LU6.2

Application Program Interface Guide

Document Version 3.0

September 2002

This manual was written, formatted and produced by technical writer Debra J. Schweiger using FrameMaker 6.0 on a Microsoft Windows Millenium platform with the help of subject matter specialists Dave Grothe and Carlton Mills, and Carleton Herrick.

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

CONTENTS

SECTION 1 ***7*** *Introducing the LU6.2 API*

8 LU6.2 Concepts
8 *source and target*
8 *transaction programs*
8 *contention winners and contention losers*
8 *CONFIRM and FLUSH*
9 LU6.2 Operations
9 *Preparing a local transaction program*
9 *Initializing session limits*
10 *Starting a connection*
12 Understanding the LU 6.2 State Machine
13 Sending Confirmations
14 Gcom Remote API

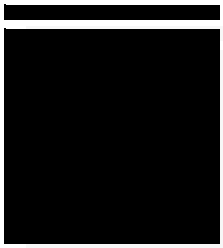
SECTION 2 ***15*** *API-provided Verbs*

17 lu62_allocate
18 lu62_allocate_luw
20 lu62_api_set_debug_level
21 lu62_change_session_limit
23 lu62_close
24 lu62_confirm
25 lu62_confirmed
26 lu62_deallocate
27 lu62_flush
28 lu62_get_attributes
29 lu62_init_log
30 lu62_initialize_session_limit
32 lu62_initialize_session_limit_luw
34 lu62_open
36 lu62_prepare_to_receive
38 lu62_receive_and_wait
41 lu62_receive_expedited_data
43 lu62_receive_immediate
46 lu62_request_to_send
47 lu62_send_data

49 lu62_send_data_record
51 lu62_send_error
53 lu62_send_expedited_data

SECTION 3 ***59*** *User-defined Callbacks*
61 lu62_allocate_t
62 lu62_confirm_t
63 lu62_confirmed_t
64 lu62_connect_t
65 lu62_deallocate_t
66 lu62_disconnected_t
67 lu62_flush_t
68 lu62_init_sess_limit_t
69 lu62_prepare_to_receive_t
70 lu62_receive_expedited_data_t
71 lu62_request_to_send_t
72 lu62_send_error_t
73 lu62_send_expedited_data_t
74 lu62_session_limit_t

SECTION 4 ***77*** *API-Provided Constants and Data Structures*
79 Logging Modes
80 Return Codes
81 Sense Codes and Sub-Codes
81 *Sense Codes*
81 *Sub-Codes*
82 allocate_data_t
83 lu62_cnos_msg_t



Introducing the LU6.2 API

Page 8 *LU6.2 Concepts*
Page 9 *LU6.2 Operations*
Page 12 *Understanding the LU 6.2 State Machine*
Page 13 *Sending Confirmations*
Page 14 *Gcom Remote API*

LU6.2 Concepts

source and target

SNA's LU6.2 (also called APPC) is a peer-to-peer communications protocol. Subsequently, it has no primary and secondary component. Instead, the local endpoint is referred to as the source, while the remote endpoint is referred to as the target. These definitions are relative, and will change with the definitions of local and remote. Assuming that an application can be said to observe, the source will always be the endpoint local to the observer and the target will always be the endpoint remote to the observer.

transaction programs

The endpoint in an LU6.2 conversation is called a transaction program or TP. The transaction program maintains one end of an LU6.2 session. A transaction program can be registered by name in order to receive connections, or can initiate a connection to a target transaction program.

contention winners and contention losers

When a session will initiate conversations, the session is termed a contention winner. When a session will register a transaction program and wait for a connection, the session is termed a contention loser. Contention losers at the source become contention winners to the target. Subsequently, the total number of sessions can be expressed in terms of contention winners available to the source and contention winners available to the target.

CONFIRM and FLUSH

For all operations which actively send a request, the operation must be followed by either an *lu62_confirm()* or an *lu62_flush()*. With an *lu62_confirm()*, no further regular data traffic will be permitted until the target responds with a CONFIRMED. When using *lu62_flush()*, your application will only receive a response if the target detects an error.

LU6.2 Operations

Preparing a local transaction program

If your local transaction program will wait for a connection from the target host, the transaction program can be started immediately and set to wait for an inbound connection.

To do this, the application should:

- Call *lu62_open()* with the *tpn_name* and *tpn_length* parameters set and the *lu_name* and *mode_name* parameters unset.
- Call *poll()* on the resource until STREAMS reported an event.
- Invoke *lu62_receive_immediate()*. Within *lu62_receive_immediate()*, the user-provided callback routines can be invoked. At this point, either the *lu62_connected_t* or the *lu62_disconnected_t* callback routine should be invoked.

If the connection is successfully set up, the application can begin to receive data at this point.

Initializing session limits

Connections between LU6.2 peers can be described in terms of LU/mode pairs. Each end of the connection has an LU address and a set of connection parameters identified with a mode name. At least once on each of these LU/mode pairs, a “change number of sessions” or “CNOS” request must be placed. This CNOS request allows the peers to configure their resource pools to make these sessions available.

To initialize this session limit, an application

- Call *lu62_open()*, specifying the LU name for which the transaction should be performed. The special mode name “SNASVCMG” should be used to indicate that this connection will be used for session management.
- Call *poll()* on the resource until activity occurs
- Call *lu62_receive_immediate()*. Either the *lu62_connected_t* or the *lu62_disconnected_t* callback should be invoked at this point.
- Call *lu62_initialize_session_limit()*.
- Call *poll()* on the resource until activity occurs.
- Call *lu62_receive_immediate()*. The *lu62_init_sess_limit_t* callback should be invoked at this point.
- Call *lu62_close()* to complete the call.

Starting a connection

Prior to initiating a connection, the LU/mode pair your application will use must have had the session limit initialized. Once this has been accomplished, the procedure for starting a conversation is fairly simple.

- First open the LU with *lu62_open()*. Use the *lu_name* and *mode* parameters to specify which LU/mode pair you wish to connect on.
- Call *poll()* on the resource until activity is indicated.
- Call *lu62_receive_immediate()*. If the connection has been established, your *lu62_connected_t* callback will be invoked. If not, your *lu62_disconnected_t* will be invoked.
- Prepare an *allocate_data_t* struct with the mode, *lu_name*, and TPN that you want to connect to. Use this structure in a call to *lu62_allocate()*.
- Call either *lu62_flush()* or *lu62_confirm()* to send the verb to the target.
- Call *poll()* on the resource until activity is indicated.
- Call *lu62_receive_immediate()*. Your *lu62_allocate_t* callback should be invoked.

At this point, if everything has gone properly, you should be ready to begin transmitting data.

