IBM Communications Server for AIX or Linux

LUA Programmer's Guide

V6.4

SC23-8590-00

IBM Communications Server for AIX or Linux

LUA Programmer's Guide

V6.4

SC23-8590-00

Note:
Before using this information and the product it supports, be sure to read the general information under Appendix B, "Notices," on page 155.

First Edition (May 2009)

This edition applies to Version 6 Release 4 of Communications Server for AIX and Linux (5765-E51 and 5724-i33) and to all subsequent releases and modifications until otherwise indicated in new editions or technical newsletters.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address below.

IBM welcomes your comments. You may send your comments to the following address:
International Business Machines Corporation
Attn: Communications Server for AIX/Linux Information Development
Department AKCA, Building 501
P.O. Box 12195, 3039 Cornwallis Road
Research Triangle Park, North Carolina
27709-2195
U.S.A.

You can send us comments electronically by using one of the following methods:
- Fax (USA and Canada):
  - 1+919-254-4028
  - Send the fax to "Attn: Communications Server for AIX/Linux Information Development"
- Internet e-mail:
  - comsvrcf@us.ibm.com

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Tables

Figures

About This Book

This book is a guide for developing C-language application programs that use the Conventional Logical Unit Application (LUA) interface to communicate with a Systems Network Architecture (SNA) host computer.

This manual applies to IBM Communications Server, which is an IBM®software product that enables a server running AIX®, or a computer running Linux, to exchange information with other nodes on an SNA network.

There are three different installation variants of IBM Communications Server, depending on the hardware on which it operates:

IBM Communications Server for AIX (CS/AIX)
IBM Communications Server for AIX operates on a server running AIX Version 5.2, 5.3 or 6.1 base operating system.

IBM Communications Server for Linux (Communications Server for Linux)
IBM Communications Server for Linux, program product number 5724–i33, operates on the following:

- 32–bit Intel workstations running Linux (i686)
- 64–bit AMD64/Intel EM64T workstations running Linux (x86_64)
- IBM pSeries computers running Linux (ppc64)

IBM Communications Server for Linux on System z (Communications Server for Linux on System z)
IBM Communications Server for Linux on System z, program product number 5724–i34, operates on System z mainframes running Linux for System z (s390 or s390x).

In this book, the name Communications Server is used to indicate any of these variants, and the term "Communications Server computer" is used to indicate any type of computer running Communications Server, except where differences are described explicitly.

The IBM Communications Server for Linux or AIX implementation of LUA is based on the IBMimplementation of the Request/Response Unit Interface (RUI) in its OS/2®products (such as **Communications Server for OS/2**), with modifications for the AIX / Linux environment.

This book applies to V6.4 of Communications Server.

Who Should Use This Book

This book is intended for experienced C programmers who write Systems Network Architecture (SNA) transaction programs for systems with Communications Server. Programmers may or may not have prior experience with SNA or the communication facilities of Communications Server.

Application programmers design and code transaction and application programs that use the Communications Server programming interfaces to send and receive data over an SNA network. They should be thoroughly familiar with SNA, the remote program with which the transaction or application program communicates, and the AIX / Linux operating system programming and operating environments.

More detailed information about writing application programs is provided in the manual for each API. For additional information about Communications Server publications, see the Bibliography.

How to Use This Book

This section explains how information is organized and presented in this book.

Organization of This Book

This book is organized as follows:

- Chapter 1, "Concepts," on page 1, introduces the fundamental concepts of LUA. It is intended for programmers who are not familiar with LUA.
- Chapter 2, "Designing and Writing LUA Applications," on page 13, contains general information a programmer needs when writing LUA applications. This chapter also includes information about SNA concepts relevant to the design of LUA applications, and on compiling and linking an LUA application.
- Chapter 3, "LUA VCB Structure," on page 45, describes the structure of the Verb Control Block (VCB) used for all LUA verbs.
- Chapter 4, "RUI Verbs," on page 55, describes each RUI verb in detail. Each description includes the following: purpose, verb record format, supplied parameters and returned values, and details on how the verb interacts with other RUI verbs.
- Chapter 5, "SLI Verbs," on page 99, describes each SLI verb in detail. Each description includes the following: purpose, verb record format, supplied parameters and returned values, and details on how the verb interacts with other SLI verbs.
- Chapter 6, "Sample LUA Application," on page 147, describes the Communications Server sample LUA application that illustrates the use of LUA RUI verbs. This chapter also includes instructions for compiling, linking, and running the sample application (including the Communications Server configuration steps necessary).
- Appendix A, "Return Code Values," on page 151, lists all the possible return codes in the LUA interface in numerical order and gives their meanings.

Typographic Conventions

Table 1 shows the typographic styles used in this document.

Table 1. Typographic Conventions

Special Element	Sample of Typography
Emphasized words	**back up files before deleting**
Document title	*IBM Communications Server for AIX or Linux APPC Programmer's Guide*
File or path name	**/usr/spool/uucp/myfile.bkp**
Program or application	**snaadmin**
Command or AIX / Linux utility	**define_node**; **cd**
General reference to all commands of a particular type	**query_*** (indicates all of the administration commands that query details of a resource)
Option or flag	**-i**
Parameter or Motif field	*opcode*; *LU name*

Table 1. Typographic Conventions (continued)

Special Element	Sample of Typography
Literal value or selection that the user can enter (including default values)	`255`; `On node startup`
Constant or signal	`AP_GET_LU_STATUS`
Return value	`AP_INVALID_FORMAT`; `0`; `-1`
Variable representing a supplied value	*filename*; *LU_name*; *user_ID*
Environment variable	`PATH`
Programming verb	GET_LU_STATUS
User input	**0p1**
Computer output	**CLOSE**
Function, call, or entry point	`ioctl`
Data structure	`termios`
3270 key	ENTER
Keyboard keys	**Ctrl+D**; **Enter**
Hexadecimal value	`0x20`

Graphic Conventions

AIX, LINUX

This symbol is used to indicate the start of a section of text that applies only to the AIX or Linux operating system. It applies to AIX / Linux servers and to the IBM Remote API Client running on AIX, Linux, Linux for pSeries or Linux for System z.

WINDOWS

This symbol is used to indicate the start of a section of text that applies to the IBM Remote API Client on Windows.

▬

This symbol indicates the end of a section of operating system specific text. The information following this symbol applies regardless of the operating system.

Where to Find More Information

See the bibliography for other books in the Communications Server library, as well as books that contain additional information about topics related to SNA and AIX / Linux workstations.

Chapter 1. Concepts

This chapter introduces the fundamental concepts of LUA—the Conventional LU (Logical Unit) Application Programming Interface (API).

The topics covered in this chapter are as follows:
- What is LUA?
- Choosing which interface to use (RUI or SLI)
- LUs and sessions
- LUA verbs
- A sample LUA communication sequence
- LUA compatibility

What Is LUA?

LUA (the Conventional LU Application Programming Interface) is an API that enables you to write Communications Server applications to communicate with host applications.

The LUA interface is provided at the request/response unit (RU) level, allowing the programmer control over the Systems Network Architecture (SNA) messages sent between Communications Server and the host. It can be used to communicate with any of the LU types 0, 1, 2, or 3 at the host; it is up to the application to send the appropriate SNA messages as required by the host application.

For example, you can use LUA to write a 3270 emulation program that communicates with a host 3270 application; a simple version of this is included as a sample LUA application with Communications Server, and described in Chapter 6, “Sample LUA Application,” on page 147.

AIX, LINUX

If your Communications Server system supports SNA Gateway for communications with downstream PUs, you can also write an LUA application that acts as the SNA primary for communications with secondary LUs on these downstream PUs. This allows you to emulate a host application on the Communications Server node, or to offload processing from a host application to the Communications Server node. This function is described as “Primary RUI”; it is specific to Communications Server and may not be provided by other LUA implementations.

Choosing Which Interface to Use

LUA includes two different programming interfaces at different levels:

- The Request Unit Interface (RUI) is provided at the request/response unit (RU) level, allowing the programmer control over the Systems Network Architecture (SNA) messages sent between Communications Server and the host. It is up to the application to build and send the appropriate SNA messages as required by the host application.

  The RUI interface supports SNA Function Management Profiles 2, 3, 4, 7, and 18, and SNA Transmission Services Profiles 2, 3, 4, and 7.
- The Session-Level Interface (SLI) is a higher-level interface, allowing the programmer to work at a logical message level rather than being concerned with the detail of individual RUs. For example:
  - The session can be established and terminated with a single SLI verb (rather than with a sequence of RUI verbs corresponding to the individual RUs involved in session startup and termination).
  - The SLI library controls chaining when the application needs to send or receive data that is longer than the maximum RU length specified in the BIND.
  - For most SNA commands sent to the host, the SLI library can build the appropriate RU at the request of the application.

  The SLI interface supports SNA Function Management Profiles 3 and 4, and SNA Transmission Services Profiles 3 and 4.

An application can use only one of these interfaces for each session. For example, if it starts a session using the RUI, it cannot subsequently issue SLI verbs on that session.

You should consider the following points before deciding which API to use.

- The SLI handles some of the detail of individual RUs and their contents, simplifying the processing required in the application. The RUI requires the application to deal with each RU individually.
- The RUI provides control over the detailed contents of RUs sent to the host, and allows the use of a wide range of SNA bind profiles. The SLI does not provide the same degree of control or flexibility.

AIX, LINUX

- The RUI includes Primary RUI(AIX / Linux only), which allows you to write an application that acts as an SNA primary for communications with downstream PUs. The SLI interface does not provide this function.

LUs and Sessions

Figure 1 on page 3, shows the SNA components used for LUA communications with a host.

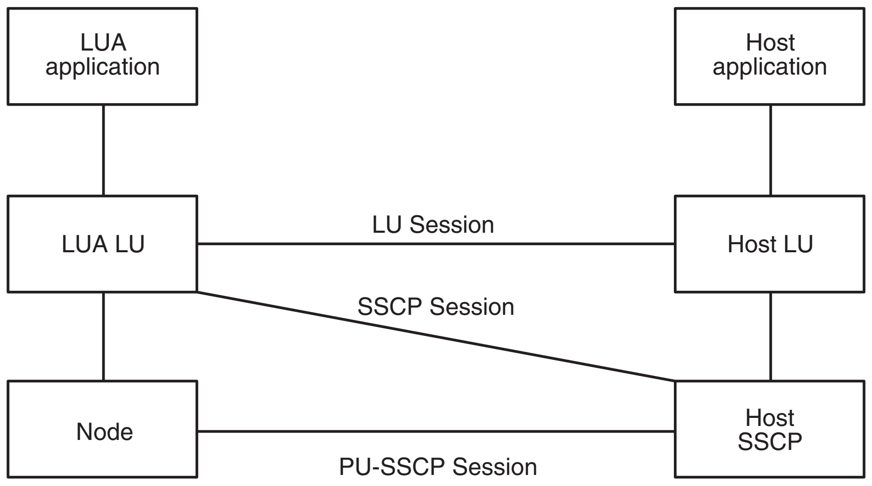

Figure 1. SNA Components Used for LUA Communications

An LUA application uses an LU of type 0–3 that communicates with the host system by means of the Communications Server node. There are three sessions between the Communications Server node and the host node, as follows:

- The physical unit-system services control point (PU-SSCP) session, between the PU 2.1 and the host's system services control point (SSCP); this is used for controlling the PU.
- The SSCP session, between the Communications Server LU and the SSCP; this is used for controlling the LU.
- The LU session, between the Communications Server LU and the host LU; this is used for data transfer between the LU and the host application.

The LUA application programming interface enables applications to send and receive data on the SSCP session and on the LU session. It does not provide access to the PU-SSCP session. An LUA application can send data on this session using the Management Services (MS) verb TRANSFER_MS_DATA; for more information, refer to the *IBM Communications Server for AIX or Linux MS Programmer's Guide*.

WINDOWS

For Windows operating systems, TRANSFER_MS_DATA is provided as part of the Common Service Verb (CSV) API; for more information, refer to the *IBM Communications Server for AIX or Linux CSV Programmer's Guide*.

AIX, LINUX

Figure 2 on page 4, shows the SNA components used for LUA communications using RUI primary to a downstream LU.

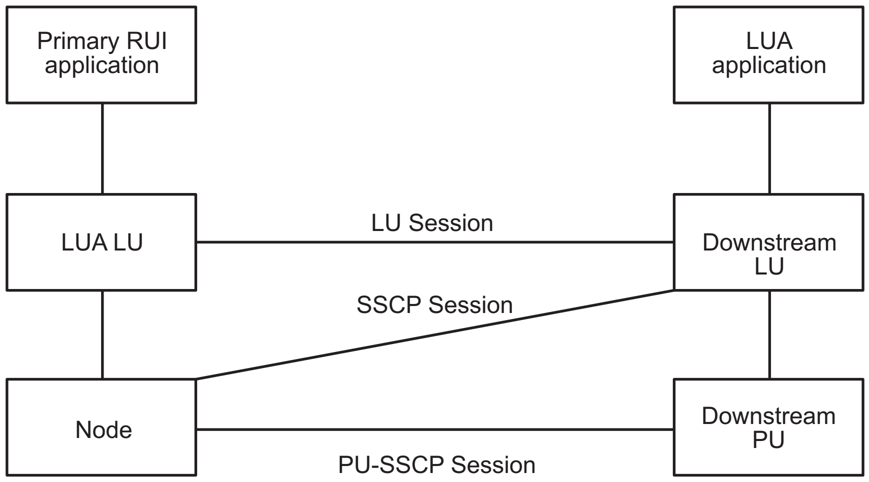

Figure 2. SNA Components Used for RUI Primary Communications

An RUI Primary application uses an LU of type 0–3 that communicates with the downstream LU by means of the Communications Server node. From the point of the downstream LU, the Communications Server LU acts as the host LU, and the Communications Server node acts as the host SSCP. The three sessions between these components, and the restrictions on access to these sessions, are equivalent to those for an LUA application communicating with a host.

