management (FM) headers with secondary	Cannot exchange Function Management (FM) headers with primary	TS (Transaction Services) profile 3
Responsible for error recovery	Can send multiple-element chains over LU-LU session	Bracketed session is used
Primary to secondary pacing not supported	Secondary to primary pacing not supported	Both code selection indicator settings are accepted
Maximum Request/Response Unit (RU) size sent by primary is checked and can be rejected	Maximum Request/Response Unit (RU) size sent by secondary is accepted	LU-LU session uses half-duplex flip-flop transmission mode
	Wins contention races	
	Cannot send end-bracket	
	First speaker	

Refer to Appendix starting on page 178 for a list of the bytes of the BIND parameter for LU type 2 and information concerning whether or not they are ignored.

How NPI/SNA Applications Communicate With a Host or Host Program

Figure 3 shows how an NPI/SNA application communicates with the host SSCP (System Services Control Point) via an LU-SSCP session, or with a host program over an LU-LU session. These are described in the following subsections.

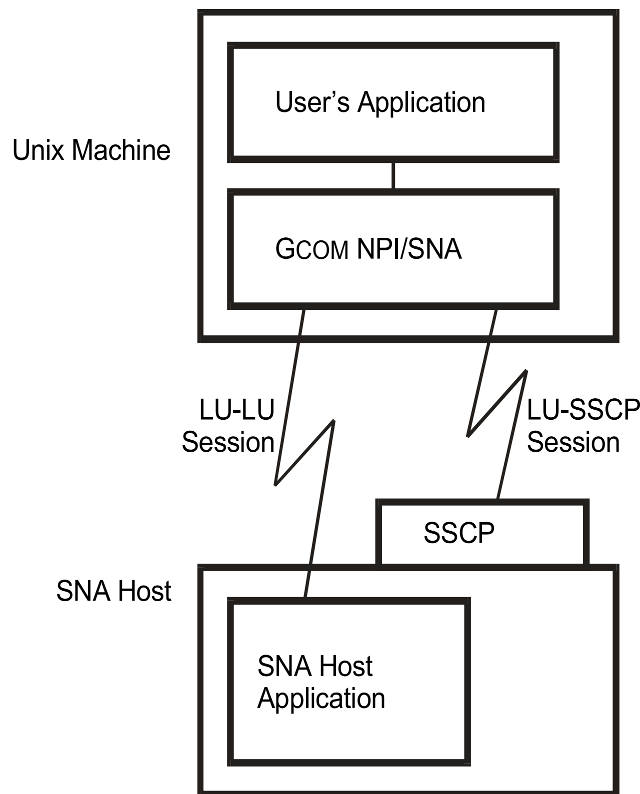


Figure 3 LU-SSCP (System Services Control Point) and LU-LU Sessions

3270 LU-SSCP Sessions

3270 LUs (Logical Units) are activated with the ACTLU command. This creates an LU-SSCP session with TS (Transaction Services) profile set to 1 and Function Management (FM) profile set to zero. As currently implemented, the LU-SSCP session is not accessible by your application while the LU-LU session exists. That is, your application communicates over the LU-SSCP session until a

BIND command is processed. Your application then communicates over the LU-LU session until the LU-LU session is terminated.

When an initial ACTLU command is received, no application-LU connection yet exists. NPI/SNA reports this to the host as a powered off condition.

When an ACTLU is received and there is no connection between the NPI/SNA and your application, an “LU unavailable” response is sent to the host.

When your application connects to the LU, a NOTIFY command indicating “power on” is sent to the host. This indicates to the host that data and commands can be transferred over the LU-SSCP session. Therefore, the host can now use a BIND command to create the LU-LU session. Should your application break its connection to the LU, a NOTIFY command indicating “power off” is sent to the host.

3270 LU-LU Sessions

After the LU has been bound using the BIND command, an LU-LU session exists with Transaction Services (TS) profile and Function Management (FM) profile both set to three. NPI/SNA provides support for SNA 3270 LU-LU sessions using LU type 2 and PU type 2.

FM profile 3 supports the following:

- Use of brackets
- Multiple-element chains
- Code selection indicator
- The following SNA commands: CANCEL, CHASE, CLEAR, BID, LUSTAT, SHUTD, SHUTC, SIGNAL, SDT, and UNBIND.

  Many of these commands modify the rules governing exchange of data (RUs) between the local LU and the remote host. An occurrence of these commands is reported to your application.

If an ACTLU command is received during the time that your application is in an LU-LU session, the NPI/SNA takes one of the following actions:

- If the ACTLU command specifies ERP (Error Recovery Procedure),

NPI/SNA sends a positive response and does not notify your application.

- If the ACTLU command specified COLD, your application is sent a disconnect request and the LU-LU session is terminated. Your application must reconnect to reestablish the LU-LU session.

3270 Data Transmission Modes

One of three modes of data transmission over the LU is always in force. These are:

full-duplex — Data transmitted on the LU-SSCP session is limited to 256 bytes, immediate request mode, OIC (only-in-chain), no brackets, no code selection indicator. Immediate request mode implies that data cannot be transmitted over an LU until any previously started transmissions have completed.

NPI/SNA places no additional constraints on when your application can transmit data. However, the 3270 data stream protocol may require a half-duplex transmission discipline, with the host initiating the first transmission.

data traffic reset — Once a 3270 SNA host BIND or CLEAR command has been processed, no data can be transferred until an SNA host SDT command has been processed.

half-duplex flip-flop — After a 3270 SNA host SDT has been accepted, bracket protocol is used in half-duplex flip-flop mode. If your application and the host simultaneously begin a bracket, your application has the right to proceed, and the host's attempt at starting a bracket fails. Only the host can end a bracket.

RU Chaining

Only complete Request/Response Unit (RU) chains cross the interface between NPI/SNA and your application. When transmitting a chain larger than the maximum transmit RU size, NPI/SNA transmits the chain in multiple chain elements including first-in-chain (FIC), middle-in-chain (MIC), and last-in-chain (LIC). Data received from the host in multiple chain-elements is reassembled and passed to your application during a single read operation as a complete chain.

NPI/SNA Response Modes

The SNA protocol defines two response modes: definite response and exception response. The BIND request specifies the response modes separately for the host and the secondary LU.

When data is sent using definite response mode, the recipient must respond with a positive or negative acknowledgment. When data is sent using exception response mode, a response is generated only when an error is detected.

Unless prohibited by the BIND request, data is sent by NPI/SNA using definite response mode. See “usr_write_complete()” on page 157 for further details. NPI/SNA accepts data sent by the host using either definite or exception response mode.

3270 Bracket Protocol

NPI/SNA enforces the correct use of the BB, EB and CDI indicators, and will fail any application write request that violates the protocol. The state of these indicators is returned to your application along with each complete chain received from the host.

When a host request to begin a bracket is received, your application is given the opportunity to accept or reject it. If your application rejects the host’s bracket request, it is then responsible for beginning the next bracket.

5250 SNA Supported Features

Figure 4 shows how NPI/SNA and the serial hardware implement the lower six layers of the SNA protocol suite, presenting an SNA Function Management Layer interface to your application software.

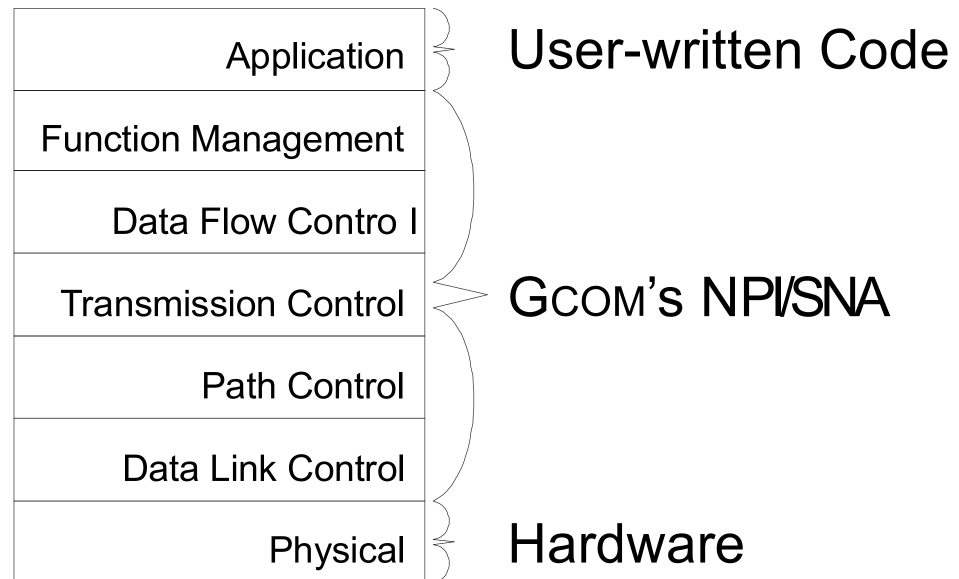


Figure 4 SNA Protocol Stack

NPI/SNA enables your application to:

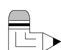
- connect to a previously activated local Logical Unit (LU)
- disconnect from the LU
- accept or reject a host BID or Begin Bracket (BB) request
- transfer chains over the LU
- access BB, End Bracket (EB) and Change Direction Indicator (CDI) in received chains
- specify BB and CDI in transmitted chains
- access and specify code selection indicator bit
- accept or reject host data
- be informed of processed host commands that affect the exchange of RUs (Request/Response Units)

5250 BIND Parameters

The SNA BIND command carries options governing the exchange of data between the remote SNA host and the local system. Table 3 lists the specific parameters that NPI/SNA enforces for BIND operations.

Table 3 5250 BIND Parameters Enforced

Primary Attributes	*Secondary Attributes*	*Other Attributes*
Must use immediate request mode	Must use immediate request mode	BIND command not negotiable
Cannot use compression	Cannot use compression	Function Management (FM) profile 3
Cannot exchange Function Management (FM) headers with secondary	Cannot exchange Function Management (FM) headers with primary	TS (Transaction Services) profile 3
Responsible for error recovery	Can send multiple-element chains over LU-LU session	Both code selection indicator settings are accepted
Primary to secondary pacing not supported	Secondary to primary pacing not supported	LU-LU session uses half-duplex flip-flop transmission mode
Maximum Request/Response Unit (RU) size sent by primary is checked and can be rejected	Maximum Request/Response Unit (RU) size sent by secondary is accepted	
	Wins contention races	
	Cannot send end-bracket	
	First speaker	



Note: *The 5250 SNA protocol does not use bracketed sessions, but the 3270 protocol does.*

Refer to Appendix starting on page 178 for a list of the bytes of the BIND parameter for LU type 2 and information concerning whether or not they are ignored.

How NPI/SNA Applications Communicate With a Host or Host Program

Figure 5 shows how an NPI/SNA application communicates with the host SSCP (System Services Control Point) via an LU-SSCP session, or with a host program over an LU-LU session. These are described in the following subsections.

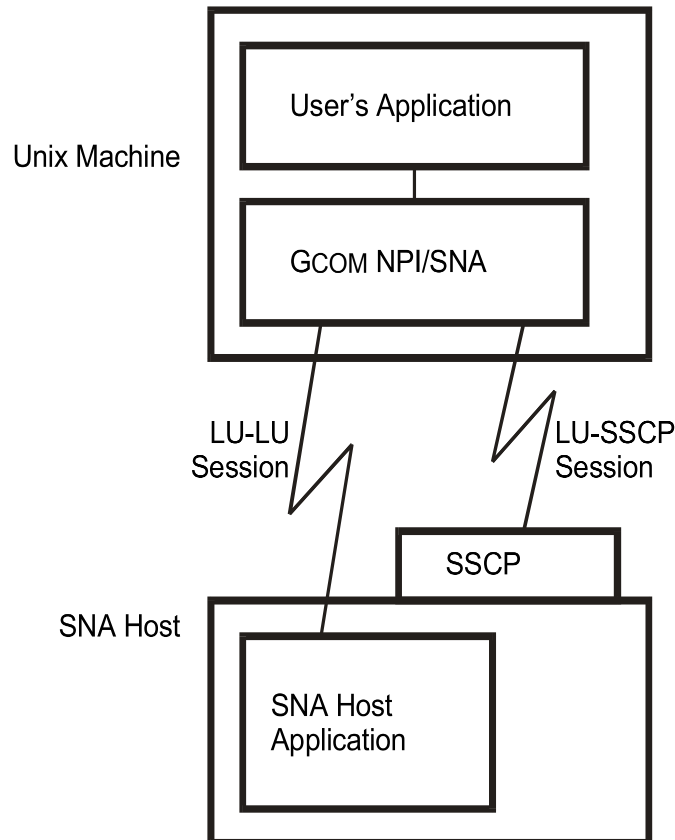


Figure 5 LU-SSCP and LU-LU Sessions

5250 LU-SSCP Sessions

5250 Logical Units (LUs) are activated with the ACTLU command. This creates the LU-SSCP session for a particular LU. When the initial ACTLU command is received, no application-LU connection yet exists. NPI/SNA reports this to the host as LU powered down in the ACTLU response.

When an application program connects to the LU, an

LUSTAT is sent on the SS-LU session with a station available indication. This indicates to the host that data and commands can be transferred over the LU-SSCP session, and that the host can use a BIND command to create the LU-LU session. Data can be sent on the SS-LU sessions after a BIND is sent to the application. Should your application break its connection to the LU, an LUSTAT command indicating “power off” is sent to the host, followed by an RSHUTD request.

5250 LU-LU Sessions

After the LU has been bound via the BIND command, an LU-LU session exists with Transaction Services (TS) profile and Function Management (FM) profile both set to 7. NPI/SNA provides support for SNA 5250 LU-LU sessions using LU type 7 and PU type 1.

FM profile 7 supports the following:

- Multiple-element chains
- Code selection indicator
- These SNA commands: CANCEL, LUSTAT, RSHUTD, SIGNAL and UNBIND.

  Many of these commands modify the rules governing exchange of data (RUs) between the local LU and the remote host. With the exception of RSHUTD, an occurrence of these commands is reported to your application.

If an ACTLU command is received during the time that your application is in an LU-LU session, the NPI/SNA takes one of the following actions:

- If the ACTLU command specifies ERP (Error Recovery Procedure), NPI/SNA sends a positive response and does not notify your application.
- If the ACTLU command specified COLD, your application is sent a disconnect request and the LU-LU session is terminated. Your application must reconnect to reestablish the LU-LU session.

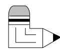
5250 Data Transmission Modes

One of three modes of data transmission over the LU is always in force. These are:

full-duplex
: Data transmitted on the LU-SSCP session is limited to 256 bytes, immediate request mode, OIC (only-in-chain), no brackets, no code selection indicator. Immediate request mode implies that data cannot be transmitted over an LU until any previously started transmissions have completed.

NPI/SNA places no additional constraints on when your application can transmit data. However, the 5250 data stream protocol may require a half-duplex transmission discipline, with the host initiating the first transmission.

half-duplex flip-flop
: After a BIND has been accepted, 5250 type devices are put into the receive state upon receipt of a BIND.



Note: *The* data traffic reset *mode is not supported for the 5250 protocol.*

RU Chaining

Only complete Request/Response Unit (RU) chains cross the interface between NPI/SNA and your application. When transmitting a chain larger than the maximum transmit RU size, NPI/SNA transmits the chain in multiple chain elements including first-in-chain (FIC), middle-in-chain (MIC), and last-in-chain (LIC). Data received from the host in multiple chain-elements is reassembled and passed to your application during a single read operation as a complete chain.

NPI/SNA Response Modes

The SNA protocol defines two response modes: definite response and exception response. The BIND request specifies the response modes separately for the host and the secondary LU.

When data is sent using definite response mode, the recipient must respond with a positive or negative acknowledgment. When data is sent using exception response mode, a response is generated only when an error is detected.

Unless prohibited by the BIND request, data is sent by NPI/SNA using definite response mode. See “usr_write_complete()” on page 157 for details. NPI/SNA accepts data sent by the host using either definite or exception response mode.

NPI/SNA Software Architecture

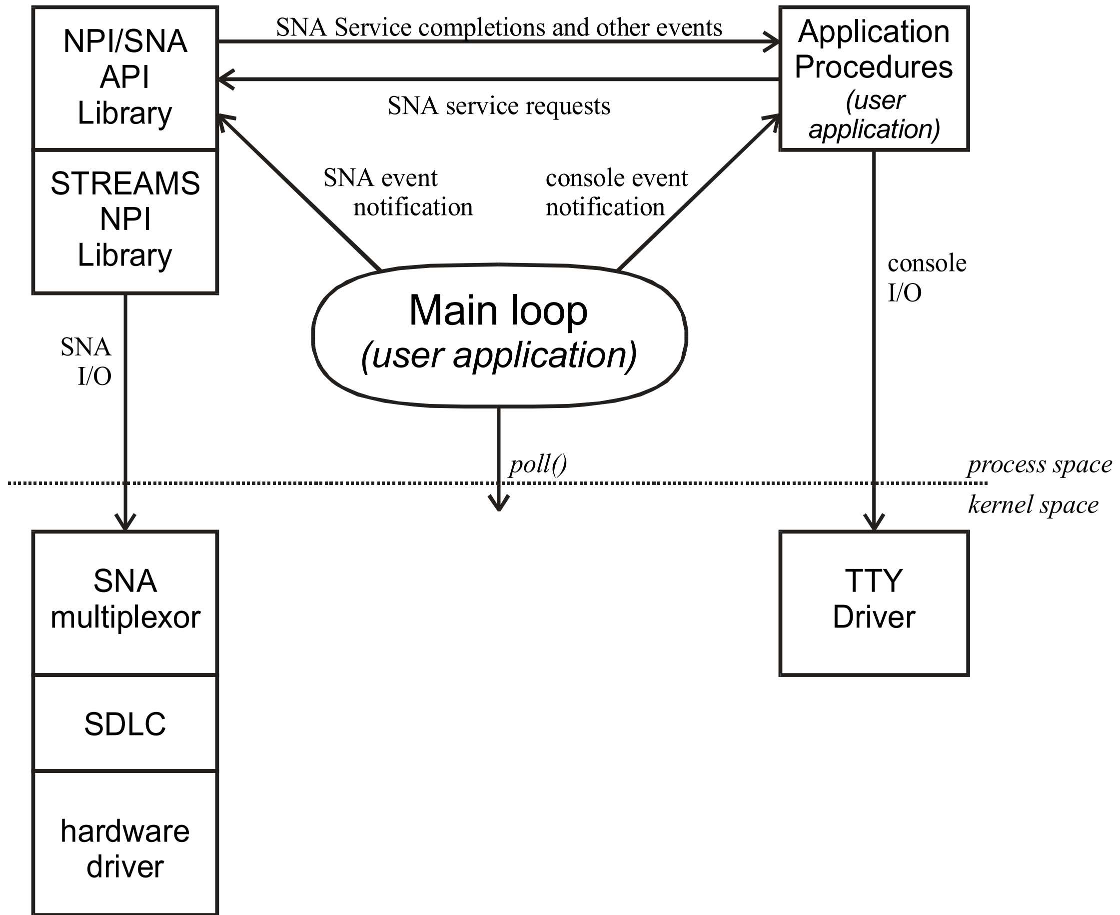


Figure 6 NPI/SNA Application Software

NPI/SNA is comprised of several software modules. The NPI/SNA API library and NPI Streams Interface library are linked with your application code to run as an ordinary UNIX process, while the other components reside in the UNIX kernel as streams drivers.

The NPI/SNA API is designed to support applications that require concurrent non-blocking I/O access to multiple UNIX STREAMS or files, which typically includes the following:

- One or more NPI/SNA streams
- One or more ASCII terminals
- Various other streams and files

Disadvantages of Blocking I/O

Blocking I/O cannot reliably guarantee that data will arrive on one stream or file while another one is blocked (reading). That is, there is no way of knowing which stream or file to read from next. A similar problem occurs when writing large data blocks to a low-speed device or a device that is blocked (typically by pressing Control-S).

Advantages of Non-Blocking I/O

The model shown in Figure 6 breaks the dependency between the two streams, in this case between the TTY and the LU stream. This is done by sending data via the callback functions that you must write as part of your application. The callback functions bypass blocking mode by implementing non-blocking asynchronous notification.

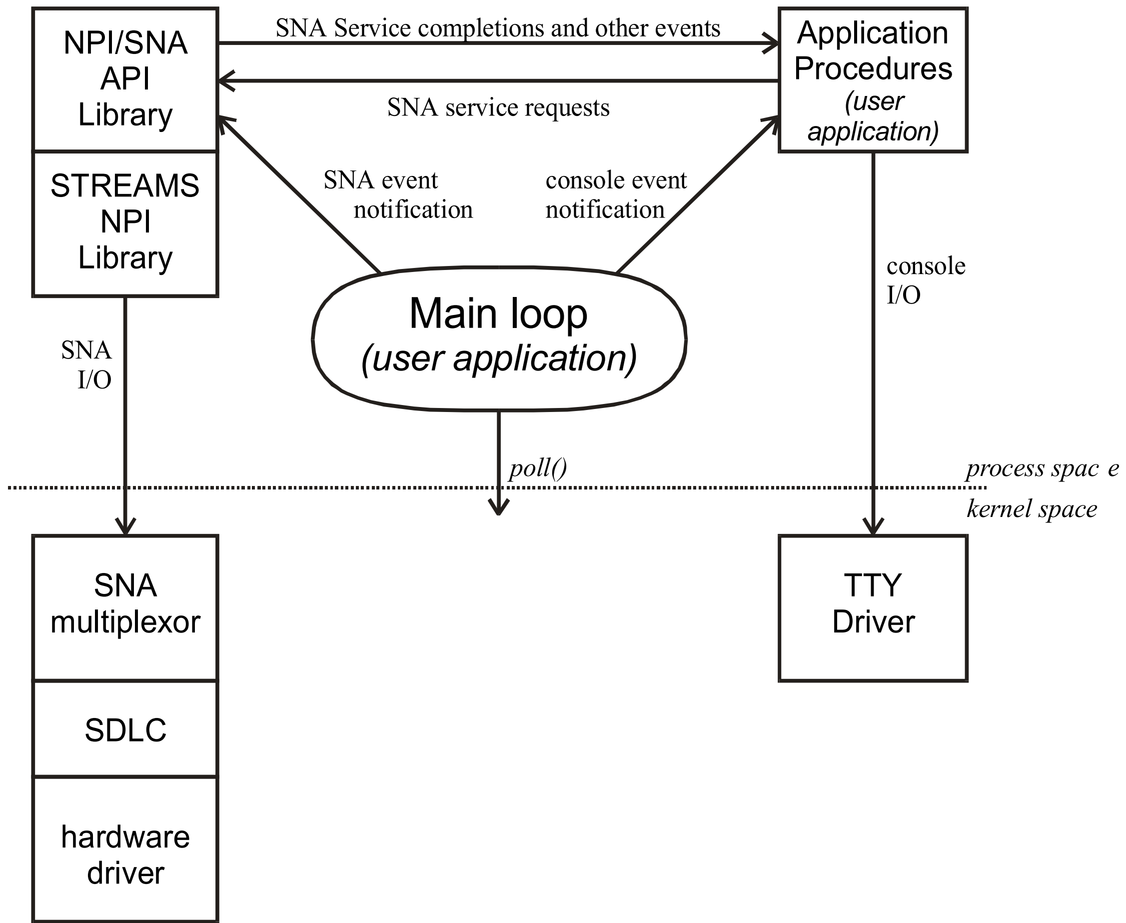


Figure 7 NPI/SNA Application Software *(repeated)*

Understanding the Event-Driven Model

Your NPI/SNA application is built using an event-driven model. Its main processing loop resembles the following pseudo code:

```
REPEAT_FOREVER
{
      WAIT_FOR_EVENT
      PROCESS_EVENT
}
```

Events are processed by calling the API library or one of your own procedures.

WAIT_FOR_EVENT is implemented using the UNIX *poll()* system call. Your main application loop polls the NPI stream and waits for an event to happen, as shown in Figure 7. The events processed are notifications from the system that one or more streams can now be read or written. If an event occurs on an NPI/SNA stream, your application calls the *sna_poll_retry()* routine in the SNA API library to process the event. For an LU 6.2 interface, *lu_62_receive_immediate()* should be called. Otherwise, one of your own user-supplied procedures is called, such as one to handle terminal I/O.

Interactions between application procedures and NPI/SNA consist primarily of service requests from your application and corresponding service completion notifications from NPI/SNA. NPI/SNA also invokes customer-supplied callback procedures when special events occur, such as a connection failure or receipt of a BID command from the host.

One of your procedures is usually called when the request completes.

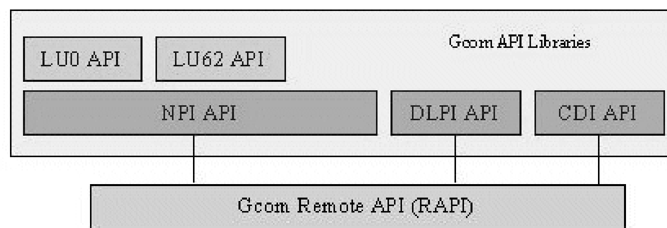
Most service requests made by your application result in one or more message exchanges between the NPI/SNA API library and GCOM SNA server software driver. Since NPI/SNA supports a non-blocking interface, the request is not normally completed by the invoked NPI/SNA API procedure. Instead, a customer supplied callback procedure is invoked by NPI/SNA when the request actually completes.

GCOM Remote API

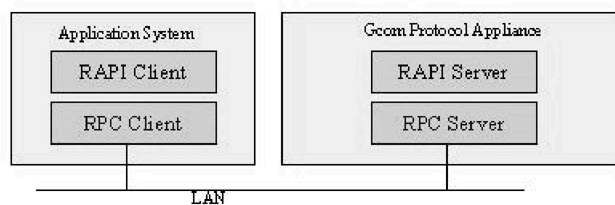
GCOM'S Remote API (RAPI) is a library of functions that allows the standard GCOM APIs to operate on protocol stacks that are configured and running on a remote machine. It is especially useful in situations in which the application code resides on a server system and the protocol processing is performed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

Architecture

The illustration, 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.



Client Server Model

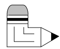


The Remote API is intended for use in a client/server environment. The user's application program, linked with the GCOM RAPI library, runs on the client system. The server system is typically a GCOM Protocol Appliance. It contains the communication hardware, protocol software and the Remote API server.

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.

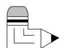


Note: *Additional information on GCOM RAPI arguments, authentication, and other API routines can be found by accessing the GCOM RAPI white paper on the www.gcom.com web site.*

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 DLPI protocol data streams, use the functions:

Open routine: *dlpi_open*
Close routine: *dlpi_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.

Writing a Sample NPI/SNA Application

This section is presented in tutorial style, showing you how to write a small NPI/SNA interactive application called **sim3270.c** for 3270 environment and **sim5250.c** for 5250 environments. Complete source code is listed starting on page 186. These sample applications send a sample message to the SNA host whenever you press a key, and send a dot to the terminal screen each time a chain is received from the host. No attempt is made to interpret the contents of the chain.

Requirements of an NPI/SNA Application

Like all NPI/SNA applications, **sim3270.c** and **sim5250.c** must accomplish the following steps in approximately this sequence:

- Initialize logging mechanism (optional).
- Open an NPI/SNA stream.
- Start a read. (This could be postponed until after the next step.)
- Connect to a local LU.
- Enter the main loop.

Header Files for sim3270.c and sim5250.c

Table 4 lists the header files included in **sim3270.c** and **sim5250.c**.

Table 4 **sim3270.c** and **sim5250.c** Header Files

Header File	*Description*
#include <stropts.h>	Required to use the UNIX *poll()* routine.
#include <sys/poll.h>	Required to use the UNIX *poll()* routine.
#include <gcom/npiapi.h>	Provides access to defines used in specifying logging options.
#include <gcom/npiext.h>	Required by the API procedures and callback procedures.
#include <gcom/snaapi.h>	Required by the API procedures and callback procedures.
#include <stdio.h>	Used for I/O because *printf()* is used.
#include <fcntl.h>	Required because the UNIX *fcntl()* routine is used to set up the terminal for non-blocking I/O.
#include <errno.h>	Contains UNIX error codes used by **sim3270.c** and **sim5250.c**.

Figure 8 **sim3270.c** and **sim5250.c** Global Variables and `#defines`

```
unsigned char     buf[6144] ;                       /* SNA input buffer */
char              tty_input_bfr[120];
int               connect_lu = 2;                   /* local LU */
int               xmt_mode;                         /* transmission mode */
int               if_state;                         /* interface state */
int               write_pending;
int               msgs_to_send;                     /* msgs to send to HOST */
int               sna_connected;                    /* to LU */

int               tty_keyboard = 0;                 /* a file descriptor */
int               tty_screen   = 1;                 /* a file descriptor */
int               sna_fid;                          /* a file descriptor */

struct pollfd     pollfd[3];                        /* used in poll loop */

int               log_options = NPI_LOG_DEFAULT;    /* for sna_init_log () */
char              *log_name    = "snausr.log";      /* for sna_init_log () */

char              dot          = '.';
unsigned char     host_msg[]  =
{
       0x7d,                                        /* 3270 ENTER AID */
       0x40, 0x40,                                  /* display address 0,0 */
       0xd5, 0xd7, 0xc9, 0x61, 0xe2, 0xd5, 0xc1     /* "NPI/SNA" */
};


/*
 * Indexes into the pollfd array.
 */

#define SNA_FD          0
#define TTY_KEYBOARD    1                          /* standard in */
#define TTY_SCREEN      2                          /* standard out */
```

Defines and Global Data

Figure 8 shows the **sim3270.c** and **sim5250.c** global variables and *#defines,* which are explained in Table 5.

Table 5 **sim3270.c** and **sim5250.c** Global Variables and `#defines` *(1 of 2)*

<table>
<tr><th>Variable or #define</th><th colspan="2">Purpose</th></tr>
<tr><td>buf[] buffer</td><td colspan="2">Used to store data received from the SNA host. In your application, be sure that this buffer is large enough to hold the largest chain that the SNA host might send.</td></tr>
<tr><td>tty_input_bfr[]</td><td colspan="2">Used to buffer keystrokes entered by the user.</td></tr>
<tr><td>connect_lu</td><td colspan="2">When invoking the API procedure, this application has the option of requesting connection to a specific LU. For sim3270.c and sim5250.c, connect_lu is used to store a selection of LU 2.</td></tr>
<tr><td rowspan="6">xmit_mode</td><td colspan="2">Used to track the transmission mode. The valid transmission modes are as follows:</td></tr>
<tr><td>Transmission Mode</td><td>Description</td></tr>
<tr><td>XM_SSCP</td><td>SSCP to LU session</td></tr>
<tr><td>XM_DTR</td><td>Data Traffic Reset</td></tr>
<tr><td>XM_FDX</td><td>Full-Duplex</td></tr>
<tr><td>XM_HDX_FF</td><td>Half-Duplex Flip-Flop</td></tr>
<tr><td rowspan="9">if_state</td><td colspan="2">Used to track the SNA interface state. The valid NPI/SNA API states are as follows:</td></tr>
<tr><td>Interface State</td><td>Meaning</td></tr>
<tr><td>SA_CONTENTION</td><td>Contention</td></tr>
<tr><td>SA_BID</td><td>BID</td></tr>
<tr><td>SA_RCV</td><td>Receive</td></tr>
<tr><td>SA_SND</td><td>Send</td></tr>
<tr><td>SA_CHAIN_CONT</td><td>Chain contention state: between chains and bound</td></tr>
<tr><td>SA_FDX</td><td>Full-duplex state: session bound, full-duplex mode</td></tr>
<tr><td colspan="2">Note: The if_state variable has meaning only when the current transmission mode is XM_HDX_FF.</td></tr>
<tr><td>write_pending</td><td colspan="2">Used to avoid allowing the application to write to NPI/SNA before a previous write has completed or failed—this is an unwanted error condition.</td></tr>
<tr><td>msgs_to_send</td><td colspan="2">Provides a trivial queuing mechanism to facilitate the ability to send a sample message to the host each time you enter a keystroke. It is incremented each time a keystroke is detected, and decremented when a message is successfully transmitted.</td></tr>
<tr><td>sna_connected</td><td colspan="2">Used to record if the application is currently connected to a local LU.</td></tr>
</table>

Figure 9 **sim3270.c** and **sim5250.c** Global Variables and `#defines` *(repeated)*

```
unsigned char     buf[6144] ;                        /* SNA input buffer */
char              tty_input_bfr[120];
int               connect_lu = 2;                    /* local LU */
int               xmt_mode;                          /* transmission mode */
int               if_state;                          /* interface state */
int               write_pending;
int               msgs_to_send;                      /* msgs to send to HOST */
int               sna_connected;                     /* to LU */

int               tty_keyboard = 0;                  /* a file descriptor */
int               tty_screen   = 1;                  /* a file descriptor */
int               sna_fid;                           /* a file descriptor */

struct pollfd     pollfd[3];                         /* used in poll loop */

int               log_options = NPI_LOG_DEFAULT;     /* for sna_init_log () */
char              *log_name    = "snausr.log";       /* for sna_init_log () */

char              dot          = '.';
unsigned char     host_msg[]  =
{
       0x7d,                                         /* 3270 ENTER AID */
       0x40, 0x40,                                   /* display address 0,0 */
       0xd5, 0xd7, 0xc9, 0x61, 0xe2, 0xd5, 0xc1      /* "NPI/SNA" */
};


/*
 * Indexes into the pollfd array.
 */

#define SNA_FD          0
#define TTY_KEYBOARD    1                           /* standard in */
#define TTY_SCREEN      2                           /* standard out */
```

Figure 9 shows the **sim3270.c** and **sim5250.c** global variables and *#defines,* which are explained in Table 5.