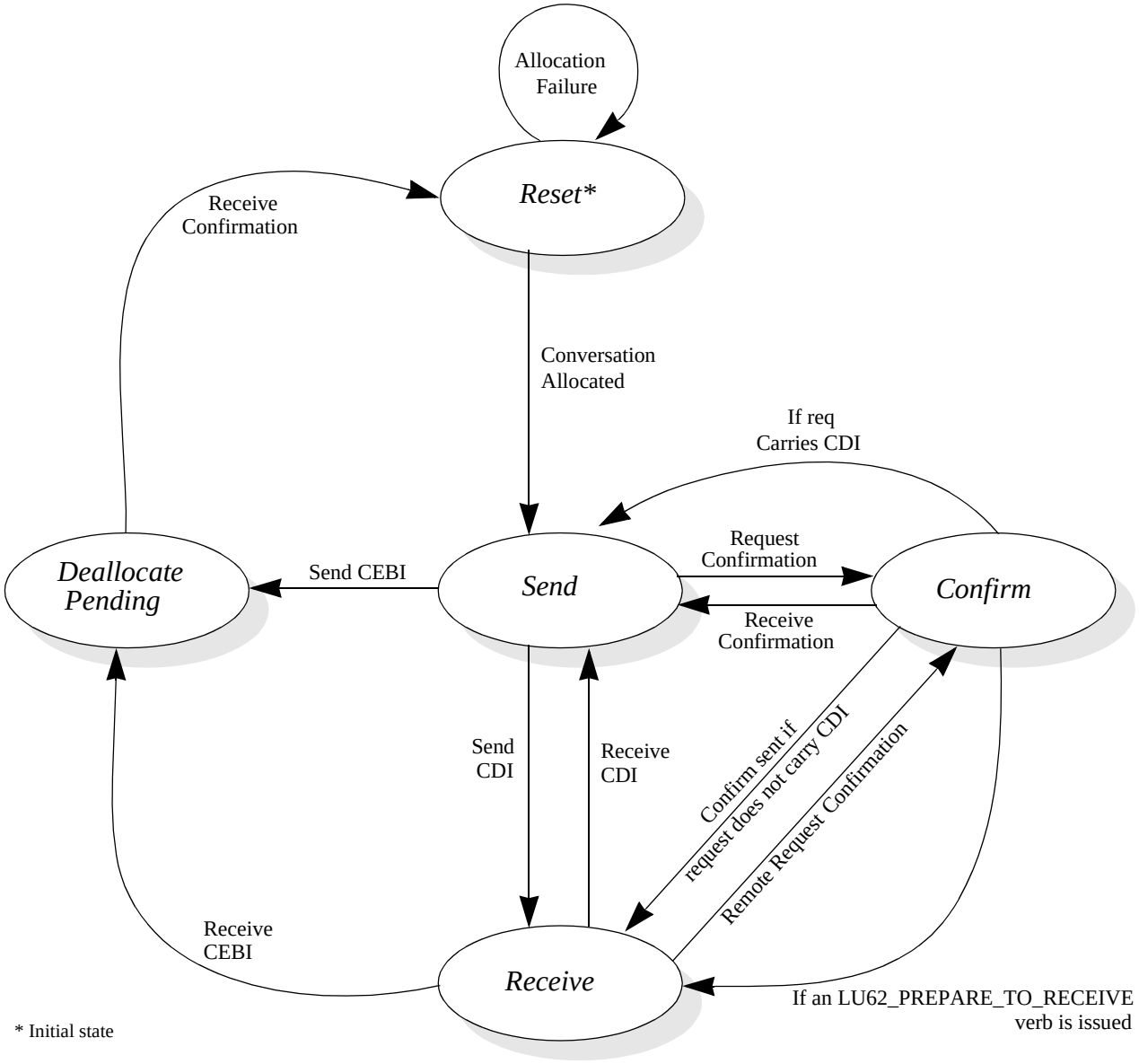


Figure 1 State Machine Diagram for LU 6.2

Understanding the LU 6.2 State Machine

Reset state: Starting a conversation

Figure 1 shows that the initial state for the LU 6.2 software is *Reset.* When an *lu62_allocate* verb is successfully executed, this starts a conversation and places the local LU in the *Send* state.

Receive state

The *Receive* state is entered when the transaction program issues the *lu62_prepare_to_receive* verb, causing the server to send a CDI to the remote transaction program.

Send state

The *Send* state is entered when the remote transaction program issues an *lu62_prepare_to_receive* verb and the local transaction program receives the CDI. While in this state, the local transaction program may send data to the remote transaction program.

Confirm state

The *Confirm* state is entered from the *Send* state when the local transaction program issues an *lu62_confirm* verb. The local transaction program cannot send additional data while in this state. The *Confirm* state is exited and the state is returned to the *Send* state when confirmation is received from the remote transaction program. If the *Confirm* state was entered as a result of an *lu62_prepare_to_receive* verb, upon receipt of confirmation, the *Receive* state is entered.

This state may also be entered while in *Receive* state when the remote transaction program requests confirmation. The *lu62_confirm* verb must be executed before any other action by the local transaction program. This *Confirm* state is exited to *Receive* state if the remote transaction program did not send CDI. This *Confirm* state is also exited to *Send* state if the remote transaction program sent CDI.

Deallocate Pending state

The *Deallocate Pending* state is entered when the local transaction program sends or receives a Conditional End Bracket Indicator (CEBI) indicating termination of the conversation. Upon receipt or sending of a confirmation to the CEBI, the *Reset* state is entered.

Sending Confirmations

In all but a very few cases, the LU6.2 API requires that applications confirm receipt of protocol objects. The following guidelines determine what kind of acknowledgements are required and when.

No confirmation When the *what_received* parameter of a receive routine returns with either DEALLOC_ABEND, LU62_BIS, LU62_SBI, DEALLOCATE_CONFIRM, or CONFIRMED, the application will not need to issue a confirmation.

Deallocation Requests When the what_received parameter of a receive routine returns CONFIRM_DEALLOCATE, the application must issue an *lu62_confirmed()* with *type* set to CONFIRM_DEALLOCATE. When *what_received* is CONFIRM_DEALLOCATE, data may have accompanied the deallocate request. Check buffers before discarding them.

Expedited Data The application should confirm expedited data by calling *lu62_confirmed()* with *type* set to CONFIRM_EXPEDITED_DATA.

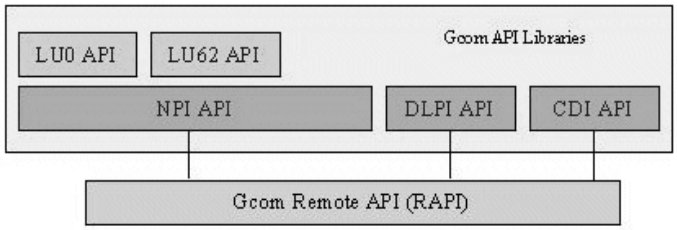
All Other Cases All other items require a call to *lu62_confirmed()* with a *type* CONFIRM_DATA.

Gcom Remote API

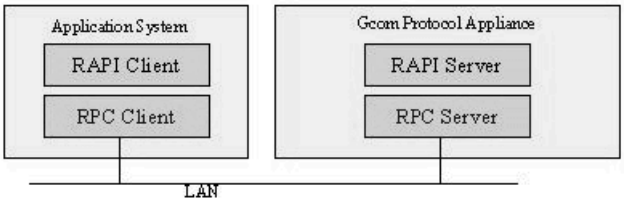
GCOM'S Remote API (RAPI) is a library of functions that allows the standard GCOM APIs to operate on protocol stacks that are configured and running on a remote machine. It is especially useful in situations in which the application code resides on a server system and the protocol processing is performed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

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 LU6.2 protocol data streams, use the functions:

Open routine: *lu62_open*
Close routine: *lu62_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.

2

API-provided Verbs

The LU6.2 API provides access to the features of LU 6.2 with a collection of C language functions. These functions correspond to the LU6.2 verbs.

lu62_allocate

```
int lu62_allocate (
allocate_data_t *allocate_data,
int resource,
lu62_allocate_t wait_object);
```

Description

The *lu62_allocate* verb allocates a conversation between the local transaction program and a remote (partner) transaction program to a session between the local LU and the remote LU. This verb is issued immediately after a call to *lu62_open* and before sending any data on the conversation. When the verb completes, the API will be in the *Send* state.

Parameters

allocate_data

This structure contains the LU name, TP name, and other information necessary to start a conversation. The structure is defined as follows:

```
typedef struct allocate_data
{
    int             sync_level;
    int             type;
    int             tpn_length;
    unsigned char   tpn[64];
    unsigned char   lu_name[9];
    unsigned char   mode[9];
    unsigned char   userid[10];
    unsigned char   passwd[10];
} allocate_data_t;
```

The Transaction Program Name (*tpn*) must be specified as an EBCDIC string. See “allocate_data_t” on page 72 for additional details.

resource

Specifies the resource for which this operation will be performed.

wait_object

Either **BLOCKING** to block, **NULL** to ignore the results of completion, or a pointer to a function to call when the request completes.