[redacted]

Each of the LU sessions provides two priorities of messages: normal and expedited. Expedited flow messages take precedence over other messages waiting to be transmitted on the same session. There are four different flows on which a message can be sent or received:

- SSCP session, expedited flow
- LU session, expedited flow
- SSCP session, normal flow
- LU session, normal flow

The LU session normal flow carries application data; the other flows are used for control messages and start-up.

The Communications Server implementation of LUA does not enable applications to send data on the SSCP expedited flow, and will not return data to an application on this flow.

Configuration

Each LU used by an LUA application must be configured using the Motif administration program, the command-line administration program, or the node operator facility (NOF) API (for more information, refer to the *IBM Communications Server for AIX Administration Guide*, the *IBM Communications Server for Linux Administration Guide*, the *IBM Communications Server for AIX NOF Programmer's Guide* or the *IBM Communications Server for Linux NOF Programmer's Guide*). In addition, the Communications Server configuration may include LU pools. A pool is a group of LUs with similar characteristics, such that an application can use any free LU from the group. This can be used to allocate LUs on a first-come, first-served basis when there are more applications than LUs available, or to provide a choice of LUs on different connections.

LUA Verbs

An application accesses LUA through LUA verbs. Each verb supplies parameters to LUA, which performs the desired function and returns parameters to the application.

RUI Verb Summary

The following list contains a brief summary of each of the LUA RUI verbs (for a detailed explanation of each verb, see Chapter 4, “RUI Verbs,” on page 55):

RUI_BID
This verb enables the application to determine when information from the host is available to be read.

RUI_INIT
This verb sets up the SSCP session for an LUA application.

AIX, LINUX

RUI_INIT_PRIMARY
This verb sets up the SSCP session for an LUA application acting as the SNA primary for communications with a downstream LU.

RUI_PURGE
This verb cancels an outstanding RUI_READ verb.

RUI_READ
This verb receives data or status information sent from the host to the LUA application’s LU, on either the SSCP session or the LU session.

AIX, LINUX

RUI_REINIT
This verb re-establishes the SSCP session for an LUA application after a session failure. It is intended for use by an application that was using an LU from a pool, and needs to re-establish the session using the same LU in order to continue its processing.

RUI_TERM
This verb ends the SSCP session for an LUA application. It also brings down the LU session if it is active.

RUI_WRITE
This verb sends data to the host on either the SSCP session or the LU session.

SLI Verb Summary

The following list contains a brief summary of each of the LUA SLI verbs (for a detailed explanation of each verb, see Chapter 5, "SLI Verbs," on page 99):

SLI_BID
: This verb enables the application to determine when information from the host is available to be read.

SLI_CLOSE
: This verb ends the session for an LUA application.

SLI_OPEN
: This verb sets up the session for an LUA application.

SLI_PURGE
: This verb cancels an outstanding SLI_RECEIVE verb.

SLI_RECEIVE
: This verb receives data or status information sent from the host to the LUA application's LU, on either the SSCP session or the LU session.

SLI_SEND
: This verb sends data to the host on either the SSCP session or the LU session.

On the SLI_OPEN verb, the application can optionally specify the addresses of its own routines to process BIND, STSN, and SDT requests from the host. If it provides these routines, and a request of the appropriate type arrives from the host, LUA sends an additional verb to the appropriate application-supplied routine to allow it to process the request, as follows.

SLI_BIND_ROUTINE
: LUA sends this verb to the application-supplied BIND routine when a BIND request arrives from the host. The application can accept the BIND, negotiate BIND parameters, or reject the BIND as described in "SNA Information" on page 34.

: If the application does not provide a BIND routine, LUA performs limited BIND checking and responds to the host appropriately.

SLI_STSN_ROUTINE
: LUA sends this verb to the application-supplied STSN routine when an STSN request arrives from the host. The application can respond to the STSN or reject it with an appropriate SNA sense code, as described in "SNA Information" on page 34.

: If the application does not provide an STSN routine, LUA returns a positive response indicating that no data is available.

SLI_SDT_ROUTINE
: LUA sends this verb to the application-supplied SDT routine when an SDT request arrives from the host. The application can respond to the SDT or reject it with an appropriate SNA sense code, as described in "SNA Information" on page 34.

: If the application does not provide an SDT routine, LUA returns a positive response.

Asynchronous Verb Completion

Some LUA verbs complete quickly, after some local processing (for example the RUI_PURGE verb); however, most verbs take some time to complete, because they require messages to be sent to and received from the node or from the host application. Because of this, LUA is implemented as an asynchronous interface; control can be returned to the application while a verb is still in progress, so the application is free to continue with further processing (including issuing other LUA verbs).

AIX, LINUX

When the verb completes, LUA calls a callback routine supplied by the application. This routine may perform further processing on the returned data, issue further LUA verbs, or simply act as an indicator that the verb has completed.

- RUI verbs may complete synchronously or asynchronously. The application should check the primary return code in the VCB to determine which completion mode applies for each verb.
- SLI verbs always complete asynchronously. After issuing the verb, the application must not access the VCB until its callback routine has been called. It can process the VCB either from within the callback routine, or from the program's main thread of execution after the callback routine has completed.

WINDOWS

When the verb completes, LUA either posts a message to a window handle supplied by the application or signals an event handle supplied by the application.

For more information, see Chapter 2, "Designing and Writing LUA Applications," on page 13.

A Sample LUA Communication Sequence

Figure 3 on page 9, shows a sample LUA communication sequence using RUI verbs, and Figure 4 on page 10, shows the equivalent sequence using SLI verbs.

In the RUI example, the application performs the following steps:

1. Issues the RUI_INIT verb to establish the SSCP session. The RUI_INIT verb does not complete until Communications Server has received an activate logical unit (ACTLU) message from the host and sent a positive response; however, these messages are handled by Communications Server and not exposed to the LUA application.
2. Sends an INITSELF message to the SSCP, to request a BIND, and reads the response.
3. Reads a BIND message from the host, and writes the response. This establishes the LU session.
4. Reads an SDT message from the host, which indicates that initialization is complete and data transfer can begin.
5. Sends a chain of data consisting of three RUs (the last indicates that a definite response is required), and reads the response.
6. Reads a chain of data consisting of two RUs, and writes the response.

7. Reads an UNBIND message from the host, and writes the response. This terminates the LU session.
8. Issues the RUI_TERM verb to terminate the SSCP session. (Communications Server sends a NOTIFY message to the host and waits for a positive response; however, these messages are handled by Communications Server and not exposed to the LUA application.)

The SLI example shows the same sequence of messages flowing between the host and the application. The SLI verbs used are similar to those used in the RUI example, but note the following differences:

- SLI_OPEN handles the complete session initialization; the application does not need to read and write each individual RU in the initialization sequence, as in the RUI example.
- LUA uses the application's BIND and SDT routines (specified on SLI_OPEN) to allow the application to process the BIND and SDT messages from the host. These routines must return synchronously. All other SLI verbs complete asynchronously.
- SLI_RECEIVE and SLI_SEND handle complete chains of data, so the application needs only one verb to receive or send the data even though it is long enough to require two or three RUs. (In the RUI example, the application must receive or send each RU with a separate verb.)

The list that follows shows the abbreviations used in Figure 3 on page 9and Figure 4 on page 10.

SSCP norm	SSCP session, normal flow
LU norm	LU session, normal flow
LU exp	LU session, expedited flow
+rsp	Positive response to the indicated message
BC	Begin chain
MC	Middle of chain
EC	End chain
CD	Change direction indicator set
RQD	Definite response required

Figure 3 on page 9, shows the RUI verbs used to start a session, exchange data, and end the session, and the SNA messages sent and received. The arrows indicate the direction in which SNA messages flow.

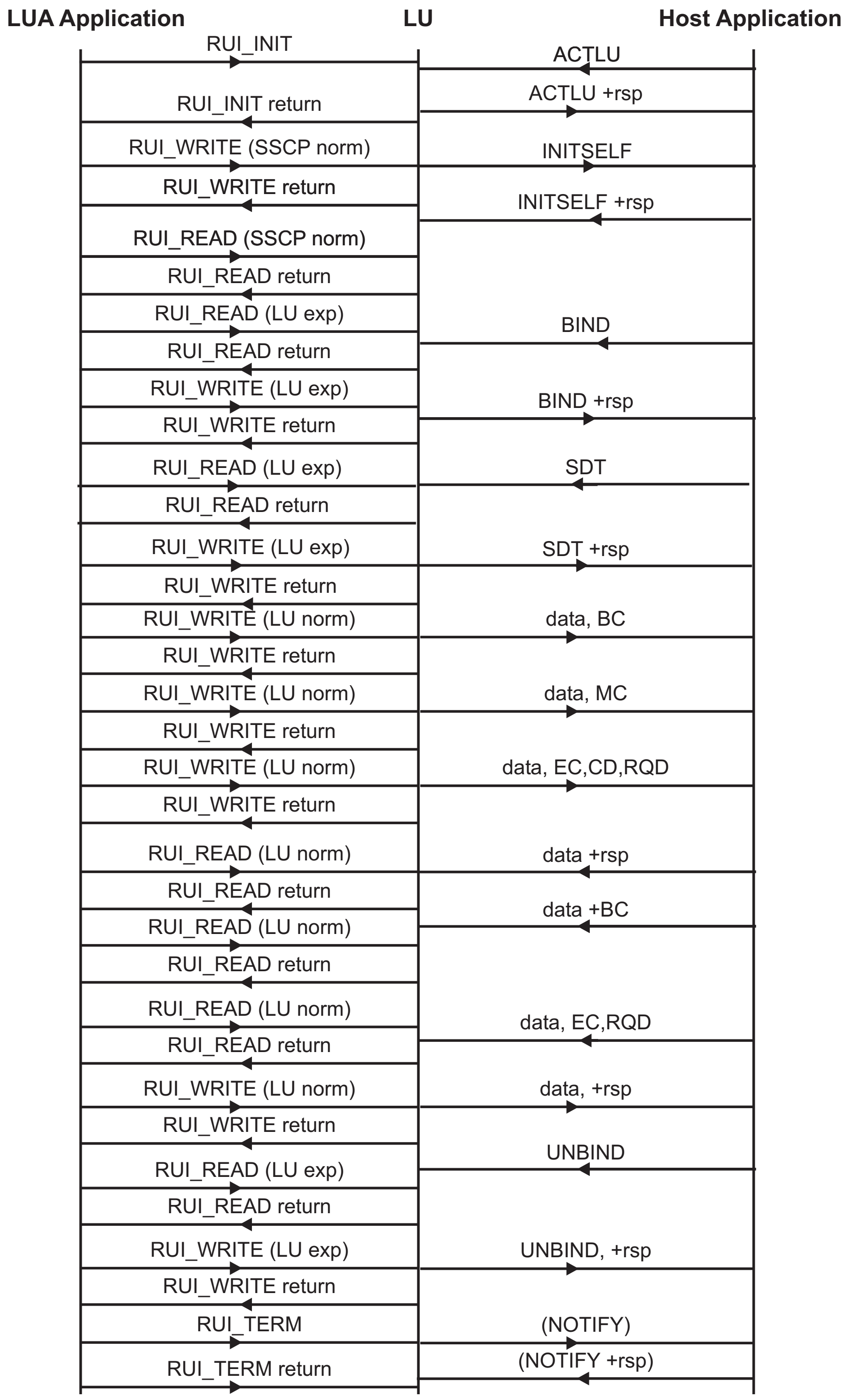

Figure 3. RUI Communication Sequence

Figure 4 on page 10, shows the equivalent SLI verbs used for the same SNA message sequence.