Table 5 sim3270.c and **sim5250.c** Global Variables and `#defines` *(2 of 2)*

Variable or #define	*Purpose*
tty_keyboard *tty_screen*	These hold the values of the file descriptors used for terminal I/O. They are initialized here to standard in and standard out.
sna_fid	Used to store the file descriptor of the NPI/SNA stream.
pollfd[]	An array of three *pollfd* structures, one each for the keyboard, screen and NPI/SNA streams. Each *pollfd* structure contains the following three members:
	`int fd /*file descriptor*/`
	`short events /*events of interest to caller*/`
	`short revents /*returned events*/`
	pollfd[] is passed to *poll()* in the application main loop, all of the array elements having first been appropriately updated so as to indicate the current events of interest. When *poll()* returns, the *revents* member of each *pollfd[]* array element indicates which (if any) events of interest have occurred on the corresponding stream.
log_options *log_name*	Provides parameters for the NPI/SNA logging mechanism, (See “Logging Options” on page 166 for details.) The application can specify the name of a file into which the log data is written. Here, *log_name* is used to hold the file name, **snausr.log**.
dot	Holds the character that will be written to the terminal screen when a host chain is received.
host_msg[]	Holds the array of characters that will be sent to the host each time the user enters a keystroke. The characters constitute a valid 3270 Data stream: the EBCDIC encoded message “NPI/SNA,” preceded by a three byte header. Byte zero of the header contains the Attention Identifier (AID) code corresponding to the Enter key. Bytes 1 and 2 hold a display buffer address of zero encoded in 3270 12-bit address form.
SNA_FD *TTY_KEYBOARD* *TTY_SCREEN*	Defines used to identify the *pollfd[]* array elements that correspond to the NPI/SNA, keyboard and screen streams, respectively.

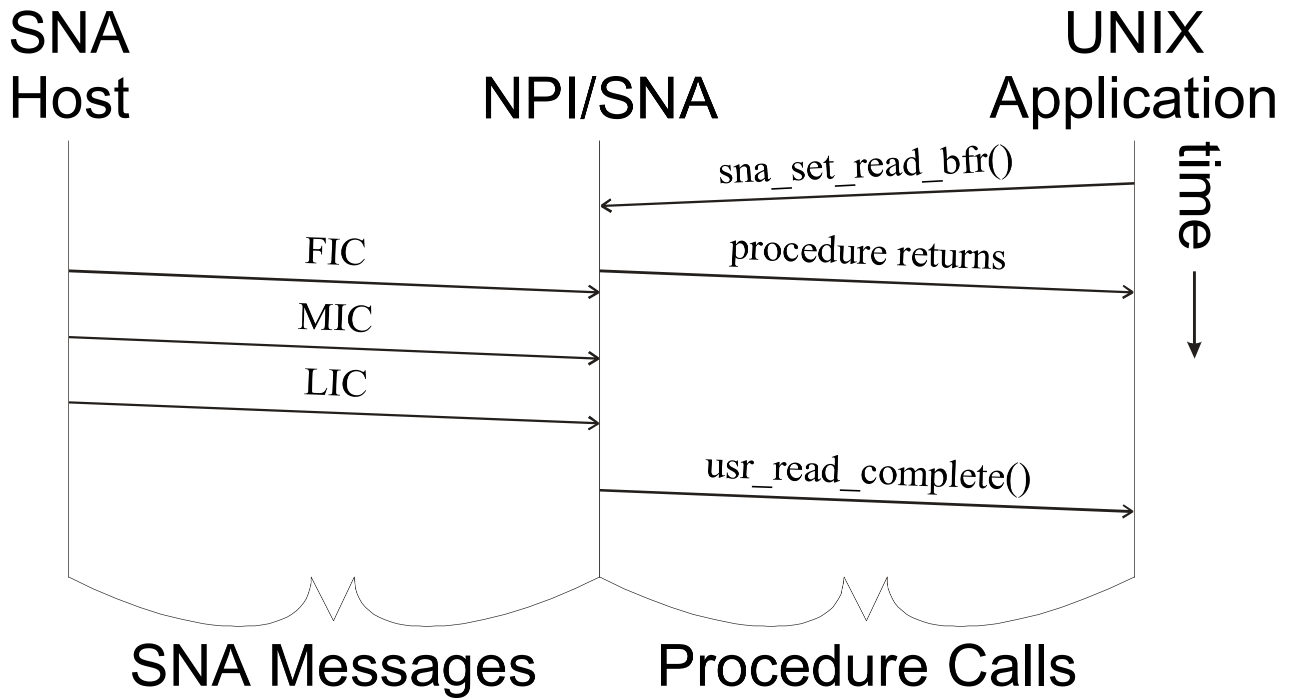


Figure 10 Reading a Host Chain

Opening an NPI/SNA Stream

The API *sna_open()* procedure is used to open an NPI/SNA stream. It returns a file descriptor that is used by the **sim3270.c** and **sim5250.c** application to identify the stream in subsequent interactions with NPI/SNA.

```
int fid ;
    .
    .
fid = sna_open (usr_read_complete,
      usr_write_complete,
      usr_connected, usr_host_event,
      usr_bid, usr_disconnected);
```

The parameters passed to *sna_open()* are pointers to the various user-supplied callback procedures, which are listed in the preceding code.

Starting a Read

Once an NPI/SNA stream has been opened, your application may call *sna_set_buffering_mode()*. The file descriptor returned by *sna_open()* is passed to *sna_set_buffering_mode(),* along with a value indicating how your application will receive the data. Appropriate values for this parameter are defined in snaapi.h, and have the following meanings:

CHAIN_MODE_CHAIN
The API will place the entire chain into an application buffer before calling the application's read complete routine.

CHAIN_MODE_RU
The API will pass through each chain element as it arrives from the host.

CHAIN_MODE_SEG
The API will pass through each segment as it arrives. If the RU is not segmented, the API will call the application's read complete routine for each chain element that is received.

CHAIN_MODE_BUFFERED_RU
The API will buffer the data until th application's buffer is full, then forward it.

If your application doesn't call *sna_set_buffering_mode()*, the API assumes a default value of CHAIN_MODE_CHAIN.

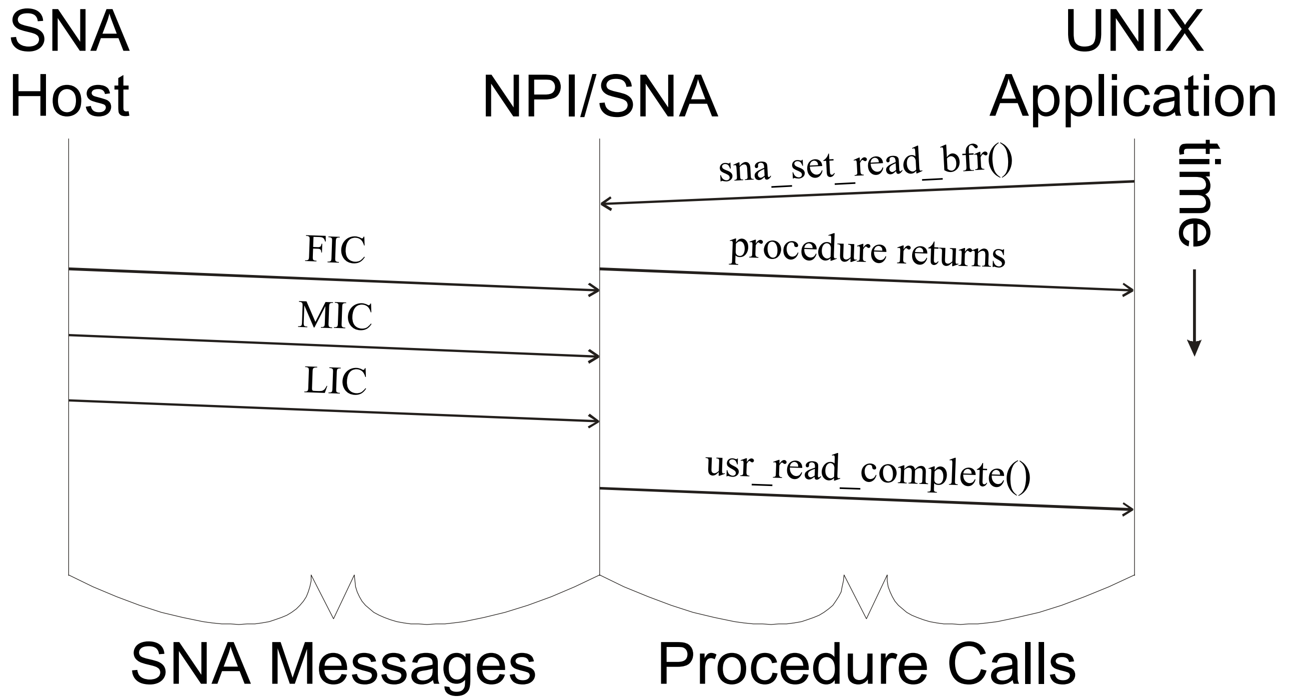


Figure 11 Reading a Host Chain (*repeated*)

Once this routine has either been called or skipped, the application should call *sna_set_read_bfr()*.This routine will take the descriptor returned by *sna_open()*, a pointer to a buffer, and a value indicating the length of the buffer as arguments. The length argument will depend heavily on the option selected with *sna_set_buffering_mode()*: the buffer must always be sufficient to hold the largest example of the element selected. For example, if the buffering method selected is CHAIN_MODE_CHAIN, the buffer must be large enough to hold the largest chain the host can send.

In the example below, the buffer size used is 6144.

```
unsigned charbuf[6144] ;
      •
      •
sna_poll_bits = sna_set_read_bfr (fid, buf, sizeof
                                    (buf)) ;

if (sna_poll_bits < 0)
      /* handle error */
```

sna_set_read_bfr() returns –1 on error. Otherwise it returns bits that are passed to the UNIX *poll()* procedure in your application's main loop to indicate which events are currently of interest on the NPI/SNA stream.

When data is eventually received from the host, NPI/SNA invokes your application's *usr_read_complete()* user-supplied callback procedure, passing along a pointer to *buf*. Figure 10 shows a case of a three element chain arriving from the host. As each element arrives, the contents of its Request/Response Unit (RU) are copied into your application's buffer. When the complete chain has been assembled, NPI/SNA invokes *usr_read_complete()*.

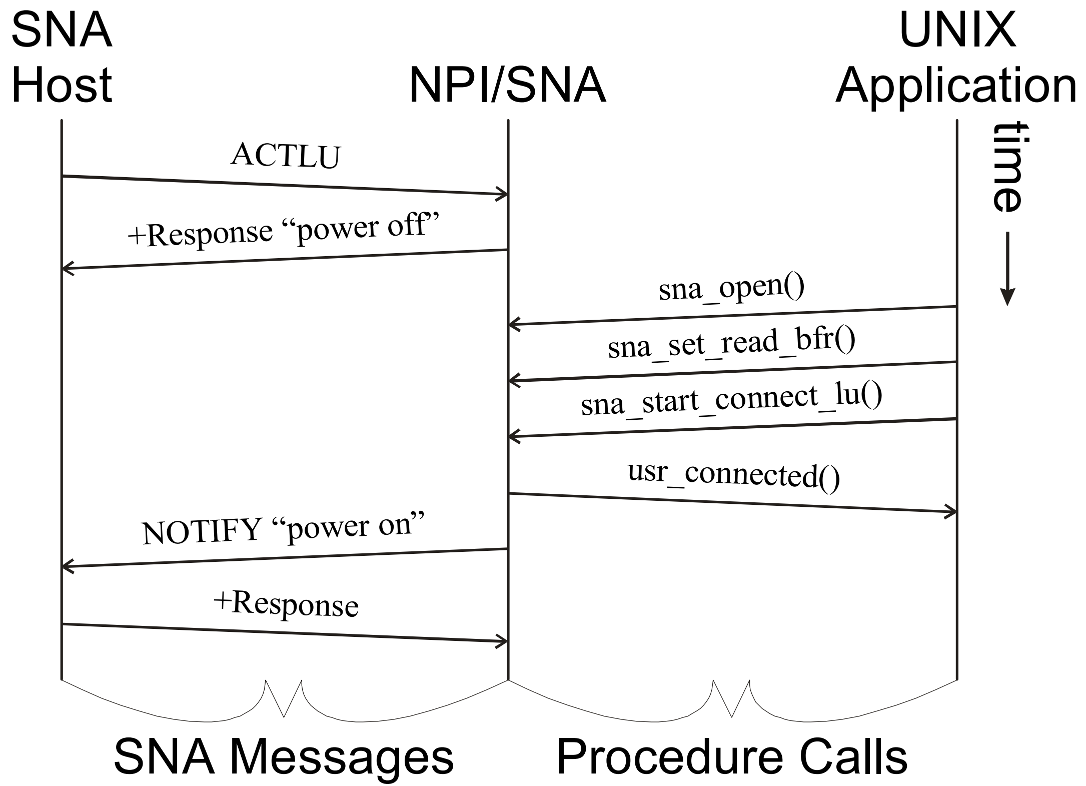


Figure 12 Connecting to an LU

Connecting to a Local LU

Before data can be sent or received over an NPI/SNA stream, a connection must be established to a local LU. Your application initiates the connection by calling *sna_start_connect_lu()*. When a connection is established, NPI/SNA calls your application's callback procedure *usr_connected()*. Figure 12 shows the typical sequence of events.

Before establishing a connection between your application and a local LU, NPI/SNA responds to any ACTLU from the SNA host with a power off indicator. A NOTIFY indicating power on is sent when a connection is successfully completed.

```
if (sna_start_connect_lu (fid, 0, connect_lu, 0,
      10) < 0)
       /* handle error */ ;
```

sna_start_connect_lu() returns –1 on error, and zero otherwise. The parameters passed to *sna_start_connect_lu()* in the above example are as follows:

fid	File descriptor of the NPI/SNA stream.
0	Number of a local SNA PU.
connect_lu	Number of a local SNA LU. If zero, NPI/SNA selects an activated LU that is not currently in use.
0	Maximum number of times NPI/SNA attempts to establish a connection should the first attempt fail. The value zero used here causes NPI/SNA to retry an indefinite number of times.
10	Number of seconds NPI/SNA delays between connection retries. (If zero, NPI/SNA does not retry a failed connection attempt, regardless of the value of the previous parameter.)

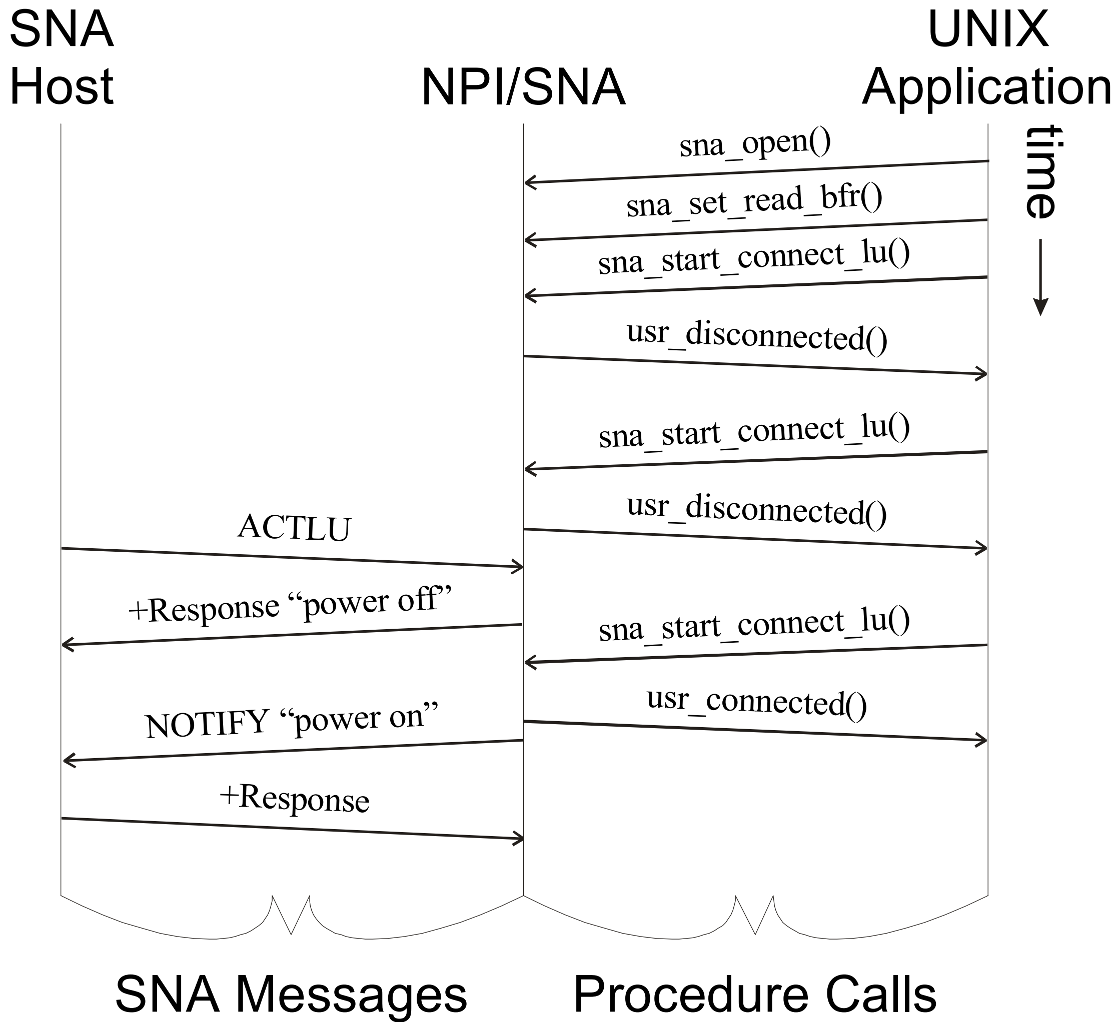


Figure 13 Connection Attempts Before Host ACTLU

Connection Attempts Prior to ACTLU

Figure 13 shows a scenario where your application tries to establish an LU connection before receiving an ACTLU command from the host. This attempt always fails, provided that no retry was specified in the *sna_start_connect_lu()* call. NPI/SNA notifies your application of the failure by invoking the *usr_disconnected()* user-supplied callback procedure.

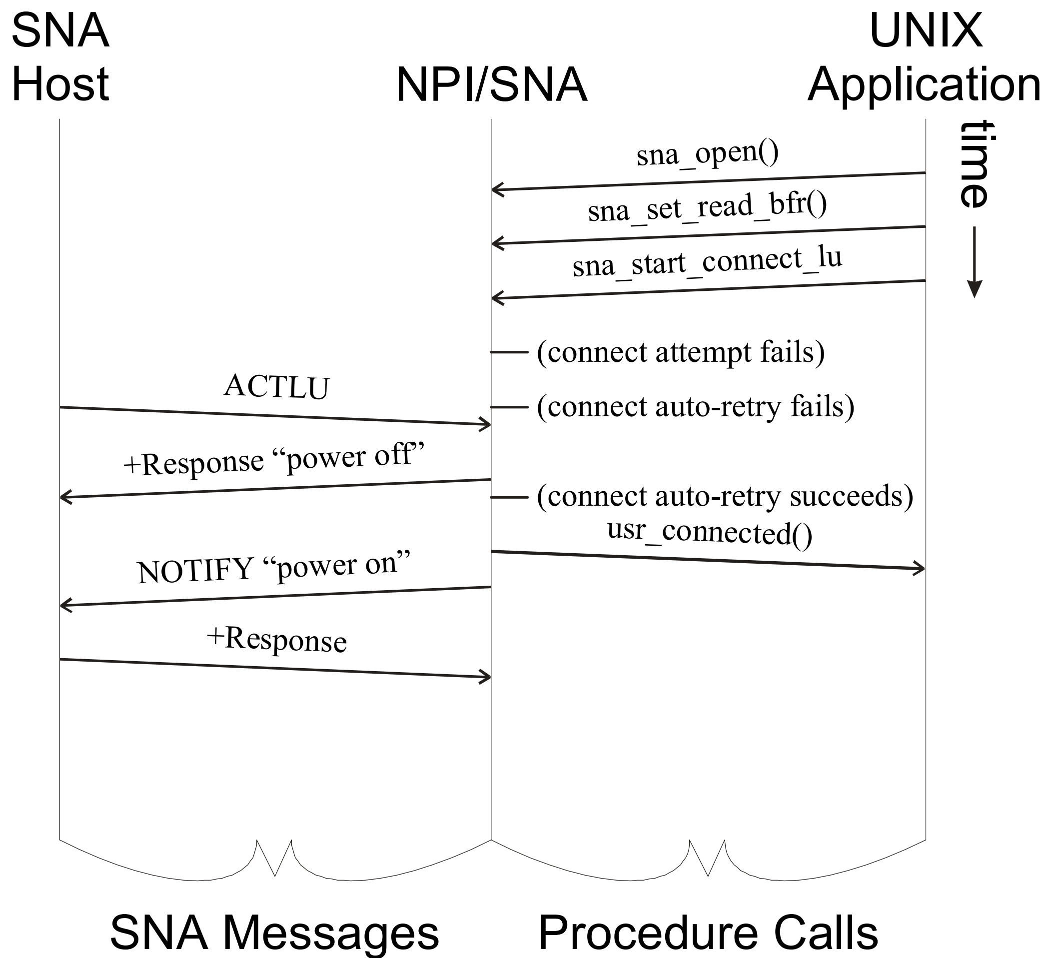


Figure 14 Connection Auto-Retry

Connection Auto-Retry

In the sequence of events shown in Figure 14, *sna_start_connect_lu()* is called with auto-retry specified before receiving an ACTLU from the host. Refer to “sna_start_connect_lu()” on page 137 further details.

Figure 15 *initialize()* Procedure

```
int                                                    /* file descriptor */
initialize ()
{
    int   sna_fid;
    int    result;

    if (sna_init_log (log_options, log_name) < 0)
          perror ("main - sna_init_log");

    printf ("snausr: starting test\n");

    sna_fid = sna_open (usr_read_complete,      /* open sna data stream */
                        usr_write_complete,
                        usr_connected,
                        usr_host_event,
                        usr_bid,
                        usr_disconnected);

    if (sna_fid < 0)                            /* open failed */
    {
          perror ("sna_open");
          return (-1);
     }

    result = sna_set_read_bfr (sna_fid, buf, sizeof (buf));

    if (result < 0)
    {
          perror ("connecting - set read bfr");
          sna_close (sna_fid);
          return (-1);
    }

    pollfd[SNA_FD].events = result;           /* from sna_set_read_bfr() */

    result = sna_start_connect_lu (sna_fid, 0, connect_lu, 0, 10);

    if (result < 0)
    {
          perror ("sna_start_connect_lu");
          sna_close (sna_fid);
          return (-1);
    }

    pollfd[SNA_FD].events = result;           /* from sna_start_connect_lu()*/
```

The *initialize()* Procedure

In **sim3270.c** and **sim5250.c**, the *initialize()* procedure accomplishes the following tasks:

- Initializes the logging mechanism
- Opens an NPI/SNA stream
- Starts the initial read
- Initiates a connection to a local LU

The *initialize()* procedure returns the file descriptor of the NPI/SNA stream, or –1 if an error occurs.

As shown in Figure 15, *initialize()* begins by calling the API procedure *sna_init_log()* to set up the logging mechanism, passing in the *log_options* and *log_name* globals. A startup message is then sent to the terminal screen.

Next, *sna_open()* is called, and the file descriptor stored in the local variable *sna_fid*.

If there was no error, *sna_set_read_bfr()* and *sna_start_connect_lu()* are called to start a read and open a connection to an LU. If these procedures encounter no problems, their return values, indicating the current events on the NPI/SNA stream, are assigned to pollfd[SNA_FD].events. This is done with *sna_set_read_bfr()* for illustration purposes only, since the value stored at that point is overwritten after a successful invocation of *sna_start_connect_lu()*.

Figure 16 *initialize()* Procedure (continued)

```
int                                                    /* file descriptor */
initialize ()
{
        •
        •
        •

    if (fcntl (tty_keyboard, F_SETFL, O_NDELAY) < 0)
    {
        sna_close (sna_fid);
        return (-1);
    }

    if (fcntl (tty_screen, F_SETFL, O_NDELAY) < 0)
    {
        sna_close (sna_fid);
        return (-1);
    }

    /*
     * Initialize SNA poll descriptor entry.
     * pollfd[SNA_FD].events is set above.
     */
    pollfd[SNA_FD].fd      = sna_fid;
    pollfd[SNA_FD].revents = 0;

    /*
     * Initialize TTY poll descriptor entries.
     */
    pollfd[TTY_KEYBOARD].fd      = tty_keyboard;   /* standard in */
    pollfd[TTY_KEYBOARD].revents = 0;
    pollfd[TTY_KEYBOARD].events  = POLLIN;

    pollfd[TTY_SCREEN].fd        = tty_screen;     /* standard out */
    pollfd[TTY_SCREEN].revents   = 0;
    pollfd[TTY_SCREEN].events    = 0;

    return (sna_fid);
}
```

Figure 16 shows the remainder of the *initialize()* procedure. The terminal keyboard and screen are set up for non-blocking I/O via the UNIX system call *fcntl()*, and then the *pollfd[]* array is initialized. For the keyboard, POLLIN (input) is specified as the only event of interest. For the screen, zero is specified, indicating no events, since there is nothing to display at this point, so it is irrelevant whether the screen can be written or not.

Figure 17 sim3270.c and **sim5250.c** *main()* Loop

```
main ()
{
    int events;

    if ((sna_fid = initialize ()) < 0)               /* initialize */
        exit (1);                                    /* failure */
    for (;;)
    {
        if (poll (&pollfd[0], 3, INFTIM) < 0)        /* wait sna/tty events */
        {
            perror ("snausr - poll");
            return;
        }

        /*
         * The called keyboard/screen event handling
         * procedures update the poll list as needed.
         */

        if (pollfd[TTY_KEYBOARD].revents)           /* keyboard events */
            keyboard_events (tty_keyboard);         /* read event */

        if (pollfd[TTY_SCREEN].revents)             /* screen events */
            write_dots (0);                         /* retry write dots */

        if (pollfd[SNA_FD].revents)                 /* process SNA events */
        {
            events = sna_poll_retry (sna_fid, pollfd[SNA_FD].revents);

            if (events < 0)                         /* sna fid not usable */
            {                                       /* corrective action? */
                pollfd[SNA_FD].events = 0;
                break;                              /* {1} */
            }

            pollfd[SNA_FD].events = events;         /* for next poll */
        }
    }
}
```

sim3270.c and sim5250.c *main()* Loop

In **sim3270.c** and **sim5250.c**, *main()* accomplishes the following tasks:

- Calls *initialize()* to handle the preliminaries
- Stores the returned NPI/SNA stream file descriptor in *sna_fid*
- Enters the main loop unless the return value from *initialize()* indicates an error

1. At the top of the loop, *poll()* is called with these three parameters: the array of *pollfd* structures, the number of elements in the array, and a time-out value. INFTIM specifies an infinite time-out, indicating that *poll()* does not return until one of the events specified in *pollfd[]* occurs. When *poll()* returns, the *revents* member of each *pollfd[]* array element indicates if any events of interest have occurred on the corresponding stream.
2. If a *revent* member is non-zero, a procedure is called to handle the event(s). In the **sim3270.c** and **sim5250.c** main loop in Figure 17, *keyboard_events()* and *write_dots()* handle events on the keyboard and screen streams, respectively.
3. *pollfd[TTY_KEYBOARD].events* and *pollfd[TTY_SCREEN].events* are updated when appropriate from within the event handlers. As a result of responding to a keyboard or screen event, application code can invoke API procedures (such as *sna_start_write()*), which can alter the set of events of interest on the NPI/SNA stream. When this occurs, *pollfd[SNA_FD].events* is updated from the value returned by the API procedure.
4. When *keyboard_events()* is called, it is passed the file descriptor of the keyboard stream. The parameter passed to *write_dots()* indicates how many additional dots are to be written to the screen, beyond those already queued. In the main loop, *write_dots()* is called merely to allow already queued dots to be written the screen, so the *pollfd[SNA_FD]* parameter is set to zero.

Figure 18 sim3270.c and **sim5250.c** *main()* Loop *(repeated)*

```
main ()
{
    int events;

    if ((sna_fid = initialize ()) < 0)               /* initialize */
        exit (1);                                    /* failure */
    for (;;)
    {
        if (poll (&pollfd[0], 3, INFTIM) < 0)        /* wait sna/tty events */
        {
            perror ("snausr - poll");
            return;
        }

        /*
         * The called keyboard/screen event handling
         * procedures update the poll list as needed.
         */

        if (pollfd[TTY_KEYBOARD].revents)           /* keyboard events */
            keyboard_events (tty_keyboard);         /* read event */

        if (pollfd[TTY_SCREEN].revents)             /* screen events */
            write_dots (0);                         /* retry write dots */

        if (pollfd[SNA_FD].revents)                 /* process SNA events */
        {
            events = sna_poll_retry (sna_fid, pollfd[SNA_FD].revents);

            if (events < 0)                         /* sna fid not usable */
            {                                       /* corrective action? */
                pollfd[SNA_FD].events = 0;
                break;                              /* {1} */
            }

            pollfd[SNA_FD].events = events;         /* for next poll */
        }
    }
}
```

5. The *sna_poll_retry()* API procedure is called to handle events on the NPI/SNA stream. *pollfd[SNA_FD].revents* is passed to *sna_poll_retry()* because the actions to be taken depend upon the specific event(s) that occurred. If *sna_poll_retry()* fails, the main loop is exited and the program terminates. Otherwise, the return value is copied to *pollfd[SNA_FD].events* and control returns to the top of the loop.

At the next invocation of *poll()*, the value passed in *pollfd[SNA_FD].events* will be as shown in the following pseudo code:

IF *sna_poll_retry()* was called in this iteration of the loop,
{
 pollfd[SNA_FD].events holds the value returned.
}

ELSE IF Application code called API procedures that altered the events of interest on the NPI/SNA stream

{
 pollfd[SNA_FD].events holds the value returned by the most recently called such API procedure.
}

ELSE *pollfd[SNA_FD].events* is unchanged since the previous *poll()* call.