Return Values

>0

OK response. See “Return Codes” on page 80 for more information.

<=0

Error return. See “Return Codes” on page 80 for more information.

lu62_allocate_luw

```
int lu62_allocate_luw (
allocate_data_t *allocate_data,
int resource,
lu62_allocate_t wait_object);
```

Description

The *lu62_allocate_luw* verb allocates a conversation between the local transaction program and a remote (partner) transaction program to a session between the local LU and the remote LU. This verb is issued immediately after a call to *lu62_open* and before sending any data on the conversation. When the verb completes, the API will be in the *Send* state unless the change direction indicator was set on the attach request.

This function is the same as the *lu62_allocate* verb except that the LU62 server will add the logical unit of work subfield to the attach request.

Parameters

allocate_data

This structure contains the LU name, TP name, and other information necessary to start a conversation. The structure is defined as follows:

```
typedef struct allocate_data
{
      int sync_level;
      int type;
      int tpn_length;
      unsigned char tpn[64];
      unsigned char lu_name[9];
      unsigned char mode[9];
      unsigned char userid[10];
      unsigned char passwd[10];
      unsigned char luw_instance_num_len;
      unsigned char luw_instance_num[6];
      unsigned char luw_seq_num_len;
      unsigned char luw_seq_num[2];
      unsigned char correlator_field_len;
      unsigned char correlator_field[8];
      unsigned char attach_seq_num_field_len;
      unsigned char attach_seq_num_field[8];
} allocate_data_t;
```

NOTE: The API assumes that the fields of the logical unit of work subfield are set up correctly. Please see the SNA Formats manual p11-11,12 for the formats of those fields.

The Transaction Program Name (tpn) must be specified as an EBCDIC string. See "allocate_data_t" on page 72 for additional details.

resource

Specifies the resource for which this operation will be performed.

wait_object

Either **BLOCKING** to block, **NULL** to ignore the results of completion, or a pointer to a function to call when the request completes.

Return Values

>0

OK response. See "Return Codes" on page 80 for more information.

<=0

Error return. See "Return Codes" on page 80 for more information.

lu62_api_set_debug_level

```
void lu62_api_set_debug_level (int level);
```

Description

This routine sets the level and amount of the debugging information generated by the API. Presently, the values 0 and 1 are recognized. The routine should only ever require a single call. Information generated by this option will mostly be useful to GCOM technicians.

Parameters

level

An integer representing the new debugging level.

Return Values

>0

OK response. See “Return Codes” on page 80 for more information.

<=0

Error return. See “Return Codes” on page 80 for more information.

lu62_change_session_limit

```
int lu62_change_session_limit (
     int                    resource,
     unsigned char         *mode_name,
     int                    lu_mode_session_limit,
     int                    min_conwinners_source,
     int                    min_conwinners_target,
     int                    responsible,
     int                    action,
     lu62_session_limit_t   wait_object);
```

Description

This routine changes the session limit and contention-winer polarities for parallel session connections. The verb acts on a group of sessions associated with the specified mode name and a remote and local LU. The new session limit and contention winner polarities are enforced until changed by a subsequent CNOS verb. LU-LU sessions may be deactivated or activated with this verb.

Parameters

resource

The resource for which the changes should be made.

mode_name

The mode name for which the changes should be made.

lu_mode_session_limit

The new session limit.

min_conwinners_source

The minimum number of contention winners for the source.

min_conwinners_target

The minimum number of contention winners for the target.

responsible

Either **LU62SOURCE** or **LU62TARGET**, depending on who would be responsible for closing down extra sessions.

action

If 0, set the new session limits. If 2, close the sessions on this LU/mode pair.

wait_object

Either **BLOCKING** to block, **NULL** to ignore the results of completion, or a pointer to a function to call when the request completes.

Return Values

>=0

OK response. See “Return Codes” on page 80 for more information.

<0

Error return. See “Return Codes” on page 80 for more information.

lu62_close

```
void lu62_close (int resource);
```

Description

This verb is called when the LU6.2 server file descriptor (*resource*) is no longer needed. The resource variable specifies which file descriptor to close.

Parameter

resource

The LU6.2 resource to close.

Return Values

>0

OK response. See “Return Codes” on page 80 for more information.

<=0

Error return. See “Return Codes” on page 80 for more information.

lu62_confirm

```
int  lu62_confirm (
        int                 resource,
        int                 confirm_type,
        int                *request_to_send,
        int                *expedited_data_received,
        lu62_confirm_t      wait_object);
```

Description

The *lu62_confirm* verb flushes the API's send buffer for this resource, sending the accumulated requests to the target with a request for confirmation of receipt. This verb can only be issued in the Send state. If the application previously issued an *lu62_prepare_to_receive* verb, then when this verb completes, the API will be placed in *Receive* state. If the application did not place a call to *lu62_prepare_to_receive*, the API will remain in *Send* state.

Parameters

resource

Specifies the resource ID.

confirm_type

The type of confirmation requested. Either **CONFIRM_RQD2** or **CONFIRM_RQD3**.

request_to_send

Set to **LU62_YES** if target has requested permission to transmit. Set to **LU62_NO** otherwise.

expedited_data_received

Set to **LU62_YES** if expedited data is available. Set to **LU62_NO** otherwise.

wait_object

Either **BLOCKING** to block, **NULL** to ignore the results of completion, or a pointer to a function to call when the request completes.

Return Values

>0

OK response. See "Return Codes" on page 80 for more information.

<=0

Error return. See "Return Codes" on page 80 for more information.

lu62_confirmed

```
int  lu62_confirmed (
                     int  resource,
                     int  sense,
                     int  type);
```

Description

The *lu62_confirmed* verb sends a confirmation reply to the remote transaction program. The local program should issue this verb according to the rules described under “Sending Confirmations” on page 13.

Parameters

resource

Specifies the resource ID.

sense

Zero if request confirmed. Otherwise, it contains the sense code sent by the remote TP.

type

One of **CONFIRM_DATA** or **CONFIRM_EXPEDITED_DATA** depending on what type of confirmation was requested.

Return Values

>0

OK response. See “Return Codes” on page 80 for more information.

<=0

Error return. See “Return Codes” on page 80 for more information.

lu62_deallocate

```
int  lu62_deallocate (int                 resource,
                      int                 confirm_type,
                      int                *log_data,
                      lu62_deallocate_t   wait_object);
```

Description

The *lu62_deallocate* verb deallocates the specified conversation from the transaction program. This allows an application to close down individual conversations.

Prototype

resource

Specifies the variable containing the resource ID of the conversation to be deallocated.

confirm_type

Specifies the variety of confirmation desired. Either **CONFIRM_RQD2** or **CONFIRM_RQD3** for confirmation or errors only with **FLUSH_RQE2** or **FLUSH_RQE3**.

log_data

Reserved for future use. Must be **NULL**.

wait_object

Either **BLOCKING** to block, **NULL** to ignore the results of completion, or a pointer to a function to call when the request completes.

Return Values

>0

OK response. See “Return Codes” on page 80 for more information.

<=0

Error return. See “Return Codes” on page 80 for more information.

lu62_flush

```
int  lu62_flush (int           resource,
                 int           confirm_type,
                 lu62_flush_t  wait_object);
```

Description

The *lu62_flush* verb flushes the local LU's send buffer. The LU sends any information presently buffered to the remote LU. This buffered information may be a result of calls to *lu62_allocate*, *lu62_send_data*, *lu62_prepare_to_receive*, or *lu62_send_error*. The information is sent with an exception-only request, meaning that the target will only report errors. If the transmission is successful, no notification will be sent.