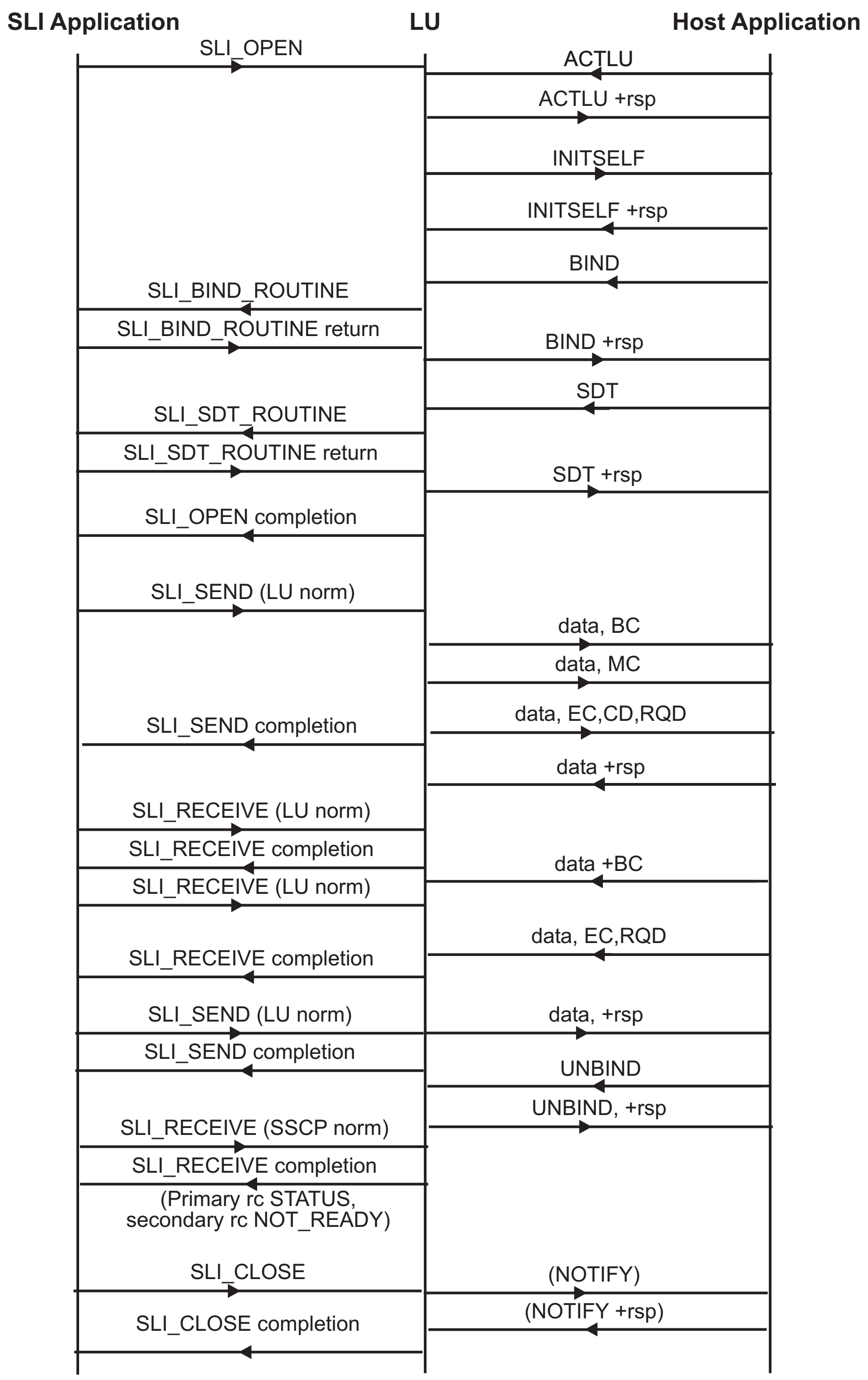

Figure 4. SLI Communication Sequence

LUA Compatibility

AIX, LINUX

The RUI_INIT_PRIMARY and RUI_REINIT verbs are extensions to the standard LUA interface specification. They are not available on a Remote API Client on Windows, and may not be available in other LUA implementations.

WINDOWS

The implementation of LUA on the Remote API Client on Windows is designed to be compatible with Windows LUA (as defined by the WOSA SNA specification); applications written for Windows LUA can be used with the Remote API Client without modification.

Chapter 2. Designing and Writing LUA Applications

The information contained in this chapter will help you write LUA application programs. The following topics are covered:

- LUA entry points for AIX or Linux applications
- LUA entry points for Windows applications
- Issuing an LUA verb
- SNA information
- Configuration information
- AIX or Linux considerations
- Windows considerations
- Writing portable applications

LUA Entry Points for AIX or Linux Applications

Applications running on AIX or Linux access LUA using the `RUI` or `SLI` function call, specifying the address of a Verb Control Block (VCB) containing information for an LUA verb. Communications Server returns control to the application immediately.

The returned VCB contains a value indicating whether verb processing is still in progress or has completed.

- In some cases, verb processing is still in progress when control returns to the application; Communications Server then uses an application-supplied callback routine to return the results of the verb processing.
- In other cases, verb processing is complete when Communications Server returns control to the application; Communications Server does not use the application's callback routine. This applies particularly if the verb failed LUA's initial parameter checks or state checks and so cannot be acted on.
- For SLI_OPEN, if the initial checks succeed, the SLI function call returns a non-zero value representing the session ID of the new session. Communications Server then uses the application-supplied callback routine in the same way as for other verbs. The application can use the new session ID to issue a limited range of subsequent verbs on the session, without waiting for the callback routine to be called. For details of which verbs can be issued in this situation, see "Interaction with Other Verbs" on page 120.

Note: Because of the way operating system callback routines operate, it is possible that the application's callback routine will be called before control returns to the application from its initial function call for the verb. This means that, if the callback routine modifies or deletes the returned VCB, the program's main thread of execution may be unable to check the VCB parameters to determine that the verb is operating asynchronously. You may need to take account of this in your application design.

The entry points RUI and SLI are defined in the LUA header file **/usr/include/sna/lua_c.h** (AIX) or **/opt/ibm/sna/include/lua_c.h** (Linux).

RUI Function Call

```
void RUI(verb)
LUA_VERB_RECORD * verb;
```

SLI Function Call

```
AP_UINT32 SLI(verb)
LUA_VERB_RECORD * verb;
```

Supplied Parameters

Supplied parameter is:

verb Pointer to a Verb Control Block (VCB) that contains the parameters for the verb being issued. The VCB structure is defined in the LUA header file **lua_c.h**, and is described in Chapter 3, "LUA VCB Structure," on page 45.

Note: The LUA VCB contains many parameters marked as "reserved"; some of these are used internally by the Communications Server software, and others are not used in this version but may be used in future versions. Your application must not attempt to access any of these reserved parameters; instead, it must set the entire contents of the VCB to zero to ensure that all of these parameters are zero, before it sets other parameters that are used by the verb. This ensures that Communications Server will not misinterpret any of its internally-used parameters, and also that your application will continue to work with future Communications Server versions in which these parameters may be used to provide new functions.

To set the VCB contents to zero, use `memset`:

memset(vcb, 0, sizeof(vcb));

Returned Values

For RUI, and for all SLI verbs except for SLI_OPEN, the entry point does not return a value. The returned parameters in the VCB indicate whether the verb has completed synchronously or will complete asynchronously; after the verb has completed, the VCB contains the results of the verb.

For SLI_OPEN, the entry point returns a value indicating whether the VCB passed LUA's initial checks:

- A return value of `0` (zero) indicates that the verb failed LUA's initial checks (for example because the application supplied incorrect parameters). Communications Server will not call the application-supplied callback routine.
- A non-zero value represents the session ID of the new session that SLI_OPEN will start.

  This return value does not indicate that the verb has completed. Communications Server will call the application-supplied callback routine to indicate SLI_OPEN completion when the session has been set up.

For more information, see "Usage."

Usage

Sometimes LUA is sometimes able to complete all the processing for a verb as soon as it is issued. This applies particularly if the verb failed LUA's initial parameter checks or state checks and so cannot be acted on. When this happens, the verb returns synchronously; the primary return code is set to a value other than `LUA_IN_PROGRESS`, and the *lua_flag2.async* bit is set to `0` (zero). (For information about these returned parameters, see Chapter 4, “RUI Verbs,” on page 55 or Chapter 5, “SLI Verbs,” on page 99.)

At other times, LUA must wait for information from the remote LU or from the node before it can complete the verb. In this case, the verb returns asynchronously; the primary return code is set to `LUA_IN_PROGRESS`, and the *lua_flag2.async* bit is set to 1. The application can now perform other processing, or wait for notification from LUA that the verb has completed. LUA issues this notification by setting the primary return code to its final value, leaving *lua_flag2.async* set to 1.

As part of the supplied VCB, the application supplies a pointer to a callback routine (in the *lua_post_handle* parameter). If the verb completes synchronously, LUA does not call the callback routine. If the verb completes asynchronously, LUA indicates the verb completion by calling the callback routine with one parameter—a pointer to the original verb control block (VCB). For more information, see “Callback Routine for Asynchronous Verb Completion.”

Note:

1. It is not possible for an application to predict whether a particular verb will complete synchronously or asynchronously.
2. If the *lua_flag2.async* parameter indicates that the verb will complete asynchronously, the program’s main thread of execution should not access any other parameters in the VCB at this point. When LUA calls the callback routine, the application can then access the VCB parameters.
3. Because of the way operating system callback routines operate, it is possible that the application’s callback routine will be called before control returns to the application from its initial function call for the verb. This means that, if the callback routine modifies or deletes the returned VCB, the program’s main thread of execution may be unable to check the VCB parameters to determine that the verb is operating asynchronously. You may need to take account of this in your application design.

Callback Routine for Asynchronous Verb Completion

To enable an LUA verb to complete asynchronously, the application must supply a pointer to a callback routine. This section describes how Communications Server uses this routine, and the functions that it must perform.

Function Call

```
void callback (verb)
LUA_VERB_RECORD * verb;
{     .
      .
}
```

Supplied Parameters

Communications Server calls the routine with the following parameter:

verb Pointer to the VCB supplied by the application, including the returned parameters set by Communications Server. The callback routine may perform all the necessary processing on the returned parameters in the VCB, or may simply set a variable to inform the main program that the verb has completed.

Note: Because of the way operating system callback routines operate, it is possible that the application's callback routine will be called before control returns to the application from its initial function call for the verb. This means that, if the callback routine modifies or deletes the returned VCB, the program's main thread of execution may be unable to check the VCB parameters to determine that the verb is operating asynchronously. You may need to take account of this in your application design.

Returned Values

There are no returned values.

LUA Entry Points for Windows Applications

WINDOWS

A Windows application accesses LUA using the following functions:

`RUI`
: Issues an RUI verb. If the verb completes asynchronously, LUA indicates the completion by signaling an event handle supplied by the application.

`WinRUIStartup`
: Registers the application as a Windows RUI user, and determines whether the LUA software supports the level of function required by the application.

`WinRUI`
: Issues an RUI verb. If the verb completes asynchronously, LUA will indicate the completion by posting a message to the application window.

`WinRUIGetLastInitStatus`
: Checks the status of an RUI session (initiated by a previous RUI_INIT verb that is still outstanding), requests notification of changes to the session status, or cancels this notification.

`WinRUICleanup`
: Unregisters the application when it has finished using RUI .

`GetLuaReturnCode`
: Generates a printable character string for the primary and secondary return codes obtained on an LUA verb.

`SLI`
: Issues an SLI verb. If the verb completes asynchronously, LUA indicates the completion by signaling an event handle supplied by the application.

`WinSLIStartup`
: Registers the application as a Windows SLI user, and determines whether the LUA software supports the level of function required by the application.

`WinSLI`
: Issues an SLI verb. If the verb completes asynchronously, LUA will indicate the completion by posting a message to the application window.

`WinSLICleanup`
: Unregisters the application when it has finished using SLI.

An RUI application must call `WinRUIStartup` before attempting to issue any LUA verbs using the `WinRUI` call.

While an RUI_INIT verb is outstanding, the application can use `WinRUIGetLastInitStatus` to determine the status of the LUA session initiated by this verb; it can then cancel the RUI_INIT verb if necessary. The

`WinRUIGetLastInitStatus` function can be used to check the current status without requesting notification of subsequent changes, to request asynchronous notification of subsequent changes to the session status, or to cancel a previous request for notification of status changes.

If a verb returns with non-LUA_OK return codes, the application can use `GetLuaReturnCode` to obtain a text string representation of these return codes, which can be used to generate standard error messages.

When it has finished issuing LUA verbs using the `WinRUI` call, it must call `WinRUICleanup` before terminating; it must not attempt to issue any more RUI verbs after calling `WinRUICleanup`.

An SLI application must call `WinSLIStartup` before attempting to issue any LUA verbs using the `WinSLI` call.

If a verb returns with non-LUA_OK return codes, the application can use `GetLuaReturnCode` to obtain a text string representation of these return codes, which can be used to generate standard error messages.

When it has finished issuing LUA verbs using the `WinSLI` call, it must call `WinSLICleanup` before terminating; it must not attempt to issue any more SLI verbs after calling `WinSLUICleanup`.

The following sections describe these functions.

RUI

The application uses this function to issue an LUA RUI verb. If the verb completes asynchronously, LUA indicates the completion by signaling an event handle supplied by the application.

The application does not need to issue a `WinRUIStartup` verb before making this call.

Function Call

```
void WINAPI RUI(verb)
LUA_VERB_RECORD FAR * verb;
```

Supplied Parameters

Supplied parameter is:

verb Pointer to a Verb Control Block (VCB) that contains the parameters for the verb being issued. The VCB structure is defined in the LUA header file **winlua.h**; this file is installed in the subdirectory **\sdk** for 32–bit applications, or **\sdk64** for 64–bit applications, within the directory where you installed the Windows Client software. For an explanation of the VCB structure, see Chapter 3, “LUA VCB Structure,” on page 45.

Note: The LUA VCB contains many parameters marked as “reserved”; some of these are used internally by the Communications Server software, and others are not used in this version but may be used in future versions. Your application must not attempt to access any of these reserved parameters; instead, it must set the entire contents of the VCB to zero to ensure that all of these parameters are zero, before it sets other parameters that are used by the verb. This ensures that Communications Server will not misinterpret any of its internally-used parameters, and also that your application will continue to work with future Communications Server versions in which these parameters may be used to provide new functions.

To set the VCB contents to zero, use `memset`:

memset(vcb, 0, sizeof(vcb));

Returned Values

The entry point does not return a value. When the call returns, the application can examine the parameters in the VCB to determine whether the verb has completed synchronously or will complete asynchronously. For more information, see "Usage."

Usage

Sometimes LUA is able to complete all the processing for a verb as soon as it is issued. When this happens, the verb returns synchronously; the primary return code is set to a value other than `LUA_IN_PROGRESS`, and the *lua_flag2.async* bit is set to `0` (zero). (For information about these returned parameters, see Chapter 4, "RUI Verbs," on page 55.)