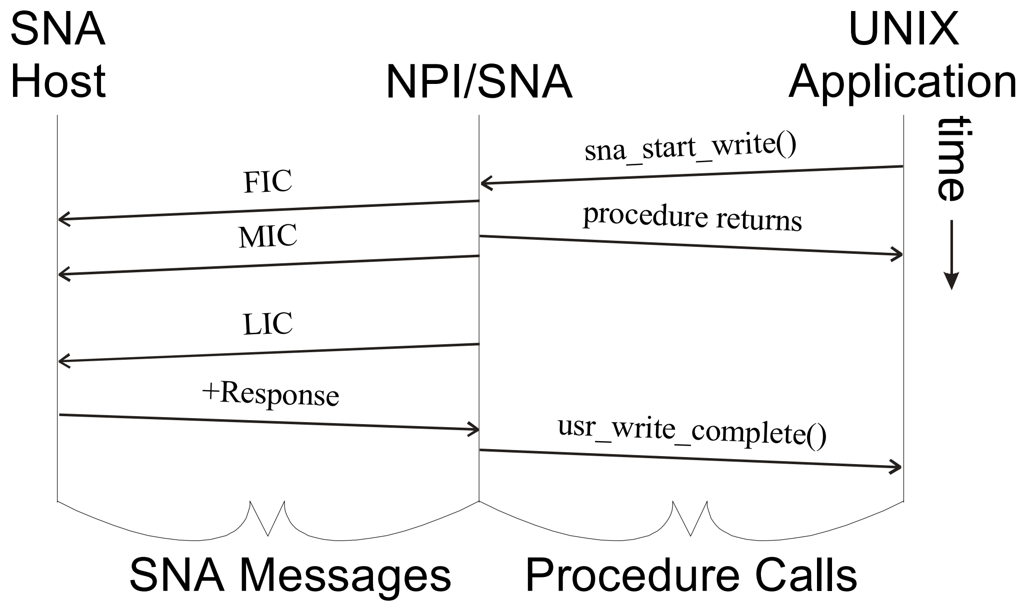


Figure 19 Transmitting a Chain

Sending Data to the Host

To send data to the SNA host, your application calls the *sna_start_write()* API procedure, passing four parameters:

- The file descriptor of the NPI/SNA stream
- A pointer to a buffer holding the data
- A count of the bytes to send
- A "bit-box" containing flags specifying the desired values of the BB, EB, and CDI indicators. (Code Selection Indicator is also selected via the *flags* parameter.) The correct handling of these indicators is discussed in section 5.9.1.

```
int bits ;
•
•
•
bits = sna_start_write (fid,    /* of NPI/SNA Stream */
                        bfr,    /* data for host     */
                        count,  /* bytes to send     */
                        flags); /* BB, EB, CDI       */

if (bits < 0)
                                /* handle error */ ;
```

sna_start_write() returns -1 on error. Otherwise it returns bits to be passed to the UNIX *poll()* procedure in the next iteration of your application's main loop. If the write buffer contains more data than can be sent in a single frame, NPI/SNA transmits it in multiple chain elements, such as the chain shown in Figure 19. Furthermore, the NPI/SNA may segment each chain element, depending on the frame size selected in the SDLC configuration parameters.

If a negative response is received from the host, NPI/SNA discards the chain automatically. When a positive response is finally received, NPI/SNA notifies your application by invoking the *usr_write_complete()* user-supplied callback procedure.

Table 6 Flag Bit Settings

Mode	*State*	*Flag Bits*
XM_HDX_FF	SA_BID	Writes not allowed
	SA_RCV	Writes not allowed
	SA_SND	BB=0; EB=0; CDI=1 or 0; MD=1 or 0
	SA_CONTENTION	BB=1; EB=0; CDI=1 or 0; MD=1 or 0
XM_FDX	-----	BB=0; EB=0; CDI=0; MD=0
XM_DTR	-----	Writes not allowed

BB = Begin Bracket; EB = End Bracket; CDI = Change Direction Indicator;
MD = more data indicator (N_SNA_More_Chain)

Begin Bracket (BB), End Bracket (EB), and Change Direction Indicator (CDI) Bits

The value of the BB, EB and CDI indicators must be specified in each call to *sna_start_write()*. The correct values are mainly dictated by the transmission mode and interface state in effect at the time of the call. (The caller has some discretion in the setting of the CDI.) A call of *sna_start_write()* fails if any of the following rules are violated.

- It is an error for your application to call *sna_start_write()* when the transmission mode is XM_DTR, or when the transmission mode is XM_HDX_FF and the interface state is either SA_BID or SA_RCV.
- The EB bit in the *sna_start_write() flags* parameter must always be zero (that is, only the host can end a bracket).
- The BB bit must be zero except in SA_CONTENTION state, where it must be 1.
- The CDI bit must be zero when in XM_FDX mode.

Table 6 shows the correct settings of the indicator bits in each mode and state. The three following macro names (as defined in **gcom/npiext.h**) are available to your application for use in setting/clearing the relevant flag bits:

N_SNA_CDI	Change direction indicator
N_SNA_BB	Begin bracket
N_SNA_EB	End bracket

The following code shows you how to set the CDI and clear the EB while leaving the BB unchanged:

```
flags |= N_SNA_CDI ;   /* set CDI  */
flags &= ~N_SNA_EB ;   /* clear EB */
```

Table 7 Flag Bit Settings (*repeated*)

Mode	*State*	*Flag Bits*
XM_HDX_FF	SA_BID	Writes not allowed
	SA_RCV	Writes not allowed
	SA_SND	BB=0; EB=0; CDI=1 or 0; MD=1 or 0
	SA_CONTENTION	BB=1; EB=0; CDI=1 or 0; MD=1 or 0
XM_FDX	-----	BB=0; EB=0; CDI=0; MD=0
XM_DTR	-----	Writes not allowed

BB = Begin Bracket; EB = End Bracket; CDI = Change Direction Indicator; MD = more data indicator (N_SNA_More_Chain)

In addition to these three write flag bits, an application also has access to two more:

N_SNA_MoreChain — This bit indicates that data to be sent to the host spans more than one buffer.

N_SNA_ALT_CODE — This bit indicates SNA alternate code. Use of the alternate code must have been set in bind.

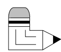
If a crypto session was requested in the bind and accepted by your application, the following flag bits must be set accordingly:

N_SNA_ENCRYPT — Indicates that data is encrypted

N_SNA_PAD — Indicates that data in the ru was not modulo 8 in length.

Furthermore, if the data has been compressed, the following flag bit should be set:

N_SNA_COMPRES — indicates compressed data



Note: *The following items are not supported or do not exist for 5250:*

- *BB and EB bits*
- *Transmission mode XM_DTR*
- *XM_HDX_FF interface states SA_BID and SA_CONTENTION*

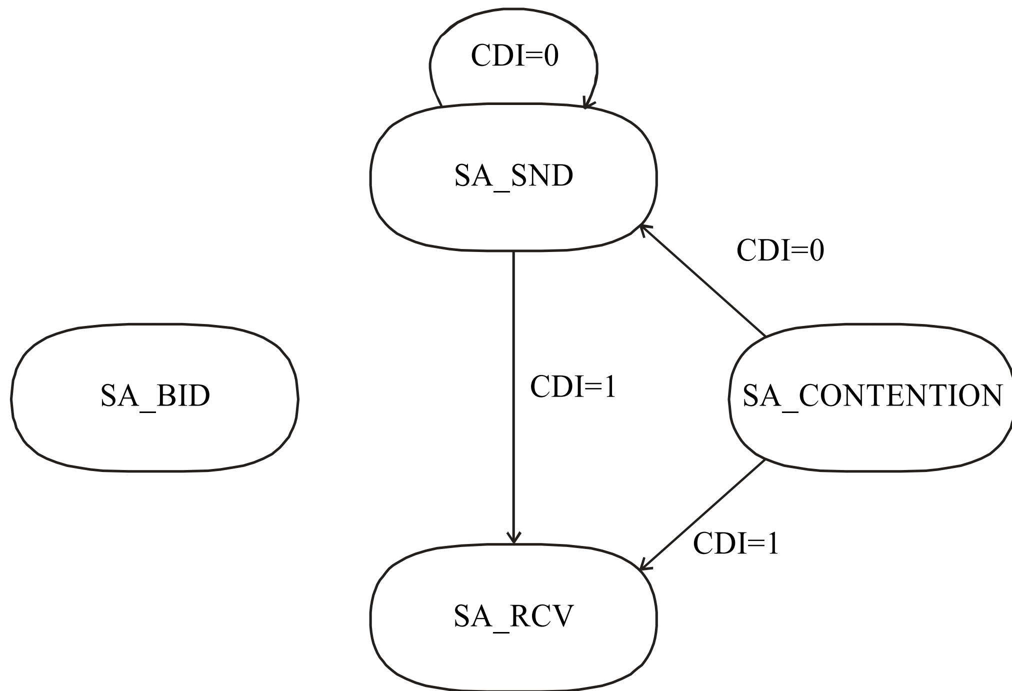


Figure 20 State Transitions Resulting From a 3270 Write Operation

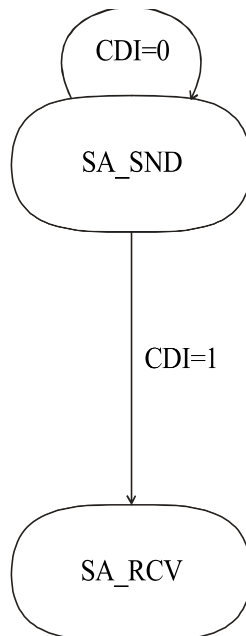


Figure 21 State Transitions Resulting From a 5250 Write Operation

State Transitions for 3270 and 5250 Writes

Figure 20 and 21 show the state transitions resulting from a write operation in XM_HDX_FF mode.

In XM_HDX_FF mode, the CDI can be set to one (no more data to follow) or zero (more data to follow) at your application's discretion whenever data is written to NPI/SNA.

A value of one drives the state to SA_RCV, allowing the SNA host to transmit and preventing further application writes until a change of state or mode occurs.

A CDI value of zero results in the state changing to SA_SND or remaining there. In this state, additional application writes are permitted after the current one completes and the SNA host is prohibited from transmitting.

Figure 22 Code Fragments for Performing a 3270 Write

```
  •
  •
int  bits ;                                    /* to pass to poll()     */
long flags ;                                   /* for BB, EB, CDI       */
int  save_if_state ;

save_if_state = if_state ;                     /* in case write failure */
flags         = 0 ;
  •
  •
  •
if ((write_pending != 0)                       /* write in progress  */
||  (sna_connected == 0))                      /* not connected      */
    ;                                          /* write not allowed  */

else
  switch (xmt_mode)
  {
  case XM_DTR:                                /* data traffic reset */
      break ;                                 /* write not allowed  */

    case XM_HDX_FF:                           /* bracket protocol   */
      if ((if_state == SA_BID)
      ||  (if_state == SA_RCV))
          break ;                             /* write not allowed */

        else
      {
        if (if_state == SA_CONTENTION)
            flags |= N_SNA_BB ;               /* start a bracket */

            if (we_choose_to_set_cdi)
        {
                flags |= N_SNA_CDI ;
            if_state = SA_RCV ;              /*     record      */
            }                                /*      the        */
                                             /*      new        */
        else                                 /*   interface     */
            if_state = SA_SND ;              /*     state       */
      }

                      /* drop through to next case to perform the write */

  •
  •
```

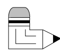
Example: Performing a 3270 Write

The code fragment in Figure 22, though not a part of **sim3270.c**, illustrates writing data to NPI/SNA in the general case. This is the sequence of events:

1. On entering this code, the current interface state is saved in *save_if_state* so it can be restored later if an error occurs.
2. Next, *flags* is set to zero so that the BB, EB and CDI bits will be zero in the *sna_start_write()* call unless explicitly changed by subsequent code.
3. Checks are then performed to make sure there is no write already pending and that an LU connection does exist. If either of these tests fail, a write is not attempted.
4. If no problems are discovered, the current transmission mode is examined. In the case of XM_DTR, a write is not attempted.

   If the mode is XM_HDX_FF, no write is attempted if the current interface state is either SA_BID or SA_RCV.

   If the state is SA_CONTENTION, the BB bit is set to 1 in *flags* as required by the protocol.



Note: *The* we_choose_to_set_cdi *variable is assumed to have been assigned a value of 1 or zero by other code prior to execution of the fragment shown here.*

5. If *we_choose_to_set_cdi* is non-zero the CDI bit is set to 1 in *flags.*
6. Next, the global variable *if_state* is changed to reflect the transition to either SA_RCV or SA_SND, as appropriate.

Figure 23 Code Fragments for Performing a 3270 Write *(repeated)*

```
/* switch (xmt_mode) */
    •
    •
    •
case XM_FDX:
    bits = sna_start_write (fid,    /* of NPI/SNA Stream  */
                            bfr,    /* data for host     */
                            count,  /* bytes to send     */
                            flags); /* BB, EB, CDI       */

    if (bits < 0)                   /* call failed       */
        if_state = save_if_state;   /* restore I/F state */

    else
    {
        write_pending   = 1;        /* one at a time     */
        sna_poll_events = bits;     /* update global     */
    }
}
    •
    •
    •
```

If the current transmission mode is XM_FDX, bracket protocol is not in effect, and so *flags* remains zero, unchanged from its original assignment.

7. Finally, *sna_start_write()* is called. If the return value is negative, the write attempt has failed. If the failure was caused by the NPI/SNA stream becoming unusable, the *usr_disconnected()* user-supplied callback procedure has been invoked by the API code.

   If *sna_start_write()* fails, the value of *if_state* on entry to this fragment is restored. Otherwise, *write_pending* is set to 1 to prevent further write attempts until the current write completes, and the global *sna_poll_events* is updated from the value returned by *sna_start_write()*.

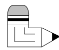
Figure 24 Code Fragments for Performing a 5250 Write

```
  •
  •
int  bits ;                                   /* to pass to poll()     */
long flags ;                                  /* for BB, EB, CDI       */
int  save_if_state ;

save_if_state = if_state ;                    /* in case write failure */
flags         = 0 ;
  •
  •
  •
if ((write_pending != 0)                      /* write in progress  */
||  (sna_connected == 0))                     /* not connected      */
    ;                                         /* write not allowed  */

else
  switch (xmt_mode)
  {
    case XM_HDX_FF:                          /* half-duplex flip-flop */
      if (if_state == SA_RCV)
          break ;                            /* write not allowed */

      else
      {

        if (we_choose_to_set_cdi)
        {
            flags |= N_SNA_CDI ;
            if_state = SA_RCV ;             /*     record      */
        }                                   /*      the        */
                                            /*      new        */
        else                                /*   interface     */
            if_state = SA_SND ;             /*     state       */
      }

                      /* drop through to next case to perform the write */

  •
  •
```

Example: Performing a 5250 Write

The code fragment in Figure 24, though not a part of **sim5250.c**, illustrates writing data to NPI/SNA in the general case. This is the sequence of events:

1. On entering this code, the current interface state is saved in *save_if_state* so it can be restored later if an error occurs.
2. Next, *flags* is set to zero so that the BB, EB and CDI bits will be zero in the *sna_start_write()* call unless explicitly changed by subsequent code.
3. Checks are then performed to make sure there is no write already pending and that an LU connection does exist. If either of these tests fail, a write is not attempted.
4. If the mode is XM_HDX_FF, no write is attempted if the current interface state is SA_RCV.



Note: *The* we_choose_to_set_cdi *variable is assumed to have been assigned a value of 1 or zero by other code prior to execution of the fragment shown here.*

5. If *we_choose_to_set_cdi* is non-zero the CDI bit is set to 1 in *flags.*
6. Next, the global variable *if_state* is changed to reflect the transition to either SA_RCV or SA_SND, as appropriate.

Figure 25 Code Fragments for Performing a 5250 Write *(continued)*

```
/* switch (xmt_mode) */
    •
    •
    •
    bits = sna_start_write (fid,    /* of NPI/SNA Stream  */
                            bfr,    /* data for host     */
                            count,  /* bytes to send     */
                            flags); /* BB, EB, CDI       */

/* SS-LU data */

case XM_FDX                         /* Data length not > 128 bytes */
    bits = sna_sslu_start_write (fid, bfr, count, flags);

    if (bits < 0)                   /* call failed       */
        if_state = save_if_state;   /* restore I/F state */

    else
    {
        write_pending   = 1;        /* one at a time     */
        sna_poll_events = bits;     /* update global     */
    }
}
    •
    •
    •
```

For 5250 data sent on the SS-LU session, the *sna_sslu_start_write()* routine is called. This routine is called when in XM_FDX mode or if an SS-LU message is to be sent when in XM_HDX_FF mode.

7. Finally, *sna_start_write()* is called. If the return value is negative, the write attempt has failed. If the failure was caused by the NPI/SNA stream becoming unusable, the *usr_disconnected()* user-supplied callback procedure has been invoked by the API code.

   If *sna_start_write()* fails, the value of *if_state* on entry to this fragment is restored. Otherwise, *write_pending* is set to 1 to prevent further write attempts until the current write completes, and the global *sna_poll_events* is updated from the value returned by *sna_start_write()*.

Figure 26 *usr_bid()* **sim3270.c** and **sim5250.c** Version

```
int
usr_bid (fid)
    int fid;
{
    if_state = SA_BID;            /* waiting begin bracket */

    return (0);                   /* Accept the BID */
}
```

Figure 27 *usr_connected()* **sim3270.c** and **sim5250.c** Version

```
void
usr_connected (fid)
    int fid;
{
    xmt_mode      = XM_FDX;
    sna_connected = 1;
    write_pending = 0;

    send_msgs (0);              /* to HOST */
}
```

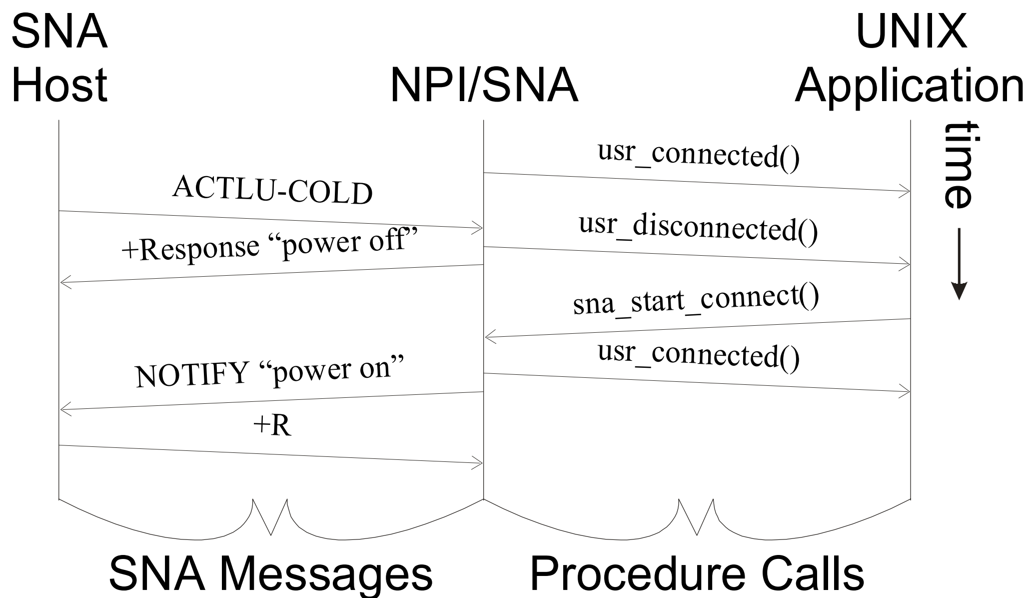


Figure 28 LU Disconnection

User-Supplied Callback Procedures For sim3270.c and sim5250.c

In this section the NPI/SNA user-supplied callback procedures are examined as they are implemented in **sim3270.c** and **sim5250.c**.

usr_bid()

Figure 27 shows the **sim3270.c** and **sim5250.c** implementation of the *usr_bid()* user-supplied callback procedure. The host's request to begin a bracket is unconditionally accepted and the interface state is adjusted accordingly.

usr_connected()

NPI/SNA invokes the *usr_connected()* user-supplied callback procedure when a connection to a local LU has been established (or reestablished). At that point the transmission mode becomes XM_FDX. The global variables *xmt_mode* and *sna_connected* are set accordingly in the **sim3270.c** and **sim5250.c** implementation shown in Figure 28. Because no write can be in progress when a connection is established, *write_pending* is set to zero.

The *send_msgs()* procedure is then called to write any queued data to NPI/SNA. A zero parameter is passed because there is nothing to add to the queue at this point.

usr_disconnected()

Figure 28 shows a scenario where an ACTLU-COLD command is received from the SNA host. NPI/SNA then disconnects the LU connection, invoking the *usr_disconnected()* user-supplied callback procedure to inform your application. Your application attempts to reconnect by calling *sna_start_connect_lu()*.

If there is a write pending when the disconnection occurs, it is silently terminated. If a read is pending, NPI/SNA flushes any data accumulated in the read buffer up to this point, but does not actually terminate the read itself.

Figure 29 *usr_disconnected()* **sim3270.c** and **sim5250.c** Version

```
usr_disconnected (fid, diagnostic, file_closed)
    int fid;
    int diagnostic;
    int file_closed;
{
    int poll_bits;

    sna_connected = 0;
    write_pending = 0;

    /*
     * If the file descriptor is closed, an attempt is made to
     * get a new file descriptor, and update the poll structure.
     */
    if (file_closed)
    {
        fid = sna_open (usr_read_complete,
                        usr_write_complete,
                        usr_connected,
                        usr_host_event,
                        usr_bid,
                        usr_disconnected);

        if (fid < 0)
        {
            perror ("usr_disconnected - sna_open");
            pollfd[SNA_FD].fd = -1;                  /* stream closed */
            return;                                  /* exit() ? */
        }

        pollfd[SNA_FD].fd = fid;                     /* new fid */
    }

    poll_bits = sna_start_connect_lu (fid, 0, connect_lu, 0, 10);

    if (poll_bits < 0)
    {
        perror ("usr_disconnected");
        return;
    }

    pollfd[SNA_FD].events = poll_bits;
}
```

Figure 29 shows the **sim3270.c** and **sim5250.c** implementation of the *usr_disconnected()* user-supplied callback procedure. The parameters are as follows:

fid	File descriptor of the NPI/SNA stream
diagnostic	Cause of the disconnection (See “SNA Disconnect Diagnostic Codes” on page 160 for details.)
file_closed	One if *fid* is no longer valid, zero otherwise.

This is the **sim3270.c** and **sim5250.c** code sequence of *usr_disconnected():*

1. Upon entry, the globals *sna_connected* and *write_pending* are set to zero to reflect the connection closure and termination of any writes in progress.
2. If *file_closed* is non-zero, an attempt is made to open a new NPI/SNA stream.

   If the attempt fails, an error message is printed and then pollfd[SNA_FD].fd is set to -1. This prevents *poll()* from attempting to report on events occurring on the now closed stream. Control is then returned to the caller. In an actual application, somewhat more drastic corrective action should be taken.
3. If the call to *sna_open()* is successful, *pollfd[SNA_FD].fd* is assigned the value of the returned file descriptor.
4. *sna_start_connect_lu()* is called to initiate a new LU connection.
5. If the call to *sna_start_connect_lu()* is successful, the return value, which indicates the current events of interest on the NPI/SNA stream, is copied to *pollfd[SNA_FD].events.*

Figure 30 *usr_host_event()* **sim3270.c** and **sim5250.c** Version

```
void
usr_host_event (fid, mode, state, event)
    int fid;
    int mode;
    int state;
    int event;
{
    xmt_mode      = mode;                    /* same definition as API */
    if_state      = state;                   /* same definition as API */
    write_pending = 0;                       /* any write is canceled */

    switch (event)
    {
    case Event_HOST_aborted_BB:
    case Event_ServiceInterrupt:            /* Reserved for future use */
    case Event_ACTLU:                       /* ERP - warm start */
    case Event_UNBIND:
    case Event_SDT:                         /* Start Data Transfer */
    case Event_CANCEL_EB:                   /* End Bracket */
    case Event_CHASE_EB:                    /* End Bracket */
    case Event_CHASE_CDI:                   /* host passed cdi */
    case Event_HostNegRes:                  /* Negative response from host */
        send_msgs (0);                      /* in case more msgs to send */
        break;

    case Event_CLEAR:                       /* wait SDT */
    case Event_BIND:                        /* wait SDT */
    case Event_Shutdown:                    /* SHUTC/SHUTD */
    case Event_SIG:                         /* host has right to send */
        break;                              /* send_msgs() will defer */
     }
}
```

usr_host_event()

The **sim3270.c** and **sim5250.c** implementation of *usr_host_event()* is shown in Figure 30. The four parameters contain the file descriptor of the NPI/SNA stream, the updated transmission mode and interface state, and the event that triggered the call.

Here are the sequence of events:

1. Upon entry, the globals *xmt_mode* and *if_state* are updated from the values passed in, and *write_pending* is set to zero to reflect the fact that any write in progress has been terminated.
2. The *event* parameter is examined to determine the next appropriate action.

   In the case of Figure 30, the thirteen cases degenerate into only two. If the event is one that does not leave the API in a mode or state that prohibits writes, *send_msgs()* is invoked to transmit any queued messages to the host.

   Control is then returned to the caller.

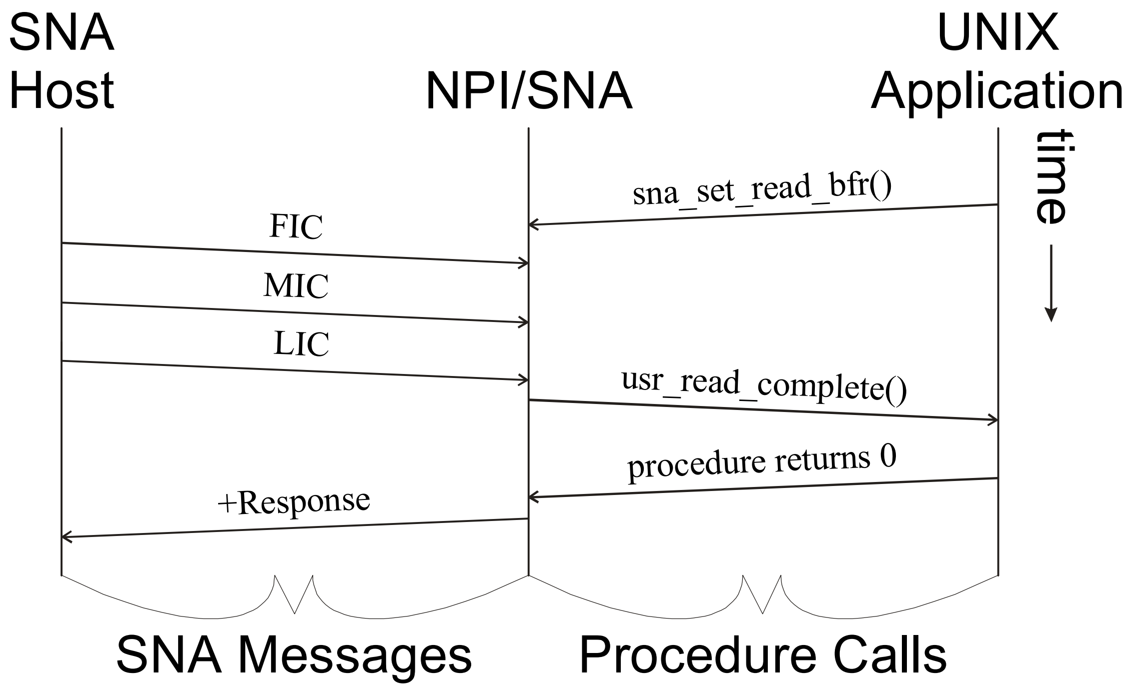


Figure 31 Host Chain Accepted

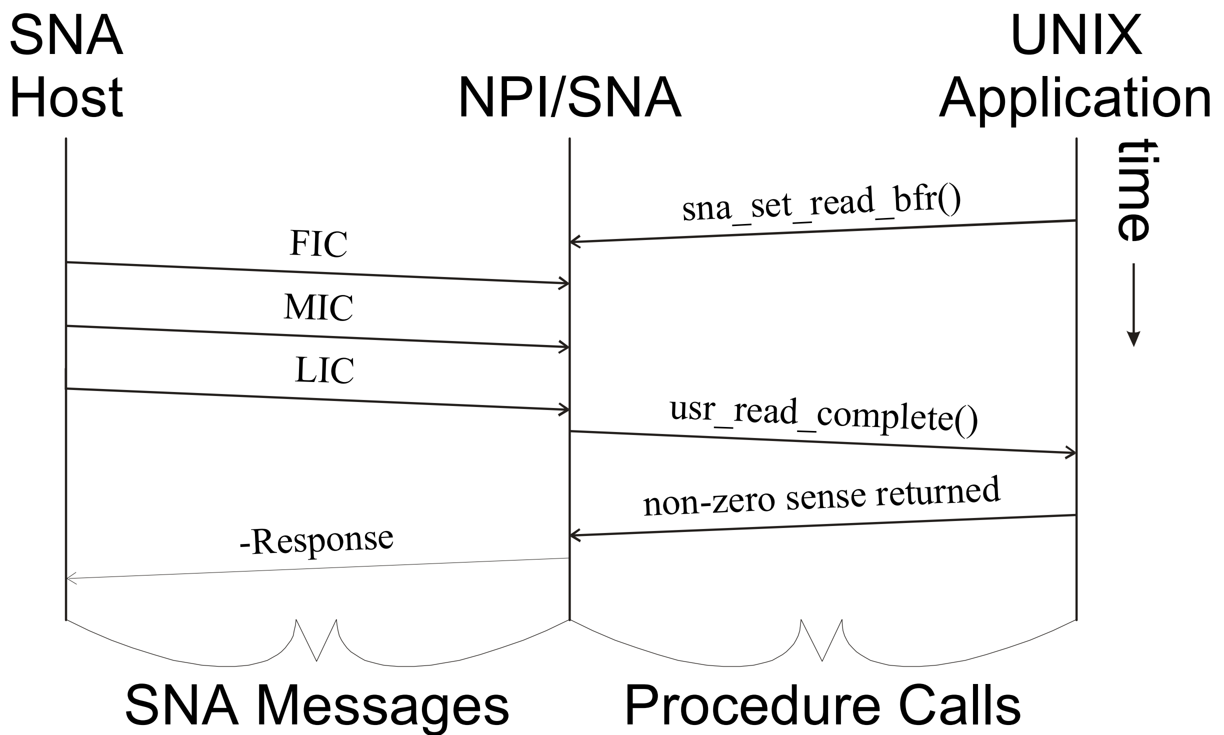


Figure 32 Host Chain Rejected

usr_read_complete()

When data arrives from the host, NPI/SNA invokes *usr_read_complete()* to deliver it to your application. If the data arrives in a multi-element chain, *usr_read_complete()* is not called until the complete chain has been assembled (last-in-chain [LIC] element is received).

Your application indicates acceptance of the chain by returning zero, or rejects it by returning a non-zero value that NPI/SNA then returns to the host as a sense code in a negative response.

Figures 31 and 32 show both cases.

Once NPI/SNA calls *usr_read_complete()*, the API module is temporarily left without a buffer into which the next host chain can be received. It is your application's responsibility to provide such a buffer via *sna_set_read_bfr()*.

NPI/SNA does provide some internal buffering at the driver level. Nevertheless, there is a danger of data being lost if *sna_set_read_bfr()* is not called within a reasonable time after *usr_read_complete()* is invoked. This is especially true if a call to *sna_set_buffering_mode()* set the buffering mode to something other than CHAIN_MODE_CHAIN.

Figure 33 *usr_read_complete()* **sim3270.c** and **sim5250.c** Version

```
unsigned long                                               /* sense */
usr_read_complete (fid, bfr, length, flags)
    int                   fid;
    unsigned char *bfr;
    int                   length;
    long               flags;
{

    unsigned long    result;
    int                   poll_bits;

    /*
     * Immediately start another read on the buffer knowing that
     * the buffer will not be modified by the API call.
     */
    poll_bits = sna_set_read_bfr (fid, buf, sizeof (buf));

    if (poll_bits < 0)                                     /* failure */
        perror ("usr_read_complete - reread");
    else
        pollfd[SNA_FD].events = poll_bits;                 /* update global */

    if (xmt_mode == XM_FDX)
        send_msgs (0);                                     /* for keystrokes */

    else if (xmt_mode == XM_HDX_FF)
    {
        if (flags & N_SNA_EB)                      /* 3270 only: End Bracket */
        {                                          /* 3270 only */
            if_state = SA_CONTENTION;              /* 3270 only */
            send_msgs  (0);                        /* 3270 only for keystrokes */
        }                                          /* 3270 only */
        else if (flags & N_SNA_CDI)                        /* Change Direction */
        {
            if_state = SA_SND;
            send_msgs  (0);                                /* for keystrokes */
        }
        else
            if_state = SA_RCV;
    }

    write_dots (1);                                       /* send another dot */
    return (0L);                                          /* accept host chain */
}
```

Figure 33 shows the **sim3270.c** and **sim5250.c** implementation of the *usr_read_complete()* user-supplied callback procedure. The four parameters are:

- The NPI/SNA file descriptor
- A pointer to a buffer (previously passed to NPI/SNA via *sna_set_read_bfr()*)
- The number of bytes of data
- The *flags* parameter holding the values of the BB, EB and CDI bits in the received chain.

Here is the sequence of events for *usr_read_complete():*

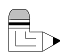
1. Before processing the data, *sna_set_read_bfr()* is called to prime NPI/SNA for receipt of the next chain.

   If no error occurs the return value, which indicates the current events of interest on the NPI/SNA stream, is copied to pollfd[SNA_FD].events.