Parameters

resource

Specifies the resource ID.

confirm_type

Specifies confirmation on error. Either **FLUSH_RQE2** or **FLUSH_RQE3**.

wait_object

Either **BLOCKING** to block, **NULL** to ignore the results of completion, or a pointer to a function to call when the request completes.

Return Values

>0

OK response. See "Return Codes" on page 80 for more information.

<=0

Error return. See "Return Codes" on page 80 for more information.

lu62_get_attributes

```
int  lu62_get_attributes (int    resource,
                          char  *partner_lu_name,
                          char  *partner_fq_lu_name,
                          int    sync_level);
```

Description

The *lu62_get_attributes* verb returns information concerning the specified conversation.

Parameters

resource

Specifies the resource ID.

partner_lu_name

Partner LU name.

partner_fq_lu_name

Partner LU's fully qualified name. If not known, **NULL** is returned.

sync_level

Specifies the synchronization level that the local and remote programs can use on this conversation. Levels **NONE** and **CONFIRM** are currently supported.

Return Values

>0

OK response. See "Return Codes" on page 80 for more information.

<=0

Error return. See "Return Codes" on page 80 for more information.

lu62_init_log

```
int lu62_init_log (int   log_options,
                   char *log_file_name);
```

Description

The *lu62_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 *lu62_open()* to the default values described below. Refer to “Logging Modes” on page 69 for further logging option information.

The NPI Streams Interface is used by the LU6.2 API to transfer messages to and from the GCOM SNA server software module. The NPI logging facility is used to log interesting events which occur in the API during the application’s execution.

Note: *The* lu62_init_log() *procedure should be called only once within your application, just prior to calling* lu62_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 Modes” on page 69 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

>0

OK response. See “Return Codes” on page 80 for more information.

<=0

Error return. See “Return Codes” on page 80 for more information.

lu62_initialize_session_limit

```
int lu62_initialize_session_limit (
        int                      resource,
        char                    *mode_name,
        int                      lu_mode_session_limit,
        int                      min_conwinners_source,
        int                      min_conwinners_target,
        int                      action,
        allocate_data_t         *allocate_data,
        lu62_init_sess_limit_t   wait_object);
```

Description

This routine sets up the initial LU-LU session limit for single- or parallel-session connections, and the contention winner polarities for parallel-session connections. The verb applies to the group of sessions between the source and target LUs with the specified mode name. The session limit and contention winning polarities are enforced until changed by a subsequent CNOS verb. As a consequence of initializing the session limit, one or more LU-LU sessions with the specified mode name may be activated.

Parameters

resource

The resource to initialize the session limit for. Should be a file descriptor connected to the target LU.

mode_name

The mode table entry name that both the source and target LUs are using to manage the session(s).

lu_mode_session_limit

Maximum number of sessions.

min_conwinners_source

Minimum number of contention-winning sessions for the local LU to have available.

min_conwinners_target

Minimum number of contention-winning sessions for the remote LU to have available.

action

0 to set a new session limit, 2 to close all conversations.

allocate_data

A pointer to a completed *allocate_data* struct. This structure is as follows:

```
typedef struct allocate_data
{
   int             sync_level;
   int             type;
   int             tpn_length;
   unsigned char   tpn[64];
   unsigned char   lu_name[9];
   unsigned char   mode[9];
   unsigned char   userid[10];
   unsigned char   passwd[10];
} allocate_data_t;
```

The transaction program name (tpn) must be formatted as an EBCDIC string. See Chapter 4 starting on page 67 for further details.

wait_object

Either **BLOCKING** to block, **NULL** to ignore the results of completion, or a pointer to a function to call when the request completes.

Return Values

>0

OK response. See “Return Codes” on page 80 for more information.

<=0

Error return. See “Return Codes” on page 80 for more information.

lu62_initialize_session_limit_luw

```
int lu62_initialize_session_limit_luw (
      int resource,
      char *mode_name,
      int lu_mode_session_limit,
      int min_conwinners_source,
      int min_conwinners_target,
      int action,
      allocate_data_t *allocate_data,
      lu62_init_sess_limit_t wait_object);
```

Description

This routine sets up the initial LU-LU session limit for single- or parallel-session connections, and the contention winner polarities for parallel-session connections. The verb applies to the group of sessions between the source and target LUs with the specified mode name. The session limit and contention winning polarities are enforced until changed by a subsequent CNOS verb. As a consequence of initializing the session limit, one or more LU-LU sessions with the specified mode name may be activated.

This function is the same as the *lu62_initialize_sessions* verb except that the LU62 server will add the logical unit of work subfield to the attach request.

Parameters

resource

The resource for initializing the session limit. Should be a file descriptor connected to the target LU.

mode_name

The mode table entry name that both the source and target LUs are using to manage the session(s).

lu_mode_session_limit

Maximum number of sessions.

min_conwinners_source

Minimum number of contention-winning sessions for the local LU to have available.

min_conwinners_target

Minimum number of contention-winning sessions for the remote LU to have available.

action

0 to set a new session limit, 2 to close all conversations.

allocate_data

A pointer to a completed allocate_data struct. This structure is as follows:

```
typedef struct allocate_data
{
      int sync_level;
      int type;
      int tpn_length;
      unsigned char tpn[64];
      unsigned char lu_name[9];
      unsigned char mode[9];
      unsigned char userid[10];
      unsigned char passwd[10];
      unsigned char luw_instance_num_len;
      unsigned char luw_instance_num[6];
      unsigned char luw_seq_num_len;
      unsigned char luw_seq_num[2];
      unsigned char correlator_field_len;
      unsigned char correlator_field[8];
      unsigned char attach_seq_num_field_len;
      unsigned char attach_seq_num_field[8];
} allocate_data_t;
```

NOTE: The API assumes that the fields of the logical unit of work sub-field are set up correctly. Please see the SNA Formats manual (GA27-3136) p11-11,12 for the formats of those fields.

The Transaction Program Name (tpn) must be specified as an EBCDIC string. See Chapter 4 starting on page 67 for further detaiils.

wait_object

Either **BLOCKING** to block, **NULL** to ignore the results of completion, or a pointer to a function to call when the request completes.

Return Values

>0

OK response. See "Return Codes" on page 80 for more information.

<=0

Error return. See "Return Codes" on page 80 for more information.

lu62_open

```
Old: int lu62_open (
(void)
   int                  *resource,
   lu62_connect_t        connect_object,
   lu62_disconnected_t   disconnect_object,
   unsigned char        *lu_name,
   unsigned char        *mode_name,
   unsigned int          tpn_length,
   unsigned char        *tpn_name,
   int                   pu,
   int                   buffer_size,
   unsigned char        *buffer);

New: extern int lu62_open_host (char *hostname
   int                  *resource,
   lu62_connect_t        connect_object,
   lu62_disconnected_t   disconnect_object,
   unsigned char        *lu_name,
   unsigned char        *mode_name,
   unsigned int          tpn_length,
   unsigned char        *tpn_name,
   int                   pu,
   int                   buffer_size,
   unsigned char        *buffer);
```

Description

This verb opens the stream between the LU6.2 application and the kernel-level protocol code, requests a connection with the appropriate LU/mode pair so that a session can be then started with an attach request. This verb is called either at the start of a conversation between a local application and a remote transaction program (TP) or between a local transaction program and a remote application.

If this verb is called to initiate a conversation, the *lu_name* and *mode_name* parameters must point to null-terminated ASCII character strings. The *tpn_length* parameter must be set to 0. The *tpn_name* parameter is not examined.

If this verb is called to register a transaction program so it can receive attach requests, the *lu_name* and *mode_name* parameters must point to a null character, and the *tpn_length* parameter must be set to the length of the *tpn_name*. The *tpn_name* parameter must point to an EBCDIC string containing the transction program name. This string must be in EBCDIC, and cannot be a simple null-terminated string, since TP names may start with a NULL.