At other times, LUA must wait for information from the remote LU or from the node before it can complete the verb. In this case, the verb returns asynchronously; the primary return code is set to `LUA_IN_PROGRESS`, and the *lua_flag2.async* bit is set to 1. The application can now perform other processing, or wait for notification from LUA that the verb has completed. LUA issues this notification by setting the primary return code to its final value, leaving the *lua_flag2.async* bit set to 1.

As part of the supplied VCB, the application supplies an event handle in the *lua_post_handle* parameter. The event must be in the nonsignaled state and the handle must have EVENT_MODIFY_STATE access to the event. If the verb completes synchronously, LUA does not signal this event handle. If the verb completes asynchronously, LUA indicates the verb completion by signaling the event handle.

The application issues a `WaitForSingleObject` or `WaitForMultipleObject` call to wait on the event handle. When the event is signaled, the application examines the primary return code and secondary return code to check for errors.

It is not possible for an application to predict whether a particular verb will complete synchronously or asynchronously.

WinRUIStartup

The application uses this function to register as a Windows RUI user, and to determine whether the LUA software supports the Windows LUA version that it requires.

Function Call

```
int WINAPI WinRUIStartup (
                           WORD           wVersionRequired;
                           LUADATA far *  lpData;
                         )
typedef struct
{
  WORD  wVersion;
  char  szDescription[41];
} LUADATA;
```

Supplied Parameters

Supplied parameter is:

wVersionRequired
: The version of Windows LUA that the application requires. Communications Server supports Version 1.0.

The low-order byte specifies the major version number, and the high-order byte specifies the minor version number. For example:

Version	wVersionRequired
1.0	0x0001
1.1	0x0101
2.0	0x0002

If the application can use more than one version, it should specify the highest version that it can use.

Returned Values

The return value from the function is one of the following:

`0` (zero)
: The application was registered successfully, and the Windows LUA software supports either the version number specified by the application or a lower version. The application should check the version number in the `LUADATA` structure to ensure that it is high enough.

`WLUAVERNOTSUPPORTED`
: The version number specified by the application is not supported by the Windows LUA software. The application was not registered.

`WLUAINITREJECT`
: The application has already called `WinRUIStartup` and registered successfully. It must not call this function more than once.

`WLUASYSNOTREADY`
: The Communications Server software has not been started, or the local node is not active. The application was not registered.

`WLUAFAILURE`
: An operating system error occurred during initialization of the Windows LUA software. The application was not registered. Check the log files for messages indicating the cause of the failure.

If the return value from `WinRUIStartup` is `0` (zero), the `LUADATA` structure contains information about the support provided by the Windows LUA software. If the return value is nonzero, the contents of this structure are undefined and the application should not check them. The parameters in this structure are as follows:

wVersion
: The Windows LUA version number that the software supports, in the same format as the *wVersionRequired* parameter (see “Supplied Parameters”). Communications Server supports Version 1.0.

If the software supports the requested version number, this parameter is set to the same value as the *wVersionRequired* parameter; otherwise it is set to the highest version that the software supports, which will be lower than the version number supplied by the application. The application must check the returned value and take action as follows:

- If the returned version number is the same as the requested version number, the application can use this Windows LUA implementation.
- If the returned version number is lower than the requested version number, the application can use this Windows LUA implementation but must not attempt to use features that are not supported by the returned version number. If it cannot do this because it requires features not available in the lower version, it should fail its initialization and not attempt to issue any LUA verbs.

szDescription
A text string describing the Windows LUA software.

WinRUI

The application uses this function to issue an RUI verb. If the verb completes asynchronously, LUA will indicate the completion by posting a message to the application's window handle.

Before using the `WinRUI` call for the first time, the application must use `RegisterWindowMessage` to obtain the message identifier that LUA will use for messages indicating asynchronous verb completion. For more information, see "Usage" on page 21.

Function Call

```
int WINAPI WinRUI (
                   HWND                     hWnd,
                   LUA_VERB_RECORD far *    lpVCB
                  );
```

For the definition of the LUA_VERB_RECORD structure, see Chapter 3, "LUA VCB Structure," on page 45.

Supplied Parameters

Supplied parameters are:

hWnd A window handle that LUA will use to post a message indicating asynchronous verb completion.

lpVCB A pointer to the VCB structure for the verb. For the `WinRUI` function, the *lua_post_handle* parameter is reserved; leave it as `0` (zero).

For more information about the VCB structure, see Chapter 3, "LUA VCB Structure," on page 45. For more information about on its usage for individual verbs, see Chapter 4, "RUI Verbs," on page 55.

Note: The LUA VCB contains many parameters marked as "reserved"; some of these are used internally by the Communications Server software, and others are not used in this version but may be used in future versions. Your application must not attempt to access any of these reserved parameters; instead, it must set the entire contents of the VCB to zero to ensure that all of these parameters are zero, before it sets other parameters that are used by the verb. This ensures that Communications Server will not misinterpret any of its internally-used parameters, and also that your application will continue to work with future Communications Server versions in which these parameters may be used to provide new functions.

To set the VCB contents to zero, use `memset`:

memset(vcb, 0, sizeof(vcb));

Returned Values

The return value from the function is one of the following:

`0` (zero)
: The function call was accepted, and the LUA verb will be processed. The application should check the *lua_flag2.async* parameter in the VCB structure to determine whether the verb has already completed synchronously or will complete asynchronously, as described in "Synchronous and Asynchronous Verb Completion."

`WLUAINVALIDHANDLE`
: The supplied *hWnd* parameter was not a valid window handle.

`WLUASTARTUPNOTCALLED`
: The application has not issued the `WinRUIStartup` call, which is required before issuing any LUA verbs.

For information about the parameters returned in the VCB structure, see the descriptions of individual verbs in Chapter 4, "RUI Verbs," on page 55.

Usage

Before using `WinRUI` for the first time, the application must use the `RegisterWindowMessage` call to obtain the message identifier that LUA will use for messages indicating asynchronous verb completion. `RegisterWindowMessage` is a standard Windows function call, not specific to LUA; refer to your Windows documentation for more information about the function. (There is no need to issue the call again before subsequent LUA verbs; the returned value will be the same for all calls issued by the application.)

The application must pass the string `WinRUI` to the function; the returned value is a message identifier (the value returned from the `RegisterWindowMessage` call).

Each time an LUA verb that was issued using the `WinRUI` entry point completes asynchronously, LUA posts a message to the window handle specified on the `WinRUI` call. The format of the message is as follows:

- The message identifier is the value returned from the `RegisterWindowMessage` call.
- The *lParam* argument contains the address of the VCB that was supplied to the original `WinRUI` call; the application can use this address to access the returned parameters in the VCB structure.
- The *wParam* argument is undefined.

Synchronous and Asynchronous Verb Completion

Sometimes LUA is able to complete all the processing for a verb as soon as it is issued. When this happens, the verb returns synchronously; the primary return code is set to a value other than `LUA_IN_PROGRESS`, and the *lua_flag2.async* bit is set to `0` (zero). (For information about these returned parameters, see Chapter 4, "RUI Verbs," on page 55.)

To enable the verb to return asynchronously, the application supplies a window handle to the LUA entry point. If the verb completes synchronously, LUA does not use this window handle. If the verb completes asynchronously, LUA indicates the verb completion by posting a message to this window handle; the message includes a pointer to the original VCB.

It is not possible for an application to predict whether a particular verb will complete synchronously or asynchronously.

Verbs can be issued from a callback, but they will not always complete asynchronously. Such verbs may be returned synchronously if they fail from within the library. The application should not reissue the failed verb from within the callback.

If the user repeatedly issues RUI_INITs in parallel from the callback context, the RUI_INITs will eventually fail with a memory error. However, if verbs are issued from the application thread, allowing the availability of all the system memory, more attempts will complete successfully.

WinRUIGetLastInitStatus

The application uses this function to determine the status of a previous RUI_INIT verb that is still outstanding. It can use the returned information to decide whether to cancel the session initiation (by issuing RUI_TERM) or to wait for the session to be established.

The function can be used to do any of the following:

- Request information about the current status of the session initiated by a specific RUI_INIT verb.
- Request asynchronous notification of changes to session status for a specific session or for all sessions. When the session status changes, LUA will indicate this by either posting a message to the application's window handle or by signaling the application's event handle.
- Cancel a previous request for asynchronous notification of changes to session status.

Before using the `WinRUI` call for the first time, the application must use `WinRUIStartup` to register as a Windows LUA application. If it requires asynchronous notification of status changes, it must also use `RegisterWindowMessage` to obtain the message identifier that LUA will use for this notification. For more information about these calls, see "WinRUIStartup" on page 18 and "Usage" on page 24.

Function Call

```
int WINAPI WinRUIGetLastInitStatus (
                                    DWORD  Sid,
                                    HANDLE StatusHandle,
                                    DWORD  NotifyType,
                                    BOOL   ClearPrevious
                                   );
```

Supplied Parameters

Supplied parameters are:

Sid To obtain information about the session status for a specific RUI_INIT verb, or to cancel a previous request for notification of session status changes for this verb, specify the session ID returned on the initial return from the RUI_INIT verb.

To request notification on session status changes for all outstanding RUI_INIT verbs, specify `0` (zero). In this case, the *StatusHandle* parameter must specify a valid Windows handle, because the information will always be returned asynchronously.

To cancel notification of session status changes for all outstanding RUI_INIT verbs, specify `0` (zero).

StatusHandle
: To obtain the current session status for a specific RUI_INIT verb, without requesting notification of subsequent changes, specify a null handle.

To request notification on session status changes, either for a specific RUI_INIT verb or for all outstanding RUI_INIT verbs, specify a Windows handle or an event handle that LUA will use when the session status changes.

If the *ClearPrevious* parameter is set to `TRUE`, to cancel a previous notification request, LUA ignores this parameter.

NotifyType
: If requesting asynchronous notification, this parameter determines how LUA should identify the RUI_INIT verb on the asynchronous notification message. Allowed values:

`WLUA_NTFY_MSG_CORRELATOR`
: The *StatusHandle* parameter contains a window handle. Identify the verb using the *lua_correlator* value supplied on the RUI_INIT verb.

`WLUA_NTFY_MSG_SID`
: The *StatusHandle* parameter contains a window handle. Identify the verb using the *lua_sid* value returned on the RUI_INIT verb.

`WLUA_NTFY_EVENT`
: The *StatusHandle* parameter contains an event handle.

If the *StatusHandle* parameter is null (to request current status information), or if the *ClearPrevious* parameter is set to `TRUE` (to cancel a previous notification request), LUA ignores this parameter.

ClearPrevious
: To cancel a previous notification request, set this parameter to `TRUE`; LUA ignores the *StatusHandle* and *ClearPrevious* parameters. To request either current status or notification of future status changes, set this parameter to `FALSE`.

Returned Values

If the function completed successfully, the return value from the function is one of the following:

`WLUALINKINACTIVE`
: The communications link to the host is not yet active.

`WLUAPUINACTIVE`
: The communications link to the host is active, but an activate physical unit (ACTPU) has not yet been received.

`WLUAPUACTIVE`
: An ACTPU has been received from the host.

`WLUAPUREACTIVATED`
: The PU has been reactivated by the host.

`WLUALUINACTIVE`
: The communications link to the host is active, and an ACTPU has been received, but an ACTLU has not yet been received.

`WLUALUACTIVE`
: The LU is active.

WLUALUREACTIVATED
: The LU has been reactivated.

WLUAGETLU
: The application is establishing contact with the node.

If the application requested notification of status changes, one of these values will be included in a Windows message sent to the application each time the status changes. For more information, see "Usage."

The following return values indicate that the function failed:

WLUASYSNOTREADY
: The SNA software is not running.

WLUANTFYINVALID
: The *NotifyType* parameter was set to a value that was not valid.

WLUAINVALIDHANDLE
: The supplied *StatusHandle* parameter was not a valid window handle.

WLUASTARTUPNOTCALLED
: The application has not issued the `WinRUIStartup` call, which is required before issuing any LUA verbs.

WLUAUNKNOWN
: Internal error: the session status is unknown.

WLUASIDINVALID
: The supplied *Sid* parameter did not match the session ID of an outstanding RUI_INIT verb.

WLUASIDZERO
: The application supplied a zero session ID (indicating `all sessions`), but did not specify either a Windows handle (to indicate asynchronous notification) or a *ClearPrevious* value of `TRUE` (to clear a previous notification request).

WLUAGLOBALHANDLER
: The application has previously requested notification of status changes for all RUI_INIT verbs; it cannot request notification for a specific session unless it first clears the "all sessions" notification.

Usage

If the application is requesting asynchronous notification of status changes using a Windows message, it must use the `RegisterWindowMessage` call before its first `WinRUIGetLastInitStatus` call, to obtain the message identifier that LUA will use for messages indicating status changes.

The `RegisterWindowMessage` call is a standard Windows function call, not specific to LUA; refer to your Windows documentation for more information about the function. (There is no need to issue the call again before subsequent calls to this function; the returned value will be the same for all calls issued by the application.)

The application must pass the string "`WinRUI`" to the function; the returned value is a message identifier (the value returned from the `RegisterWindowMessage` call).

Each time the session status changes, LUA posts a message to the window handle specified on the `WinRUI` call. The format of the message is as follows:

- The message identifier is the value returned from the `RegisterWindowMessage` call.
- The *lParam* argument contains either the correlator value supplied to the original RUI_INIT verb or the session ID returned on the original RUI_INIT verb, as defined by the *NotifyType* parameter. The application can use this value to correlate the message with the original verb.
- The *wParam* argument contains the session status (one of the values listed for successful execution in "Returned Values" on page 23), or the value `WLUAUNKNOWN` if an internal error occurred during processing.