   NPI/SNA does not write into the buffer passed to *sna_set_read_bfr()* until after *usr_read_complete()* returns, so no danger of data corruption exists.

2. *xmt_mode* is checked and if the current transmission mode is XM_HDX_FF, the *flags* parameter is examined and the interface state updated accordingly.

3. If writes are allowed in the new mode/state, *send_msgs()* is called to send queued data to the host.



Note: *As shown in Figure 33, 5250 does not test the (*flags & N_SNA_EB*) condition.*

4. Then, *write_dots()* is called with a parameter of 1, to indicate that an additional chain has arrived.

5. Finally, the chain is unconditionally accepted by returning zero.

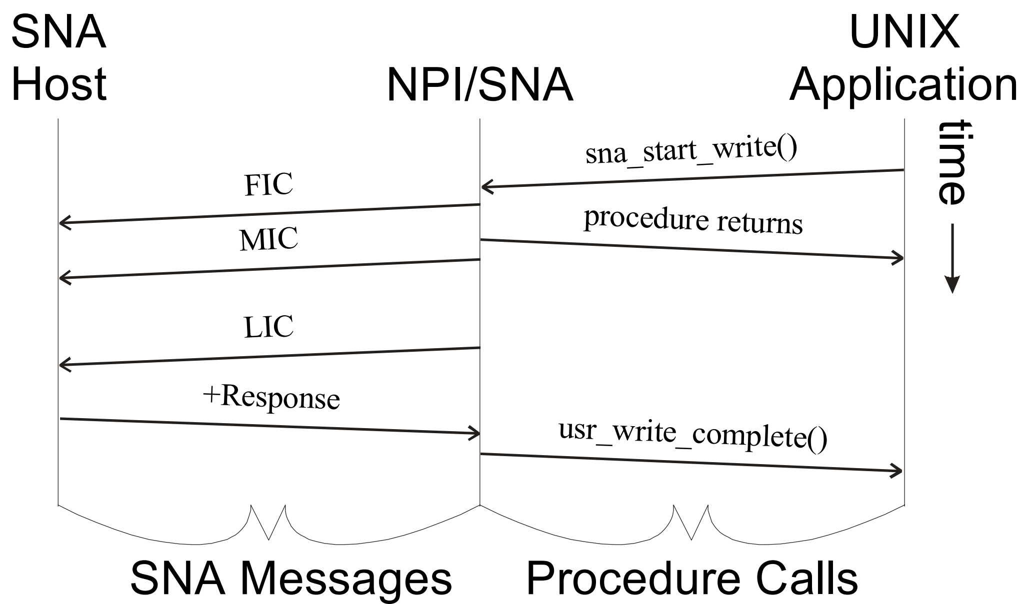


Figure 34 Chain Sent to Host

Figure 35 *usr_write_complete()* **sim3270.c** and **sim5250.c**

```
void
usr_write_complete (fid)
    int fid;
{
    msgs_to_send--;        /* one less to send */
    write_pending = 0;     /* write completed */
    send_msgs (0);         /* in case more msgs to send */
}
```

usr_write_complete()

Figure 34 shows how an application calls the *sna_start_write()* API procedure to send data to the host. If there is more data than can be sent in a single Request/Response Unit (RU), NPI/SNA transmits it in multiple chain elements.

The behavior of write complete notification is controlled by the host-specified BIND parameters. If the BIND requires that the secondary LU send data in exception response mode, the host does not send positive responses to correctly received data. That is, the positive response shown in Figure 34 is not sent by the host. Therefore, your application's *usr_write_complete()* procedure is invoked when the complete chain has been assembled.

If a positive response is received from the host, the *usr_write_complete()* user-supplied procedure is invoked, assuming that the BIND parameters enable the NPI/SNA to transmit data using definite response mode.

If the N_SNA_More_Chain flag bit was set in the write buffer flags, when the NPI/SNA completes sending the data in the buffer it will invoke your application's usr_write_complete() with the type parameter set to Write_Intermediate. This indicates the data in the buffer is complete and the host has received the data correctly. This parameter value only indicates that the buffer has been transmitted.

If a negative response is received from the host, *usr_write_complete()* is not called. Instead, your application's *usr_host_event()* procedure is invoked with the *events* parameter set to Event_HostNegRes.

Figure 35 shows the **sim3270.c** and **sim5250.c** implementation of this procedure, whose details are as follows:

1. *msgs_to_send* is decremented to reflect the successful transmission of the current message.
2. *write_pending* is set to zero.
3. *send_msgs()* is called to start another write if there are more messages queued.

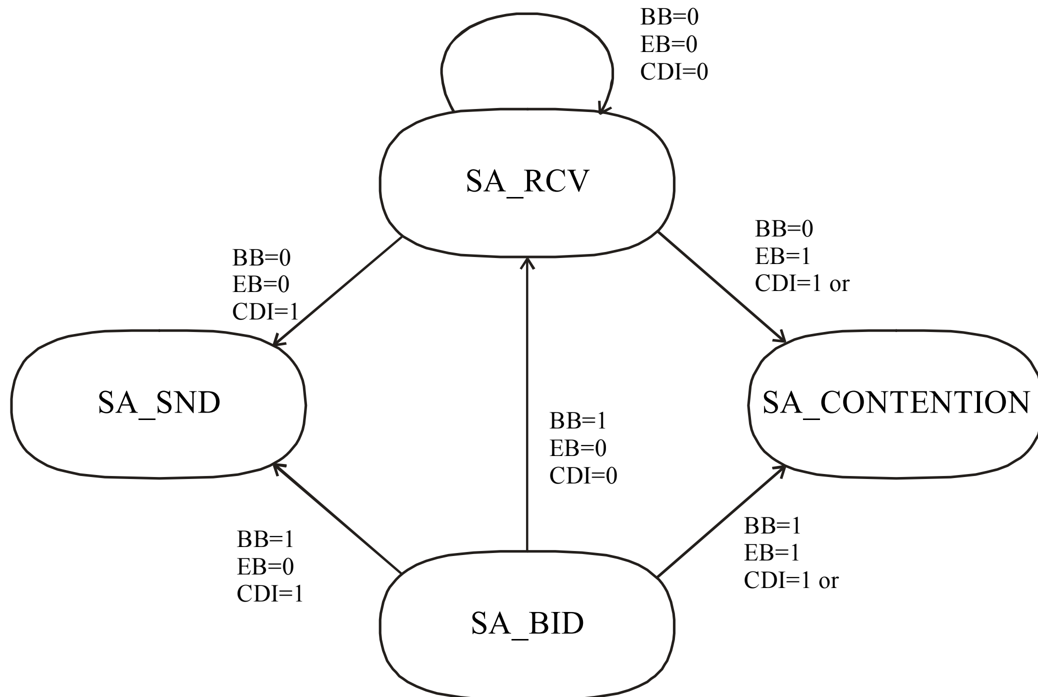
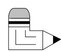


Figure 36 API State Transitions Upon 3270 Read Completion



Note: *Figure 36 applies only for positive responses.*

User-Supplied Callback Procedures in sim3270.c Only

In this section the NPI/SNA user-supplied callback procedures are examined as they are implemented in **sim3270.c** only.

State Changes Caused by Receipt of a Chain

In XM_HDX_FF mode, the interface state is affected by Begin Bracket (BB), End Bracket (EB) and/or Change Direction Indicator (CDI) bits in chains received from the host. The values of these bits are passed by NPI/SNA to the *usr_read_complete()* user-supplied callback procedure.

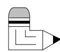
Figure 36 illustrates the state transitions occurring when *usr_read_complete()* is invoked.

No transitions are shown from SA_SND or SA_CONTENTION. Receipt of data from the host while in SA_SND is a protocol violation and will be rejected by NPI/SNA. (Receipt of chains with bit settings other than those shown are similarly rejected.)

If data arrives while in SA_CONTENTION, the *usr_bid()* callback procedure is invoked. NPI/SNA only calls *usr_read_complete()* after your application accepts the BID, thus changing the state to SA_BID.

From the perspective of *usr_read_complete(),* the new state in Figure 36 is not dependent upon the previous state, nor on BB. It is only dependent on EB and CDI. The following are the state transitions if the chain is accepted:

EB	*CDI*	*New State*
1	1 or 0	SA_CONTENTION
0	1	SA_SND
0	0	SA_RCV



Note: *If the application rejects the chain, the resulting state is SA_RCV.*

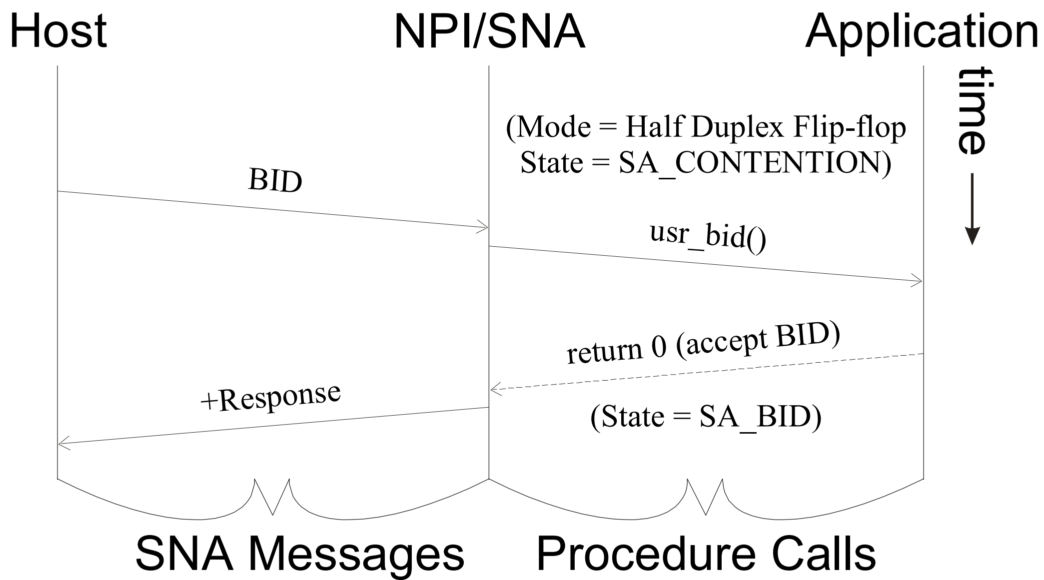


Figure 37 Host BID Accepted (3270 Only)

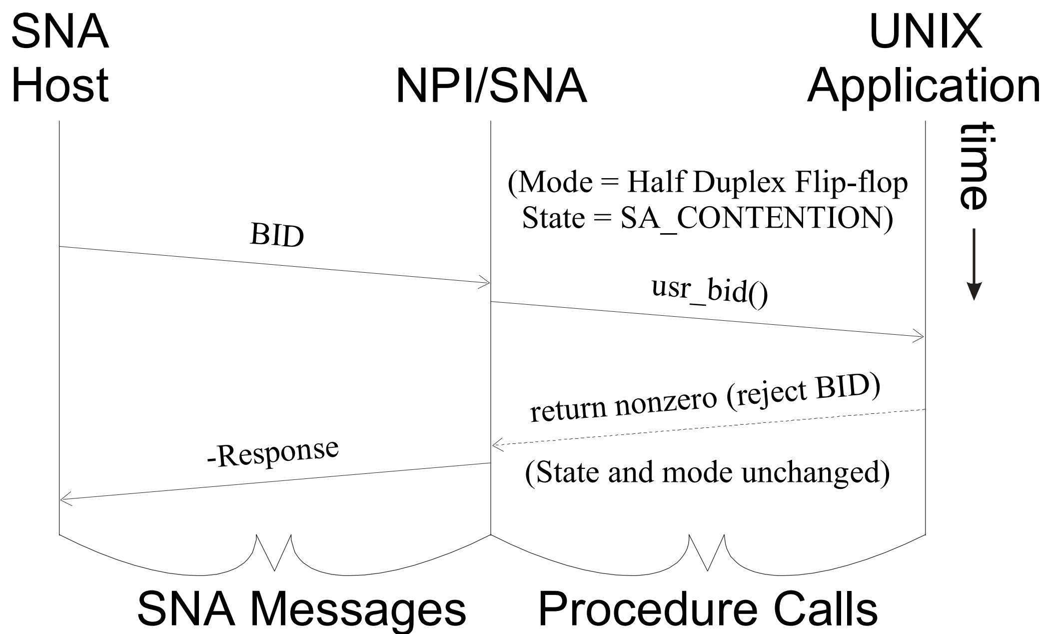


Figure 38 Host BID Rejected (3270 Only)

usr_bid()

In SA_CONTENTION state (XM_HDX_FF mode), both the SNA host and your application can initiate a bracket by sending a chain containing a BB indicator. The host can also attempt to begin a bracket by first sending a BID command.

When either a BID command or a chain with BB (not preceded by a BID command) is received, NPI/SNA invokes *usr_bid()*, passing the file descriptor of the NPI/SNA stream. Your application can then grant or deny the host's request to begin a bracket by returning zero or non-zero, respectively. Granting the request changes the interface state to SA_BID. Otherwise the state remains SA_CONTENTION.

Figure 37 and 38 show examples of a host BID command being accepted and rejected, respectively.

Once a BID command has been accepted, NPI/SNA does not invoke *usr_bid()* when the subsequent chain arrives from the host.

Your application has no way of knowing if *usr_bid()* has been invoked because of a host BID command or chain with BB.

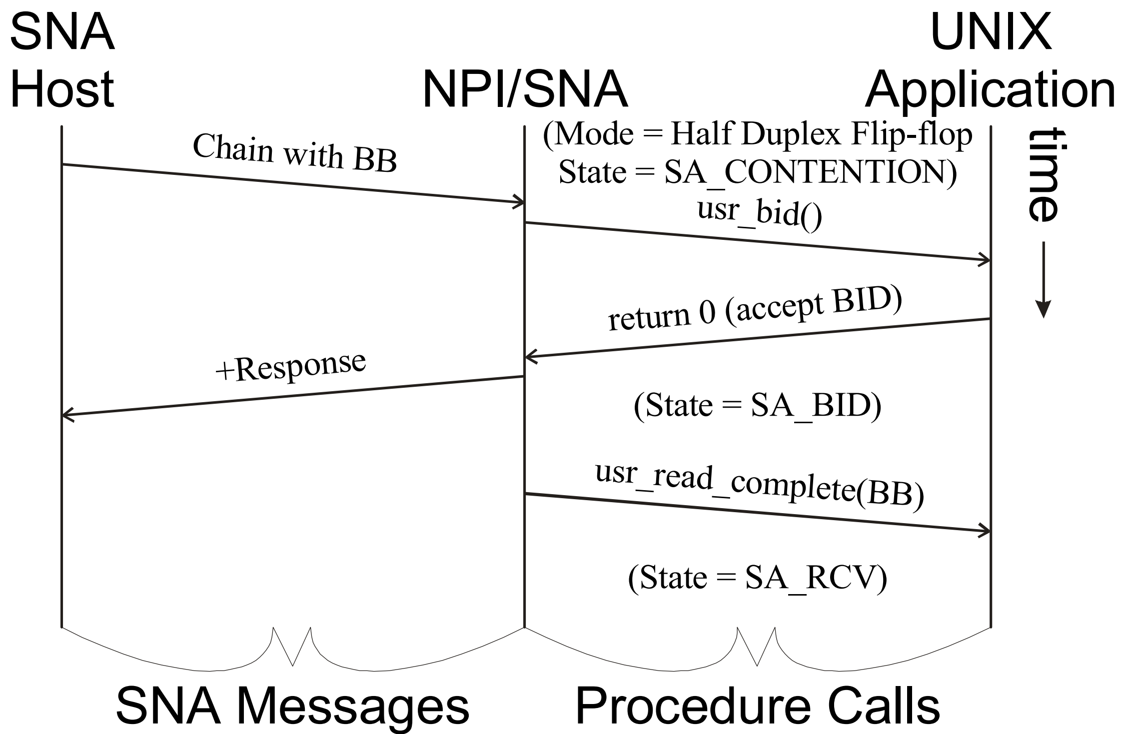


Figure 39 Host BB Accepted (3270 Only)

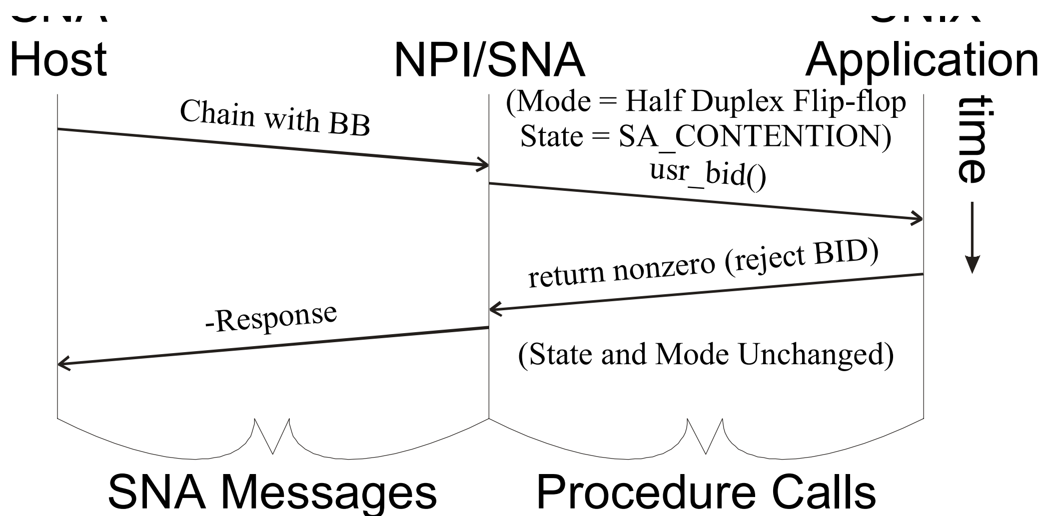


Figure 40 Host BB Rejected (3270 Only)

BB Without Preceding BID

Figure 39 and 40 show the host sending a chain with BB not preceded by a BID command.

If your application grants the request to begin a bracket, NPI/SNA passes the data contained in the chain to your application via *usr_read_complete()*.

If the chain has multiple elements, the *usr_bid()* user-supplied callback procedure is invoked when the first element, which carries the BB, is received. Assuming your application accepts the BID, the *usr_read_complete()* user-supplied callback procedure is invoked when the complete chain has been assembled.

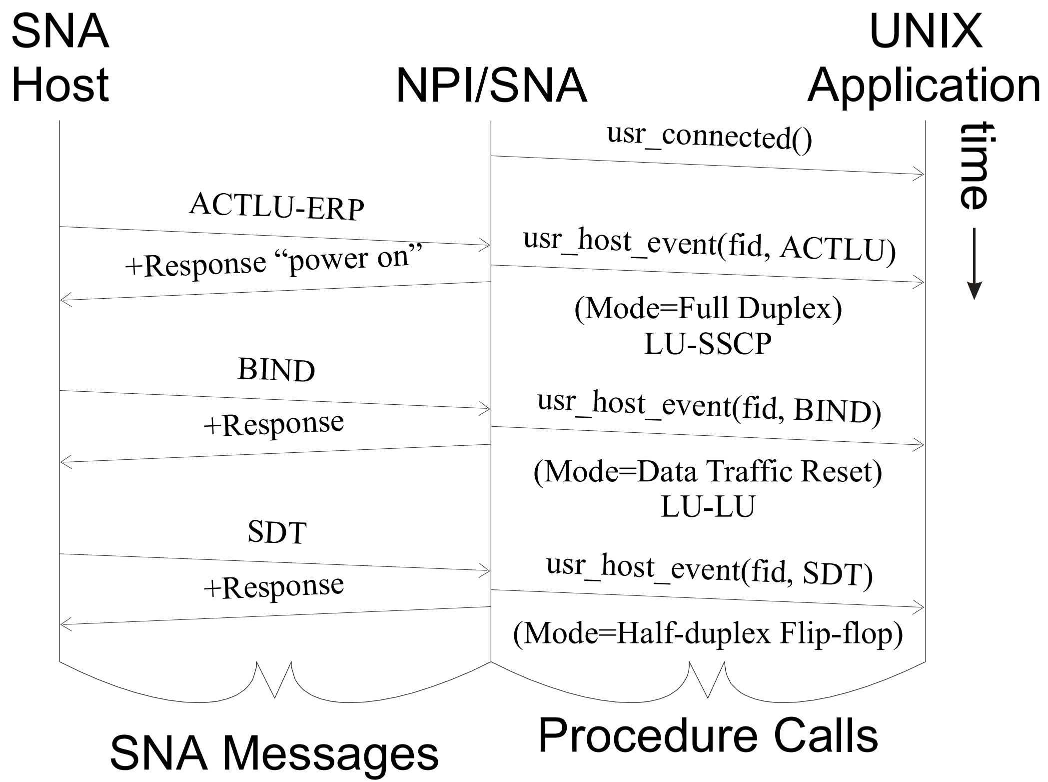


Figure 41 Host Events: ACTLU, BIND, and SDT (3270 Only)

usr_host_event()

Figure 41 shows how *usr_host_event()* informs your application when NPI/SNA receives ACTLU-ERP, BIND, and SDT commands. When any one of the (usually host-initiated) special events occur, NPI/SNA informs your application by calling *usr_host_event()*.

Because the various events can affect the transmission mode and the interface state, *usr_host_event()* parameters carry updated values. Table 8 shows the mode and state resulting from each of the events.

Table 8 New *usr_host_event()* States for 3270 Only

Host Event	*New Mode*	*New State*
Event_HOST_aborted_BB	XM_HDX_FF	SA_CONTENTION
Event_ServiceInterrupt	XM_FDX	-----
Event_ACTLU	XM_FDX	-----
Event_CLEAR	XM_DTR	-----
Event_SDT	XM_HDX_FF	SA_CONTENTION
Event_BIND	XM_DTR	-----
Event_UNBIND	XM_FDX	-----
Event_SIG	XM_HDX_FF	SA_RCV
Event_Shutdown	XM_DTR	-----
Event_CANCEL_EB	XM_HDX_FF	SA_CONTENTION
Event_CANCEL_CDI	XM_HDX_FF	SA_SND
Event_CHASE_EB	XM_HDX_FF	SA_CONTENTION
Event_CHASE_CDI	XM_HDX_FF	SA_SND
Event_HostNegRes	no change	The *state* parameter sets the appropriate state.
Event_SetBidState	no change	SA_BID
Event_CRV	no change	-----

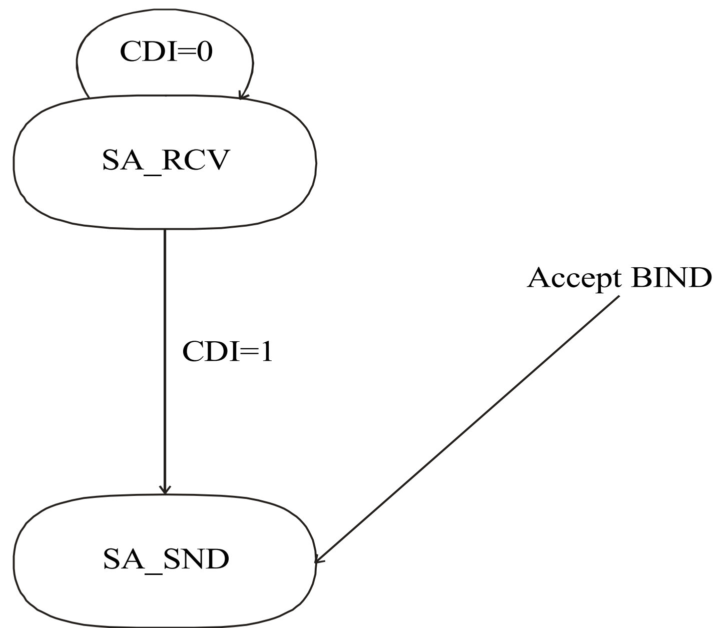


Figure 42 API State Transitions Upon 5250 Read Completion

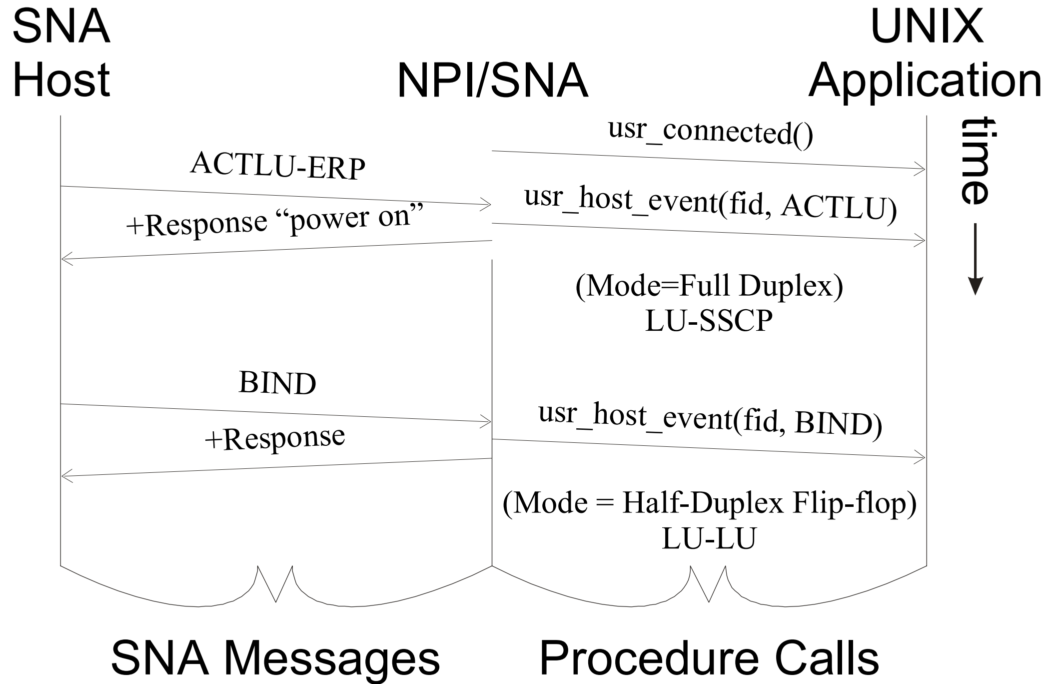


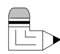
Figure 43 Host Events: ACTLU and BIND (5250 Only)

User-Supplied Callback Procedures in sim5250.c Only

In this section the NPI/SNA user-supplied callback procedures are examined as they are implemented in **sim5250.c** only.

State Changes Caused by Receipt of a Chain

For 5250 connections, acceptance of the BIND causes your application to enter SA_RCV state, as shown in Figure 42. The LU-LU session interface is affected only by the Change Direction Indicator (CDI). The value of this bit is passed by NPI/SNA to the *usr_read_complete()* user-supplied callback routine.



Note: *If the application rejects the chain, the resulting state is SA_RCV..*

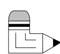
usr_host_event()

Figure 43 shows how *usr_host_event()* informs your application when NPI/SNA receives ACTLU-ERP and BIND commands. When any one of the (usually host-initiated) special events occur, NPI/SNA informs your application by calling *usr_host_event()*.

Because the various events can affect the transmission mode and the interface state, *usr_host_event()* parameters carry updated values. Table 9 shows the mode and state resulting from each of the events.

Table 9 New *usr_host_event()* States for 5250 Only

Host Event	*New Mode*	*New State*
Event_ServiceInterrupt	XM_FDX	-----
Event_ACTLU	XM_FDX	-----
Event_BIND	XM_DTR	-----
Event_UNBIND	XM_FDX	-----
Event_SIG	XM_HDX_FF	SA_RCV
Event_CANCEL_CDI	XM_HDX_FF	SA_SND
Event_CHASE_CDI	XM_HDX_FF	SA_SND
Event_HostNegRes	no change	Same as when *sna_start_write()* was invoked



Note: *ACTLU-ERPs (error recovery procedures) are not reported to applications for 5250.*

Figure 44 *usr_log_error_complete()* in **sim5250.c**

```
/*********************************************************************
*                  usr_log_error_complete                            *
**********************************************************************
*    This routine would receive the results of an attempt to write an   *
*    error log entry                      *
*********************************************************************/
void
usr_log_error_complete(fid, status, sense)
    int fid;
    int status;
    int sense;
{
   return;
}
```

Figure 45 *usr_request_complete()* in **sim5250.c**

```
/*********************************************************************
*                  usr_request_complete                              *
**********************************************************************
*    This routine would receive the results of a user request          *
*********************************************************************/
void
usr_request_complete(fid, request, status, sense)
    int fid;
    int request;
    int status;
    int sense;
{
   return;
}
```

usr_log_error_complete()

Although the routine in Figure 44 is never called in **sim5250.c**, it would be used to receive the results of a *sna_log_error()* call. The *status* parameter indicates the success or failure of the *sna_log_error()* call and tells your application if it was rejected by the host or locally (by the API or the GCOM SNA server software). If the *status* parameter is zero, the *sense* parameter has no meaning. Otherwise, if the *sense* parameter is -1, an invalid log request was received. See “usr_log_error_complete()” on page 151 for the seven reasons why it would be invalid.

If *status* is -2, then the *sense* parameter contains the negative response from the host.

usr_request_complete()

Although the routine in Figure 45 is never called in **sim5250.c**, it would be used to receive the results of a user request. The *status* parameter indicates the success or failure of the request and tells your application if it was rejected by the host or locally (by the API or the GCOM SNA server software itself). If the *status* parameter is zero, the *sense* parameter has no meaning. Otherwise, if the *sense* parameter is -1, an invalid request was received. See “usr_request_complete()” on page 153 for the four reasons why it would be invalid. If *status* is -2, then the *sense* parameter contains the negative response from the host.

Figure 46 *usr_sslu_read_complete()* in **sim5250.c**

```
/*************************************************************************
*                         usr_sslu_read_complete                         *
**************************************************************************
*                                                                        *
* An sslu message has been received from the host.  If conditions permit,  *
* attempt to write a possibly queued message to the host.               *
*                                                                        *
* In response to the received message, enqueue a "S" and attempt to     *
* write it to the screen.                                                *
*                                                                        *
*************************************************************************/
unsigned long                                                /* sense */
usr_sslu_read_complete (fid, bfr, length, flags)
    int                   fid;
    unsigned char        *bfr;
    int                   length;
    long                  flags;
{
    unsigned long         result;
    int                   poll_bits;

    if (xmt_mode == XM_FDX)
        send_sslu_msgs (0);                               /* for keystrokes */

    write_dots (1,'S');                                   /* write a S */
    return (0L);                                          /* accept host chain */
}
```

usr_sslu_read_complete()

Figure 46 shows the **sim5250.c** implementation of the *usr_sslu_read_complete()* user-supplied callback procedure. The four parameters are:

- The NPI/SNA file descriptor
- A pointer to a temporary buffer. It is valid only until this routine returns.
- The number of bytes of data
- This is always zero.

If still in full-duplex mode, this routine sends another message and an “S” is written to the screen.

Figure 47 *usr_sslu_write_complete()* in **sim5250.c**

```
/*****************************************************************************
*                        usr_sslu_write_complete                             *
******************************************************************************
*                                                                            *
* A message previously written to the host has been acknowledged.           *
*                                                                            *
* send_msgs() is invoked to attempt to send another message to the          *
* host as appropriate.                                                       *
*****************************************************************************/
void
usr_sslu_write_complete (fid,status,sense)
    int fid;
    int status;
    int sense;
{
    sslu_msgs = 0;
    sslu_write_pending = 0;                             /* write completed */
    send_sslu_msgs (0);                        /* in case more msgs to send */
}
```

usr_sslu_write_complete()

For SSLU data, the *usr_sslu_write_complete()* routine is invoked. The *sense* parameter indicates whether a positive or negative response has been received from the host.

Figure 47 shows the **sim5250.c** implementation of this procedure, whose details are as follows:

1. *msgs_to_send* is set to zero.
2. *write_pending* is set to zero.
3. *send_msgs()* is called to start another write if there are more SSLU messages queued.

Non-Callback Procedures In sim3270.c and sim5250.c

See the listings starting on page 186 to find the following three non-callback procedures that are specific to **sim3270.c** and **sim5250.c**:

- *send_msgs()*
- *write_dots()*
- *keyboard_events().*

API Overview

The GCOM NPI/SNA API consists of eleven procedure calls that are entry points to the API library. As complements, your appliation must provide ten callback procedures, usually referred to as *user-supplied callback procedures*. As a collective, this set of routines provides SNA connectivity to an application for the following lu types: lu type 0, 1, 2, 3, 4, 7. The entry points and callback procedures which provide this support are described in detail below.