If multiple instances of the LU6.2 server are active, the *pu* parameter must be set to the value of the *sp_pu_id* field in the configuration file so that the open request is routed to the correct instance.

Parameters

resource

The file descriptor of the stream opened. This will be repeatedly used.

connect_object

A callback routine to be invoked when the connection to the source LU/mode pair has been established. May be **NULL** to disregard the event, but may not be **BLOCKING**.

disconnect_object

A callback routine to be invoked in the event that the connection attempt fails or an established connection is disconnected. May be **NULL** to disregard the event, but may not be **BLOCKING**.

lu_name

Name for the local LU. ASCII null-terminated string.

mode_name

Name for the local mode. ASCII null-terminated string.

tpn_length

Length of the transaction program name.

tpn_name

Name of the transaction program. Must be in EBCDIC, and cannot be null-terminated, since the name may start with a null character.

pu

The PU ID number. Should match the configured value of *sp_pu_id* for the instance of the LU6.2 Server to connect with.

buffer_size

The size of the data buffer pointed to by buffer.

buffer

This points to a buffer to be used for receiving request units from the remote TP. This buffer must be large enough to accomodate the largest chain of request units that the remote TP may send.

Return Values

>0

OK response. See “Return Codes on page 80 for more information.

<=0

Error return. See “Return Codes on page 80 for more information.

lu62_prepare_to_receive

```
int  lu62_prepare_to_receive (
    int                          resource,
    int                          confirm_type,
    lu62_prepare_to_receive_t    wait_object,
    int                         *request_to_send,
    int                         *expedited_data_received);
```

Description

The *lu62_prepare_to_receive* verb changes conversation from the *Send* to *Receive* state in preparation to receive data. The change to *Receive* state can be completed either as part of this verb or deferred until the transaction program issues an *lu62_flush* or an *lu62_confirm* verb. The message currently being prepared for transmission will have the change direction indicator (CDI) set.

When the *confirm_type* parameter has a nonzero value, the transition to the *Receive* state is completed as part of this verb. When this verb completes the transition, it performs the function of the *lu62_flush* or *lu62_confirm* verb.

If the *confirm_type* parameter has a zero value, the transition to *Receive* state will be delayed until an *lu62_flush* or an *lu62_confirm* verb is issued.

Parameters

resource

Specifies the resource ID.

confirm_type

Specifies the variety of confirmation desired. Either **CONFIRM_RQD2** or **CONFIRM_RQD3** for confirmation or errors only with **FLUSH_RQE2** or **FLUSH_RQE3**.

wait_object

Either **BLOCKING**, which specifies a blocking routine, **NULL**, which will ignore the completion status, or a pointer to a user-defined routine to call when the request is completed.

request_to_send

Set to **LU62_YES** if target has requested permission to transmit. Set to **LU62_NO** otherwise.

expedited_data_received

Set to **LU62_YES** if expedited data is available. Set to **LU62_NO** otherwise.

Return Values

>0

OK response. See “Return Codes” on page 80 for more information.

<=0

Error return. See “Return Codes” on page 80 for more information.

lu62_receive_and_wait

```
int  lu62_receive_and_wait (
              int               resource,
              int               fill,
              int              *length,
              int              *request_to_send,
              int              *expedited_data_received,
              char            **data,
              int              *what_received);
```

Description

The *lu62_receive_and_wait* verb waits for information to arrive on the specified conversation and then receives the information. If information is already available, the transaction program returns it without waiting. Otherwise, this verb always blocks. The information can be data, conversation status or a request for confirmation . Control is returned to the transaction program with an indication of the type of information in *what_received.*

The transaction program can issue this verb when in the *Send* state. The LU flushes its send buffer, sending all buffered information and the SEND indication to the remote program. This changes the conversation to the *Receive* state. The LU then waits for information to arrive. The remote program can send data to the local LU after receiving the SEND indication.

Parameters

resource

Specifies the resource ID.

fill

Specifies whether posting is to occur in terms of the logical record format of the data. The following values are possible:

- **LL**. Logical record.
- **BUFFER**. When data received is at least equal to the length specified by the *length* parameter or the end of data, which ever is first.

length

Specifies the maximum length of data that the transaction program can receive. When control is returned to the program, this variable indicates the actual amount of data received up to the maximum. If the program receives information other than data, this variable remains unchanged.

Note: *The maximum size buffer that the API can receive is 6134 bytes.*

request_to_send

Set to **LU62_YES** if target has requested permission to transmit. Set to **LU62_NO** otherwise.

expedited_data_received

Set to **LU62_YES** if expedited data is available. Set to **LU62_NO** otherwise.

data

Pointer to the data buffer pointer. The buffer space pointed at will be swapped with a full buffer if data is available.

what_received

Returns one of the following indications of what the transaction program received:

- **DATA**. Indicated when *fill*(BUFFER) is specified and data is received by the program.
- **SEND**. Indicated when the remote program has entered the *Receive* state. The local program can now issue *lu62_send_data*.
- **CONFIRM**. Indicated when the remote program has issued an *lu62_confirm*. The local program should respond with either an *lu62_confirmed* or an *lu62_send_error*.
- **CONFIRM_SEND**. Indicates that the remote program has issued *lu62_prepare_to_receive* with *type*(CONFIRM). The local program should respond with an *lu62_confirmed* or an *lu62_send_error* verb.
- **CONFIRM_DEALLOCATE.** This is returned when the peer application sends a data request with the Conditional End Bracket Indicator (CEBI) bit set in the request header (RH). The peer is requesting that the conversation be closed.
- **CONFIRM_*xxx***. The *xxx* extension indicates what type of confirmation the peer application requested. RE2 and RE3 are exception response only. RD2 and RD3 require a confirmation response be sent in response to the data request. They are all confirmation requests, but some applications need to know the type of confirmation requested by the peer in order to maintain synchronization of resources. In any event, the receiving application MUST confirm the received request. The LU62 server will determine if a response must be sent to the requesting application.
- **DEALLOCATE_ABEND**. The peer application has terminated abnormally. The local application should clean up its resources, close the stream, and retry the conversation.
- **ERROR, ERROR_SEND**. Both indicate an error condition was found. ERROR_SEND indicates that the peer application sent the

data with the Change Direction Indicator (CDI) bit set in the RH. This puts the receiving application into send state. If an error indication is received, the receiving application will transition to receive state.

Return Values

>0

OK response. See “Return Codes” on page 80 for more information.

<=0

Error return. See “Return Codes” on page 80 for more information.

lu62_receive_expedited_data

```
int  lu62_receive_expedited_data (
  int                               resource,
  int                               return_control,
  lu62_receive_expedited_data_t     wait_object,
  int                              *length,
  int                              *request_to_send,
  int                              *expedited_data_received,
  char                             *data);
```

Description

The *lu62_receive_expedited_data* verb receives data sent by the remote transaction program in an expedited manner, via the *lu62_send_expedited_data* verb. The transaction program can issue this verb in any conversation state except the *Reset* state.

In addition to the standard usage, the transaction program can also issue the *lu62_receive_expedited_data* verb to test for the presence of expedited data. This can be done by setting *length* to zero and *return_control* to **IMMEDIATE**. If expedited data is available, the return code **INSUFFICIENT_BUFFER_SIZE** is returned to the transaction program. The transaction program can then decide what to do next. If expedited data is not available, a return code of **UNSUCCESSFUL** is returned and the length field is not meaningful.

Parameters

resource

Specifies the resource ID.

return_control

Specifies when the local LU is to return control to the local program, in relation to the receipt of expedited data. The following values are valid:

- **WHEN_EXPEDITED_DATA_RECEIVED**. Specifies when expedited data is received. If the *wait_object()* function is to be called and the *wait_object* parameter must be set to a valid function pointer.
- **IMMEDIATE**. Specifies to receive the expedited data for the conversation and return control to the transaction program with a return code indicating whether or not data has been received.

wait_object

Specifies whether the verb should be processed in non-blocking mode. If *return_control* specifies **WHEN_EXPEDITED_DATA_RECEIVED**, then *wait_object* must specify a valid callback function. Otherwise, this value is not checked.

length

Specifies the maximum amount of data that the transaction program is to receive. When control is returned to the transaction program, this variable indicates the actual amount of data received up to the maximum.