If the application is requesting asynchronous notification of status changes using an event handle, implement it as follows:

```
WinRUIGetLastInitStatus(Sid,EventHandle,WLUA_NOTIFY_EVENT,FALSE);
```

The event whose handle is given will be signaled when a change in state occurs. Since no information is returned when an event is signaled, a further call must be issued to determine the status, as follows:

```
Status = WinRUIGetLastInitStatus(Sid,NULL,0,FALSE);
```

Note: In this case, a *Sid* must be specified.

WinRUICleanup

The application uses this function to unregister as a Windows RUI user, after it has finished issuing RUI verbs.

Function Call

```
  BOOL WINAPI WinRUICleanup (void);
```

Supplied Parameters

There are no supplied parameters for this function.

Returned Values

The return value from the function is one of the following:

TRUE The application was unregistered successfully.

FALSE An error occurred during processing of the call, and the application was not unregistered. Check the log files for messages indicating the cause of the failure.

GetLuaReturnCode

The application uses this function to obtain a printable character string indicating the primary and secondary return codes from a supplied VCB. The string can be used to generate application error messages for non-LUA_OK return codes.

Function Call

```
         int WINAPI GetLuaReturnCode (
                                     struct LUA_COMMON FAR *  vcbptr,
                                     unsigned int             buffer_length,
                                     unsigned char far *      buffer_addr
                                    );
```

Supplied Parameters

Supplied parameters are:

vcbptr A pointer to the VCB structure for the verb. For more information about the VCB structure and on its usage for individual verbs, see Chapter 4, "RUI Verbs," on page 55.

buffer_length
The length (in bytes) of the buffer supplied by the application to hold the returned data string. The recommended length is 256 bytes.

buffer_addr
The address of the buffer supplied by the application to hold the returned data string.

Returned Values

The return value from the function is one of the following:

`0` (zero)
The function completed successfully.

`0x20000001`
LUA could not read from the supplied VCB, or could not write to the supplied data buffer.

`0x20000002`
The supplied data buffer is too small to hold the returned character string.

`0x20000003`
The dynamic link library, **LUASTR32.DLL**, which generates the returned character strings for this function, could not be loaded.

If the return value is `0` (zero), the returned character string is in the buffer identified by the *buffer_addr* parameter. This string is terminated by a null character (binary zero), but does not include a trailing new-line *(\n)* character.

SLI

The application uses this function to issue an LUA SLI verb. If the verb completes asynchronously, LUA indicates the completion by signaling an event handle supplied by the application.

Function Call

```
void WINAPI SLI(verb)
LUA_VERB_RECORD FAR * verb;
```

Supplied Parameters

Supplied parameter is:

verb Pointer to a Verb Control Block (VCB) that contains the parameters for the verb being issued. The VCB structure is defined in the LUA header file **winlua.h**; this file is installed in the subdirectory **/sdk** within the directory where you installed the Windows Client software. For an explanation of the VCB structure, see Chapter 3, "LUA VCB Structure," on page 45.

Note: The LUA VCB contains many parameters marked as "reserved"; some of these are used internally by the Communications Server software, and others are not used in this version but may be used in future versions. Your application must not attempt to access any of these reserved parameters; instead, it must set the entire contents of the VCB to zero to ensure that all of these parameters are zero, before it sets other parameters that are used by the verb. This ensures that Communications Server will not misinterpret any of its internally-used parameters, and also that your application will continue to work with future Communications Server versions in which these parameters may be used to provide new functions.

To set the VCB contents to zero, use `memset`:

memset(vcb, 0, sizeof(vcb));

Returned Values

The entry point does not return a value. When the call returns, the application can examine the parameters in the VCB to determine whether the verb has completed synchronously or will complete asynchronously. For more information, see "Usage."

Usage

Sometimes LUA is able to complete all the processing for a verb as soon as it is issued. When this happens, the verb returns synchronously; the primary return code is set to a value other than `LUA_IN_PROGRESS`, and the *lua_flag2.async* bit is set to `0` (zero). (For information about these returned parameters, see Chapter 5, "SLI Verbs," on page 99.)

At other times, LUA must wait for information from the remote LU or from the node before it can complete the verb. In this case, the verb returns asynchronously; the primary return code is set to `LUA_IN_PROGRESS`, and the *lua_flag2.async* bit is set to 1. The application can now perform other processing, or wait for notification from LUA that the verb has completed. LUA issues this notification by setting the primary return code to its final value, leaving the *lua_flag2.async* bit set to 1.

As part of the supplied VCB, the application supplies an event handle in the *lua_post_handle* parameter. The event must be in the nonsignaled state and the handle must have EVENT_MODIFY_STATE access to the event. If the verb completes synchronously, LUA does not signal this event handle. If the verb completes asynchronously, LUA indicates the verb completion by signaling the event handle.

The application issues a `WaitForSingleObject` or `WaitForMultipleObject` call to wait on the event handle. When the event is signaled, the application examines the primary return code and secondary return code to check for errors.

It is not possible for an application to predict whether a particular verb will complete synchronously or asynchronously.

WinSLIStartup

The application uses this function to register as a Windows SLI user, and to determine whether the LUA software supports the Windows LUA version that it requires.

Function Call

```
int WINAPI WinSLIStartup (
                          WORD           wVersionRequired;
                          LUADATA far *  lpData;
                         )
typedef struct
{
  WORD  wVersion;
  char  szDescription[41];
} LUADATA;
```

Supplied Parameters

Supplied parameter is:

wVersionRequired
: The version of Windows LUA that the application requires. Communications Server supports Version 1.0.

  The low-order byte specifies the major version number, and the high-order byte specifies the minor version number. For example:

Version	wVersionRequired
1.0	0x0001
1.1	0x0101
2.0	0x0002

If the application can use more than one version, it should specify the highest version that it can use.

Returned Values

The return value from the function is one of the following:

`0` (zero)
The application was registered successfully, and the Windows LUA software supports either the version number specified by the application or a lower version. The application should check the version number in the `LUADATA` structure to ensure that it is high enough.

`WLUAVERNOTSUPPORTED`
The version number specified by the application is not supported by the Windows LUA software. The application was not registered.

`WLUAINITREJECT`
The application has already called `WinSLIStartup` and registered successfully. It must not call this function more than once.

`WLUASYSNOTREADY`
The Communications Server software has not been started, or the local node is not active. The application was not registered.

`WLUAFAILURE`
An operating system error occurred during initialization of the Windows LUA software. The application was not registered. Check the log files for messages indicating the cause of the failure.

If the return value from `WinSLIStartup` is `0` (zero), the `LUADATA` structure contains information about the support provided by the Windows LUA software. If the return value is nonzero, the contents of this structure are undefined and the application should not check them. The parameters in this structure are as follows:

wVersion
The Windows LUA version number that the software supports, in the same format as the *wVersionRequired* parameter (see “Supplied Parameters” on page 19). Communications Server supports Version 1.0.

If the software supports the requested version number, this parameter is set to the same value as the *wVersionRequired* parameter; otherwise it is set to the highest version that the software supports, which will be lower than the version number supplied by the application. The application must check the returned value and take action as follows:

- If the returned version number is the same as the requested version number, the application can use this Windows LUA implementation.
- If the returned version number is lower than the requested version number, the application can use this Windows LUA implementation but must not attempt to use features that are not supported by the returned version number. If it cannot do this because it requires features not available in the lower version, it should fail its initialization and not attempt to issue any LUA verbs.

szDescription
A text string describing the Windows LUA software.

WinSLI

The application uses this function to issue an SLI verb. If the verb completes asynchronously, LUA will indicate the completion by posting a message to the application's window handle.

Before using the `WinSLI` call for the first time, the application must use `RegisterWindowMessage` to obtain the message identifier that LUA will use for messages indicating asynchronous verb completion. For more information, see "Usage" on page 30.

Function Call

```
int WINAPI WinSLI (
                   HWND                 hWnd,
                   LUA_VERB_RECORD far *  lpVCB
                  );
```

For the definition of the LUA_VERB_RECORD structure, see Chapter 3, "LUA VCB Structure," on page 45.

Supplied Parameters

Supplied parameters are:

hWnd A window handle that LUA will use to post a message indicating asynchronous verb completion.

lpVCB A pointer to the VCB structure for the verb. For the `WinSLI` function, the *lua_post_handle* parameter is reserved; leave it as `0` (zero).

For more information about the VCB structure, see Chapter 3, "LUA VCB Structure," on page 45. For more information about on its usage for individual verbs, see Chapter 5, "SLI Verbs," on page 99.

Note: The LUA VCB contains many parameters marked as "reserved"; some of these are used internally by the Communications Server software, and others are not used in this version but may be used in future versions. Your application must not attempt to access any of these reserved parameters; instead, it must set the entire contents of the VCB to zero to ensure that all of these parameters are zero, before it sets other parameters that are used by the verb. This ensures that Communications Server will not misinterpret any of its internally-used parameters, and also that your application will continue to work with future Communications Server versions in which these parameters may be used to provide new functions.

To set the VCB contents to zero, use `memset`:

memset(vcb, 0, sizeof(vcb));

Returned Values

The return value from the function is one of the following:

`0` (zero)
: The function call was accepted, and the LUA verb will be processed. The application should check the *lua_flag2.async* parameter in the VCB structure to determine whether the verb has already completed synchronously or will complete asynchronously, as described in "Synchronous and Asynchronous Verb Completion."

`WLUAINVALIDHANDLE`
: The supplied *hWnd* parameter was not a valid window handle.

`WLUASTARTUPNOTCALLED`
: The application has not issued the `WinSLIStartup` call, which is required before issuing any SLI verbs.

For information about the parameters returned in the VCB structure, see the descriptions of individual verbs in Chapter 5, "SLI Verbs," on page 99.

Usage

Before using `WinSLI` for the first time, the application must use the `RegisterWindowMessage` call to obtain the message identifier that LUA will use for messages indicating asynchronous verb completion. `RegisterWindowMessage` is a standard Windows function call, not specific to LUA; refer to your Windows documentation for more information about the function. (There is no need to issue the call again before subsequent LUA verbs; the returned value will be the same for all calls issued by the application.)

The application must pass the string `WinSLI` to the function; the returned value is a message identifier (the value returned from the `RegisterWindowMessage` call).

Each time an LUA verb that was issued using the `WinSLI` entry point completes asynchronously, LUA posts a message to the window handle specified on the `WinSLI` call. The format of the message is as follows:

- The message identifier is the value returned from the `RegisterWindowMessage` call.
- The *lParam* argument contains the address of the VCB that was supplied to the original `WinSLI` call; the application can use this address to access the returned parameters in the VCB structure.
- The *wParam* argument is undefined.

Synchronous and Asynchronous Verb Completion

Sometimes LUA is able to complete all the processing for a verb as soon as it is issued. When this happens, the verb returns synchronously; the primary return code is set to a value other than `LUA_IN_PROGRESS`, and the *lua_flag2.async* bit is set to `0` (zero). (For information about these returned parameters, see Chapter 5, "SLI Verbs," on page 99.)

To enable the verb to return asynchronously, the application supplies a window handle to the LUA entry point. If the verb completes synchronously, LUA does not use this window handle. If the verb completes asynchronously, LUA indicates the verb completion by posting a message to this window handle; the message includes a pointer to the original VCB.

It is not possible for an application to predict whether a particular verb will complete synchronously or asynchronously.

Verbs can be issued from a callback, but they will not always complete asynchronously. Such verbs may be returned synchronously if they fail from within the library. The application should not reissue the failed verb from within the callback.

If the user repeatedly issues SLI_OPENs in parallel from the callback context, the SLI_OPENs will eventually fail with a memory error. However, if verbs are issued from the application thread, allowing the availability of all the system memory, more attempts will complete successfully.

WinSLICleanup

The application uses this function to unregister as a Windows SLI user, after it has finished issuing SLI verbs.

Function Call

```
BOOL WINAPI WinSLICleanup (void);
```

Supplied Parameters

There are no supplied parameters for this function.

Returned Values

The return value from the function is one of the following:

TRUE The application was unregistered successfully.

FALSE An error occurred during processing of the call, and the application was not unregistered. Check the log files for messages indicating the cause of the failure.

Issuing an LUA Verb

The steps required to issue an LUA verb are as follows. The examples indicate the use of the RUI_INIT verb.

1. Include the LUA header file in the application's source code.

   AIX, LINUX

   ```
    #include < lua_c.h>
   ```

   WINDOWS

   ```
   #include < winlua.h >
   ```

   AIX, LINUX

2. Set up a callback function that LUA will use to indicate that the verb has completed asynchronously. (For more information, see "LUA Entry Points for AIX or Linux Applications" on page 13.)

```
void callback(verb)
LUA_VERB_RECORD * verb;
{
       .
       .
       .
}
```

WINDOWS

If this is the first LUA verb from the application, and the application will be issuing RUI verbs using the `WinRUI` call, issue the `WinRUIStartup` call to initialize the application's use of LUA. Similarly, if the application will be issuing SLI verbs using the `WinSLI` call, issue the `WinSLIStartup` call to initialize the application's use of LUA. (For more information, see "LUA Entry Points for Windows Applications" on page 16.) This call must be issued once before the application's first LUA verb; it must not be repeated before subsequent verbs.

Also issue the `RegisterWindowMessage` call, to obtain the message identifier that LUA will use when posting messages to indicate the completion of an LUA verb. (For more information, see "LUA Entry Points for Windows Applications" on page 16.) This call must be issued once before the application's first LUA verb; there is no need to repeat it before subsequent verbs.