NPI Streams Interface

Your application can directly access the GCOM NPI Streams Interface, which underlies the NPI/SNA API. In general, the application should avoid using direct NPI Streams Interface calls, with the exception of routines that provide logging support, such as *npi_printf()* and *npi_error()*.

NPI/SNA API Routine Entry Points

Common API Routines

Ten of the NPI/SNA API routines apply to both the 5250 and 3270 protocols:

sna_api_set_debug_level	Sets the level of debugging reporting
sna_close()	Close an NPI/SNA stream
sna_get_api_state()	Determine the current transmission mode and interface state
sna_init_log()	Specify log options and a log file name
sna_open()	Open an NPI/SNA stream
sna_poll_retry()	Inform NPI/SNA that a read or write should be retried
sna_send_request()	Causes one of three SNA command types to be sent to the remote SNA host: SIGNAL, LUSTAT, REQTEST, or INIT-SELF.
sna_set_read_bfr()	Initiate a non-blocking read on an NPI/SNA stream
sna_sslu_start_write()	Initiates a non-blocking write of data on the SS-LU session.
sna_start_connect_lu()	Initiate a connection to a local LU
sna_start_write()	Initiate a non-blocking write on an NPI/SNA stream

5250-Specific API Routines

One of the NPI/SNA API routines applies only to the 5250 protocol:

sna_log_error()	Allows you to enter data into the log in a non-blocking mode.

User-Supplied Callback Procedures

Common Callback Routines

Nine of the callback routines apply to both the 5250 and 3270 protocols:5250-Specific Callback Routines

usr_bid()	The host requests to begin a bracket. Your application can accept or reject the host's request.
usr_connected()	A request to connect to an LU has succeeded.
usr_disconnected()	A request to connect to an LU has failed, or the connection to an LU has been terminated.
usr_host_event()	A host command or other action affecting the data stream has occurred that may be of interest to your application.
usr_read_complete()	A host transmitted complete chain is available to the application.
usr_request_complete()	User-supplied callback procedure invoked when the GCOM SNA server software receives a response to an LUSTAT, SIGNAL or REQTEST from the host.
usr_sslu_read_complete()	User-supplied callback procedure that is called when a previously initiated SSLU read completes.
usr_sslu_write_complete()	User-supplied callback procedure invoked when a previously initiated SSLU write completes.
usr_write_complete()	A previous application attempt to write a complete chain has successfully completed.

5250-Specific Callback Routines

One of the callback routines applies to only to the 5250 protocol:

usr_log_error_complete()	User-supplied callback procedure invoked when a previously initiated log error request completes.

NPI/SNA API Procedures

This section describes the following GCOM-supplied procedures, which are entry points to the GCOM NPI/SNA API library. Three of these procedures do not apply to the 3270 protocol—the rest of them apply to both 5250 and 3270.

Routine	*5250 Protocol*	*3270 Protocol*
sna_api_change_userid_passwd()	x	x
sna_api_set_debug_level()	x	x
sna_api_verify_userid()	x	x
sna_close()	x	x
sna_get_api_state()	x	x
sna_init_log()	x	x
sna_log_error()	x	
sna_open()	x	x
sna_poll_retry()	x	x
sna_send_request()	x	x
sna_set_buffering_mode()	x	x
sna_set_read_bfr()	x	x
sna_sslu_start_write()	x	x
sna_start_connect_lu()	x	x
sna_start_write()	x	x

sna_api_change_userid_passwd()

Prototype

```
int    sna_api_change_userid_passwd (
                    const char *userid
                    const char *old_password,
                    const char *new_password ) ;
```

Description This routine allowes an application to change the password found in the **.gcompasswd** file. The old password is changed to the new password. The call will fail if the old password has expired.

Parameters

userid	An ASCII string from 2-10 characters. Must correspond to a userid in the **.gcompasswd** file.
old_password	An ASCII character string of at least 8 characters and no more than 10 characters. Must match the password currently in the **.gcompasswd** file for *userid*.
new_password	An ASCII character string of at least 8 characters and no more than 10 characters. If successful, this will be the new password stored in the **.gcompasswd** file for *userid*.

Return Values

PASSWORD_VERIFIED	Password updated
PASSWORD_INVALID	Old password supplied is invalid.
PASSWORD_EXPIRED	Password for *userid* has expired.
USERID_UNKNOWN	Supplied *userid* not in file.

sna_api_set_debug_level()

Prototype

```
void  sna_api_set_debug_level (int level) ;
```

Description

This routine controls the amount of debugging information the API will generate.

Parameters

level The degree of debugging information to generate. The greater the value, the more debugging information generated.

sna_api_verify_userid()

Prototype

```
int   sna_api_verify_userid (
                  const char *userid
                  const char *password ) ;
```

Description

This routine compares the userid and password given to the matching *userid* and *password* found in the **/usr/lib/gcom/.gcompasswd** file. The **.gcompasswd** file must be created with the *Gcom_passwd* utility by the system administrator. Password durations can be set by the system administrator when the *userid* and *password* are entered.

Parameters

userid	An ASCII string from 2-10 characters. Must correspond to a userid in the **.gcompasswd** file.
password	An ASCII character string of at least 8 characters and no more than 10 characters. Must match the password currently in the **.gcompasswd** file for *userid*.

Return Values

PASSWORD_VERIFIED	Supplied *userid* and *password* match.
PASSWORD_INVALID	Password incorrect.
PASSWORD_EXPIRED	Password for *userid* has expired.
USERID_UNKNOWN	Supplied *userid* not in file.

sna_close()

Prototype `void sna_close (int fid) ;`

Description The *sna_close()* procedure closes the specified stream to the SNA driver.

Parameters *fid* File descriptor returned by *sna_open()*.

sna_get_api_state()

Prototype

```
int     sna_get_api_state (int fid, int *mode_ptr, int
        *ff_state_ptr);
```

Description

The *sna_get_api_state()* procedure determines the current transmission mode and NPI/SNA interface state.

Parameters

fid — File descriptor returned by *sna_open()*.

mode_ptr — Pointer to caller's variable where the current transmission mode will be stored. The value stored will be one of the following:

XM_DTR	Data Traffic Reset: wait SDT
XM_FDX	Pre-bind
XM_HDX_FF	Post-SDT

ff_state_ptr — Pointer to caller's variable where the current SNA API state will be stored. The value stored will be one of the following:

SA_CONTENTION	Between brackets
SA_BID	Accepted BID; wait BB
SA_RCV	Receiving Chains
SA_SND	Sending Chains

Return Values

-1	A connection does not exist
0	Call succeeds

sna_init_log()

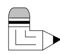
Prototype

```
int sna_init_log (int log_options, char *log_file_name) ;
```

Description

The *sna_init_log()* procedure specifies log options and a log file name. If the log options and name are not specified using this call, they are set by *sna_open()* to the default values described below. Refer to “Logging Options” on page 166 for further logging option information.

The NPI Streams Interface is used by the NPI/SNA API to transfer messages to and from the GCOM SNA server software module. The NPI logging facility is used to log interesting events.



Note: *The* sna_init_log() *procedure should be called only once within your application, just prior to calling sna_open() for the first time.*

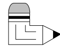
Parameters

log_options	Options controlling what data is logged and where the output is sent. The default log options are: NPI_LOG_FILE, NPI_LOG_STDERR, and NPI_LOG_ERRORS. See “Logging Options” on page 166 for details.
log_file_name	Pointer to a null terminated string containing the name of the file where logged data is to be saved. The default log file name is **/usr/spool/gcom/npi.log**.

Return Values

–1	Call failed. The likely reason is an inability to open the specified file for writing.
0	Call succeeds

sna_log_error()



Note: *This procedure is specific to the 5250 SNA protocol. It does not apply to 3270.*

Prototype

```
poll_t sna_log_error ( int fid, uns16 src,
                       unsigned char *ptr,
                       int count );
```

Description

NPI/SNA maintains an error log on a per-PU basis for FID type 3 sessions. The *sna_log_error()* procedure provides the user with a means to enter data into the log in a non-blocking mode. When the data has been entered into the log, NPI/SNA will notify the user by invoking the *usr_log_error_complete()* user supplied callback. The *sna_log_error()* procedure will fail if a previously initiated write to the log is still pending. The error log is sent to the host under two conditions: when the host sends a REQMS command to NPI/SNA or when the log entry table is full.

Parameters

fid	File descriptor returned by *sna_open()*.
src	5250 System Reference Code.
ptr	Pointer to from zero to five bytes of sense code data.
count	Number of bytes of sense code data.

Return Values

-1	Call failed
bits	Poll bits suitable for the *events* field of a *pollfd* structure to be passed by your application to the UNIX *poll()* system call.

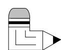
sna_open()

Prototype

```
Old: int sna_open (sna_read_complete_t usr_read_complete
(void)
    sna_write_complete_t          usr_write_complete,
    sna_connected_t               usr_connected,
    sna_host_event_t              usr_host_event,
    sna_bid_t                     usr_bid,
    sna_disconnected_t            usr_disconnected,
    sna_request_complete_t        usr_request_complete,
    usr_sslu_write_complete_t    usr_sslu_write_complete,
    sna_log_error_complete_t      usr_log_error_complete,
    sna_sslu_read_complete_t      usr_sslu_read_complete
) ;
New: int sna_open (char    *hostname,
    sna_write_complete_t          usr_write_complete,
    sna_connected_t               usr_connected,
    sna_host_event_t              usr_host_event,
    sna_bid_t                     usr_bid,
    sna_disconnected_t            usr_disconnected,
    sna_request_complete_t        usr_request_complete,
    usr_sslu_write_complete_t    usr_sslu_write_complete,
    sna_log_error_complete_t      usr_log_error_complete,
    sna_sslu_read_complete_t      usr_sslu_read_complete
) ;
```

Description

The *sna_open()* procedure opens an NPI/SNA stream and returns the associated file descriptor. The file descriptor can then be passed to *sna_start_connect_lu()* to establish a connection to a local LU. Should the LU connection be disconnected, *sna_start_connect()* can be called again using the same file descriptor.



Note: *The NPI/SNA API currently supports a single open file descriptor.*

Parameters

The parameters to *sna_open()* are pointers to the following user-supplied procedures described in Section starting on page 141:

- *usr_read_complete()*
- *usr_write_complete()*
- *usr_connected()*
- *usr_host_event()*
- *usr_bid()*
- *usr_disconnected()*

5250-Specific Parameters

The 5250-specific parameters for *sna_open()* are pointers to the following user-supplied procedures (for SNA 3270 protocol , these parameters must be set to NULL):

- *usr_request_complete()*
- *usr_sslu_write_complete()*
- *usr_log_error_complete()*
- *usr_disconnected()*

Return Values

fid Valid file descriptor

–1 Failure for one of the following reasons:

- A null user callback procedure *ptr* was detected.
- The NPI/SNA API had not previously been initialized and could not be initialized here. (See *sna_init_log()*.)
- The call to open an SNA driver file descriptor failed.
- The NPI BIND call failed.
- The attempt to set non-blocking I/O failed.

sna_poll_retry()

Prototype

```
poll_t sna_poll_retry (int fid, int poll_out_bits) ;
```

Description

The *sna_poll_retry()* procedure tells NPI/SNA to retry a read or write. Since the NPI/SNA API supports a non-blocking I/O interface, it is your application's responsibility to inform NPI/SNA when a read or write action should be retried. Your application should use the UNIX *poll()* system call to determine when a retry is appropriate.

If your application is executing as a result of an NPI/SNA callback procedure invocation, and calls back into an NPI/SNA API procedure that returns poll bits, those poll bits can be ignored because updated poll bits will be returned by *sna_poll_retry()*.

The poll bits indicate which potential NPI/SNA stream events are of interest to the API library (such as data arriving or the stream becoming writeable after having been blocked). These poll bits are intended to be passed by your application to the UNIX *poll()* routine.

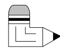
Parameters

fid	File descriptor returned by *sna_open()*.
poll_out_bits	Poll bits returned by the UNIX *poll()* system call in the *revents* field of a *pollfd* structure.

Return Values

-1	*fid* is not a valid file descriptor returned by *sna_open()*, or the file has subsequently been closed. (See *usr_disconnected()*.)
bits	Poll bits suitable for the *events* field of a *pollfd* structure to be passed by your application to the UNIX *poll()* system call.

sna_send_request()



Note: *This procedure is designed for the 5250 SNA protocol. It may be used by LU type applications to send INIT-SELF requests.*

Prototype

```
poll_t sna_send_request ( int             fid,
                          int             request,
                          unsigned char *ptr,
                          int             count,
                          long            flags ) ;
```

Description

The *sna_send_request()* procedure causes one of several types of SNA commands to be sent to the remote SNA host, depending upon the value of the *request* parameter. NPI/SNA invokes the *usr_request_complete()* user-supplied callback procedure when a response has been received from the host.

The *sna_send_request()* procedure imposes a one-at-a-time discipline on requests on a per request type basis. That is, a call to *sna_send_request()* for request type X will fail if a previous request of type X is still pending, but pending requests of other types will not cause such a failure.

The GCOM SNA server software supplies a TH/RH bracket and 0 to 3 bytes of command type for the specified request. The user's data is then appended to the command type bytes. A summary of the request types supported by the SNA server and the amount of data which may accompany them can be found in "Sending SNA Requests" on page 183.

The data passed in by your application is examined by NPI/SNA in those cases it deems appropriate. For example, your application will not be allowed to indicate "power off" in an LUSTAT request. The user will be informed of locally detected errors via the return code of this procedure or via the *usr_request_complete()* user-supplied callback procedure.

Unless the Bracket and/or CDI bits are set in the *flags* parameter, the SNA request will be sent to the host with the RH Bracket and CDI bits cleared. The need to set these bits is SNA command type- and LU type-specific.

The GCOM SNA server software performs error checking on your application's request. For example, the LUSTAT command must be sent with the N_SNA_CDI flags bit set.

The SNAREQ_UNBIND request is designed to allow an application to

terminate an SNA session gracefully. The SNA Server can't tell the difference between an abnormal end of session and a call to *sna_close()*. In both cases, it would send the indication "session end unrecoverable." By sending a SNAREQ_UNBIND with no other data or flags, a normal end of session request will be sent. Any data will treated as sense data and will be sent with the vector 0xFE. Refer to IBM's Formats Manual GA27-3136-12 for more details.

For the format type 1 init-self request, the SNA API assumes that the application has correctly formatted the control vectors stored in the control vectors field of the *init_self_fmt1_rq* field of the *init_self* structure. The SNA multiplexor will insert this field in the appropriate place in the init-self request **but** will not check it for correctness. For further information on the structure of the control vectors refer to IBM's Formats Manual GA27-3136-12.

Parameters

fid File descriptor returned by *sna_open()*.

request Type of SNA command. Legal values include SNAREQ_SIGNAL, SNAREQ_LUSTAT_LULU, SNAREQ_LUSTAT_SSLU, SNAREQ_REQTEST, SNAREQ_UNBIND, SNAREQ_NMVT, SNAREQ_SHUTC, and SNAREQ_INIT_SELF. See "Sending SNA Requests" on page 183 for a complete list.

ptr Pointer to a buffer containing data to accompany the command. For the init-self request this is the address of the structure containing the init_self structure as defined in the **snaapi.h** header file.

count Size of the buffer. In the case of SNAREQ_REQTEST, setting the *count* parameter to zero will cause NPI/SNA to append six bytes of value zero to the internally generated three byte header. For the init-self request this field is ignored API assumes the application has allocated sufficient space to hold the *init_self_t* structure (see **snaapi.h**)

flags Flags used to specify the settings of bits in the Request Header. (See **npiext.h**)

Return Values

If the return value is negative, one of the following return value codes is possible:

- -3 Invalid SNA state
- -7 Invalid data (probably too much or too little)
- -8 Request outstanding
- -9 Unknown Request

Otherwise, a non-negative return value indicates poll bits suitable for the *events* field of a *pollfd* structure to be passed by your application to the UNIX *poll()* system call.

sna_set_buffering_mode()

Prototype

```
void     sna_set_buffering_mode (int fid,
                                 int buffering_mode) ;
```

Description

The *sna_set_buffering_mode()* procedure determines the SNA message buffering scheme between the API and your application. The three methods of message buffering are provided in the description of the *buffering_mode* parameter below.

Parameters:

fid — A file descriptor returned by *sna_open()*.

buffering_mode — One of the following buffering schemes, which are defined in the **snaapi.h** include file:

CHAIN_MODE_CHAIN. Your application wants to receive complete chains only (First, middle and last elements). This default mode does not require the *sna_set_buffering_mode()* routine to be called.

Note: *The buffer that your application provides in a call to* sna_set_read_bfr() *must be as large as the largest chain to be received.*

CHAIN_MODE_BUFFERED_RU. Your application wants the API to buffer messages until either the last-in-chain is received or the buffer is full. Data received from the host can span buffers depending upon the size of the buffer that your application gives to the API and the size of the message being sent by the host.

CHAIN_MODE_RU. Your application wants to receive each RU as it is received from the host.

CHAIN_MODE_SEG. Your application wants the API to receive each RU segment as it is received from the host. If the RU's are not segmented the application will receive each chain element as it is received.

sna_set_read_bfr()

Prototype

```
poll_t      sna_set_read_bfr (int fid, unsigned char
                              *ptr, int length) ;
```

Description

The *sna_set_read_bfr()* procedure initiates a non-blocking read on an NPI/SNA stream. This procedure supports non-blocking delivery of host data to your application. Your application buffer is used to accumulate chain elements arriving from the host . When the complete chain has been assembled, the read buffer is returned to your application via *usr_read_complete().*

The SNA API does not buffer host data for your application. That is, your application should keep a read buffer within the API at all times. The *sna_set_read_bfr()* procedure should be called after *fid* is obtained from *sna_open()* to initially supply the API with a read buffer. When your *usr_read_complete()* callback routine is called, this buffer is passed back to your application. The *sna_set_read_bfr()* procedure should be called before your application's *usr_read_complete()* callback routine exits.

If a chain is received that is larger than the buffer supplied by *sna_set_read_bfr(),* NPI/SNA flushes the data and terminates the LU connection, invoking the *usr_disconnected()* user-supplied callback procedure to inform your application. The pending read is not itself terminated, however, and so another call to *sna_set_read_bfr()* at that point would fail.

Parameters:

fid — A file descriptor returned by *sna_open()*.

ptr — A pointer to a buffer in your application.

length — Length of the specified application buffer pointed to by *ptr*. The buffer must be long enough to hold the largest chain that the host program can send.

Return Values

–1 — *sna_set_read_bfr()* fails because:

- *fid* is not a valid file descriptor returned by *sna_open().*
- *sna_set_read_bfr()* has been called twice in a row without an intervening call of your application's *usr_read_complete()* callback procedure.
- *ptr* is NULL
- *length* is non-positive

bits — Poll bits suitable for the *events* field of a *pollfd* structure to be passed by your application to the UNIX *poll()* system call.

sna_sslu_start_write()

Prototype

```
poll_t sna_sslu_start_write ( int  fid,
                              unsigned char *ptr,
                              int  count, long  flags );
```

Description The *sna_sslu_start_write()* procedure initiates a non-blocking write of data on the SS-LU session. If the call succeeds, the data is sent in a single UNFORMATTED Request/Response Unit (RU) on the NORMAL SS-LU flow. When a (positive or negative) response is received from the host, the *usr_sslu_write_complete()* user-supplied callback procedure is invoked.

Parameters

fid File descriptor returned by *sna_open()*.

ptr Pointer to data.

count Number of bytes data (maximum 256 for 3270 and maximum 128 for 5250).

flags Reserved for future use (must be 0).

Return Values

-1 Call failed

bits Poll bits suitable for the *events* field of a *pollfd* structure to be passed by your application to the UNIX *poll()* system call.

sna_start_connect_lu()

Prototype

```
poll_t   sna_start_connect_lu ( int fid,
                                int pu,
                                int lu,
                                int n_retries,
                                int retry_delay,
                                int lu_type ) ;
```

Description

The *sna_start_connect_lu()* procedure initiates a connection to a local LU that has been activated by the host via an ACTLU command. When the connection to the local LU is established, the *usr_connected()* user-supplied callback procedure is invoked. If the connection attempt fails, the *usr_disconnected()* user-supplied callback procedure is invoked.

If *sna_start_connect_lu()* fails, your application should call *sna_close()* and *sna_open()* to close the file and open a new one, and then retry *sna_start_connect_lu()*. The specified file descriptor can only be connected to a single local LU at a time.

Parameters

fid — A file descriptor returned by *sna_open()*.

pu — The number of a local SNA PU. This corresponds to an SDLC (Synchronous Data Link Control) Secondary Station. The numbers of valid local PUs are specified in a configuration file. A typical value in systems with only a single PU is zero.

lu — The number of a local SNA LU. If the *lu* parameter is zero, NPI/SNA searches for a LU that is not in use.

n_retries — Maximum number of times *sna_start_connect_lu()* attempts to reestablish the connection before invoking the *usr_disconnected()* user-supplied callback procedure. If *retry_delay* is zero, no limit is imposed.

If *retry_delay* is non-zero and *n_retries* is zero, *sna_start_connect_lu()* retries indefinitely every *retry_delay* number of seconds. If *retry_delay* is non-zero and *n_retries* is non-zero, up to *n_retries* retries are attempted before invoking *usr_disconnected()*.

retry_delay — The number of seconds to wait before retrying the lu connect. The delay is implemented via the UNIX *sleep()* system call. If the blocking sleep interferes with your application, the retry interval can be incorporated into your application's poll loop.

lu_type — The LU type, which supports types 0,1,2, 3, 4 and 7.

Return Values

–1 — *sna_start_connect_lu()* fails because:

- The *fid* was not previously opened using *sna_open()*.
- NPI/SNA's attempt to write an NPI CONNECT Request failed.

0 — A connection already exists or a previous attempt to connect is still pending.

bits — Poll bits suitable for the *events* field of a *pollfd* structure to be passed by your application to the UNIX *poll()* system call.

sna_start_write()

Prototype

```
poll_t    sna_start_write (int fid, unsigned char *ptr,
                                       int length, long flags) ;
```

Description

The *sna_start_write()* procedure initiates a non-blocking write on an NPI/SNA stream. If the BIND parameters force NPI/SNA to transmit data using exception response mode, the *usr_write_complete()* user-supplied procedure is invoked when the complete chain has been assembled. Otherwise, *usr_write_complete()* is invoked when a positive response is received from the host.

If the host rejects the transmitted data, *usr_write_complete()* is not called. Instead, the *usr_host_event()* user-supplied procedure is invoked with the *events* parameter set to Event_HostNegRes.

If a disconnect occurs before a host acknowledgment is received, the *usr_disconnected()* user-supplied callback procedure is invoked, implicitly aborting the write. The write is also implicitly aborted when the *usr_host_event()* user-supplied callback procedure is invoked.

If the buffer is larger than the maximum Request/Response Unit (RU) being accepted by the host, the buffer is sent using multiple chain elements. If the host has not sent a BIND, the buffer cannot exceed 256 bytes.

Once the host BIND request has been accepted, the half-duplex flip-flop transmission mode is enforced. That is, the user must set the BB flag in *flags* when the interface is in SA_CONTENTION state, and must clear the BB flag when the interface is in the SA_SND state. It is a protocol violation to call *sna_start_write()* when the interface is in SA_RCV state.

Parameters

fid File descriptor returned by *sna_open()*.

ptr A pointer to a buffer in your application.

length Length of the specified application buffer pointed to by *ptr*.

flags Flags used to specify the settings of bits in the Request Header. The bits that can be set are: BB Indicator, CDI, Code Selection Indicator, Encrypted Data Indicator, Padded Data Indicator, and Compressed Data Indicator. See **npiext.h**.

Return Values

–1 — *sna_start_write()* fails because:

- Connection to the SNA driver over the *fid* does not exist.
- Previous write has not completed.
- *ptr* is Null.
- Invalid settings of bits in *flags*, including incorrect use of BB, EB and/or CDI.
- Non-positive buffer length.
- Attempt to transmit while in Data Traffic Reset mode.
- Attempt to transmit while in SA_RCV or SA_BID state.
- Attempt to transmit a buffer larger than 256 bytes in XM_FDX mode.
- An error occurred while writing to the SNA driver.

bits — Poll bits suitable for the *events* field of a *pollfd* structure to be passed by your application to the UNIX *poll()* system call.

User-Supplied Callback Procedures

This section documents the various callback procedures that must be supplied by your application using the NPI/SNA API. One of these procedures does not apply to the 3270 protocol—the rest of them apply to both 5250 and 3270.

Routine	*5250 Protocol*	*3270 Protocol*
usr_bid()	x	x
usr_connected()	x	x
usr_disconnected()	x	x
usr_host_event()	x	x
usr_log_error_complete()	x	
usr_read_complete()	x	x
usr_request_complete()	x	x
usr_sslu_read_complete()	x	x
usr_sslu_write_complete()	x	x
usr_write_complete()	x	x

usr_bid()

Prototype

```
typedef int (*sna_bid_t) (int fid) ;
```

Description

The *usr_bid()* callback procedure is invoked when the SNA host requests to begin a bracket. This happens in two cases: when an SNA BID command is received and when a chain with a BB is received that was *not* preceded by a BID command. If your application rejects the bracket request, then it is responsible for starting a bracket at a later time. The host, having had its bracket request rejected, waits for the secondary LU to initiate a bracket.

If the application has started a bracket, the host bracket request is rejected by the GCOM SNA server software transparently to your application.

Parameters

fid	File descriptor of a currently connected LU.

Return Values

0	Host bracket request accepted.
non-zero	Host bracket request rejected.

usr_connected()

Prototype

```
typedef void (*sna_connected_t) (int fid) ;
```

Description

The *usr_connected()* callback procedure notifies your application of a successful connection to a local LU, and is invoked when a previous call on *sna_start_connect_lu()* succeeds. That is, the host has activated the LU, and a NOTIFY (indicating power on) has been sent to the host. At this point, an LU-SSCP session exists.

If there is no read pending when *usr_connected()* is invoked, your application should pass a read buffer to NPI/SNA as soon as possible using the *sna_set_read_bfr()* API procedure.

Parameters

fid File descriptor previously passed to *sna_start_connect_lu()*.

usr_disconnected()

Prototype

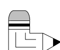
```
typedef void (*sna_disconnected_t) (int fid,
                    int diagnostic, int file_closed);
```

Description

The *usr_disconnected()* callback procedure notifies your application of a failure to establish a connection or the termination of an existing connection. Both the failure to establish a connection and the termination of an existing session are reported to your application using *usr_disconnected()*. The *diagnostic* parameter carries a code indicating the reason for the connection failure (see Appendix starting on page 169). If the disconnect resulted from the termination of the SNA driver (possibly resulting from an administrative action), the file descriptor is closed, and the *file_closed* parameter set to 1.

Your application can attempt to reconnect by calling *sna_open()* to obtain a new file descriptor, and then calling *sna_start_connect_lu()* to attempt to reestablish the connection. However, depending upon the administrative action that resulted in the file closure, either of the above two procedure calls can fail. In particular, *sna_open()* is not likely to succeed, because NPI/SNA invokes *usr_disconnected()* only after an internal attempt to reinstate the file descriptor fails. Consequently, your application may have to set up a retry loop for the *sna_open()* call.

If *file_closed* is zero, your application can immediately call *sna_start_connect_lu()* to attempt to re-establish the connection without first calling *sna_open()*.



Note: *The LU connection is terminated when an ACTLU-ERP is received from the host. However,* usr_host_event() *is called instead of* usr_disconnected() *in this case.*

Parameters

fid	File descriptor previously passed to *sna_start_connect_lu()*.
diagnostic	Diagnostic code indicating reason for disconnect. See Appendix starting on page 169 for details.
file_closed	One if *fid* is no longer valid, zero indicates that your application can immediately call *sna_start_connect_lu()* to attempt to re-establish the connection without first calling *sna_open()*.

usr_host_event()

Prototype

```
typedef unsigned long (*sna_host_event_t)(int   fid,
                                          int   mode,
                                          int   state,
                                          int   event,
                                      unsigned char *ptr,
                                          int   count ) ;
```

Description

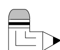
The *usr_host_event()* callback procedure notifies your application of special events resulting from the host's actions (See "Host Events" on page 164). Most of the events directly result from an accepted host message, such as BIND. Other events, such as Event_Shutdown*,* result from a sequence of events starting with a host SHUTD command followed by a transition to SA_CONTENTION state, and a SHUTC being returned to the host.

The one abnormal event is Event_HOST_aborted_BB. This event is caused by the host sending a BB request that is accepted by your application followed by an error condition that occurs before the chain containing the BB is successfully processed.

The *mode* parameter contains the transmission mode that results from the event. The *state* parameter is the interface state.

A host event causes a pending write to be terminated. No other notification of this termination is given to your application. A pending read is not affected.

The return type is unsigned long so that your application can indicate that it wants to reject the SNA command (primarily BIND), which precipitated the invocation of *usr_host_event().*



Note: *The* usr_host_event() *procedure is called only if a connection exists between your application and a local LU. See* usr_connected() *and* usr_disconnected *starting on page 145.*

BIND processing

The GCOM, INC. Sna Server will verify bytes 0 - 14 and byte 26 (see "" starting on page 183). The SNA Server expects the application to examine bytes 15-25 and accept/reject the bind based on the values in those bytes. The sense code returned for a BIND rejection will take the format xxyyzzzz where xx = the byte in error; yy = the bit of the byte in error (set to 0 if whole byte is in error) and zzzz = the sense code for bind parameters errors (0821).

The SNA Server will also examine byte 26. If byte 26 is 0 the SNA Server will pass the bind to the application for it to verify. If the value of byte 26 indicates the primary lu (PLU) is requesting this session use enchiphered data, the SNA Server will verify the configuration parameters allow this station to accept enciphered data. The bind will then be checked for sufficient length to include the session cryptography key and sent to the application. The application must verify the key and change the value as appropriate. The SNA Server will then return value of the session seed to the PLU.

Host Emulation

The GCOM SNA server can be configured to act as a host. With this configuration, the usr_host_event callback can be invoked with an event_CHAIN host.

The host event_CHAIN indicates that the SNA code has received an inbound message from the host while in CHAIN_CONTENTION state. It has queued the message and sent this event to the application because the host is the contention loser and the application is the contention winner.

When an application receives this host event it has two choices:

1. If the application has started to send, or is about to send, it should return a non-zero return code. (The value does not have to be a specific sense code as it will never be returned to the host).

   A non-zero return code causes the interface to transition to send state. The data received from the host will be queued until the application has finished sending and a) returns the interface to CHAIN_CONTENTION state or b) sends a change direction indicator (CDI) to the host. The SNA code will always queue outbound host messages when the host loses the contention race.
2. A zero return code to this host event will place the interface into receive state. The data received from the host will be sent to the application. The interface will remain in receive state until the host sends a “last in chain” message. The interface will transition to CHAIN_CONTENTION state if CDI is not set and to SEND state if CDI was received from the host.

Cryptography sessions

The GCOM, INC. SNA Server accepts binds with cryptography sessions requested. (See BIND processing). The GCOM, INC. SNA Server expects the application to do the encryption/decryption of the data.

The CRV (cryptography request verification) will be length checked by the SNA Server and passed to the application for verification.

The application must indicate to the API when an ru is enciphered by setting the N_SNA_ENCRYPT flag (see **npiext.h**) in the flags field of a *sna_start_write()* request. If the ru requires that the padded data indicator (PDI) bit be set in the request header (RH) the application must indicate this by setting the N_SNA_PAD (see **npiext.h**) bit in the flags field of a *sna_start_write()* request.

It is highly recommended when a session is sending encrypted data the application send only request unit (RU) size writes to the SNA API. This allows the SNA Server to correctly set the PDI bit in the RH.

Compressed data sessions

The GCOM, INC. Server allows compressed data sessions. It expects the application to do the compression/decompression of the data. The application indicates to the SNA API that the data is compressed by setting the N_SNA_COMPRES flag (see **npiext.h**) in the flags parameter of the *sna_start_write()* function call.