Valid expedited data lengths may vary from 0-86 bytes.

If the transaction program receives information other than data, this variable remains unchanged. If the length provided is smaller than the data received or is set to zero, a return code of **INSUFFICIENT_BUFFER_SIZE** is returned to the transaction program.

request_to_send

Set to **LU62_YES** if target has requested permission to transmit. Set to **LU62_NO** otherwise.

expedited_data_received

Set to **LU62_YES** if expedited data is available. Set to **LU62_NO** otherwise.

data

Pointer to data buffer.

Return Values

>0

OK response. See “Return Codes” on page 80 for more information.

<=0

Error return. See “Return Codes” on page 80 for more information.

lu62_receive_immediate

```
int  lu62_receive_immediate (
       int                        resource,
       int                        in_poll_bits,
       int                        fill,
       int                       *length,
       int                       *request_to_send,
       int                       *expedited_data_received,
       char                     **data,
       int                       *what_received );
```

Description

The *lu62_receive_immediate* verb receives any information that is available from the specified conversation, but does not wait for the information to arrive. The information (if any) can be data, conversation status or a request for confirmation. Control is returned to the transaction program with an indication of whether any information was received and, if so, its type.

When the send-receive queue for half-duplex conversations is not empty, the *lu62_receive_immediate* verb can be blocked just like the *lu62_receive_and_wait* verb. The *lu62_receive_and_wait* verb waits for information to arrive on the specified conversation and then returns the information. If information is already available, the transaction program receives it without waiting. Otherwise, the *lu62_receive_and_wait* verb always blocks.

Parameters

resource

Specifies the resource ID.

in_poll_bits

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

fill

Specifies whether posting is to occur in terms of the logical record format of the data. The following values are possible:

- **LL**. Logical record.
- **BUFFER**. When data received is at least equal to the length specified by the *length* parameter or the end of data, which ever is first.

length

Specifies the maximum amount of data that the transaction program is to receive. When control is returned to the program, this variable indicates the actual amount of data received up to the maximum. If the program receives information other than data, this variable remains unchanged.

Note: *The maximum size buffer that the API can receive is 6134 bytes.*

request_to_send

Set to **LU62_YES** if target has requested permission to transmit. Set to **LU62_NO** otherwise.

expedited_data_received

Set to **LU62_YES** if expedited data is available. Set to **LU62_NO** otherwise.

data

Pointer to a data buffer pointer. If the API has data to receive, the API will retain this empty buffer. When the call returns, the pointer at this address will point to the received data. If no data is available, the pointer at this address will still point to the empty buffer.

what_received

Returns one of the following indications of what the transaction program received:

- **DATA**. Indicated when **BUFFER** is specified for *fill* and data is received by the program that exceeds the size of *length.*
- **SEND**. Indicated when the remote program has entered the *Receive* state. The local program can now issue *lu62_send_data*. Data may be present.
- **CONFIRM**. Indicated when the remote program has issued an *lu62_confirm*. The local program should respond with either an *lu62_confirmed* or an *lu62_send_error*.
- **CONFIRM_SEND**. Indicates that the remote program has issued *lu62_prepare_to_receive* with a *type* **CONFIRM**. The local program should respond with an *lu62_confirmed* or an *lu62_send_error* verb.
- **CONFIRMED**. Indicates that the remote program has sent confirmation to a previous data transmission.
- **CONFIRM_DEALLOCATE.** This is returned when the peer application sends a data request with the Conditional End Bracket Indicator (CEBI) bit set in the request header (RH). The peer is requesting that the conversation be closed.
- **CONFIRM_*xxx***. The *xxx* extension indicates what type of confirmation the peer application requested. RE2 and RE3 are

exception response only. RD2 and RD3 require a confirmation response be sent in response to the data request. They are all confirmation requests, but some applications need to know the type of confirmation requested by the peer in order to maintain synchronization of resources. In any event, the receiving application MUST confirm the received request. The LU62 server will determine if a response must be sent to the requesting application.

- **DEALLOCATE_ABEND**. This is sent when either side determines an outaage has occurred. The outage can be the result of the peer application terminating without ending the conversation, a link down condition, or some other error that causes the local server to determine that the conversation has ended abnormally.
- **ERROR, ERROR_SEND**. Both indicate an error condition was found. ERROR_SEND indicates that the peer application sent the data with the Change Direction Indicator (CDI) bit set in the RH. This puts the receiving application into send state. If an error indication is received, the receiving application will transition to receive state.

Return Values

>0

OK response. See “Return Codes” on page 80 for more information.

<=0

Error return. See “Return Codes” on page 80 for more information.

lu62_request_to_send

```
int  lu62_request_to_send (
   int                      resource,
   lu62_request_to_send_t   wait_object );
```

Description

The *lu62_request_to_send* verb notifies the remote program that the local program is requesting to enter the *Send* state for the conversation. The conversation will be changed to the *Send* state when the local program subsequently receives a **SEND** indication from the remote program.

Parameters

resource

Specifies the resource ID.

wait_object

Either **BLOCKING** to block, **NULL** to ignore the results of completion, or a pointer to a function to call when the request completes.

Return Values

>0

OK response. See “Return Codes” on page 80 for more information.

<=0

Error return. See “Return Codes” on page 80 for more information.

lu62_send_data

```
int lu62_send_data (
             int              resource,
             unsigned char   *data,
             int              length,
             int             *request_to_send,
             int             *expedited_data_received );
```

Description

The *lu62_send_data* verb transfers data to the API. The amount of data is specified independently of the data format in the *length* parameter. The data is not transmitted to the remote transaction program until the transaction program issues an *lu62_prepare_to_receive*, *lu62_flush*, or *lu62_confirm* verb.

Parameters

resource

Specifies the resource ID.

data

Pointer to the data to be sent. The data consists of logical records. Each record starts with a two byte length field.

length

Length of the data to be sent. This length is not related to the length of the logical record. It is used to determine the length of the data pointed to by *data*.

wait_object

Either **BLOCKING** to block, **NULL** to ignore the results of completion, or a pointer to a function to call when the request completes.

request_to_send

Set to **LU62_YES** if target has requested permission to transmit. Set to **LU62_NO** otherwise.

expedited_data_received

Set to **LU62_YES** if expedited data is available. Set to **LU62_NO** otherwise.

Return Values

>0

OK response. See “Return Codes on page 80 for more information.

<=0

Error return. See “Return Codes on page 80 for more information.

lu62_send_data_record

```
lu62_send_data_record (
            int         resource,
            unsigned char *data,
            int         length,
            int         *request_to_send,
            int         *expedited_data_received)
```

Description

lu62_send_data_record allows the application to send logical records as individual chain elements. The first call to this routine results in the record being sent first in chain. Subsequent calls result in the records being sent as middle of chain elements.

Upon a succesful return, the application must call lu62_flush to have the the Lu62 Server send the record to the remote application. The flush callback function must be valid in the call to this routine. The application should transition from send state to a temporary waiting state until the call back routine is invoked. This will indicate the record has been sent and the application can reclaim the buffer used. The applilcation can then transition back to send state and send another record.

To terminate the chain the application must call lu62_send_data followed by a call to lu62_confirm, lu62_flush or lu62_prepare_to_receive. This will cause the Lu62 Server to send the last record as a last in chain element. The application should transition to CONFIRM state and wait for confirmation that the record has been sent and received by the other side.

The application is responsible for insuring that the record in the data buffer is a complete record. The Lu62 server does not check the validity of the record header.