3. Create a variable for the VCB structure.

   ```
   LUA_VERB_RECORD rui_init;
   ```

   The LUA_VERB_RECORD structure is declared in the header file **lua_c.h** (AIX / Linux applications) or **winlua.h** (Windows applications); for an explanation of the VCB structure, see Chapter 3, "LUA VCB Structure," on page 45.
4. Clear (set to `0`) the variables within the VCB.

   ```
    memset( rui_init, 0, sizeof( rui_init) );
   ```

   LUA requires that all reserved parameters, and all parameters not required by the particular verb being issued, must be set to `0` (zero). For details about reserved parameters, see "LUA Verb Control Block (VCB) Format" on page 45. The simplest way to do this is to set the entire VCB to zeros before setting the parameters required for this particular verb.
5. Assign values to the VCB parameters that supply information to LUA.

   ```
   rui_init.common.lua_verb = LUA_VERB_RUI
   rui_init.common.lua_verb_length = sizeof(LUA_COMMON);
   rui_init.common.lua_opcode = LUA_OPCODE_RUI_INIT;
   memcpy (rui_init.common.lua_luname, "THISLU  ", 8);
   ```

   AIX, LINUX

   ```
   rui_init.common.lua_post_handle = (unsigned long) callback;
   ```

   WINDOWS

   The *rui_init.common.lua_post_handle* parameter is reserved; leave it as `0` (zero).

   The values **LUA_VERB_RUI** and **LUA_OPCODE_RUI_INIT** are symbolic constants. These constants are defined in the header file **lua_c.h** (AIX / Linux applications) or **winlua.h** (Windows applications); you are recommended to use the symbolic constants and not the integer values, for portability between different systems. (For more information, see “Writing Portable Applications” on page 42.)

6. Invoke LUA. The address of the VCB structure is a parameter to the function call.

   AIX, LINUX

   ```
   RUI ( &rui_init );
   ```

   WINDOWS

   The `WinRUI` entry point requires an additional parameter, which is a window handle for the window to which LUA will post a message indicating asynchronous completion of the verb.

   ```
   WinRUI ( handle, (LUA_VERB_RECORD far * ) &rui_init );
   ```

7. Check the *lua_flag2.async* parameter to find out whether the verb has completed synchronously or will complete asynchronously.

   ```
   if (rui_init.common.lua_flag2.async )
   {
         /* verb will complete asynchronously   */
         /* using the supplied callback routine */
         /* continue with other processing      */
         .
         .
         .
   }
   else
   {

         /* verb has completed synchronously    */
         /* callback routine will not be called */
         /* process the returned values here    */
         .
         .
         .
   }
   ```

   If the *lua_flag2.async* parameter indicates that the verb will complete asynchronously, the program’s main thread of execution should not access any other parameters in the VCB at this point. When LUA calls the callback routine, the application can then access the VCB parameters.

8. Use the variables returned by LUA. If Step 7 indicates that the verb will complete asynchronously, this step must not be performed until the verb has completed; on AIX / Linux systems, the processing is typically done by the callback routine. If Step 7 indicates that the verb has completed synchronously, the processing should be done by the main code path because the callback routine will not be called.

   ```
   if( rui_init.common.lua_prim_rc == LUA_OK )
   {
         /* Init OK */
             .
             .
             .
   }
   else
   {
         /* Do error routine */
   ```

```
                      .
                      .
                      .
          }
```

SNA Information

This section explains some SNA information that you need to consider when writing Communications Server LUA applications for communications with a host. If you are writing an RUI Primary application for communications with a downstream LU, see "SNA Information for RUI Primary" on page 38.

This guide does not attempt to explain SNA concepts in detail. If you need specific information about SNA message flows, refer to the documentation for the host application for which you are designing your Communications Server LUA application.

BIND Checking: RUI

During initialization of the LU session, the host sends a BIND message to the Communications Server LUA application that contains information such as RU sizes to be used by the LU session. Communications Server returns this message to the LUA application on an RUI_READ verb. It is the responsibility of the LUA application to check that the parameters specified on the BIND are suitable. The application has the following options:

- Accept the BIND as it is, by issuing an RUI_WRITE verb containing an `OK` response to the BIND. No data needs to be sent on the response.
- Try to negotiate one or more BIND parameters (this is only permitted if the BIND is negotiable). To do this, the application issues an RUI_WRITE verb containing an `OK` response, but including the modified BIND as data.
- Reject the BIND by issuing an RUI_WRITE verb containing a negative response, using an appropriate SNA sense code as data.

For more information about the RUI_WRITE verb, see Chapter 4, "RUI Verbs," on page 55.

Validation of the BIND parameters, and ensuring that all messages sent are consistent with them, is the responsibility of the LUA application. However, the following two restrictions apply:

- Communications Server rejects any RUI_WRITE verb that specifies an RU length greater than the size specified on the BIND.
- Communications Server requires the BIND to specify that the secondary LU is the contention winner, and that error recovery is the responsibility of the contention loser.

BIND Checking: SLI

During initialization of the LU session, the host sends a BIND message to the Communications Server LUA application that contains information such as RU sizes to be used by the LU session.

On the SLI_OPEN verb, the application can optionally specify the address of its own routine to process BIND requests from the host. If it has done so, LUA sends an additional verb SLI_BIND_ROUTINE to the application-supplied routine to allow it to process the request, as follows. It is the responsibility of the LUA application to check that the parameters specified on the BIND are suitable. The application has the following options:

- Accept the BIND as it is, by returning the SLI_BIND_ROUTINE verb with a primary return code of `OK`. The application does not modify the data buffer containing the BIND.
- Try to negotiate one or more BIND parameters (this is only permitted if the BIND is negotiable). To do this, the application returns the SLI_BIND_ROUTINE verb with a primary return code of `OK`, but including the modified BIND in the data buffer.
- Reject the BIND by returning the SLI_BIND_ROUTINE verb with a primary return code of `LUA_NEGATIVE_RESPONSE`, and replacing the BIND request in the data buffer with an appropriate SNA sense code.

Validation of the BIND parameters, and ensuring that all messages sent are consistent with them, is the responsibility of the LUA application. However, Communications Server requires the BIND to specify that the secondary LU is the contention winner, and that error recovery is the responsibility of the contention loser.

Negative Responses and SNA Sense Codes

SNA sense codes may be returned to an LUA application in the following cases:

- When the host sends a negative response to a request from the LUA application, this includes an SNA sense code indicating the reason for the negative response. This is reported to the application on a subsequent RUI_READ or SLI_RECEIVE verb, as follows:
  - The primary return code is `LUA_OK`.
  - The Request/Response Indicator, Response Type Indicator, and Sense Data Included Indicator are all set to `1`, indicating a negative response that includes sense data.
  - The data returned by the RUI_READ or SLI_RECEIVE verb is the SNA sense code.
- When Communications Server receives data that is not valid from the host, it generally sends a negative response to the host and does not pass the data that is not valid to the LUA application. This is reported to the application on a subsequent RUI_READ or RUI_BID verb, or SLI_RECEIVE / SLI_BID, as follows:
  - The primary return code is `LUA_NEGATIVE_RSP`.
  - The secondary return code is the SNA sense code sent to the host.
- In some cases, Communications Server detects that data supplied by the host is not valid, but cannot determine the correct sense code to send. In this case, it passes the data that is not valid in an Exception Request (EXR) to the LUA application on an RUI_READ or SLI_RECEIVE verb as follows:
  - The Request/Response Indicator is set to `0` (zero), indicating a request.
  - The Sense Data Included Indicator is set to `1`, indicating that sense data is included (this indicator is normally used only for a request).
  - The message data is replaced by a suggested SNA sense code.

  The application must then send a negative response to the message; it may use the sense code suggested by Communications Server, or may alter it.

- Communications Server may send a sense code to the application to indicate that data supplied by the application was not valid. This is reported to the application on the RUI_WRITE or SLI_SEND verb that supplied the data, as follows:
  - The primary return code is `LUA_UNSUCCESSFUL`.
  - The secondary return code is the SNA sense code.

Distinguishing SNA Sense Codes from Other Secondary Return Codes

Note: The byte ordering used in LUA secondary return codes means that the most significant byte of the numeric value is the last byte, not the first byte.

For a secondary return code that is not a sense code, the two most significant bytes of this value are always `0` (zero). As an example, `0x01000000` (`LUA_INVALID_LUNAME`) is a standard LUA secondary return code and not a sense code.

For an SNA sense code, the two most significant bytes are nonzero; the most significant byte gives the sense code category, and the next byte identifies a particular sense code within that category. (The remaining bytes may contain additional information, or may be `0`.) As an example, `0x00000108` (`LUA_RESOURCE_NOT_AVAILABLE`) is a sense code.

All LUA secondary return codes, including those that are SNA sense codes, are listed in Appendix A, "Return Code Values," on page 151.

Information about SNA Sense Codes

If you need information about a returned sense code, refer to IBM's *Systems Network Architecture: Formats*. The sense codes are listed in numerical order by category.

You can also retrieve online help information about a specific SNA sense code generated on the Communications Server computer, by typing **sna -getsense** followed by either the category and modifier (the first four digits) or the entire sense code (all eight digits) on the command line. For more information, see *IBM Communications Server for AIX Diagnostics Guide* or *IBM Communications Server for Linux Diagnostics Guide*.

Pacing

Pacing is handled by the LUA interface; an LUA application does not need to control pacing, and should never set the Pacing Indicator flag.

If pacing is being used on data sent from the LUA application to the host (this is determined by the BIND), an RUI_WRITE or SLI_SEND verb may take some time to complete. This is because Communications Server has to wait for a pacing response from the host before it can send more data.

If an LUA application is used to transfer large quantities of data in one direction, either to the host or from the host (for example, a file transfer application), then the host configuration should specify that pacing is used in that direction; this is to ensure that the node receiving the data is not flooded with data and does not run out of data storage.

Segmentation

Segmentation of RUs is handled by the LUA interface. LUA always passes complete RUs to the application, and the application should pass complete RUs to LUA.

Modification of Nonstandard Host Response/Request Header (RH) Bits

A host may send data to an LUA application with the BB (begin bracket) and RQE (request exception) options set but without the EB (end bracket) option (begin bracket and exception response but no end bracket). This combination of options is not strictly valid in SNA, but is used by some host applications.

In order to support these host applications, Communications Server modifies the host data to specify definite response rather than exception response before sending it to the application.

Courtesy Acknowledgments

Communications Server keeps a record of requests received from the host in order to correlate any response sent by the application with the appropriate request. When the application sends a response, Communications Server correlates this with the data from the original request, and can then free the storage associated with it.

If the host specifies exception response only (a negative response can be sent but a positive response should not be sent), Communications Server must still keep a record of the request in case the application subsequently sends a negative response. If the application does not send a response, the storage associated with this request cannot be freed.

Because of this, Communications Server allows the LUA application to issue a positive response to an exception-response-only request from the host (this is known as a courtesy acknowledgment). The response is not sent to the host, but is used by Communications Server to clear the storage associated with the request.

Purging Data to End of Chain

When the host sends a chain of request units to an LUA application, the application may wait until the last RU in the chain is received before sending a response, or it may send a negative response to an RU that is not the last in the chain. If a negative response is sent mid-chain, Communications Server purges all subsequent RUs from this chain, and does not send them to the application.

When Communications Server receives the last RU in the chain, it indicates this to the application by setting the primary return code of an RUI_READ or RUI_BID verb, or SLI_RECEIVE / SLI_BID, to `LUA_NEGATIVE_RSP` with a `0` (zero) secondary return code.

The host may terminate the chain by sending a message such as CANCEL while in mid-chain. In this case, the CANCEL message is returned to the application on an RUI_READ or SLI_RECEIVE verb, and the `LUA_NEGATIVE_RSP` return code (see "Negative Responses and SNA Sense Codes" on page 35) is not used.

SNA Information for RUI Primary

This section explains some SNA information that you need to consider when writing Communications Server RUI Primary applications for communications with a downstream LU.

This guide does not attempt to explain SNA concepts in detail. If you need specific information about SNA message flows, refer to the documentation for the host application for which you are designing your Communications Server LUA application.

Responsibilities of the Primary RUI application

A Primary RUI application has control of both LU-SSCP and PLU-SLU sessions at the Request/Response Unit (RU) level, and can send and receive SNA RUs on these sessions. The PU-SSCP session is internal to Communications Server and the Primary RUI application cannot access it.

Because a Primary RUI application works at the RU level, it has a large degree of control over the data flow to and from the secondary LU. However, it takes greater responsibility than a regular LUA application for ensuring that the SNA messages it sends are valid and that the RU level protocols (for example bracketing and chaining) are used correctly. In particular, note that Communications Server does not attempt to verify the validity of RUs sent by a Primary RUI application.

The Primary RUI application is responsible for:

- Initializing downstream LUs using RUI_INIT_PRIMARY, and terminating them using RUI_TERM
- Processing NOTIFY messages from the secondary LU as secondary applications start and stop
- Processing INIT-SELF and TERM-SELF to activate and deactivate the PLU-SLU session
- Building, sending, receiving and parsing 3270 datastream messages in data RUs
- Implementing RU level protocols (request control, bracketing, chaining, direction)
- Cryptography (if required)
- Compression (if required).

Pacing

Pacing is handled by the LUA interface; an LUA application does not need to control pacing, and should never set the Pacing Indicator flag.

If pacing is being used on data sent from the LUA application to the host (this is determined by the BIND), an RUI_WRITE verb may take some time to complete. This is because Communications Server has to wait for a pacing response from the host before it can send more data.

If an LUA application is used to transfer large quantities of data in one direction, either to the host or from the host (for example, a file transfer application), then the host configuration should specify that pacing is used in that direction; this is to ensure that the node receiving the data is not flooded with data and does not run out of data storage.