Parameters	*fid*	File descriptor of a currently connected LU.	
	mode	The current transmission mode, which can be any of the following:	
		XM_DTR	Data Traffic Reset: wait SDT
		XM_FDX	Pre bind
		XM_HDX_FF	Post SDT
	state	The current SNA interface state, which can be any of the following:	
		SA_CONTENTION	Between brackets
		SA_BID	Accepted BID; wait BB
		SA_RCV	Receiving Chains
		SA_SND	Sending Chains
	event	The host event. See "Host Events" on page 164 for details.	
	ptr	Pointer to a buffer containing data accompanying the received SNA command which precipitated the call. The byte contained in *ptr[0]* occupied byte 0 of the Request/Response Unit (RU).	
	count	Number of bytes in the buffer.	
Return Values	0	Host event is accepted.	
	non-zero	The host event is rejected. In the case of a BIND, the value returned is the offset from *ptr[0]* of the first byte of the BIND data to which the application objects. (There is no way for the application to indicate an objection to byte 0.) In the case of an event other than a BIND, a non-zero return value is interpreted by NPI/SNA as a sense code to be sent to the host in a negative response.	

usr_log_error_complete()

Note: *This procedure is specific to the 5250 SNA protocol. It does not apply to 3270.*

Prototype

```
typedef void (*sna_log_error_complete_t) ( int fid,
                                           int status )
                                           long sense );
```

Description

The *usr_log_error_complete()* callback procedure notifies your application of the success or failure of a previously initiated write to the error log (see “sna_log_error()” on page 128).

Parameters

fid File descriptor previously passed to *sna_log_error()*.

status

0 Write succeeded.

-1 Request failed locally. The *sense* parameter is returned locally.

-2 Negative response from host. The *sense* parameter is returned from host.

sense If *status* returned -1, this value is returned locally. If *status* returned -2, this value is returned from the host. It can have the following values:

-3 Invalid SNA state

-4 Append error

-5 SNA_LOG_FULL

-6 log table send failed

-7 Invalid data (probably too much or too little)

-8 Request outstanding

-9 Unknown Request

usr_read_complete()

Prototype

```
typedef int  (*sna_read_complete_t) (int   fid,
                 unsigned char *bfr_ptr, int chain_length,
                 long flags) ;
```

Description

The *usr_read_complete()* callback procedure passes received data (in the form of a chain) to your application. Chain elements received from the host are accumulated until a chain element containing the end chain indicator is received.

Your application can call *sna_start_write()* before returning from the *usr_read_complete()* procedure. If so, the received host chain is automatically accepted with a positive response before starting the write, and your application's returned sense code is ignored.

Before exiting the *usr_read_complete()* procedure, your application should process the contents of the buffer, and then pass the buffer (or a replacement) back to NPI/SNA using *sna_set_read_bfr()*.

The *flags* parameter contains a set of bits indicating BB, EB, change direction, and code selection indicator. Look at **npiext.h** for details.

Parameters

fid	File descriptor previously passed to *sna_set_read_bfr()*.
bfr_ptr	Pointer to a buffer containing data received from the host.
chain_length	Number of bytes in the buffer.
flags	Flags indicating the presence or absence of code selection indicator, BB, EB and CDI bits in the received chain. See "Data Transfer Flags" on page 167 for details.

Return Value

0	Chain accepted
non-zero	Sense code to be sent to host in a negative response

usr_request_complete()

Note: *This procedure is specific to the 5250 SNA protocol. It does not apply to 3270.*

Prototype

```
typedef void (*sna_request_complete_t) ( int fid,
                                         int request,
                                         int status
                                         long sense ) ;
```

Description

The *usr_request_complete()* callback procedure notifies your application of the success or failure of a previously initiated SNA request (see *sna_send_request).*

Parameters

fid File descriptor previously passed to *sna_send_request()*.

request Type of request. SNAREQ_SIGNAL, SNAREQ_LUSTAT_LULU, SNAREQ_LUSTAT_SSLU or SNAREQ_REQTEST.

status

- 0 Request succeeded.
- -1 Request failed locally. The *sense* parameter is returned locally.
- -2 Negative response from host. The *sense* parameter is returned from host.

sense If *status* returned -1, this value is returned locally. If *status* returned -2, this value is returned from the host. It can have the following values:

- -3 Invalid SNA state
- -7 Invalid data (probably too much or too little)
- -8 Request outstanding
- -9 Unknown Request

usr_sslu_read_complete()

Prototype

```
typedef unsigned long (*sna_sslu_read_complete_t)
                      ( int            fid,
                        unsigned char *ptr,
                        int            count,
                        long           flags ) ;
```

Description

The *usr_sslu_read_complete()* callback procedure is used to pass SS-LU session data to your application. The data passed is that received from the host in a single Request/Response Unit (RU). Once your application returns, the API buffer containing the SS-LU data may be overwritten by the API.

Your application's read buffer is not affected by the deliver of SS-LU data. That is, your application need not call *sna_set_read_bfr()* from *usr_sslu_read_complete()*.

Note: *The delivery of data received on the SS-LU session interacts with the delivery of data received on the LU-LU session. As explained in the description of* sna_set_read_bfr(), *your application must have a read buffer outstanding to the API to receive LU-LU data. Otherwise, the API will not read streams messages until your application calls* sna_set_read_bfr().

Obviously, if the API is not reading streams messages, SS-LU data will not be delivered to your application. However, there exist protocol exchanges between the API and SNA MUX where the API reads for streams messages even though a read buffer does not exist. These protocol exchanges only occur when LU-LU data cannot be in transit from the SNA MUX to the API. However, SS_LU data can be in transit during one of these protocol exchanges.

In other words, the lack of a read buffer does not guarantee that your application's usr_sslu_read_complete() *procedure will not be invoked.*

If the address to a *usr_sslu_read_complete()* routine is not provided in a call to *sna_open()* in the *usr_sslu_read_complete* parameter (for example, it might be set to NULL), then NPI/SNA will silently discard the data and return a positive response to the host.

Parameters

fid File descriptor previously passed to *sna_set_read_bfr()*.

ptr Pointer to a buffer containing received SS-LU data.

	count	Number of bytes in the buffer.
	flags	Reserved for future use.
Return Values	0	Request/Response Unit (RU) accepted.
	non-zero	Sense code to be sent to the host in a negative response.

usr_sslu_write_complete()

Prototype

```
typedef void (*sna_sslu_write_complete_t) (int fid,
                                            int status
                                            long sense ) ;
```

Description

The *usr_sslu_write_complete()* callback procedure notifies your application of the success or failure of transmission of a Request/Response Unit (RU) to the host on the SS-LU session (see “sna_sslu_start_write()” on page 136 for details).

Parameters

fid File descriptor previously passed to *sna_sslu_start_write()*.

status

- 0 Request succeeded.
- -1 Request failed locally. The *sense* parameter is returned locally.
- -2 Negative response from host. The *sense* parameter is returned from host.

sense If *status* returned -1, this value is returned locally. If *status* returned -2, this value is returned from the host. It can have the following values:

- -3 Invalid SNA state
- -7 Invalid data (probably too much or too little)
- -8 Request outstanding
- -9 Unknown Request

usr_write_complete()

Prototype

```
typedef void (*sna_write_complete_t) (int fid,
                                      int type) ;
```

Description

The *usr_write_complete()* callback procedure notifies your application of the successful transmission of a chain to the host ir the intermediate completion of a write that spans several buffers.. The specific behavior is controlled by the the setting of the N_SNA_More_Chain flag in the flags field of the *sna_start_write()*.

If the N_SNA_More_Chain flag was set in the flags field of *sna_start_write()*, then a call to *usr_write_complete()* with the *type* parameter set to Write_Intermediate indicates that the NPI/SNA has completed assembling the data in the user buffer for transmission to the host and is ready to process another buffer. This does not indicate that the data has been successfully transmitted to the host. Successful transmission of the data is only indicated when the *type* parameter is set to Write_Complete.

If the BIND requires that the secondary LU send data in definite response mode, the host sends a positive response to each correctly received chain. NPI/SNA invokes *usr_write_complete()* when the positive response is received.

If the BIND requires that the secondary LU send data in exception response mode, the host does *not* send a positive response to correctly received data. Therefore, when sending data in exception response mode, NPI/SNA invokes *usr_write_complete()* when the complete chain has been assembled.

If the BIND allows the secondary LU to send data in either definite or exception response mode, NPI/SNA defaults to definite response.

Regardless of response mode, the host can send a negative response when an error is detected in a received chain. NPI/SNA invokes the *usr_host_event()* callback procedure with the *events* parameter set to Event_HostNegRes. If the host rejects a chain that has been sent by NPI/SNA using definite response mode, *usr_write_complete()* is not called.

When NPI/SNA is sending data in exception response mode, it is possible for a negative response from the host to arrive after *usr_write_complete()* has been called. Therefore, when NPI/SNA calls this procedure, delivery is not guaranteed in this case.

Parameters	*fid*	File descriptor previously passed to *sna_start_write()*.
	type	Indicates whether this is either an intermediate or full write completion.

NPI/SNA User-Accessible Defines

This section lists the defines accessible by your application from the **snaapi.h** file.

SNA Disconnect Diagnostic Codes

Figure 48

Disconnects Resulting from a Badly Formatted Call Request

Disc_short_call_req	Call format error
Disc_long_address	Too many digits in called address
Disc_short_address	Too few digits in called address
Disc_format_error	Malformed called address
Disc_lu_type_error	Unsupported LU type in address
Disc_invalid_LU	Range error

Calls Rejected Resulting From the State of the PU/LU

Disc_not_DLC_connected	Data link control not setup
Disc_not_PU_activated	Valid ACTPU not received
Disc_not_LU_activated	Specified LU not activated
Disc_LU_not_available	When LU = 0 requested
Disc_LU_in_use	LU being used

Disconnects Resulting From Administrative Actions

Disc_sna_mux_terminated	SNA driver de-configured
Disc_sna_mux_turned_off	Administrator turned driver off
Disc_sna_mux_restarted	Administrator request

Disconnects Resulting From Host Actions

Disc_ACTPU_COLD	Host ACTPU cold start
Disc_ACTLU_COLD	Host ACTLU cold start
Disc_DACTLU	Host DACTLU
Disc_DACTPU	Host DACTPU diagnostic

Disconnects Resulting From Data Link Events

Disc_DLC_error	Possibly messages lost
Disc_DLC_disconnected	DLC station disconnected

Disconnects Resulting From the Receipt of Unsupported Rsystem Tokens

Your application is likely using an NPI feature that is not supported within the GCOM SNA server software.

Disc_interrupt_request	Currently unsupported request
Disc_interrupt_confirm	Currently unsupported request
Disc_request_res	Currently unsupported request

Disconnects Resulting From State Errors

Disc_lu_not_known	Received request on LU not in use
Disc_state_error	Received unexpected message from NPI
Disc_state_error_1	Event in unexpected state
Disc_state_error_2	Event in pre-call state
Disc_state_error_3	Event in call setup state
Disc_state_error_4	Event in data transfer state
Disc_state_error_5	Event in termination state

Disconnects Resulting From Client/Application/Stream Errors

Disc_client_error	Unrecoverable client error
Disc_upstream_overrun	Chain too large for application bfr
Disc_hangup	From *getmsg()*
Disc_stream_unusable	*getmsg()/putmsg()* failure

Disconnect Diagnostics Interpreted from NPI Reason Codes

Disc_PU_unknown	NPI could not route call . Likely configuration error
Disc_call_retry_limit	NPI failed to send call request
Disc_call_perm_error	N_REJ_NSPA_UNREACH_P
Disc_rej_unspecified	N_REJUNSPECIFIED
Disc_connect_reject	NPI connection failure

Disconnect Diagnostics Produced by Other Errors

Disc_rej_lu_type	Incompatible LU type requested for requested address in START_CONNECT_REQ
Disc_rej_pu_sess	PU_SESSION_REQ not allowed
Disc_rej_bind	A BIND was rejected for this LU
Disc_rej_table_ovflw	Correlation table overflowed
Disc_conn_time_exp	Connection request placed prior to an LU's activation failed due to timeout

Transmission Modes

XM_SSCP	SSCP to LU session
XM_DTR	Data Traffic Reset: wait SDT
XM_FDX	Pre bind
XM_HDX_FF	Post SDT

SNA/API State Values

SA_CONTENTION	Between brackets
SA_BID	Accepted BID; wait BB
SA_RCV	Receiving Chains
SA_SND	Sending Chains
SA_CHAIN_CONT	Chain contention state: between chains and bound
SA_FDX	Full-duplex state: session bound, full-duplex mode

Host Events

Event_HOST_aborted_BB	BID or host chain with BB canceled
Event_ServiceInterrupt	Reserved: DLC (Data Link Control) outage, ACTPU-warm
Event_ACTLU	ACTLU warm start
Event_CLEAR	Valid host CLEAR
Event_SDT	Valid host SDT
Event_BIND	Valid host BIND
Event_UNBIND	Valid host UNBIND
Event_SIG	Take back CDI
Event_Shutdown	SHUTD/SHUTC sequence
Event_CANCEL_EB	CANCEL carries EB
Event_CANCEL_CDI	CANCEL carries CDI
Event_CHAIN	Received an inbound message from the host while in CHAIN_CONTENTION state. Message queued and sent to the application. (Host is contention loser and application is contention winner.)
Event_CHASE_CDI	CHASE carries CDI
Event_CHASE_EB	CHASE carries EB
Event_HostNegRes	Received negative response to FMD data
Event_SetBid_State	Forces the interface state to SA-BID.
Event_NotifyRcvd	Notify request received from host.
Event_CRV	Verification of crypto request.
Event_BIND_REJ	SNA rejected host’s BIND
Event_RTR	Received Ready To Read request

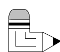
Event_BIS	Received Bracket Initiation Stop request
Event_SBI	Received Stop Bracket Initiation request
Event_QEC	Received Quiescent End Chain request
Event_QC	Received Quiescent Chain request
Event_RELQ	Received Release Quiescent Chain request
Event_NSPE	Received NS Procedure Error
Event_CHASE	Received CHASE command
Event_LUSTAT	Received LUSTAT
Event_LUSTAT_BB	Received LUSTAT with Begin Bracket set
Event_LUSTAT_EB	Received LUSTAT with End Bracket set
Event_LUSTAT_CDI	Received LUSTAT with Change Direction Indicator set
Event_BIND_REJ	Server rejected a BIND for this LU; the BIND being rejected is attached to this event

Logging Options

NPI/SNA can log error messages to a file, print them out on *stderr,* or both or neither. The behavior is controlled by the following defines, which are located in **npiapi.h**:

NPI_LOG_FILE	Log to file
NPI_LOG_STDERR	Log to *stderr*
NPI_LOG_RX_PROTOS	Log received M_PROTOS
NPI_LOG_TX_PROTOS	Log transmitted M_PROTOS
NPI_LOG_ERRORS	Log UNIX errors
NPI_LOG_RX_DATA	Log received M_DATA
NPI_LOG_TX_DATA	Log transmitted M_DATA
NPI_LOG_SIGNALS	Log NPI generated UNIX signals
NPI_LOG_CONINDS	Log NPI connection indications
NPI_LOG_OPTIONS	Log initialization options
NPI_LOG_DEFAULT	NPI_LOG_FILE | NPI_LOG_STDERR | NPI_LOG_ERRORS

These can be "OR-ed" together in your code for combinations. These options are passed to the *npi_init()* routine by NPI/SNA. If your application does not specify log options by calling *sna_init_log(),* NPI_LOG_DEFAULT is used.



Note: *A zero means no error reporting at all.*

Data Transfer Flags

The data transfer flags below are defined in **npiext.h**:

N_SNA_ALT_CODE	SNA code selection indicator
N_SNA_CDI	SNA CDI
N_SNA_BB	SNA BB
N_SNA_EB	SNA EB
N_SNA_ENCRYPT	Encrypted data indicator (RH byte 2 bit 5)
N_SNA_PAD	Padded data indicator (RH byte 2 bit 6)
N_SNA_COMPRES	Compressed data indicator (RH byte 1 bit 1)

UNIX Errno Codes

The following are UNIX *errno* codes are returned by the NPI/SNA API procedures:

EPIPE — LU Session not established, or *sna_set_read_bfr()* called before *sna_open()*.

EPROTO — One of the following:

- *sna_open()* detected a null callback procedure *ptr*.
- *sna_set_read_bfr()* called when previous read still pending.
- *sna_start_write()* called when previous write still pending.

ENOEXEC — At *sna_open()*, NPI/SNA had not been initialized, and the internal call on *sna_init_log()* with default initialization parameters failed.

EXDEV — At *sna_open()*, the NPI BIND request failed.

EBADF — *sna_start_connect-lu()* was called using a file descriptor not obtained from *sna_open()*.

EEXIST — *sna_start_connect_lu()* was called when an LU connection already existed.

E2BIG — Invalid flags were set in a *sna_start_write()* call.

EACCESS — A call was made on *sna_start_write()* while in the “Data Traffic Reset” state.

EFAULT — While in XM_FDX mode, a call on *sna_start_write()* was made with a buffer larger than 256 bytes, or a call on *sna_start_write()* was made while in SA_RCV state.

ENOTBLK — *sna_start_write()* was called with BB in the *flags* parameter cleared to zero when in SA_CONTENTION state.

EBUSY — *sna_start_write()* was called with BB bit in the *flags* parameter set to one when in SA_SND state.

Appendix A

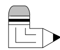
Error Return Codes

Figure 49 he **/usr/include/sys/errno.h** file contains defines for error codes used by the frame relay API routines. They are useful to programmers for debugging purposes. You can also check for certain error return codes in your application code and take appropriate action, such as:

```
If (return_code == ENOSTR) ...
```

GCOM recommends including this in your application using the following line of C code:

```
#include <errno.h>
```



Note: *UNIX Errno codes are listed on page 168 because they are part of the NPI/SNA user-accessible defines.*

Normal Error Codes

EPERM	1	Not super-user
ENOENT	2	No such file or directory
ESRCH	3	No such process
EINTR	4	Interrupted system call
EIO	5	I/O error
ENXIO	6	No such device or address

E2BIG	7	Argument list too long
ENOEXEC	8	Executive format error
EBADF	9	Bad file number
ECHILD	10	No children processes
EAGAIN	11	No more processes
ENOMEM	12	Not enough core
EACCES	13	Permission denied
EFAULT	14	Bad address
ENOTBLK	15	Block device required
EBUSY	16	Mount device busy
EEXIST	17	File exists
EXDEV	18	Cross-device link
ENODEV	19	No such device
ENOTDIR	20	Not a directory
EISDIR	21	Is a directory
EINVAL	22	Invalid argument
ENFILE	23	File table overflow
EMFILE	24	Too many open files
ENOTTY	25	Not a typewriter
ETXTBSY	26	Text file busy
EFBIG	27	File too large
ENOSPC	28	No space left on device
ESPIPE	29	Illegal seek
EROFS	30	Read-only file system

EMLINK	31	Too many links
EPIPE	32	Broken pipe
EDOM	33	Math argument out of domain of function
ERANGE	34	Math result not representable
ENOMSG	35	No message of desired type
EIDRM	36	Identifier removed
ECHRNG	37	Channel number out of range
EL2NSYNC	38	Level 2 not synchronized
EL3HLT	39	Level 3 halted
EL3RST	40	Level 3 reset
ELNRNG	41	Link number out of range
EUNATCH	42	Protocol driver not attached
ENOCSI	43	No CSI structure available
EL2HLT	44	Level 2 halted
EDEADLK	45	Deadlock condition
ENOLCK	46	No record locks available

Convergent Error Returns

EBADE	50	Invalid exchange
EBADR	51	Invalid request descriptor
EXFULL	52	Exchange full
ENOANO	53	No A-node
EBADRQC	54	Invalid request code
EBADSLT	55	Invalid slot
EDEADLOCK	56	File locking deadlock error
EBFONT	57	Bad font file format

Stream Problems

ENOSTR	60	Device not a stream
ENODATA	61	No data (for no delay I/O)
ETIME	62	Timer expired
ENOSR	63	Out of streams resources
ENONET	64	Machine is not on the network
ENOPKG	65	Package not installed
EREMOTE	66	The object is remote
ENOLINK	67	The link has been severed
EADV	68	Advertise error
ESRMNT	69	*srmount* error
ECOMM	70	Communication error on send
EPROTO	71	Protocol error
EMULTIHOP	74	Multi-hop attempted
ELBIN	75	I-node is remote
EDOTDOT	76	Cross mount point
EBADMSG	77	Trying to read unreadable message
ENAMETOOLONG	78	
ENOTUNIQ	80	Given log name not unique
EBADFD	81	f.d. invalid for this operation
EREMCHG	82	Remote address changed

Shared Library Problems

ELIBACC	83	Cannot access a needed shared library
ELIBBAD	84	Accessing a corrupted shared library
ELIBSCN	85	*.lib* section in **a.out** corrupted.
ELIBMAX	86	Attempting to link in too many libraries
ELIBEXEC	87	Attempting to exec a shared library
ENOSYS	89	

Socket Errors

Socket errors are calculated as an offset from TCPERR, which is defined as 90 in **errno.h** at this writing.

Non-Blocking And Interrupt I/O Errors

EWOULDBLOCK	(TCPERR+0)	Operation would block
EINPROGRESS	(TCPERR+1)	Operation now in progress
EALREADY	(TCPERR+2)	Operation already in progress

IPC/Network Software Argument Errors

ENOTSOCK	(TCPERR+3)	Socket operation on non-socket
EDESTADDRREQ	(TCPERR+4)	Destination address required
EMSGSIZE	(TCPERR+5)	Message too long
EPROTOTYPE	(TCPERR+6)	Protocol wrong type for socket
EPROTONOSUPPORT	(TCPERR+7)	Protocol not supported
ESOCKTNOSUPPORT	(TCPERR+8)	Socket type not supported
EOPNOTSUPP	(TCPERR+9)	Operation not supported on socket
EPFNOSUPPORT	(TCPERR+10)	Protocol family not supported
EAFNOSUPPORT	(TCPERR+11)	Address family not supported by protocol family
EADDRINUSE	(TCPERR+12)	Address already in use
EADDRNOTAVAIL	(TCPERR+13)	Cannot assign requested address

Operational Errors

ENETDOWN	(TCPERR+14)	Network is down
ENETUNREACH	(TCPERR+15)	Network is unreachable
ENETRESET	(TCPERR+16)	Network dropped connection on reset
ECONNABORTED	(TCPERR+17)	Software caused connection abort
ECONNRESET	(TCPERR+18)	Connection reset by peer
ENOBUFS ENOSR	(TCPERR+19)	No buffer space available
EISCONN	(TCPERR+20)	Socket is already connected
ENOTCONN	(TCPERR+21)	Socket is not connected
ESHUTDOWN	(TCPERR+22)	Cannot send after socket shutdown
ETOOMANYREFS	(TCPERR+23)	Too many references: cannot splice
ETIMEDOUT	(TCPERR+24)	Connection timed out
ECONNREFUSED	(TCPERR+25)	Connection refused
EHOSTDOWN	(TCPERR+26)	Host is down
EHOSTUNREACH	(TCPERR+27)	No route to host
ENOPROTOOPT	(TCPERR+28)	Protocol not available

XENIX Error Numbers

EUCLEAN	135	Structure needs cleaning
ENOTNAM	137	Not a name file
ENAVAIL	138	Not available
EISNAM	139	Is a name file
EREMOTEIO	140	Remote I/O error
EINIT	141	Reserved for future
EREMDEV	142	Error
ENOTEMPTY	145	
ELOOP	150	Too many symbolic links in path

Appendix B

BIND Check Tables

The BIND check table is different for the 3270 and 5250 protocol.

3270 BIND Check Table

Table 10 for LU type 2 shows the bytes of the BIND parameter and whether or not they are ignored.

Table 10 3270 BIND Check Table *(1 of 2)*

Byte	*Bits*	*Check*	*Reject if:*	*Parameter Description*
1	0-3	C	not X'0'	BIND type and format
	4-7	C	not X'1'	
2		C	not X'03'	FM profile
3		C	not X'03'	TS profile
4	0	Ignored		Primary LU protocols
	1	C	B'1'	
	2, 3	C	B'00'	
	4, 5	Ignored		
	6	Ignored		
	7	C	B'0'	
5	0	Ignored		Secondary LU protocols
	1	Ignored		
	2, 3	C	B'00'	

Table 10 3270 BIND Check Table *(2 of 2)*

Byte	*Bits*	*Check*	*Reject if:*	*Parameter Description*
	4-7	Ignored		
6	0	Ignored		Common protocols
	1	C	B'1'	
	2	C	B'0'	
	3	C	B'0'	
	4	C	*	
	5-7	Ignored		
7	0, 1	C	not B'10'	Common protocols
	2	C	B'1'	
	3	C	B'1'	
	4-7	Ignored		
8		Ignored		Not applicable to LU 2
9	0, 1	Ignored		Pacing not used by LU 2
	2-7	Ignored		
10		C		Maximum RU size sent by secondary
11		C	too large	Maximum RU size sent by primary
12, 13		Ignored		
14		C	not correct device	Device type of LU
15-19		Ignored		
20-23		C		Device screen size—checked if byte 24 is X'E' or X'F'
24		C		Device screen size
25		Ignored		
26		C	not X'00'	Bytes 26-35 are reserved for encrypt/decrypt function

5250 BIND Check Table

Table 11 shows the format of the BIND command and response for LU 7 display stations.

Table 11 5250 BIND Check Table *(1 of 3)*

Byte	*Bits*	*Field*	*Reject if:*	*Parameter Description*
0	0-7	Request code	not X'31'	BIND command
1	0-3	Format	not B'0000'	Format 0[a]
	4-7	Type	not B'0000'	Cold (negotiable)[a]
2	0-7	FM profile	not X'07'	FM profile 7
3	0-7	TS profile	not X'07'	TS profile 7
4		FM Usage (primary)		
	0	Chaining use	not B'1'	Multiple RU chains allowed
	1	Request mode	not B'0'	Immediate Request mode
	2-3	Chain response	not B'11'	Definite or exception response
	4-5	Ignored	Not checked	
	6	Compression	not B'0'	No compression
	7	Send end bracket	not B'0'	Will not send EB
5		FM Usage (secondary)		
	0	Chaining use	not B'1'	Multiple RU chains allowed
	1	Request mode	not B'0'	Immediate Request mode
	2-3	Chain response	not B'11'	Definite or exception response
	4-5	Ignored	Not checked	
	6	Compression	not B'0'	No compression
	7	Send end bracket	not B'0'	Will not send EB

Table 11 5250 BIND Check Table *(2 of 3)*

Byte	*Bits*	*Field*	*Reject if:*	*Parameter Description*
6		FM Usage (common)		
	0	Ignored	Not checked	
	1	FM headers	not B'0'	FM headers not allowed
	2	Bracket reset	not B'0'	Reset state is in-bracket
	3	Bracket termination rule	not B'1'	Rule 1 (conditional)
	4	Code set	not B'0'	No alternate code set
	5	Sequence number	not B'0'	Not available
	6	Bracket initiation stop (BIS)	not B'0'	BIS not sent
	7	Ignored	Not checked	
7	0-1	Normal Flow mode	not B'00' or not B'10'	Full-duplex for pass-through data stream
			not B'10'	Half-duplex flip-flop (HDX-FF)
	2	Recovery responsibility	not B'0'	Contention loser (primary)
	3	Brackets contention winner or loser	not B'0'	Secondary is winner
	4-6	Ignored	not B'000'	
	7	HDX-FF reset state	not B'1'	BIND sender reset to send state
8	0	Staging indicator (secondary)	not B'x'	Same as BIND request[b]
	1	Ignored	not B'x'	
	2-7	Secondary send pacing count	not B'xxxxxx'	Same as BIND request[b]
9	0-1	Reserved	not B'00'	See footnote a.
	2-7	Secondary receive pacing count		No pacing count[a]
10	0-7	Max. RU size sent by secondary on normal flow[c]	not X'85 or not X'86	256 bytes (for IBM 5294 Emulation mode).[a] 512 bytes (for IBM 5394 mode) if BIND request byte 10 Š X'86'.
11	0-7	Maximum RU size sent by primary on normal flow[c]	not X'85 or not X'86	If BIND byte 11 is larger than BIND byte 10, the value used for maximum RU size sent by primary is set to the value of BIND byte 10

Table 11 5250 BIND Check Table *(3 of 3)*

Byte	*Bits*	*Field*	*Reject if:*	*Parameter Description*
12	0	Staging indicator (primary)	not B'0'	If bit 0 of BIND byte 12 is not set, then the primary send pacing count is set to the value of BIND byte bits 2-7. If bit 0 of BIND byte 12 is set, then the primary send pacing count is set to the value of bits 2-7 of BIND byte 9.
	1	Ignored	Not checked	
13	0-1	Ignored	Not checked	
	2-7	Primary receive pacing count		
		PS Profile (14-25)		
14	0	PS usage format	not B'0'	Basic format
	1-7	LU type	not B'0000111'	LU 7 (display station)
15-23	0-7	Ignored	Not checked	
24	0	Ignored	Not checked	
	1-7	Display screen size	not B'0000010'	1920 (24 x 80)
25	0-6	Ignored	Not checked	
	7	IPL (download)	B'1'	No download

a. Negative response generated if invalid parameter specified in this field of BIND request.
b. For detailed description, see IBM's *SNA Format and Protocol Reference Manual: Architectural Logic.*
c. Support for these parameters must also be configured at the host system.

Appendix C

Sending SNA Requests

This appendix documents some of the features the API provides for sending SNA requests to the host.