Parameters

resource

the resource on which the data record is to be sent

data

a pointer to the data buffer

length

the length (in binary) of the data buffer

request_to_send

a pointer to an integer. This is set to 0 if request to send is not received. It is set to 1 if request to send is received.

expedited_data_received

a pointer to an integer. Set to 0 if expedited data is not received. Set to 1 if expedited data is received.

Return values

0>

Success. The return code contains the bits to set in the events field of the pollfd structure.

0<=

Error return.

PROGRAM_PARAMETER_CHECK:
1 resource is invalid or not connected.
2 an acknowledgement is owed by the remote application.
3 another write is already in progress

PROGRAM_STATE_CHECK
returned if not in SEND State.

See “Return Codes” on page 80 for more information.

lu62_send_error

```
int  lu62_send_error (
             int                 resource,
             int                 type,
             int                *log_data,
             lu62_send_error_t   wait_object,
             int                *request_to_send,
             int                *expedited_data_received);
```

Description

The *lu62_send_error* verb informs the remote transaction program that the local program detected an error. If the conversation is in the *Send* state, the LU flushes its send buffer. Upon successful completion of this verb, the local program is in the *Send* state and the remote program is in the *Receive* state.

Parameters

resource

Specifies the resource ID.

type

Specifies the level of application or service error being reported. The following values are valid:

- **PROG**. Application program error is being reported.
- **SVC**. LU services detected error.

log_data

Log data in system error logs. Set to one of these values:

- **LU62_NO**. Do not log data (NULL pointer).
- **LU62_YES**. Log product-specific error information.

wait_object

Either **BLOCKING** to block, **NULL** to ignore the results of completion, or a pointer to a function to call when the request completes.

request_to_send

Set to **LU62_YES** if target has requested permission to transmit. Set to **LU62_NO** otherwise.

expedited_data_received

Set to **LU62_YES** if expedited data is available. Set to **LU62_NO** otherwise.

Return Values

>0

OK response. See “Return Codes” on page 80 for more information.

<=0

Error return. See “Return Codes” on page 80 for more information.

lu62_send_expedited_data

```
int  lu62_send_expedited_data (
             int                            resource,
             char                          *data,
             int                            length,
             int                            confirm_type,
             lu62_send_expedited_data_t     wait_object );
```

Description

The *lu62_send_expedited_data* verb sends data to the remote transaction program in an expedited manner. This means it may arrive at the remote program before data sent earlier via a send queue verb, such as *lu62_send_data*.

Prototype

resource

Specifies the resource ID.

data

Pointer to the data to be sent. The data consists of logical records. Each record starts with a two byte length field and has a maximum length of 86 bytes total.

length

Length of the data to be sent. This length is not related to the length of the logical record. It is used to determine the length of the data found at *data.*

confirm_type

Specifies the variety of confirmation desired. Either **CONFIRM_RQD2** or **CONFIRM_RQD3** for confirmation or errors only with **FLUSH_RQE2** or **FLUSH_RQE3**.

wait_object

Either **NULL** to ignore the results of completion, or a pointer to a function to call when the request completes.

Return Values

>0

OK response. See “Return Codes” on page 80 for more information.

<=0

Error return. See “Return Codes” on page 80 for more information.

3

User-defined Callbacks

The LU6.2 API informs the user of completed requests and other events through a collection of user-defined callback routines. When the event or completion occurs, the API will call the appropriate user-supplied callback routine.

lu62_allocate_t

```
typedef void (*lu62_allocate_t) (int resource,
                                 int confirmed,
                                 int sense_code);
```

Description

When the API receives a confirm in response to an allocate request, it calls this routine. In the event of a negative response, the application should keep the connection open until the remote LU has sent a *SEND_ERROR* message containing the error information before either reattempting the allocate request or closing the process.

Parameters

resource

The resource for which the allocate request was performed.

confirmed

TRUE or FALSE depending on whether the allocate request was confirmed or not confirmed.

sense_code

If confirmed == FALSE, this contains the negative response sent by the remote LU.

lu62_confirm_t

```
typedef void (*lu62_confirm_t) (int resource,
                                int request_to_send,
                                int sense_code,
                                int expedited_data);
```

Description

This call back is used to indicate to the local TPN that a confirm was received from the remote LU. If a signal request or expedited data are waiting to be delivered at the same time they will be indicated to the local transaction program which should then issue a *receive_immediate* or a *receive_and_wait* verb to get the information.

Parameters

resource

The resource to which the confirm applies.

request_to_send

TRUE if a request to send was received.

sense_code

Sense code in the event of a negative response.

expedited_data

TRUE if expedited data has been received.

lu62_confirmed_t

```
typedef void (*lu62_confirmed_t) (int resource);
```

Description

Indicates to the the local TPN that a confirmed has been received.

Parameters

resource

The resource for which the confirmed was received.

lu62_connect_t

```
typedef void (*lu62_connect_t)(int resource);
```

Description

This routine is invoked when a call to *lu62_start_connect_lu()* completes. At this point, the application program can call *lu62_flush()* to send an allocate request to the server.

Parameters

resource

The resource for which the connect request was performed.

lu62_deallocate_t

```
typedef void (*lu62_deallocate_t) (int resource,
                                   int sense_code);
```

Description

This callback indicates that a deallocate request has completed. If the type was set to CONFIRM or the conversation sync level was set to CONFIRM and the type was set to SYNC_LEVEL, this call will indicate the success or failure of the deallocate.

If the deallocate was successful, then the resource ID will be set to -1. Otherwise, the resource has not been closed and the application should keep the resource available for a send_error message from the remote LU.

Parameters

resource

Set to -1 if the deallocate succeeded, otherwise the resource for which the deallocate was requested.

sense_code

If the deallocate failed, this parameter will contain the sense code sent by the remote LU.

lu62_disconnected_t

```
typedef void (*lu62_disconnected_t) (int resource,
                                     int diagnostic);
```

Description

This callback is invoked in the event of either a failure to establish a connection or the termination of an existing session.

Parameters

resource

The resource that the disconnect occured on.

diagnostic

In the event of a failure to connect, this parameter will contain a diagnostic code indicating the reason for failure. If the disconnect is the result of terminating an existing session, this will likely be 0.

lu62_flush_t

```
typedef void (*lu62_flush_t) (int resource,
                              int sense_code);
```

Description

When a flush operation has completed, this callback will be invoked to notify the application that the data has been sent. It does not indicate that the transmission of data was successful.

If the data buffer indicated more data to come (by setting the N_SNA_MoreData flag in the buffer), or if a flush was initiated on a new connection, a call to this routine indicates that more data can now be sent.

Parameters

resource

The resource for which the flush was performed.

sense_code

In the event of a failure, the sense code indicating what failed will be provided in this parameter.

lu62_init_sess_limit_t

```
typedef void (*lu62_init_sess_limit_t) (int resource,
                                        int sense,
                                        int sub_code,
                                lu62_cnos_msg_t *cnos_msg);
```

Description

The API invokes this routine when a request to initialize the session limit has completed.

Parameters

resource

The resource for which the session limit was initialized.

sense

A sense code in the event of an error.

value	*meaning*
0x00	*Normal:* no negotiation performed
0x01	*Abnormal:* command race detected
0x02	*Abnormal:* mode name not recognized
0x03	Reserved
0x04	*Normal:* negotiated reply
0x05	*Abnormal:* (LU, mode) session limit is 0

sub_code

A second sense code reflecting the nature of the error more closely.

cnos_msg

The content of this structure will reflect the current capabilities of this *resource*'s connection. The structure is defined as follows:

```
typedef struct
{
   short           session_limit;
   short           min_conwinners_source;
   short           min_conwinners_target;
   short           responsible;
   short           action;
   unsigned char   mode_name[9];
   unsigned char   lu_name[9];
}  lu62_cnos_msg_t;
```

lu62_prepare_to_receive_t

```
typedef void (*lu62_prepare_to_receive_t) (int resource,
                                           int request_to_send,
                                                 int sense_code,
                                           int expedited_data);
```

Description

This routine will be invoked when the interface has changed its state to RECEIVE.