Segmentation

Segmentation of RUs is handled by the LUA interface. LUA always passes complete RUs to the application, and the application should pass complete RUs to LUA.

Restrictions

Communications Server does not support the following for Primary RUI applications:

- Downstream PUs over DLUR
- Dynamically Defined Dependent LUs (DDDLU)
- Sending STSN (to reset sequence numbers, the application should UNBIND and re-BIND the session).

Courtesy Acknowledgments

Communications Server keeps a record of requests received from the host in order to correlate any response sent by the application with the appropriate request. When the application sends a response, Communications Server correlates this with the data from the original request, and can then free the storage associated with it.

If the host specifies exception response only (a negative response can be sent but a positive response should not be sent), Communications Server must still keep a record of the request in case the application subsequently sends a negative response. If the application does not send a response, the storage associated with this request cannot be freed.

Because of this, Communications Server allows the LUA application to issue a positive response to an exception-response-only request from the host (this is known as a courtesy acknowledgment). The response is not sent to the host, but is used by Communications Server to clear the storage associated with the request.

Purging Data to End of Chain

When the host sends a chain of request units to an LUA application, the application may wait until the last RU in the chain is received before sending a response, or it may send a negative response to an RU that is not the last in the chain. If a negative response is sent mid-chain, Communications Server purges all subsequent RUs from this chain, and does not send them to the application.

When Communications Server receives the last RU in the chain, it indicates this to the application by setting the primary return code of an RUI_READ or RUI_BID verb to `LUA_NEGATIVE_RSP` with a `0` (zero) secondary return code.

The host may terminate the chain by sending a message such as CANCEL while in mid-chain. In this case, the CANCEL message is returned to the application on an RUI_READ verb, and the `LUA_NEGATIVE_RSP` return code (see "Negative Responses and SNA Sense Codes" on page 35) is not used.

Configuration Information

The Communications Server configuration file, which is set up and maintained by the System Administrator, contains information that is required for LUA applications to communicate. For additional information, refer to the *IBM Communications Server for AIX Administration Guide* or the *IBM Communications Server for Linux Administration Guide*.

AIX, LINUX

For a Primary RUI application communicating with a downstream LU, the only configuration required is the downstream LU (or a Downstream PU template).

The following components must be configured for use with an LUA application communicating with a host:

Data Link Control (DLC), Port, and Link Station (LS)

The communications components that Communications Server uses to communicate with the remote host computer.

LU

An LU of type 0–3, with an LU number that matches that of a suitable LU on the host.

LU Pool (Optional)

If required, you can configure more than one LU for use by the application, and group the LUs into a pool. This means that an application can specify the pool rather than a specific LU when attempting to start a session, and will be assigned the least recently used LU from the pool.

An LUA application indicates to Communications Server that it wants to start a session by issuing an RUI_INIT or SLI_OPEN verb with an LU name. This name must match the name of an LU of type 0–3, or of an LU pool, in the configuration file. Communications Server uses this name as follows:

- If the name supplied is the name of an LU that is not in a pool, a session will be assigned using that LU if it is available (that is, if it is not already in use by a program).
- If the name supplied is the name of an LU pool, or the name of any LU within the pool, a session will be assigned using the named LU, if it is available, or otherwise the least recently used LU in the pool. The RUI_INIT or SLI_OPEN verb returns the name of the actual LU assigned (which may not be the same as the name specified). The application can then use this returned LU name on subsequent LUA verbs to identify the session.

AIX or Linux Considerations

AIX, LINUX

This section summarizes processing considerations of which you must be aware when developing LUA applications on an AIX or Linux computer.

LUA Header File

The header file to be used with LUA applications is **lua_c.h**. This file contains the definitions of the LUA entry points and the LUA VCBs. It also includes the common interface header file **values_c.h**; these two files contain all the constants defined for supplied and returned parameter values at the LUA interface. The file **values_c.h** also includes definitions of parameter types such as `AP_UINT16` that are used in the LUA VCBs.

These two files are stored in **/usr/include/sna** (AIX) or **/opt/ibm/sna/include** (Linux).

Multiple Processes and Multiple Sessions

If the process that issued RUI_INIT, RUI_INIT_PRIMARY, or SLI_OPEN then forks to create a child process, the child process cannot issue any LUA verbs on the session started by the parent process; the verbs will fail with return codes `LUA_UNSUCCESSFUL` and `LUA_INVALID_PROCESS`. It can, however, issue another RUI_INIT, RUI_INIT_PRIMARY, or SLI_OPEN to obtain its own session.

A single process may simultaneously use more than one LUA session, by issuing multiple RUI_INIT, RUI_INIT_PRIMARY, or SLI_OPEN verbs. Each session must use a different LU, but two or more sessions may use the same pool.

Two or more instances of the same LUA application can be run as different processes, but they must use different LUs. This can be done either by providing a mechanism for specifying the LU name at run time, or by using LU pools; if the two processes specify the same pool, they will be allocated different LUs from that pool.

Compiling and Linking the LUA Application

Windows Considerations

WINDOWS

This section summarizes processing considerations of which you must be aware when developing LUA applications on a Windows client.

Multiple Sessions and Multiple Tasks

A single task may simultaneously use more than one LUA session, by issuing multiple RUI_INIT or SLI_OPEN verbs. Each session must use a different LU, but two or more sessions may use the same pool.

Two or more instances of the same LUA application can be run as different tasks, but they must use different LUs. This can be done by using LU pools; the two tasks can specify the same pool, and will be allocated different LUs from that pool.

Compiling and Linking LUA Programs

This section provides information about compiling and linking LUA programs on a Windows client.

Compiler Options for Structure Packing

The VCB structures for LUA verbs are not packed. Do not use compiler options that change this packing method.

DWORD parameters are on DWORD boundaries, *WORD* parameters are on WORD boundaries, and *BYTE* parameters are on BYTE boundaries.

Header File

The header file to be included in Windows LUA applications is named **winlua.h**. This file is installed in the subdirectory **\sdk** for 32–bit applications, or **\sdk64** for 64–bit applications, within the directory where you installed the Remote API Client on Windows software.

Load-Time Linking

To link the program to LUA at load time, link the program to the **winsli32.lib** library (for SLI) or **winrui32.lib** library (for RUI). This file is stored in the subdirectory **\sdk** for 32–bit applications, or **\sdk64** for 64–bit applications,

within the directory where you installed the Windows Client software

Run-Time Linking

To link the program to LUA at run-time, include the following calls in the program:

- `LoadLibrary` to load the LUA dynamic link library **winsli32.dll** (for SLI) or **winrui32.dll** (for RUI).
- `GetProcAddress` to specify each of the LUA entry points required (such as `SLI`)
- `FreeLibrary` when the library is no longer required.

Terminating Applications

If an application must close (for example, if it receives a WM_CLOSE message as a result of an **ALT F4** from a user), it should call the `WinRUICleanup` or `WinSLICleanup` function before terminating. If it does not do this, then the application is left in an indeterminate state, although as much cleanup as possible is done when the Windows LUA software detects that the application has terminated.

[redacted]

Writing Portable Applications

Communications Server's implementation of LUA is designed to be compatible with the implementation provided by IBM's OS/2 Extended Edition. However, there are a few differences between the implementations that are due to fundamental operating system differences. These operating system differences are indicated in the individual verb descriptions. In particular:

- The RUI_REINIT verb is an extension to the standard LUA interface specification. It is not available on the Remote API Client on Windows, and may not be available in other LUA implementations.
- Other LUA implementations generate certain additional return codes that are not returned by the Communications Server implementation; they may also make use of parameters that are reserved for Communications Server.

- OS/2 and Windows implementations use far pointers (far *) in all cases; AIX / Linux implementations do not have a concept of far and near pointers, so the word far must be omitted for AIX / Linux implementations.
- The asynchronous verb return feature is supported differently by different operating systems. You may need to rewrite the sections of an LUA application written for one operating system that relate to asynchronous verb returns if you are porting the application to another operating system.
- Other LUA implementations may not support LU pools.

The following guidelines are provided for writing Communications Server LUA applications so that they will be portable to other environments:

- Include the LUA header file without any path name prefix. This enables the application to be used in an environment with a different file system. Use include options on the compiler to locate the file (see "Compiling and Linking the LUA Application" on page 41).
- Use the symbolic constant names for parameter values and return codes, not the numeric values shown in the header file; this ensures that the correct value will be used regardless of the way these values are stored in memory.
- When accessing SNA sense codes in a data buffer, use the symbolic constants rather than the numeric values; this ensures that the byte storage order will be correct for your particular system.
- Include a check for return codes other than those applicable to your current operating system (for example using a "default" case in a switch statement), and provide appropriate diagnostics.
- Ensure that any parameters shown as reserved are set to `0` (zero).
- Set the *lua_verb_length* parameter as described in Chapter 4, "RUI Verbs," on page 55 or Chapter 5, "SLI Verbs," on page 99..

Chapter 3. LUA VCB Structure

This chapter contains details of the LUA verb control block structure used for all LUA verbs.

Symbolic constants are defined in the header files **lua_c.h** and **values_c.h** (AIX / Linux operating system) or **winlua.h** (Windows operating system)for many parameter values. For portability, use the symbolic constant and not the numeric value when setting values for supplied parameters, or when testing values of returned parameters. The file **values_c.h** also includes definitions of parameter types such as `AP_UINT16` that are used in the LUA VCBs.

Parameters marked as "reserved" should always be set to `0` (zero).

LUA Verb Control Block (VCB) Format

The verb control block consists of two parts:
- **Common** data structure, used for all verbs
- **Specific** data structure, used only for the following verbs:
  - RUI_BID
  - The extended version of RUI_INIT (in the AIX / Linux environment)
  - SLI_BID
  - SLI_OPEN
  - SLI_SEND

The definition of some parts of the VCB structure, in particular the ordering of bit fields, varies between different operating systems. For clarity, only one version of the ordering is shown here, although both versions are defined in the header file. When setting or testing values in bit fields, the application should access individual bits by name, to avoid dependencies on the bit ordering, rather than using bitwise AND or OR operations on complete bytes.

AIX, LINUX

To allow for these differences, the LUA header file contains the following information:
- A `#include` statement for the file **/usr/include/sna/svconfig.h** (AIX) or **/opt/ibm/sna/include/svconfig.h** (Linux).
- The type definition for bit fields in the LUA data structures. This definition ensures that the data structures are stored in the correct format. The definition depends on the setting of PUCHARQD, which is in the file **svconfig.h**.

Note: The LUA VCB contains many parameters marked as "reserved"; some of these are used internally by the Communications Server software, and others are not used in this version but may be used in future versions. Your application must not attempt to access any of these reserved parameters;

instead, it must set the entire contents of the VCB to zero to ensure that all of these parameters are zero, before it sets other parameters that are used by the verb. This ensures that Communications Server will not misinterpret any of its internally-used parameters, and also that your application will continue to work with future Communications Server versions in which these parameters may be used to provide new functions.

To set the VCB contents to zero, use memset:

memset(vcb, 0, sizeof(vcb));

LUA_VERB_RECORD Data Structure

```
typedef struct
{
  struct LUA_COMMON common;
  struct LUA_SPECIFIC specific;
} LUA_VERB_RECORD;
```