Table 12 List of supported request types

Request type	*Request type define*	*# bytes generated by SNA Server*	*max. bytes that can accompany request*
SIGNAL	SNAREQ_SIGNAL	1	4
LUSTAT (on LU-LU session)	SNAREQ_LUSTAT_LULU	1	5
LUSTAT (on SSCP-LU session)	SNAREQ_LUSTAT_SSLU	1	5
REQTEST	SNAREQ_REQTEST	3	5
INITSELF	SNAREQ_INIT_SELF	3	256
QEC	SNAREQ_QEC	1	0
QC	SNAREQ_QC	1	0
SBI	SNAREQ_SBI	1	0
BIS	SNAREQ_BIS	1	0
RTR	SNAREQ_RTR	1	0
RQR	SNAREQ_RQR	1	0
RELQ	SNAREQ_RELQ	1	0
CHASE	SNAREQ_CHASE	1	0
RSHUTD	SNAREQ_RSHUTD	1	0
REQDISCONT	SNAREQ_REQDISCONT	3	0
UNBIND	SNAREQ_UNBIND	1	4
NMVT	SNAREQ_NMVT	3	256

Table 12 List of supported request types

Request type	*Request type define*	*# bytes generated by SNA Server*	*max. bytes that can accompany request*
SHUTC	SNAREQ_SHUTC	1	0

Data structures for INITSELF requests

Format 0 INITSELF:

```
typedef struct
{
   unsigned char     format;
   unsigned char     type;                /* specifies queuing options      */
   unsigned char     mode_name[LU0_MODE_NAME_SIZE];  /* 8 char symbolic name  */
   name_t            dlu_unintrp_name;     /* dlu name                     */
   unsigned char     user_data_length; /* length in binary of user data field */
   unsigned char     user_data_key;    /* x'00' structured fields follow      */
                                       /* ~x'00' first byte of unstructed data*/
   unsigned char     *user_data;               /* user data field             */
   name_t             cv;                 /* control vectors field */
} init_self_fmt0_rq;
```

Format 1 INITSELF:

```
typedef struct
{
   unsigned char     format;
   unsigned char     type;                /* specifies queuing options      */
   unsigned char     dlu_queuing_conditions;   /* queuing conditions for DLU*/
   unsigned char     mode_name[LU0_MODE_NAME_SIZE];  /* 8 char symbolic name  */
   name_t            dlu_unintrp_name;     /* dlu name                      */
   name_t            urc;                 /* urc data field                 */
   name_t            cv;                  /* control vectors field          */

} init_self_fmt1_rq;
```

Appendix D

sim3270.c Listing

The **sim3270.c** application is a sample interactive 3270 NPI/SNA application with very limited functionality. Its is presented for tutorial purposes only. A test message is sent to the SNA host whenever the user presses a key, and each time a chain is received from the host a dot is printed on the terminal screen. No attempt is made to interpret the contents of the chain.

```
/***************************************************************************
*                Simple NPI/5250 SNA API Application.                      *
****************************************************************************
*                                                                          *
* This is a very simple demonstration NPI/SNA API 5250 application.        *
*                                                                          *
***************************************************************************/

#include <stropts.h>                          /* for poll() INFTIM */
#include <poll.h>
#include <gcom/npiapi.h>
#include <gcom/npiext.h>
#include <gcom/snaapi.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>

unsigned char  buf[6144] ;                    /* SNA input buffer */
char           tty_input_bfr[120];
int            connect_lu = 2;                /* local LU */
int            xmt_mode;                      /* transmission mode */
int            if_state;                      /* interface state */
int            write_pending;
int            sslu_write_pending;
int            msgs_to_send;                  /* msgs to send to HOST */
int            sna_connected;                 /* to LU */

int            tty_keyboard = 0;              /* a file descriptor */
int            tty_screen   = 1;              /* a file descriptor */
int            sna_fid;                       /* a file descriptor */

struct pollfd  pollfd[3];                     /* used in poll loop */

int            log_options = NPI_LOG_DEFAULT;       /* for sna_init_log () */
char           *log_name   = "snausr.log";          /* for sna_init_log () */

char           dot         = '.';
unsigned char  host_msg[]  =
{
       0x7d,                                  /* 3270 ENTER AID */
       0x40, 0x40,                            /* display address 0,0 */
       0xd5, 0xd7, 0xc9, 0x61, 0xe2, 0xd5, 0xc1     /* "NPI/SNA" */
};

unsigned char sslu_msg[] =
{
       0xf5, 0xf2, 0xf5, 0xf0                 /* "5250" */
       0xd5, 0xd7, 0xc9, 0x61, 0xe2, 0xd5, 0xc1     /* "NPI/SNA" */
};
/*
 * Indexes into the pollfd array.
 */
#define SNA_FD          0
#define TTY_KEYBOARD    1                           /* standard in */
#define TTY_SCREEN      2                           /* standard out */

/***************************************************************************
*                                send_msgs                                 *
****************************************************************************
*                                                                          *
* Send a message to the host if a message is required and conditions       *
* permit.                                                                  *
*                                                                          *
***************************************************************************/
void
send_msgs (additional_msgs)                         /* to HOST */
    int  additional_msgs;
{
    int  poll_bits;
    int  save_if_state = if_state;                  /* in case of write failure */
    long flags = 0L;                                /* specifies bracket bits   */

    msgs_to_send += additional_msgs;

    if ((msgs_to_send  == 0)                        /* nothing to write   */
    ||  (write_pending != 0)                        /* write in progress  */
    ||  (sna_connected == 0))                       /* not connected      */
        return;

    switch (xmt_mode)
    {
    case XM_HDX_FF:                                 /* bracket protocol */
        if (if_state == SA_RCV)
            return ;

        if (msgs_to_send == 1)                      /* the last message */
        {
            flags   |= N_SNA_CDI;                   /* let the host send */
            if_state = SA_RCV;
        }
        else
        {
            if_state = SA_SND;
        }
        break;
    }

    poll_bits = sna_start_write (sna_fid, host_msg, sizeof (host_msg), flags);

    if (poll_bits < 0)                              /* call failed */
    {
        perror ("send_msgs - write failure");
        if_state = save_if_state;
```

```
    }
    else
    {
        write_pending   = 1;                      /* one at a time */
        sna_poll_events = poll_bits;              /* into global */
    }
}

/****************************************************************************
*                              send_sslu_msgs                              *
****************************************************************************
*                                                                          *
* Send a message to the SSCP if a message is required and conditions      *
* permit.                                                                  *
*                                                                          *
****************************************************************************/
void
send_sslu_msgs (additional_msgs)                      /* to HOST */
    int  additional_msgs;
{
    int  poll_bits;
    long flags = 0L;                              /* specifies bracket bits   */

    if ((msgs_to_send  == 0)                      /* nothing to write   */
    ||  (sslu_write_pending != 0)                 /* write in progress  */
    ||  (sna_connected == 0))                     /* not connected      */
        return;

   if (msgs_to_send > 1)                          /* only one message allowed */
           return ;

    poll_bits = sna_sslu_start_write (sna_fid, sslu_msg, sizeof (sslu_msg),
0);

    if (poll_bits < 0)                            /* call failed */
    {
        perror ("send_msgs - sslu write failure");
    }
    else
    {
        sslu_write_pending   = 1;                      /* one at a time */
        sna_poll_events = poll_bits;              /* into global */
    }
}

/****************************************************************************
*                              write_dots                                  *
****************************************************************************
*                                                                          *
* Write any queued dots to the screen while flow-control permits.         *
*                                                                          *
****************************************************************************/
void
write_dots (additional_dots, output)
    int additional_dots;
    char output;
{
    static int    dots_to_write = 0;              /* SNA msgs received */
    int count;

    dots_to_write += additional_dots;
    while (dots_to_write > 0)                     /* work to do */
    {
        count = write (tty_screen, &output, 1);     /* try to write a '.' */
        if (count < 0)                            /* write failed */
        {
            if (errno == EINTR)                   /* interrupted system call */
                continue;                         /* try again */
            if (errno == EAGAIN)                  /* output blocked */
            {
                pollfd[TTY_SCREEN].events |= POLLOUT;
                return;
            }
            exit (2);                             /* can't proceed */
        }
        else
        {
            dots_to_write -= count;
        }
    }
    pollfd[TTY_SCREEN].events &= ~POLLOUT;      /* reset the POLLOUT bit */
}
/****************************************************************************
*                              keyboard_events                             *
****************************************************************************
*                                                                          *
* Read and discard all queued keyboard input, and enqueue a message for   *
* the host for each read keystroke.                                        *
*                                                                          *
* Notes:                                                                   *
* 1) Most of the time, the read() call will acquire all the existing      *
*    tty input.  It is inefficient to call read an extra time just to     *
*    be informed "EAGAIN".  If there is additional tty input, the         *
*    application's poll() call will immediately return POLLIN for the     *
*    tty fid, and this procedure is called again.                         *
*                                                                          *
****************************************************************************/
void
keyboard_events (fid)
    int fid;
{
    int count;

    for (;;)
```

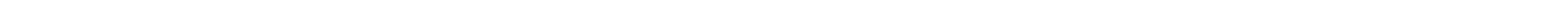
```
    {
        count = read (fid, tty_input_bfr, sizeof (tty_input_bfr));

        if (count > 0)                      /* read a few characters */
        {
            send_msgs (count);              /* additional msgs required  */
            return;
        }
        if (errno == EINTR)                 /* interrupted system call */
            continue;                       /* try again */

        if (errno == EAGAIN)                /* no tty input available */
            return;                         /* retry later */

        exit (3);                           /* punt even if EOF */
    }
}


/**************************************************************************
*                       usr_read_complete                                 *
***************************************************************************
*                                                                         *
* A chain has been received from the host.  If conditions permit,         *
* attempt to write a possibly queued message to the host.                 *
*                                                                         *
* In response to the received message, enqueue a dot and attempt to       *
* write it to the screen.                                                 *
*                                                                         *
**************************************************************************/
unsigned long                                           /* sense */
usr_read_complete (fid, bfr, length, flags)
    int                 fid;
    unsigned char      *bfr;
    int                 length;
    long                flags;
{
    unsigned long       result;
    int                 poll_bits;
    /* Immediately start another read on the buffer knowing that
     * the buffer will not be modified by the API call.
     */
    poll_bits = sna_set_read_bfr (fid, buf, sizeof (buf));

    if (poll_bits < 0)                              /* failure */
        perror ("usr_read_complete - reread");
    else
        pollfd[SNA_FD].events = poll_bits;          /* update global */

    if (xmt_mode == XM_FDX)
        send_sslu_msgs (0);                         /* for keystrokes */
    else if (xmt_mode == XM_HDX_FF)
    {
     if (flags & N_SNA_CDI)                             /* Change Direction */
        {
            if_state = SA_SND;
            send_msgs  (0);                             /* for keystrokes */
        }
        else
            if_state = SA_RCV;
    }
    write_dots (1,dot);                                 /* send another dot */
    return (0L);                                        /* accept host chain */
}
/**************************************************************************
*                       usr_sslu_read_complete                            *
***************************************************************************
*                                                                         *
* An sslu message has been received from the host.  If conditions permit, *
* attempt to write a possibly queued message to the host.                 *
*                                                                         *
* In response to the received message, enqueue a "S" and attempt to       *
* write it to the screen.                                                 *
*                                                                         *
**************************************************************************/
unsigned long                                               /* sense */
usr_sslu_read_complete (fid, bfr, length, flags)
    int                 fid;
    unsigned char      *bfr;
    int                 length;
    long                flags;
{
    unsigned long       result;
    int                 poll_bits;

    if (xmt_mode == XM_FDX)
        send_sslu_msgs (0);                             /* for keystrokes */

    write_dots (1,'S');                                 /* write a S */
    return (0L);                                        /* accept host chain */
}

/**************************************************************************
*                       usr_write_complete                                *
***************************************************************************
*                                                                         *
* A message previously written to the host has been acknowledged.         *
*                                                                         *
* send_msgs() is invoked to attempt to send another message to the        *
* host as appropriate.                                                    *
**************************************************************************/
void
usr_write_complete (fid)
```

```
    int fid;
{
    msgs_to_send--;                          /* one less to send */
    write_pending = 0;                          /* write completed */
    send_msgs (0);                           /* in case more msgs to send */
}
/**************************************************************************
*                          usr_sslu_write_complete                        *
***************************************************************************
*                                                                         *
* A message previously written to the host has been acknowledged.        *
*                                                                         *
* send_msgs() is invoked to attempt to send another message to the       *
* host as appropriate.                                                    *
**************************************************************************/
void
usr_sslu_write_complete (fid,status,sense)
    int fid;
    int status;
    int sense;
{
    sslu_msgs = 0;
    sslu_write_pending = 0;                          /* write completed */
    send_sslu_msgs (0);                      /* in case more msgs to send */
}

/**************************************************************************
*                              usr_host_event                             *
***************************************************************************
*                                                                         *
* The host took an action relevant to this application.  If a message    *
* was being sent to the host, the message has been terminated.           *
*                                                                         *
* send_msgs() is invoked to attempt to send another message to the       *
* host as appropriate.                                                    *
*                                                                         *
**************************************************************************/
void
usr_host_event (fid, mode, state, event, ptr, count)
    int   fid;
    int   mode;
    int   state;
    int   event;
    char *ptr;
    int   count;
{
    xmt_mode      = mode;                    /* same definition as API */
    if_state      = state;                   /* same definition as API */
    write_pending = 0;                       /* any write is canceled */

    switch (event)
    {
    case Event_HOST_aborted_BB:
    case Event_ServiceInterrupt:             /* Reserved for future use */
    case Event_ACTLU:                        /* ERP - warm start */
    case Event_UNBIND:
    case Event_SDT:                          /* Start Data Transfer */
    case Event_CANCEL_EB:                    /* End Bracket */
    case Event_CHASE_EB:                     /* End Bracket */
    case Event_CHASE_CDI:                    /* host passed cdi */
    case Event_HostNegRes:                   /* Negative response from host */
        send_msgs (0);                       /* in case more msgs to send */
        break;

    case Event_CLEAR:                        /* wait SDT */
    case Event_BIND:                         /* wait SDT */
    case Event_Shutdown:                     /* SHUTC/SHUTD */
    case Event_SIG:                          /* host has right to send */
        break;                               /* send_msgs() will defer */
     }
}

/**************************************************************************
*                  usr_log_error_complete                                 *
***************************************************************************
*    This routine would receive the results of an attempt to write an    *
*    error log entry*
**************************************************************************/
void
usr_log_error_complete(fid, status, sense)
    int fid;
    int status;
    int sense;
{
   return;
}


/**************************************************************************
*                  usr_request_complete                                   *
***************************************************************************
*    This routine would receive the results of an user request           *
**************************************************************************/
void
usr_request_complete(fid, request, status, sense)
    int fid;
    int request;
    int status;
    int sense;
{
   return;
}
```

```
/***********************************************************************
*                              usr_connected                            *
************************************************************************
*                                                                       *
* A previous attempt to connect to an LU has succeeded.  The initial    *
* mode is XM_FDX.                                                       *
*                                                                       *
* If any messages are queued to be sent to the host, a message is       *
* written.                                                              *
*                                                                       *
************************************************************************/
void
usr_connected (fid)
    int fid;
{
    xmt_mode      = XM_FDX;
    sna_connected = 1;
    write_pending = 0;

    send_sslu_msgs (0);                             /* to HOST */
}

/***********************************************************************
*                              usr_disconnected                         *
************************************************************************
*                                                                       *
* The connection to LU has been disconnected.  This occurs as a result  *
* of a DACTPU, DACTLU, ACTLU-cold, administrative actions affecting     *
* the SNA multiplexor, internal protocol errors, ...                    *
*                                                                       *
* The file being closed is a likely indication of an administrative     *
* action disabling the SNA multiplexor.  That is, the sna_open() call   *
* to obtain a file descriptor will likely fail.  An application retry   *
* mechanism may be appropriate.                                         *
*                                                                       *
************************************************************************/
usr_disconnected (fid, diagnostic, file_closed)
    int fid;
    int diagnostic;
    int file_closed;
{
    int poll_bits;

    sna_connected = 0;
    write_pending = 0;
    /*
     * If the file descriptor is closed, an attempt is made to
     * get a new file descriptor, and update the poll structure.
     */
    if (file_closed)
    {
        fid = sna_open (usr_read_complete,
                        usr_write_complete,
                        usr_connected,
                        usr_host_event,
                        usr_bid,
                        usr_disconnected);

        if (fid < 0)
        {
            perror ("usr_disconnected - sna_open");
            pollfd[SNA_FD].fd = 1;                     /* stream closed */
            return;                                    /* exit() ? */
        }

        pollfd[SNA_FD].fd = fid;                       /* new fid */
    }

    poll_bits = sna_start_connect_lu (fid, 0, connect_lu, 0, 10);

    if (poll_bits < 0)
    {
        perror ("usr_disconnected");
        return;
    }

    pollfd[SNA_FD].events = poll_bits;
}

/***********************************************************************
*                              initialize                               *
************************************************************************
*                                                                       *
* Called once by the main procedure to setup the SNA stream and         *
* keyboard any screen file descriptors for non-blocking I/O.            *
* The polling list is also initialized here.                            *
*                                                                       *
************************************************************************/
int                                             /* file descriptor */
initialize ()
{
    int       sna_fid;
    int       poll_bits = 0;
    if (sna_init_log (log_options, log_name) < 0)
        perror ("main - sna_init_log");

    printf ("snausr: starting test\n");

    sna_fid = sna_open (usr_read_complete,    /* open sna data stream */
                        usr_write_complete,
                        usr_connected,
                        usr_host_event,
                        usr_bid,
                        usr_disconnected,
```

```
                    usr_request_complete,
                    usr_sslu_write_complete,
                    usr_log_error_complete,
                    usr_sslu_read_complete);

    if (sna_fid < 0)                        /* open failed */
    {
        perror ("sna_open");
        return (-1);
    }

    result = sna_set_read_bfr (sna_fid, buf, sizeof (buf));
    if (result < 0)
    {
        perror ("connecting - set read bfr");
        sna_close (sna_fid);
        return (-1);
    }

    pollfd[SNA_FD].events = result;          /* from sna_set_read_bfr() */

    result = sna_start_connect_lu (sna_fid, 0, connect_lu, 0, 10);

    if (result < 0)
    {
        perror ("sna_start_connect_lu");
        sna_close (sna_fid);
        return (-1);
    }

    pollfd[SNA_FD].events = result;          /* from sna_start_connect_lu()*/

    if (fcntl (tty_keyboard, F_SETFL, O_NDELAY) < 0)
    {
        sna_close (sna_fid);
        return (-1);
    }

    if (fcntl (tty_screen, F_SETFL, O_NDELAY) < 0)
    {
        sna_close (sna_fid);
        return (-1);
    }     * Initialize SNA poll descriptor entry.
     * pollfd[SNA_FD].events is set above.
     */
    pollfd[SNA_FD].fd      = sna_fid;
    pollfd[SNA_FD].revents = 0;

    /*
     * Initialize TTY poll descriptor entries.
     */
    pollfd[TTY_KEYBOARD].fd      = tty_keyboard;        /* standard in */
    pollfd[TTY_KEYBOARD].revents = 0;
    pollfd[TTY_KEYBOARD].events  = POLLIN;

    pollfd[TTY_SCREEN].fd        = tty_screen;          /* standard out */
    pollfd[TTY_SCREEN].revents   = 0;
    pollfd[TTY_SCREEN].events    = 0;

    return (sna_fid);
}

/*************************************************************************
*                                    main                                *
**************************************************************************
*                                                                        *
* initialize() is called to handle preliminaries and then the main       *
* loop is entered to wait for events of interest and call appropriate    *
* event handling procedures.                                             *
*                                                                        *
* Notes:                                                                 *
* 1) If the closing of the SNA file descriptor is reported to            *
*    usr_disconnected, and if the SNA stream file can not be re-opened,  *
*    pollfd[SNA_FD].fd is set to -1.  While this condition is true,      *
*    poll() will not report any events on the SNA_FD entry.              *
*                                                                        *
*    Not shown here is code to retry opening an SNA file descriptor every *
*    few seconds.                                                        *
*                                                                        *
*************************************************************************/
main ()
{
    int events;

    if ((sna_fid = initialize ()) < 0)                   /* initialize */
        exit (1);                                        /* failure */
    for (;;)
    {
        if (poll (&pollfd[0], 3, INFTIM) < 0)        /* wait sna/tty events*/
        {
            perror ("snausr - poll");
            return;
        }

        /*
         * The called keyboard/screen event handling
         * procedures update the poll list as needed.
         */

        if (pollfd[TTY_KEYBOARD].revents)            /* keyboard events */
            keyboard_events (tty_keyboard);          /* read event */

        if (pollfd[TTY_SCREEN].revents)              /* screen events */
            write_dots (0,dot);                      /* retry write dots */
```

```
        if (pollfd[SNA_FD].revents)                   /* process SNA events */
        {
            events = sna_poll_retry (sna_fid, pollfd[SNA_FD].revents);

            if (events < 0)                           /* sna fid not usable */
            {                                         /* corrective action? */
                pollfd[SNA_FD].events = 0;
                break;                                /* {1} */
            }

            pollfd[SNA_FD].events = events;           /* for next poll */
        }
    }
}
```



Appendix E

sim5250.c Listing

The **sim5250.c** application is a sample interactive 5250 NPI/SNA application with very limited functionality. Its is presented for tutorial purposes only. A test message is sent to the SNA host whenever the user presses a key, and each time a chain is received from the host a dot is printed on the terminal screen. No attempt is made to interpret the contents of the chain.

```
/*************************************************************************
*                  Simple NPI/3270 SNA API Application.                 *
*************************************************************************
*                                                                       *
* This is a very simple demonstration NPI/SNA API 3270 application.     *
*                                                                       *
*************************************************************************/

#include <stropts.h>                            /* for poll() INFTIM */
#include <poll.h>
#include <gcom/npiapi.h>
#include <gcom/npiext.h>
#include <gcom/snaapi.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>

unsigned char  buf[6144] ;                      /* SNA input buffer */
char           tty_input_bfr[120];
int            connect_lu = 2;                  /* local LU */
int            xmt_mode;                        /* transmission mode */
int            if_state;                        /* interface state */
int            write_pending;
int            msgs_to_send;                    /* msgs to send to HOST */
int            sna_connected;                   /* to LU */

int            tty_keyboard = 0;                /* a file descriptor */
int            tty_screen   = 1;                /* a file descriptor */
int            sna_fid;                         /* a file descriptor */

struct pollfd  pollfd[3];                       /* used in poll loop */

int            log_options = NPI_LOG_DEFAULT;   /* for sna_init_log () */
char           *log_name   = "snausr.log";      /* for sna_init_log () */

char           dot         = '.';
unsigned char  host_msg[]  =
{
       0x7d,                                    /* 3270 ENTER AID */
       0x40, 0x40,                              /* display address 0,0 */
       0xd5, 0xd7, 0xc9, 0x61, 0xe2, 0xd5, 0xc1 /* "NPI/SNA" */
};
/*
 * Indexes into the pollfd array.
 */
#define SNA_FD          0
#define TTY_KEYBOARD    1                       /* standard in */
#define TTY_SCREEN      2                       /* standard out */

/*************************************************************************
*                                send_msgs                              *
*************************************************************************
*                                                                       *
* Send a message to the host if a message is required and conditions    *
* permit.                                                               *
*                                                                       *
*************************************************************************/
void
send_msgs (additional_msgs)                     /* to HOST */
    int  additional_msgs;
{
    int  poll_bits;
    int  save_if_state = if_state;              /* in case of write failure */
    long flags = 0L;                            /* specifies bracket bits   */

    msgs_to_send += additional_msgs;

    if ((msgs_to_send  == 0)                    /* nothing to write   */
    ||  (write_pending != 0)                    /* write in progress  */
    ||  (sna_connected == 0))                   /* not connected      */
        return;

    switch (xmt_mode)
    {
    case XM_DTR:                                /* data traffic reset */
        return;

    case XM_FDX:                                /* data traffic reset */
        break;

    case XM_HDX_FF:                             /* bracket protocol */
        if ((if_state == SA_RCV)
        ||  (if_state == SA_BID))
            return ;

        if (if_state == SA_CONTENTION)          /* between brackets */
            flags |= N_SNA_BB;                  /* start a bracket  */

        if (msgs_to_send == 1)                  /* the last message */
        {
            flags   |= N_SNA_CDI;               /* let the host send */
            if_state = SA_RCV;
        }
        else
        {
            if_state = SA_SND;
        }
        break;
    }

    poll_bits = sna_start_write (sna_fid, host_msg, sizeof (host_msg), flags);

    if (poll_bits < 0)                          /* call failed */
```

```
    {
        perror ("send_msgs - write failure");
        if_state = save_if_state;
    }
    else
    {
        write_pending   = 1;                        /* one at a time */
        sna_poll_events = poll_bits;                /* into global */
    }
}

/************************************************************************
*                              write_dots                               *
*************************************************************************
*                                                                       *
* Write any queued dots to the screen while flow-control permits.       *
*                                                                       *
************************************************************************/
void
write_dots (additional_dots)
    int additional_dots;
{
    static int    dots_to_write = 0;            /* SNA msgs received */
    int count;

    dots_to_write += additional_dots;
    while (dots_to_write > 0)                   /* work to do */
    {
        count = write (tty_screen, &dot, 1);     /* try to write a '.' */
        if (count < 0)                          /* write failed */
        {
            if (errno == EINTR)                 /* interrupted system call */
                continue;                       /* try again */
            if (errno == EAGAIN)                /* output blocked */
            {
                pollfd[TTY_SCREEN].events |= POLLOUT;
                return;
            }
            exit (2);                           /* can't proceed */
        }
        else
        {
            dots_to_write -= count;
        }
    }
    pollfd[TTY_SCREEN].events &= ~POLLOUT;      /* reset the POLLOUT bit */
}
/************************************************************************
*                              keyboard_events                          *
*************************************************************************
*                                                                       *
* Read and discard all queued keyboard input, and enqueue a message for *
* the host for each read keystroke.                                     *
*                                                                       *
* Notes:                                                                *
* 1) Most of the time, the read() call will acquire all the existing    *
*    tty input.  It is inefficient to call read an extra time just to   *
*    be informed "EAGAIN".  If there is additional tty input, the       *
*    application's poll() call will immediately return POLLIN for the   *
*    tty fid, and this procedure is called again.                       *
*                                                                       *
************************************************************************/
void
keyboard_events (fid)
    int fid;
{
    int count;

    for (;;)
    {
        count = read (fid, tty_input_bfr, sizeof (tty_input_bfr));

        if (count > 0)                          /* read a few characters */
        {
            send_msgs (count);                  /* additional msgs required  */
            return;
        }
        if (errno == EINTR)                     /* interrupted system call */
            continue;                           /* try again */

        if (errno == EAGAIN)                    /* no tty input available */
            return;                             /* retry later */

        exit (3);                               /* punt even if EOF */
    }
}

/************************************************************************
*                              usr_bid                                  *
*************************************************************************
*                                                                       *
* The host requests to initiate a bracket.  This simple test program    *
* always accepts this request.                                          *
*                                                                       *
************************************************************************/
int
usr_bid (fid)
    int fid;
{
    if_state = SA_BID;                              /* waiting begin bracket */

    return (0);                                     /* Accept the BID */
}
```

```
/***************************************************************************
*                          usr_read_complete                               *
****************************************************************************
*                                                                          *
* A chain has been received from the host.  If conditions permit,          *
* attempt to write a possibly queued message to the host.                  *
*                                                                          *
* In response to the received message, enqueue a dot and attempt to        *
* write it to the screen.                                                  *
*                                                                          *
***************************************************************************/
unsigned long                                            /* sense */
usr_read_complete (fid, bfr, length, flags)
    int                   fid;
    unsigned char        *bfr;
    int                   length;
    long                  flags;
{
    unsigned long         result;
    int                   poll_bits;
    /* Immediately start another read on the buffer knowing that
     * the buffer will not be modified by the API call.
     */
    poll_bits = sna_set_read_bfr (fid, buf, sizeof (buf));

    if (poll_bits < 0)                                 /* failure */
        perror ("usr_read_complete - reread");
    else
        pollfd[SNA_FD].events = poll_bits;             /* update global */

    if (xmt_mode == XM_FDX)
        send_msgs (0);                                 /* for keystrokes */

    else if (xmt_mode == XM_HDX_FF)
    {
        if (flags & N_SNA_EB)                          /* End Bracket */
        {
            if_state = SA_CONTENTION;
            send_msgs  (0);                            /* for keystrokes */
        }
        else if (flags & N_SNA_CDI)                    /* Change Direction */
        {
            if_state = SA_SND;
            send_msgs  (0);                            /* for keystrokes */
        }
        else
            if_state = SA_RCV;
    }
    write_dots (1);                                    /* send another dot */
    return (0L);                                       /* accept host chain */
}
/***************************************************************************
*                          usr_write_complete                              *
****************************************************************************
*                                                                          *
* A message previously written to the host has been acknowledged.          *
*                                                                          *
* send_msgs() is invoked to attempt to send another message to the         *
* host as appropriate.                                                     *
***************************************************************************/
void
usr_write_complete (fid)
    int fid;
{
    msgs_to_send--;                              /* one less to send */
    write_pending = 0;                              /* write completed */
    send_msgs (0);                               /* in case more msgs to send */
}
/***************************************************************************
*                          usr_host_event                                  *
****************************************************************************
*                                                                          *
* The host took an action relevant to this application.  If a message      *
* was being sent to the host, the message has been terminated.             *
*                                                                          *
* send_msgs() is invoked to attempt to send another message to the         *
* host as appropriate.                                                     *
*                                                                          *
***************************************************************************/
void
usr_host_event (fid, mode, state, event)
    int fid;
    int mode;
    int state;
    int event;
{
    xmt_mode      = mode;                       /* same definition as API */
    if_state      = state;                      /* same definition as API */
    write_pending = 0;                          /* any write is canceled */

    switch (event)
    {
    case Event_HOST_aborted_BB:
    case Event_ServiceInterrupt:                /* Reserved for future use */
    case Event_ACTLU:                           /* ERP - warm start */
    case Event_UNBIND:
    case Event_SDT:                             /* Start Data Transfer */
    case Event_CANCEL_EB:                       /* End Bracket */
    case Event_CHASE_EB:                        /* End Bracket */
    case Event_CHASE_CDI:                       /* host passed cdi */
    case Event_HostNegRes:                      /* Negative response from host */
        send_msgs (0);                          /* in case more msgs to send */
        break;
```

```
    case Event_CLEAR:                        /* wait SDT */
    case Event_BIND:                         /* wait SDT */
    case Event_Shutdown:                     /* SHUTC/SHUTD */
    case Event_SIG:                          /* host has right to send */
        break;                               /* send_msgs() will defer */
     }
}
/****************************************************************************
*                            usr_connected                                  *
*****************************************************************************
*                                                                           *
* A previous attempt to connect to an LU has succeeded.  The initial        *
* mode is XM_FDX.                                                           *
*                                                                           *
* If any messages are queued to be sent to the host, a message is           *
* written.                                                                  *
*                                                                           *
****************************************************************************/
void
usr_connected (fid)
    int fid;
{
    xmt_mode      = XM_FDX;
    sna_connected = 1;
    write_pending = 0;

    send_msgs (0);                               /* to HOST */
}

/****************************************************************************
*                            usr_disconnected                               *
*****************************************************************************
*                                                                           *
* The connection to LU has been disconnected.  This occurs as a result      *
* of a DACTPU, DACTLU, ACTLU-cold, administrative actions affecting         *
* the SNA multiplexor, internal protocol errors, ...                        *
*                                                                           *
* The file being closed is a likely indication of an administrative         *
* action disabling the SNA multiplexor.  That is, the sna_open() call       *
* to obtain a file descriptor will likely fail.  An application retry       *
* mechanism may be appropriate.                                             *
*                                                                           *
****************************************************************************/
usr_disconnected (fid, diagnostic, file_closed)
    int fid;
    int diagnostic;
    int file_closed;
{
    int poll_bits;

    sna_connected = 0;
    write_pending = 0;

    /*
     * If the file descriptor is closed, an attempt is made to
     * get a new file descriptor, and update the poll structure.
     */
    if (file_closed)
    {
        fid = sna_open (usr_read_complete,
                        usr_write_complete,
                        usr_connected,
                        usr_host_event,
                        usr_bid,
                        usr_disconnected);

        if (fid < 0)
        {
            perror ("usr_disconnected - sna_open");
            pollfd[SNA_FD].fd = 1;                  /* stream closed */
            return;                                 /* exit() ? */
        }

        pollfd[SNA_FD].fd = fid;                    /* new fid */
    }

    poll_bits = sna_start_connect_lu (fid, 0, connect_lu, 0, 10);

    if (poll_bits < 0)
    {
        perror ("usr_disconnected");
        return;
    }

    pollfd[SNA_FD].events = poll_bits;
}

/****************************************************************************
*                             initialize                                    *
*****************************************************************************
*                                                                           *
* Called once by the main procedure to setup the SNA stream and             *
* keyboard any screen file descriptors for non-blocking I/O.                *
* The polling list is also initialized here.                                *
*                                                                           *
****************************************************************************/
int                                          /* file descriptor */
initialize ()
{
    int       sna_fid;
    int       poll_bits = 0;
    if (sna_init_log (log_options, log_name) < 0)
        perror ("main - sna_init_log");

    printf ("snausr: starting test\n");
```

```
sna_fid = sna_open (usr_read_complete,    /* open sna data stream */
                    usr_write_complete,
                    usr_connected,
                    usr_host_event,
                    usr_bid,
                    usr_disconnected);

if (sna_fid < 0)                          /* open failed */
{
    perror ("sna_open");
    return (-1);
}

result = sna_set_read_bfr (sna_fid, buf, sizeof (buf));
if (result < 0)
{
    perror ("connecting - set read bfr");
    sna_close (sna_fid);
    return (-1);
}

pollfd[SNA_FD].events = result;           /* from sna_set_read_bfr() */

result = sna_start_connect_lu (sna_fid, 0, connect_lu, 0, 10);

if (result < 0)
{
    perror ("sna_start_connect_lu");
    sna_close (sna_fid);
    return (-1);
}

pollfd[SNA_FD].events = result;           /* from sna_start_connect_lu()*/

if (fcntl (tty_keyboard, F_SETFL, O_NDELAY) < 0)
{
    sna_close (sna_fid);
    return (-1);
}

if (fcntl (tty_screen, F_SETFL, O_NDELAY) < 0)
{
    sna_close (sna_fid);
    return (-1);
}     * Initialize SNA poll descriptor entry.
 * pollfd[SNA_FD].events is set above.
 */
pollfd[SNA_FD].fd      = sna_fid;
pollfd[SNA_FD].revents = 0;

/*
     * Initialize TTY poll descriptor entries.
     */
    pollfd[TTY_KEYBOARD].fd      = tty_keyboard;        /* standard in */
    pollfd[TTY_KEYBOARD].revents = 0;
    pollfd[TTY_KEYBOARD].events  = POLLIN;

    pollfd[TTY_SCREEN].fd        = tty_screen;          /* standard out */
    pollfd[TTY_SCREEN].revents   = 0;
    pollfd[TTY_SCREEN].events    = 0;

    return (sna_fid);
}

/*****************************************************************************
*                                    main                                    *
******************************************************************************
*                                                                           *
* initialize() is called to handle preliminaries and then the main          *
* loop is entered to wait for events of interest and call appropriate       *
* event handling procedures.                                                *
*                                                                           *
* Notes:                                                                    *
* 1) If the closing of the SNA file descriptor is reported to               *
*    usr_disconnected, and if the SNA stream file can not be re-opened,     *
*    pollfd[SNA_FD].fd is set to -1.  While this condition is true,         *
*    poll() will not report any events on the SNA_FD entry.                 *
*                                                                           *
*    Not shown here is code to retry opening an SNA file descriptor every  *
*    few seconds.                                                          *
*                                                                           *
*****************************************************************************/
main ()
{
    int events;

    if ((sna_fid = initialize ()) < 0)                   /* initialize */
        exit (1);                                        /* failure */
    for (;;)
    {
        if (poll (&pollfd[0], 3, INFTIM) < 0)        /* wait sna/tty events*/
        {
            perror ("snausr - poll");
            return;
        }

        /*
         * The called keyboard/screen event handling
         * procedures update the poll list as needed.
         */

        if (pollfd[TTY_KEYBOARD].revents)             /* keyboard events */
            keyboard_events (tty_keyboard);           /* read event */
```

```
        if (pollfd[TTY_SCREEN].revents)                /* screen events */
            write_dots (0);                            /* retry write dots */

        if (pollfd[SNA_FD].revents)                    /* process SNA events */
        {
            events = sna_poll_retry (sna_fid, pollfd[SNA_FD].revents);

            if (events < 0)                            /* sna fid not usable */
            {                                          /* corrective action? */
                pollfd[SNA_FD].events = 0;
                break;                                 /* {1} */
            }

            pollfd[SNA_FD].events = events;            /* for next poll */
        }
    }
}
```

Appendix F

LU0 Host Emulation

The GCOM software includes LU0 host emulation capability. The same API library is used for both secondary and host emulation. The GCOM LU0 host emulation provides all necessary SSCP services to start and maintain the necessary SSCP-PU, SSCP-LU and LU-LU sessions required for data transfer.