Parameters

resource

The resource for which the state has changed.

request_to_send

TRUE if a request is pending.

sense_code

If an error has occured, the sense code for the error will be provided in this parameter.

expedited_data

TRUE if expedited data is waiting to be received.

lu62_receive_expedited_data_t

```
typedef unsigned long (*lu62_receive_expedited_data_t)
                      (int resource, int length,
                       int request_to_send, char *data);
```

Description

This routine is invoked when the interface has received expedited data.

Parameters

resource

The resource for which the expedited data was received.

length

The length of the data received.

request_to_send

TRUE if a request to send has been received.

data

Pointer to a buffer containing the expedited data received.

lu62_request_to_send_t

```
typedef void (*lu62_request_to_send_t) (int resource);
```

Description

This routine is invoked when a request to send verb which the local TPN issued has completed successfully.

Parameters

resource

The resource for which the request to send was sent.

lu62_send_error_t

```
typedef void (*lu62_send_error_t) (int resource,
                                   int request_to_send,
                                   int expedited_data);
```

Description

This callback will be invoked when the server has transmitted the *send_error* message and been placed in the SEND state.

Parameters

resource

The resource for which the *send_error* message was transmitted.

request_to_send

TRUE if a request to send has been received.

expedited_data

TRUE if expedited data is awaiting receipt.

lu62_send_expedited_data_t

```
typedef void (*lu62_send_expedited_data) (int resource,
                                          int sense);
```

Description

The API invokes this callback to indicate that expedited data has been sent to the remote TPN. If CONFIRM was requested, the routine will be called when the remote TPN acknowledges receipt. If CONFIRM was not requested, the routine will be called as soon as the data has been transmitted.

Parameters

resource

The resource on which the expedited data was sent.

sense

If CONFIRM was requested, this parameter will be 0 if the data was successfully transmitted and acknowledged, or a sense code if the data was rejected or the request failed. If CONFIRM was not requested, this parameter will be non-0 only if the request failed, and the application will not be notified if the remote TPN rejects the data.

lu62_session_limit_t

```
typedef void (*lu62_session_limit_t) (int resource,
                                      int sense,
                                      int sub_code,
                           lu62_cnos_msg_t *cnos_msg);
```

Descripton

The API calls this routine when a request to change the number of sessions has completed.

Parameters

resource

The resource for which the number of sessions was changed.

sense

A sense code in the event of an error.

sub_code

A second sense code reflecting the nature of the error more closely.

cnos_msg

The content of this structure will reflect the current capabilities of this *resource*'s connection. The structure is defined as follows:

```
typedef struct
{
   short           session_limit;
   short           min_conwinners_source;
   short           min_conwinners_target;
   short           responsible;
   short           action;
   unsigned char   mode_name[9];
   unsigned char   lu_name[9];
}  lu62_cnos_msg_t;
```

4

API-Provided Constants and Data Structures

The LU6.2 API provides several preprocessor defines to use as constants throughout the API, and the API provides several pre-defined data structures. This section of the manual explains these specialized constants and data structures.

Logging Modes

The LU6.2 API 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.*

Return Codes

The LU62 API verbs have been designed to work closely with the poll routine. The following psuedocode sample illustrates how the mechanism might be used:

```
r = lu62_receive_immediate( . . . );
if(r <= 0)
{
        /* print error message using errno */
}
else
        xfds.events = r;
```

LU62_OK

The LU62_OK response indicates that the routine returned without error. This response can be used as a bitmask for the *events* field of a *pollfd* struct.

0

Unspecified error. Unable to complete request.

PROGRAM_PARAMETER_CHECK (-1)

One or more of the parameters supplied to the verb is invalid.

ALLOCATION_FAILURE_NO_RETRY (-3)

An *lu62_allocate()* verb failed. The reason for failure was not transient, and a retry will not be successful either.

INSUFFICIENT_BUFFER_SIZE (-4)

The buffer provided for receiving data was not large enough to hold the data awaiting receipt.

UNSUCCESSFUL (-5)

Returned when *lu62_receive_expedited_data()* is called with a timeout 0 and no expedited data is available.

PROGRAM_STATE_CHECK (-6)

The verb was called from an invalid state. Usually this would be a consequence of calling the verb out of sequence.

Sense Codes and Sub-Codes

The following sense codes can be provided in user-supplied callback routines.

Sense Codes

CNOS_OK. The CNOS verb completed successfully. The sub-code will indicate how success was achieved.

LU_MODE_LIMIT_ZERO. The LU/mode pair requested presently has a limit of 0 and the application called *lu62_change_session_limit()* instead of *lu62_initialize_session_limit()*.

LU_SESSION_LIMIT_EXCEEDED. The application attempted to allocate a conversation on an LU/mode pair that has not had its session limit initialized.

PARAMETER_ERROR. One of the parameters provided to the verb was invalid.

RESOURCE_FAILURE_NO_RETRY. A non-transient error occured while attempting to change the number of sessions.

UNRECOGNIZED_MODE_NAME. The mode name provided doesn't appear in the local mode table.

COMMAND_RACE_REJECT. Both source and target have issued a CNOS command. The target's CNOS command will take effect, and the CNOS command sent by the local transaction program has failed.

Sub-Codes

CNOS_OK

- **CNOS_OK_AS_SPECIFIED.** The CNOS command was completed with the parameters originally specified.
- **CNOS_OK_AS_NEGOTIATED.** The CNOS command was completed with a negotiated set of parameters.

PARAMETER_ERROR

- **INVALID_LU_NAME.** The LU name specified was not valid.
- **INVALID_MODE_NAME.** The mode name specified was not valid.

allocate_data_t

```
typedef struct allocate_data
{
    int                 sync_level;
    int                 type;
    int                 tpn_length;
    unsigned char       tpn[64];
    unsigned char       lu_name[9];
    unsigned char       mode[9];
    unsigned char       userid[10];
    unsigned char       passwd[10];
} allocate_data_t;
```

Elements

sync_level

The sync level for this conversation:

- NONE. No resynchronization will be possible after an error. The only possible response to an error is to close the conversation.
- CONFIRM. Resynchronization can occur on the most recently confirmed transaction.

type

What type of conversation this should be:

- LU62_BASIC_CONVERSATION
- LU62_MAPPED_CONVERSATION
- LU62_FDX_BASIC_CONVERSATION
- LU62_FDX_MAPPED_CONVERSATION

tpn_length

Length of the data in *tpn.*

tpn

Transaction program name.

lu_name

Locally-designated name for this LU.

mode

Locally-designated name for this mode.

userid

Userid for login to target. Use is application-defined.

passwd

Password for login to target. Use is application-defined.

lu62_cnos_msg_t

```
typedef struct
{
    short            session_limit;
    short            min_conwinners_source;
    short            min_conwinners_target;
    short            responsible;
    short            action;
    unsigned char    mode_name[9];
    unsigned char    lu_name[9];
} lu62_cnos_msg_t;
```

Elements

session_limit

Maximum number of sessions for this conversation.

min_conwinners_source

Minimum number of contention-winning sessions available to the source.

min_conwinners_target

Minimum number of contention-winning sessions available to the target. (Also the number of contention-losing sessions for the source.)

responsible

Either LU62SOURCE or LU62TARGET, depending on who will be responsible for closing down sessions in excess of the new limits.

action

Either 0 if the CNOS verb will set a new session limit or 2 if the CNOS verb will close down all sessions in this conversation.

mode_name

The locally-assigned mode name for this conversation.

lu_name

The locally-assigned LU name for this conversation.