The LU0 host/secondary emulation sftware has the capability to support the three transmission modes as as defined in the bind command: full duples (FDX), Half-duplex flip-flop (HDX-ff) and Half-Duplex contention (HDX cont). These modes define the transmission behavior at the SNA layer not the link layer of the protocol stack.

When in FDX transmission mode either side can send a data frame at any time. Both sides are considered to be in Send/Receive (S,R) state. That is, they can send and receive at the same time.

When in FDX-ff state one side is in send state (S,*R) and the other side is in the receive state (*S,R) The state is changed by the send state sending a change direction indication to the other side resulting in the state “flip-flopping”. If use of the bracket protocol is specified in the bind an additional contention state exists (Between Brackets). Either side may send when the session is in this state. The bracket race condition is resolved by a bind parameter that specifies which side (primary or secondary) of the session is the contention winner. The contention winner wins all contention races. Messages received from the contention loser in a bracket race condition are rejected by the contention winner with the appropriate sense code.

When in HDX-cont mode both sides enter a contention state between messages. Either side can send a message in this state. The contention state is entered when an end chain is received. Contention races are resolved in the same manner that bracket contention races are resolved. The contention winner as defined in the bind always wins races. In the case of HDX-cont mode the contention winner has the option of queuing messages from the contention loser or rejecting them. the GCOM SNA Server will always queue messages from the contention loser when the GCOM code is the contention winner.

Glossary

The meanings of the terms defined in this glossary are specific to the way GCOM uses them in the text. Some of meanings may extend or contradict other industry-standard definitions.

Glossary

API *Application Program Interface.* Either a process running in the Rsystem or a library of routines available to your application.

basic conversation verbs LU 6.2 SNA verbs that can be used for basic or mapped conversations.

BB *Begin Bracket.*

callback function A routine in your application that the API process calls to deliver data, signal flow control and signal data transmit complete.

CDI *Change Direction Indicator.*

CEBI *Conditional End Bracket Indicator.*

conversation (verbs) An LU 6.2 transaction that takes place between a remote and local LU. Every conversation specifies either a basic or mapped conversation type. See *basic conversation* and *mapped conversation.*

data traffic reset Once a 3270 SNA host BIND or CLEAR command has been processed, no data can be transferred until an SNA host SDT command has been processed.

definite response mode When data is sent using definite response mode, the recipient must respond with a positive or negative acknowledgment.

DLC *Data Link Control.* This refers to ISO layer 2, called the data link layer.

DLPI *Data Link Provider Interface.* A UNIX STREAMS interface that handles frame level protocols.

EB *End Bracket.*

exception response mode When data is sent using exception response mode, a response is generated only when an error is detected.

FIC *First (RU) In Chain.*

FM *Function Management.*

FMD *Function Management Data.*

full-duplex Data transmitted on the LU-SSCP session is limited to 256 bytes, immediate request mode, OIC (only-in-chain), no brackets, no code selection indicator. Immediate request mode implies that data cannot be transmitted over an LU until any previously started transmissions have completed.

NPI/SNA places no additional constraints on when your application can transmit data. However, the 3270 data stream protocol may require a half-duplex transmission discipline, with the host initiating the first transmission.

half-duplex flip-flop After a 3270 SNA host SDT has been accepted, bracket protocol is used in half-duplex flip-flop mode. If your application and the host simultaneously begin a bracket, your application has the right to proceed, and the host's attempt at starting a bracket fails. Only the host can end a bracket.

.

ISO *International Standards Organization.* The standards committee that developed the seven layer model for Open Systems Interconnection (OSI).

LIC *Last (RU) In Chain.*

local LU 6.2 The LU 6.2 from whose point-of-view an activity is described.

LU *Logical Unit.*

mapped conversation verbs LU 6.2 SNA verbs that can be used only in a mapped conversation. Basic conversations are not possible using this verb type. The application that communicates with this API must provide any necessary mapped conversation LU services component program.

MIC *Middle (RU) In Chain.*

NPI *Network Provider Interface.* A UNIX STREAMS interface that handles layer three protocols.

OIC *Only (RU) In Chain.*

OSI *Open Systems Interconnection.* An internationally recognized 7-layered protocol model for data transport.

PU *Physical Unit.*

remote LU 6.2 The LU 6.2 that is the local LU 6.2's actual or potential session partner.

Rsystem *Ring System.* GCOM's proprietary ring-based token passing software engine. Rsystem is short for Ring System.

RU *Request/Response Unit.*

SNA *Systems Network Architecture.* A protocol architecture developed by IBM.

SSCP *System Services Control Point.*

transaction program In a conversation between two transaction programs, the source program is the one that initiates the conversation with the target program.

TS *Transaction Services.*

Index

Numerics

3270
- BB w/o preceding BID 103
- BIND check table 178
- BIND parameters 23
- bracket protocol 28
- data transmission modes 27
- LU-LU sessions 25
- LU-SCCP sessions 24
- performing a write 79
- sim3270.c only callback procedures 87, 99
- SNA features 22
- usr_bid() 101
- usr_host_event() 105

5250
- API routines, listed 118
- BIND check table 180
- bind parameters 32
- callback routines, listed 119
- data transmission modes 35
- LU-LU Sessions 34
- LU-LU sessions 34
- LU-SSCP sessions 33
- parameters for sna_open() 129
- performing a write 83
- sim5250.c only callback procedures 107
- SNA features 31
- usr_host_event() 107

a

ACTLU
- connection attempts prior to 59
- host cold start diagnostic 161
- host event 106
- warm start event 164

ACTLU command
- for LU-SSCP sessions 33
- LU-LU 5250 sessions 34
- LU-LU session 25
- LU-SSCP session 25
- purpose 24

ACTLU-ERP, LU connection terminated upon receipt of a 146

ACTPU
- host cold start diagnostic 161
- not received diagnostic 160
- warm diagnostic 164

address
- already in use 175
- bad 170
- does not exist 169
- family not supported 175
- malformed diagnostic 160

Administrator request diagnostic 161

Advertise error 173

angle bracket conventions 14

A-node not available 172

API
- defined 205
- for SNA routines 19
- Routines 118

argument
- invalid 170
- list too long 170
- out of domain of function 171

array 51

Attention Identifier (AID) code 51

auto-retry for connection 61

b

Badly Formatted Call Request codes, error 160
BB 205
- 3270 correct usage 28
- bit box 71
- bits 73
- EBUSY error code 168
- ENOTBLK error code 168
- flag 167
- host accepted 102
- host, rejected 102
- receipt of a chain 99
- set flag, sna_start_write() 139
- sna_start_write() 79, 83
- usr_bid() 101, 144
- usr_read_complete() 95, 152
- w/o preceding BID 103
- wait, accepted BID 126
- wait, BID accepted 150
- wait, BID accepted diagnostic 163
- with host chain canceled, diagnostic 164

bfr_ptr parameter 152
BID
- accepted, wait BB diagnostic 163
- BB w/o preceding 103
- command, sending 101
- host accepted 100
- host rejected 100
- usr_bid() 144
- with BB cancelled diagnostic 164

BIND
- 5250 acceptance of 107
- accepted for 5250 half-duplex flip-flop 35
- accepted, xmt mode enforced for sna_start_write() 139
- check table for 3270 178
- check table for 5250 180
- check table, 3270 178
- command 27
- command, 3270 application informed of 105
- command, 5250 application informed of 107
- command, for LU-SSCP sessions 34
- command, LU-LU session usage 25
- command, LU-SSCP session usage 25
- command, options for 3270 23
- command, options for 5250 32
- data returned, usr_host_event() 150
- diagnostic 164
- host event 106
- host specified in usr_write_complete() 157
- Parameters, 5250 32
- reject, usr_host_event() 147
- request failed error code 168
- request specifying response modes 28
- request, specifies response modes 36
- requirements for usr_write_complete() 97

BIS request 183
Blocking I/O disadvantage 39
boldface conventions 14
bracket
- ending a 27
- Protocol, for 3270 28
- protocol, used in half-duplex flip-flop mode 27
- request rejected 28
- when to request beginning one 144

buffer 49
- maximum size 139
- scheme 134
- space not available 176

buffering_mode parameter, described 134

c

C language reference material 11
call
- could not be routed by NPI 162
- Rejected Resulting From the State of the PU/LU, error 160
- request, NPI failed to send, diagnostic 162
- setup state event diagnostic 162

callback
procedure typedefs 19
Routines 119
CANCEL
carries CDI diagnostic 164
carries EB diagnostic 164
cautions, purpose of 14
CDI
3270 correct usage 28
bit box 71
bits 73
defined 205
receipt of a chain 99
SNA flag 167
sna_start_write() 79, 83
take back 164
usr_read_complete() 95, 152
CEBI, defined 205
chain
accepted 93
being sent 163
multiple elements passed 27
multiple sent 97, 139
passed to your application 152
receipt of 99
rejected 93
too large for application bfr, diagnostic 162
transmitted to the host 157
transmitting a 71
chain_length parameter 152
CHAIN_MODE_BUFFERED_RU buffering scheme 134
CHAIN_MODE_CHAIN buffering scheme 134
CHAIN_MODE_RU buffering scheme 134
Chains being sent 163
Channel number out of range 171
CHASE carries CDI diagnostic 164
CHASE carries EB diagnostic 164
CHASE request 183
children processes, none 170
CLEAR
command 27
diagnostic 164
client error diagnostic 162
code selection indicator flag 167
cold start diagnostic 161
COLD, ACTLU 5250 command specifies 34
commands sent to remote SNA host 132
configuration error 162
connect_lu
application 49
parameter 57
connection
abort 176
attampts prior to ACTLU 59
dropped upon reset 176
indications log option 166
refused 176
reset by peer 176
successful notification, to a local LU 145
timed-out 176
to a local LU 137
unsuccessful notification 146
conventions
notes, cautions and warnings 14
text 14
Convergent Error Returns 172
core, not enough 170
corrupted shared library 174
Cross mount point 173
Cross-device link 170
CSI structure not available 171

d

DACTLU, host diagnostic 161
DACTPU, host diagnostic 161
data
link control not setup 160
passed to your application 152
sending to the host 71

SS-LU session passed to your application 154
traffic reset diagnostic 163
traffic reset error code 168
traffic reset mode 27
traffic reset, attempt to xmt in 140
traffic reset, wait SDT 126, 150
transfer flags 167
transfer state 162
transmission modes
3270 27
5250 35
data traffic reset, defined 27
deadlock
condition 171
error 172
defines accessible to the programmer 160
definite response mode, defined 28, 36
Destination address required 175
device
busy, mount 170
does not exist 169, 170
full 170
not a stream 173
required, block 170
diagnostic
codes 19
codes, SNA disconnect 160
host event 164
parameter 146
SNA/API state value 163
transmission modes 163
directory, does not exist 169
Disc_ACTLU_COLD diagnostic 161
Disc_ACTPU_COLD diagnostic 161
Disc_call_perm_error diagnostic 162
Disc_call_retry_limit diagnostic 162
Disc_client_error diagnostic 162
Disc_connect_rej diagnostic 162
Disc_DACTLU diagnostic 161
Disc_DACTPU diagnostic 161
Disc_DLC_disconnected diagnostic 161
Disc_DLC_error diagnostic 161
Disc_format_error diagnostic 160
Disc_hangup diagnostic 162
Disc_interrupt_confirm diagnostic 161
Disc_interrupt_request diagnostic 161, 162
Disc_invalid_LU diagnostic 160
Disc_long_address diagnostic 160
Disc_LU_in_use diagnostic 160
Disc_LU_not_available diagnostic 160
Disc_lu_not_known diagnostic 162
Disc_lu_type_error diagnostic 160
Disc_not_DLC_connected diagnostic 160, 161
Disc_not_LU_activated diagnostic 160
Disc_not_PU_activated diagnostic 160
Disc_PU_unknown diagnostic 162, 163
Disc_rej_unspecified diagnostic 162
Disc_request_res diagnostic 161
Disc_short_address diagnostic 160
Disc_short_call_req diagnostic 160
Disc_sna_mux_restarted diagnostic 161
Disc_sna_mux_terminated diagnostic 161
Disc_sna_mux_turned_off diagnostic 161
Disc_state_error diagnostic 162
Disc_state_error_1 diagnostic 162
Disc_state_error_2 diagnostic 162
Disc_state_error_3 diagnostic 162
Disc_state_error_4 diagnostic 162
Disc_state_error_5 diagnostic 162
Disc_stream_unusable diagnostic 162
Disc_upstream_overrun diagnostic 162
disconnect
Diagnostics Interpreted from NPI Reason Codes, error 162
Resulting From Administrative Actions, error 161
Resulting From Client/Application/Stream Errors 162
Resulting From Data Link Events, error 161
Resulting From Host Actions, error 161
Resulting From State Errors 162

Resulting From the Receipt of Unsupported Rsystem Tokens, error 161
DLC
defined 205
outage diagnostic 164
station disconnected diagnostic 161
DLPI, defined 205
dlpi_close 43
dlpi_open 43
dot parameter 51
driver
de-configured diagnostic 161
turned off by administrator diagnostic 161

e

E2BIG error code 168, 170
EACCES error code 170
EACCESS error code 168
EADDRINUSE error code 175
EADDRNOTAVAIL error code 175
EADV error code 173
EAFNOSUPPORT error code 175
EAGAIN error code 170
EALREADY error code 175
EB
3270 correct usage 28
bit box 71
bits 73
defined 205
receipt of a chain 99
SNA flag 167
sna_start_write() 79, 83
usr_read_complete() 95, 152
EBADE error code 172
EBADF error code 168, 170
EBADFD error code 173
EBADMSG error code 173
EBADR error code 172
EBADRQC error code 172
EBADSLT error code 172
EBFONT error code 172
EBUSY error code 168, 170
ECHILD error code 170
ECHRNG error code 171
ECOMM error code 173
ECONNABORTED error code 176
ECONNREFUSED error code 176
ECONNRESET error code 176
EDEADLK error code 171
EDEADLOCK error code 172
EDESTADDRREQ 175
EDOM error code 171
EDOTDOT error code 173
EEXIST error code 168, 170
EFAULT error code 168, 170
EFBIG error code 170
EHOSTDOWN error code 176
EHOSTUNREACH error code 176
EIDRM error code 171
EINIT error code 177
EINPROGRESS error code 175
EINTR error code 169
EINVAL error code 170
EIO error code 169
EISCONN error code 176
EISDIR error code 170
EISNAM error code 177
EL2HLT error code 171
EL2NSYNC error code 171
EL3HLT error code 171
EL3RST error code 171
ELBIN error code 173
ELIBACC error code 174
ELIBBAD error code 174
ELIBEXEC error code 174
ELIBMAX error code 174
ELIBSCN error code 174
ELNRNG error code 171
ELOOP error code 177

EMFILE error code 170
EMLINK error code 171
EMSGSIZE 175
EMULTIHOP error code 173
ENAMETOOLONG error code 173
ENAVAIL error code 177
ENETDOWN error code 176
ENETRESET error code 176
ENETUNREACH error code 176
ENFILE error code 170
ENOANO error code 172
ENOBUFS ENOSR error code 176
ENOCSI error code 171
ENODATA error code 173
ENODEV error code 170
ENOENT error code 169
ENOEXEC error code 168, 170
ENOLCK error code 171
ENOLINK error code 173
ENOMEM error code 170
ENOMSG error code 171
ENONET error code 173
ENOPKG error code 173
ENOPROTOOPT error code 176
ENOSPC error code 170
ENOSR error code 173
ENOSTR error code 173
ENOSYS error code 174
ENOTBLK error code 168, 170
ENOTCONN error code 176
ENOTDIR error code 170
ENOTEMPTY error code 177
ENOTNAM error code 177
ENOTSOCK error code 175
ENOTTY error code 170
ENOTUNIQ error code 173
enter vs. type 14
ENXIO error code 169
EOPNOTSUPP error code 175
EPERM error code 169
EPFNOSUPPORT error code 175
EPIPE error code 168, 171
EPROTO error code 168, 173
EPROTONOSUPPORT error code 175
EPROTOTYPE error code 175
ERANGE error code 171
EREMCHG error code 173
EREMDEV error code 177
EREMOTE error code 173
EREMOTEIO error code 177
EROFS error code 170
ERP (Error Recovery Procedure) 25
ERP, ACTLU specifies for 5250 sessions 34
errno codes 19, 168
errno.h file 47, 169
error
    codes used by sim3270.c & sim5250.c 47
    codes, normal 169
    communication, on send 173
    IPC/network software argument 175
    log maintained on a per-PU basis 128
    numbers, XENIX 177
    operational 176
    recovery procedure (ERP) 26, 34
    responsible for recovery 23, 32
    returns, convergent 172
    shared library 174
    SNA disconnect diagnostic codes 160
    socket 175
    stream problems 173
    UNIX errno codes 168
ESHUTDOWN error code 176
ESOCKTNOSUPPORT error code 175
ESPIPE error code 170
ESRCH error code 169
ESRMNT error code 173
ETIME error code 173
ETIMEDOUT error code 176
ETOOMANYREFS error code 176
ETXTBSY error code 170

EUCLEAN error code 177
EUNATCH error code 171
event
  driven model 41
  notification 147
  parameter 150
  short 51
Event_ACTLU diagnostic 105, 107, 164
Event_BIND diagnostic 105, 107
Event_BIND diagnostic diagnostic* 164
Event_BIND_REJ 164, 165
Event_BIS 165
Event_CANCEL_CDI diagnostic 105, 107
Event_CANCEL_CDI diagnostic * 164
Event_CANCEL_EB diagnostic 105, 164
Event_CHAIN 164
Event_CHASE 165
Event_CHASE_CDI diagnostic 105, 107, 164
Event_CHASE_EB diagnostic 105, 164
Event_CLEAR diagnostic 105, 164
Event_CRV 164
Event_CRV diagnostic 105
Event_HOST_aborted_BB 164
  abnormal event in usr_host_event() 147
Event_HOST_aborted_BB diagnostic 105, 164
Event_HostNegRes diagnostic 105, 107
Event_HostNegRes diagnostic * 164
Event_LUSTAT 165
Event_LUSTAT_BB 165
Event_LUSTAT_CDI 165
Event_LUSTAT_EB 165
Event_NotifyRcvd 164
Event_NSPE 165
Event_QC 165
Event_QEC 165
Event_RELQ 165
Event_RTR 164
Event_SBI 165
Event_SDT diagnostic 105, 164
Event_ServiceInterrupt diagnostic 105, 107
Event_ServiceInterrupt diagnostic* 164
Event_SetBid_State 164
Event_SetBidState diagnostic 105
Event_Shutdown diagnostic 105, 164
Event_SIG diagnostic 105, 107
Event_SIG diagnostic diagnostic* 164
Event_UNBIND diagnostic 105, 107
Event_UNBIND diagnostic diagnostic* 164
EWOULDBLOCK error code 175
exception response mode, defined 28, 36
exchange
  full 172
  invalid 172
EXDEV error code 168, 170
Executive format error 170
EXFULL error code 172

f

fcntl.h file 47
fd file descriptor parameter 51
ff_state_ptr parameter 126
FIC, defined 205
fid parameter 57, 125
file
  descriptor support 129
  does not exist 169
  log to option 166
  number bad 170
  system read-only 170
  table overflow 170
  too large 170
  too many open 170
file_closed parameter 89, 146
flag
  definitions 19
  parameter 133
FM
  defined 205
  layer interface 22

profile 3 25
profile set to 7 34
FMD
data received negative response to, diagnostic 164
defined 205
font file format bad 172
format
error diagnostic 160
error, exec 170
full-duplex 35
full-duplex, defined 27, 35

g

Gcom Protocol Appliance 42
Gcom Remote API 42
architecture 42
client server model 42
running the RAPI server 43
using the RAPI library 43
getmsg()
diagnostic 162
getmsg(), failure diagnostic 162

h

half-duplex flip-flop
enforced 139
half-duplex flip-flop, defined 27, 35
header files included in sim3270.c and sim5250.c 47
highlighted term conventions 14
host
ACTLU
cold start diagnostic 161
ACTPU cold start diagnostic 161
chain
with BB canceled, diagnostic 164
DACTLU 161
DACTPU diagnostic 161
down 176
event diagnostics 164
receives data 71

i

I/O error 169
Identifier removed 171
if_state define 49
immediate request mode 27, 35
initialization options log option 166
initialize() procedure 63
INITSELF request 183
I-node is remote 173
interface state determined 126
interrupt errors 175
IPC/Network Software Argument Errors 175
italic text conventions 14

k

keyboard_events() in sim3270.c or sim5250.c 114

l

level 2
halted 171
not synchronized 171
level 3
halted 171
reset 171
LIC, defined 206
link
has been severed 173
number out of range 171
too many 171
log
errors for 5250 only 109
file default name 127

- name not unique 173
- options and log file name specified 127

log_file_name parameter 127
log_name parameter 51
log_options parameter 51, 127
logging
- mechanism 51
- options 47, 166

LU
- being used 160
- connected to application 49
- connecting to a local 57
- connection 87
- connection already existed, error code 168
- connection terminated 135
- connection terminated upon ACTLU-ERP 146
- defined 206
- disconnection 87
- establish a local connection 129
- local connection 137
- local connection check, usr_host_event() 147
- not activated 160
- not in use diagnostic 162
- request specific connection 49
- requirement 47
- secondary send data in a mode 157
- session not established, error code 168
- simple.c connects to an 63
- successful connection notification 145
- type 2 supported 19
- type in address unsupported 160
- unavailable response 25
- zero requested 160

lu parameter 137
lu_62_receive_immediate()
- event processing 41

lu_type parameter 137
LU-LU session
- for 3270 25
- for LU-LU 5250 sessions 34
- supported 19

LU-SSCP session
- for 3270 24
- for 5250 33

LU-SSCP session is limited to 256 bytes 27
LUSTAT request 183

m

M_DATA received log option 166
M_PROTOS received option 166
M_PROTOS transmitted log option 166
macro names used to set/clear flag bits 73
main() Loop 67
Malformed called address diagnostic 160
message
- content and conformance 19
- desired type not availabe 171
- from NPI unexpected, diagnostic 162
- lost diagnostic 161
- too long 175

MIC, defined 206
mode parameter 150
mode_ptr parameter 126
msgs_to_send 49
Multi-hop attempted 173
multiple chain-elements 27

n

N_REJ_NSPA_UNREACH_P diagnostic 162
N_REJUNSPECIFIED diagnostic 162
n_retries parameter 137
N_SNA_ALT_CODE
- bits 75

N_SNA_ALT_CODE transfer flags 168
N_SNA_BB
- bits 73
- transfer flags 167

N_SNA_CDI

bits 73
transfer flags 167
N_SNA_COMPRES
bits 75
transfer flags 167
N_SNA_EB
bits 73
transfer flags 167
N_SNA_ENCRYPT
bits 75
transfer flags 167
N_SNA_MoreChain
bits 75
N_SNA_PAD
bits 75
transfer flags 167
network
down 176
unreachable 176
NMVT request 183
non-blocking
delivery of host data 135
errors 175
I/O applications 39
I/O, advantages of 39
notes, purpose of 14
NOTIFY command 25
NPI
BIND request failed, error code 168
defined 206
npi.h 19
npi.log default log name 127
NPI_LOG_CONINDS log option 166
NPI_LOG_DEFAULT log option 166
NPI_LOG_ERRORS log option 166
NPI_LOG_FILE log option 166
NPI_LOG_OPTIONS log option 166
NPI_LOG_RX_DATA log option 166
NPI_LOG_RX_PROTOS log option 166
NPI_LOG_SIGNALS log option 166
NPI_LOG_STDERR log option 166
NPI_LOG_TX_DATA log option 166
NPI_LOG_TX_PROTOS log option 166
npiapi.h file 47
npiext.h 19
npiext.h file 47

o

object is remote 173
OIC, defined 206
Operation not supported on socket 175
Operational Errors 176

p

Permission denied 170
pipe broken 171
poll.h file 47
poll_out_bits parameter 131
Pre bind diagnostic 163
pre-call state 162
process
does not exist 169
no children 170
no morees 170
Protocol
error error 173
family not supported 175
not available 176
not supported 175
wrong type for socket 175
protocol
driver not attached 171
PU
defined 206
error log based on 128
type 2 supported 19
pu parameter 137
PU type 1 34

putmsg()
    failure diagnostic 162

q

QC request 183
QEC request 183

r

Range error diagnostic 160
read
    non-blocking 135
    previous still pending error code 168
    retry 131
    SSLU for 5250 111
    starting a 55
received M_DATA log option 166
record locks not available 171
reject SNA (BIND) commands 147
RELQ request 183
Remote address changed 173
Remote API 42
Remote I/O error 177
REQDISCONT request 183
REQTEST request 183
request
    code invalid 172
    descriptor invalid 172
    parameter 133
    SNA, success or failure notification 153
    unsupported diagnostic 161
requirements, knowledge 11
response modes 28
result not representable 171
retry_delay parameter 137
revents 51
RH Bracket 132
RQR request 183
RSHUTD request 183
RTR request 183
RU
    accepted 155
    chaining 27, 35
    copied into simple.c buffer 55
    defined 206
    transmission success or failure to host 156
    unformatted data 136

s

SA_BID
    attempt to transmit 140
    diagnostic 163
    flag bit setting 72, 74
    force into diagnostic 164
    interface state 49
    interface state, sna_get_api_state() 126
    receipt of chain 99
    usr_bid() 101
    usr_host_event() 150
SA_CHAIN_CONT mode 49, 163
SA_CONTENTION 163
    3270 write 79
    diagnostic 164
    flag bit setting 72, 74
    interface state 49
    interface state, sna_get_api_state() 126
    receipt of chain 99
    sna_start_write() 139, 168
    usr_bid() 101
    usr_host_event() 105, 150
SA_FDX mode 49, 163
SA_RCV
    3270 write 79
    5250 write 83
    attempt to transmit 140
    diagnostic 163
    flag bit setting 72, 74
    interface state 49

- interface state, sna_get_api_state() 126
- receipt of chain 99
- sna_start_write() 139
- state transitions 77
- usr_host_event() 105, 107, 150

SA_SND
- 3270 write 79
- 5250 write 83
- diagnostic 163
- flag bit setting 72, 74
- interface state 49
- interface state, sna_get_api_state() 126
- receipt of chain 99
- sna_start_write() 139, 168
- state transitions 77
- usr_host_event() 105, 107, 150

sample 3270 NPI/SNA application 19
sample 5250 NPI/SNA application 19
SBI request 183
screen display 15
SDT
- diagnostic 164
- post 163

secondary LU
- response modes 36
- send data in a mode 157

seek illegal 170
send_msgs() in sim3270.c or sim5250.c 114
sense parameter 151
shared library, cannot access 174
SHUTC request 184
SHUTD/SHUTC sequence diagnostic 164
SIGNAL request 183
sim3270.c
- application listing 186
- described 19
- global variables and defines 49

sim5250.c
- application listing 195
- described 19
- global variables and defines 49

slot invalid 172
SNA
- API state value diagnostics 163
- defined 206
- disconnect diagnostic codes 160
- request success or failure notification 153

sna_close(), procedure description 125
sna_connected 49
SNA_FD define 51
sna_fid variable 51
sna_get_api_state(), procedure description 126
sna_init_log(), procedure description 127
sna_log_error(), procedure description 128
sna_open(), procedure description 129
sna_poll_retry()
- event processing 41

sna_poll_retry(), procedure description 131
sna_send_request(), procedure description 132
sna_set_buffering_mode(), procedure description 134
sna_set_read_bfr(), procedure description 135
sna_sslu_start_write(), procedure description 136
sna_start_connect_lu(), procedure description 137
sna_start_write(), procedure description 139
snaapi.c 19
snaapi.h 19, 47
SNAREQ_BIS 183
SNAREQ_CHASE 183
SNAREQ_INIT_SELF 183
SNAREQ_LUSTAT_LULU 183
SNAREQ_LUSTAT_SSLU 183
SNAREQ_NMVT 183
SNAREQ_QC 183
SNAREQ_QEC 183
SNAREQ_RELQ 183
SNAREQ_REQDISCONT 183
SNAREQ_REQTEST 183
SNAREQ_RQR 183
SNAREQ_RSHUTD 183
SNAREQ_RTR 183

SNAREQ_SBI 183
SNAREQ_SHUTC 184
SNAREQ_SIGNAL 183
SNAREQ_UNBIND 183
snausr.log 51
socket
- errors 175
- is already connected 176
- is not connected 176
- operation on non-socket 175
- shutdown 176
- type not supported 175

src parameter 128
srmount error 173
SS-LU session data passed to your application 154
state
- call setup, event diagnostic 162
- data transfer, event diagnostic 162
- parameter 150
- pre-call event diagnostic 162
- termination, event diagnostic 162
- transitions for 3270 and 5250 Writes 77
- unexpected event diagnostic 162

status parameter 151
stderr, log to option 166
stdio.h file 47
stream
- closed 125
- opened 129
- opening a 53
- problems 173

stropts.h file 47
super-user, you are not 169
system call interrupted 169
System Services Control Point, defined 206

t

termination state 162
terminology conventions 14
text conventions 14
Text file busy 170
Timer expired 173
transmission
- mode determined 126
- mode diagnostics 163

transmit
- attemp while in SA_RCV or SA_BID 140

transmitted M_DATA log option 166
TS profile
- set to 1 24
- set to 3 25
- set to 7 34

TS, defined 206
tty_keyboard 51
TTY_KEYBOARD define 51
tty_screen 51
TTY_SCREEN define 51
type vs. enter 14
typewriter, not a 170

u

UNBIND request 183
UNBIND, diagnostic 164
UNFORMATTED RU 136
UNIX errors log option 166
unreadable message 173
User-Supplied Callback Procedures 119
usr_bid()
- procedure description 144
- sim3270.c and sim5250 87
- sim3270.c only 101

usr_connected()
- procedure description 145
- sim3270.c and sim5250 87

usr_disconnected()
- procedure description 146
- sim3270.c and sim5250 87, 89

usr_host_event()

procedure description 147
sim3270.c and sim5250 91
sim3270.c only 105
sim5250.c only 107
usr_log_error_complete()
procedure description 151
sim5250.c only 109
usr_read_complete()
procedure description 152
sim3270.c and sim5250 93, 95
usr_request_complete()
procedure description 153
sim5250.c only 109
usr_sslu_read_complete()
procedure description 154
sim5250.c only 111
usr_sslu_write_complete()
procedure description 156
sim5250.c only 113
usr_write_complete()
procedure description 157
sim3270.c and sim5250 97

w

warm start, ACTLU 164
warnings, purpose of 14
write
complete notification 97
for 3270 79
for 5250 83
non-blocking of data 136
non-blocking on an NPI/SNA stream 139
previous still pending error code 168
retry 131
SSLU for 5250 113
write_dots() in sim3270.c or sim5250.c 114
write_pending 49

x

XENIX Error Numbers 177
XM_DTR mode
defined 49
diagnostic 163
flag bit setting 72, 74
sna_get_api_state() 126
sna_start_write() 73
usr_host_event() 150
XM_DTR transmission mode 163
XM_FDX mode
defined 49
diagnostic 163
error code 168
flag bit setting 72, 74
sna_get_api_state() 126
usr_host_event() 150
XM_HDX_FF mode
defined 49
diagnostic 163
enforced 139
flag bit setting 72, 74
sna_get_api_state() 126
sna_start_write() 73
usr_host_event() 150
write operation in 77
XM_SSCP transmission mode 49, 163
xmit_mode define 49