Common Data Structure

AIX, LINUX

```
struct LUA_COMMON
{
  AP_UINT16       lua_verb;                   /* Verb Code                   */
  AP_UINT16       lua_verb_length;            /* Length of Verb Record       */
  AP_UINT16       lua_prim_rc;                /* Primary Return Code         */
  AP_UINT32       lua_sec_rc;                 /* Secondary Return Code       */
  AP_UINT16       lua_opcode;                 /* Verb Operation Code         */
  AP_UINT32       lua_correlator;             /* User Correlation Field      */
  unsigned char   lua_luname[8];              /* Local LU Name               */
  AP_UINT16       lua_extension_list_offset;  /* Offset of DLL Extention List*/
  AP_UINT16       lua_cobol_offset;           /* Offset of Cobol Extension   */
  AP_UINT32       lua_sid;                    /* Session ID                  */
  AP_UINT16       lua_max_length;             /* Receive Buffer Length       */
  AP_UINT16       lua_data_length;            /* Data Length                 */
  char *          lua_data_ptr;               /* Data Buffer Pointer         */
  unsigned long   lua_post_handle;            /* Posting handle              */

  struct LUA_TH {                            /* LUA TH Fields               */
    BIT_FIELD_TYPE   flags_fid  : 4;         /* Format Identification Type 2 */
    BIT_FIELD_TYPE   flags_mpf  : 2;         /* Segmenting Mapping Field    */
    BIT_FIELD_TYPE   flags_odai : 1;         /* OAF-DAF Assignor Indicator  */
    BIT_FIELD_TYPE   flags_efi  : 1;         /* Expedited Flow Indicator    */
    BIT_FIELD_TYPE              : 8;         /* Reserved Field              */
    unsigned char    daf;                    /* Destination Address Field   */
    unsigned char    oaf;                    /* Originating Address Field   */
    unsigned char    snf[2];                 /* Sequence Number Field       */
    } lua_th;

  struct LUA_RH {                            /* LUA RH Fields               */
    BIT_FIELD_TYPE   rri  : 1;               /* Request-Response Indicator  */
    BIT_FIELD_TYPE   ruc  : 2;               /* RU Category                 */
    BIT_FIELD_TYPE        : 1;               /* Reserved Field              */
    BIT_FIELD_TYPE   fi   : 1;               /* Format Indicator            */
    BIT_FIELD_TYPE   sdi  : 1;               /* Sense Data Included Ind     */
    BIT_FIELD_TYPE   bci  : 1;               /* Begin Chain Indicator       */
    BIT_FIELD_TYPE   eci  : 1;               /* End Chain Indicator         */

    BIT_FIELD_TYPE   dr1i : 1;               /* DR 1 Indicator              */
    BIT_FIELD_TYPE   lcci : 1;               /* LCC Indicator               */
    BIT_FIELD_TYPE   dr2i : 1;               /* DR 2 Indicator              */
    BIT_FIELD_TYPE   ri   : 1;               /* Response Indicator          */
    BIT_FIELD_TYPE        : 2;               /* Reserved Field              */
    BIT_FIELD_TYPE   qri  : 1;              /* Queued Response Indicator    */
    BIT_FIELD_TYPE   pi   : 1;              /* Pacing Indicator             */

    BIT_FIELD_TYPE   bbi  : 1;              /* Begin Bracket Indicator      */
    BIT_FIELD_TYPE   ebi  : 1;              /* End Bracket Indicator        */
    BIT_FIELD_TYPE   cdi  : 1;              /* Change Direction Indicator   */
    BIT_FIELD_TYPE        : 1;              /* Reserved Field               */
    BIT_FIELD_TYPE   csi  : 1;              /* Code Selection Indicator     */
    BIT_FIELD_TYPE   edi  : 1;              /* Enciphered Data Indicator    */
    BIT_FIELD_TYPE   pdi  : 1;              /* Padded Data Indicator        */
    BIT_FIELD_TYPE        : 1;              /* Reserved Field               */
    } lua_rh;
  struct LUA_FLAG1 {                        /* LUA_FLAG1                    */
    BIT_FIELD_TYPE   bid_enable  : 1;       /* Bid Enabled Indicator        */
    BIT_FIELD_TYPE   reserv1     : 1;       /* reserved                     */
    BIT_FIELD_TYPE   close_abend : 1;       /* Close Immediate Flag         */
    BIT_FIELD_TYPE   nowait      : 1;       /* Don't Wait for Data Flag     */
    BIT_FIELD_TYPE   sscp_exp    : 1;       /* SSCP expedited flow          */
    BIT_FIELD_TYPE   sscp_norm   : 1;       /* SSCP normal flow             */
    BIT_FIELD_TYPE   lu_exp      : 1;       /* LU expedited flow            */
    BIT_FIELD_TYPE   lu_norm     : 1;       /* lu normal flow               */
    } lua_flag1;
  unsigned char      lua_message_type;      /* sna message command type     */
  struct LUA_FLAG2 {                        /* LUA_FLAG2                    */
    BIT_FIELD_TYPE   bid_enable  : 1;       /* Bid Enabled Indicator        */
    BIT_FIELD_TYPE   async       : 1;       /* flags asynchronous verb      */
                                            /*  completion                  */
    BIT_FIELD_TYPE               : 2;       /* reserved                     */
    BIT_FIELD_TYPE   sscp_exp    : 1;       /* SSCP expedited flow          */
    BIT_FIELD_TYPE   sscp_norm   : 1;       /* SSCP normal flow             */
    BIT_FIELD_TYPE   lu_exp      : 1;       /* LU expedited flow            */
    BIT_FIELD_TYPE   lu_norm     : 1;       /* lu normal flow               */
    } lua_flag2;
  unsigned char      lua_resv56[7];         /* Reserved Field               */
  unsigned char      lua_encr_decr_option;  /* Cryptography Option          */
} ;
```

WINDOWS

```
struct LUA_COMMON
{
  unsigned short  lua_verb;                  /* Verb Code                    */
  unsigned short  lua_verb_length;           /* Length of Verb Record        */
  unsigned short  lua_prim_rc;               /* Primary Return Code          */
  unsigned long   lua_sec_rc;                /* Secondary Return Code        */
  unsigned short  lua_opcode;                /* Verb Operation Code          */
  unsigned long   lua_correlator;            /* User Correlation Field       */
  unsigned char   lua_luname[8];             /* Local LU Name                */
  unsigned short  lua_extension_list_offset; /* Offset of DLL Extention List*/
  unsigned short  lua_cobol_offset;          /* Offset of Cobol Extension    */
  unsigned long   lua_sid;                   /* Session ID                   */
  unsigned short  lua_max_length;            /* Receive Buffer Length        */
  unsigned short  lua_data_length;           /* Data Length                  */
  char far        *lua_data_ptr;             /* Data Buffer Pointer          */
  unsigned long   lua_post_handle;           /* Posting handle               */
  struct LUA_TH {                           /* LUA TH Fields                */
    unsigned char    flags_fid  : 4;        /* Format Identification Type 2 */
    unsigned char    flags_mpf  : 2;        /* Segmenting Mapping Field     */
    unsigned char    flags_odai : 1;        /* OAF-DAF Assignor Indicator   */
    unsigned char    flags_efi  : 1;        /* Expedited Flow Indicator     */
    unsigned char               : 8;        /* Reserved Field               */
    unsigned char    daf;                    /* Destination Address Field    */
    unsigned char    oaf;                    /* Originating Address Field    */
    unsigned char    snf[2];                 /* Sequence Number Field        */
    } lua_th;
  struct LUA_RH {                            /* LUA RH Fields                */
    unsigned char    rri  : 1;               /* Request-Response Indicator   */
    unsigned char    ruc  : 2;               /* RU Category                  */
    unsigned char         : 1;               /* Reserved Field               */
    unsigned char    fi   : 1;               /* Format Indicator             */
    unsigned char    sdi  : 1;               /* Sense Data Included Ind      */
    unsigned char    bci  : 1;               /* Begin Chain Indicator        */
    unsigned char    eci  : 1;               /* End Chain Indicator          */

    unsigned char    dr1i : 1;               /* DR 1 Indicator               */
    unsigned char    lcci : 1;               /* LCC Indicator                */
    unsigned char    dr2i : 1;               /* DR 2 Indicator               */
    unsigned char    ri   : 1;               /* Response Indicator           */
    unsigned char         : 2;               /* Reserved Field               */
    unsigned char    qri  : 1;               /* Queued Response Indicator    */
    unsigned char    pi   : 1;               /* Pacing Indicator             */

    unsigned char    bbi  : 1;               /* Begin Bracket Indicator      */
    unsigned char    ebi  : 1;               /* End Bracket Indicator        */
    unsigned char    cdi  : 1;               /* Change Direction Indicator   */
    unsigned char         : 1;               /* Reserved Field               */
    unsigned char    csi  : 1;               /* Code Selection Indicator     */
    unsigned char    edi  : 1;               /* Enciphered Data Indicator    */
    unsigned char    pdi  : 1;               /* Padded Data Indicator        */
    unsigned char         : 1;               /* Reserved Field               */
    } lua_rh;
  struct LUA_FLAG1 {                         /* LUA_FLAG1                    */
    unsigned char    bid_enable  : 1;        /* Bid Enabled Indicator        */
    unsigned char    reserv1     : 1;        /* reserved                     */
    unsigned char    close_abend : 1;        /* Close Immediate Flag         */
    unsigned char    nowait      : 1;        /* Don't Wait for Data Flag     */
    unsigned char    sscp_exp    : 1;        /* SSCP expedited flow          */
    unsigned char    sscp_norm   : 1;        /* SSCP normal flow             */
    unsigned char    lu_exp      : 1;        /* LU expedited flow            */
    unsigned char    lu_norm     : 1;        /* lu normal flow               */
    } lua_flag1;
  unsigned char      lua_message_type;       /* sna message command type     */
  struct LUA_FLAG2 {                         /* LUA_FLAG2                    */
    unsigned char    bid_enable  : 1;        /* Bid Enabled Indicator        */
    unsigned char    async       : 1;        /* flags asynchronous verb      */
                                                completion                   */
    unsigned char                : 2;        /* reserved                     */
    unsigned char    sscp_exp    : 1;        /* SSCP expedited flow          */
    unsigned char    sscp_norm   : 1;        /* SSCP normal flow             */
    unsigned char    lu_exp      : 1;        /* LU expedited flow            */
    unsigned char    lu_norm     : 1;        /* lu normal flow               */
    } lua_flag2;
  unsigned char      lua_resv56[7];          /* Reserved Field               */
  unsigned char      lua_encr_decr_option;   /* Cryptography Option          */
} ;
```

The following list explains the fields in these data structures.

lua_verb

Identifies this as an LUA verb.

Possible values:

LUA_VERB_RUI
RUI verb.

LUA_VERB_SLI
SLI verb.

lua_verb_length
Length of the verb control block (VCB).

lua_prim_rc
Primary return code set by LUA.

lua_sec_rc
Secondary return code set by LUA.

lua_opcode
Verb operation code that identifies the LUA verb being issued.

lua_correlator
A four-byte correlator that you can use to correlate this verb with other processing in your application. LUA does not use this parameter.

lua_luname
The LU name used by the LUA session (in ASCII). This can be an LU name or an LU pool name; for more information, see "RUI_INIT" on page 61 or "SLI_OPEN" on page 112.

AIX, LINUX

For RUI_INIT_PRIMARY, this must match the *dslu_name* parameter of a downstream LU configured for use with SNA Gateway (or a downstream LU created implicitly by defining a downstream LU template).

lua_extension_list_offset
This field is reserved.

lua_cobol_offset
This field is reserved.

lua_sid The session ID of the LUA session on which this verb is issued.

lua_max_length
The length of the buffer supplied to RUI_READ, RUI_INIT_PRIMARY, or SLI_RECEIVE to receive data, or the total length of a waiting RU returned to RUI_BID.

lua_data_length
The length of the data to be sent, or the actual length of data received.

lua_data_ptr
A pointer to the data to be sent, or the data buffer to receive data.

lua_post_handle

AIX, LINUX

A pointer to a callback routine that LUA will call to indicate completion if the verb completes asynchronously.

WINDOWS

If the VCB is used in an `RUI` or `SLI` function call, set this field to an event handle. If the VCB is used in a `WinRUI` or `WinSLI` function call, this field is reserved.

lua_th
A data structure containing the TH (transmission header) of the message sent or received, as follows:

lua_th.flags_fid
Format Identification type 2: 4 bits

lua_th.flags_mpf
Segmenting mapping field: 2 bits

lua_th.flags_odai
Origin Address Field-Destination Address Field (OAF-DAF) Assignor Indicator

lua_th.flags_efi
Expedited Flow Indicator

lua_th.daf
DAF (Destination address field)

lua_th.oaf
OAF (Originating address field)

lua_th.snf
Sequence Number Field

lua_rh
A data structure containing the RH (request/response header) of the message sent or received, as follows:

lua_rh.rri
Request-Response Indicator

lua_rh.ruc
RU category: 2 bits

lua_rh.fi
Format Indicator

lua_rh.sdi
Sense Data Included Indicator

lua_rh.bci
Begin Chain Indicator

lua_rh.eci
End Chain Indicator

lua_rh.dr1i
Definite Response 1 Indicator

lua_rh.lcci
Length-Checked Compression Indicator

lua_rh.dr2i
Definite Response 2 Indicator

lua_rh.ri
: Exception Response Indicator (for a request), or Response Type Indicator (for a response)

lua_rh.qri
: Queued Response Indicator

lua_rh.pi
: Pacing Indicator

lua_rh.bbi
: Begin Bracket Indicator

lua_rh.ebi
: End Bracket Indicator

lua_rh.cdi
: Change Direction Indicator

lua_rh.csi
: Code Selection Indicator

lua_rh.edi
: Enciphered Data Indicator

lua_rh.pdi
: Padded Data Indicator

lua_flag1
: A data structure containing flags for messages supplied by the application, as follows:

lua_flag1.bid_enable
: Bid Enable Indicator

lua_flag1.close_abend
: Close Immediate Indicator

lua_flag1.nowait
: No Wait For Data flag

lua_flag1.sscp_exp
: SSCP expedited flow

lua_flag1.sscp_norm
: SSCP normal flow

lua_flag1.lu_exp
: LU expedited flow

lua_flag1.lu_norm
: LU normal flow

lua_message_type
: The type of SNA message received by an RUI_READ or SLI_RECEIVE verb (or indicated to an RUI_BID or SLI_BID verb)

lua_flag2
: A data structure containing flags for messages returned by LUA, as follows:

lua_flag2.bid_enable
: Bid Enabled Indicator

lua_flag2.async
: Asynchronous verb completion flag

lua_flag2.sscp_exp
: SSCP expedited flow

lua_flag2.sscp_norm
: SSCP normal flow

lua_flag2.lu_exp
: LU expedited flow

lua_flag2.lu_norm
: LU normal flow

lua_encr_decr_option
: Cryptography option. For SLI, this parameter is reserved and must be set to zero.

Specific Data Structure

The *specific*data structure is included for the following verbs:

- RUI_BID
- Extended form of RUI_INIT
- SLI_BID
- SLI_OPEN
- SLI_SEND

```
union LUA_SPECIFIC
{
struct SLI_OPEN  open;
unsigned char    lua_sequence_number[2];
unsigned char    lua_peek_data[12];
struct RUI_INIT  init;
} ;
```

AIX, LINUX

```
struct SLI_OPEN
{
  unsigned char  lua_init_type;             /* Type of Session Initiation  */
  unsigned char  lua_session_type;          /* How to process host UNBIND  */
  AP_UINT16      lua_wait;                  /* Secondary Retry Wait Time   */

  struct LUA_EXT_ENTRY
  {
    unsigned char lua_routine_type;         /* Extension Routine Type      */
    unsigned long lua_routine_ptr;          /* Ptr to Extension Routine    */
  } lua_open_extension[MAX_EXTENSIONS];

  char reserved[93];                        /* Padding                     */
  unsigned char lua_ending_delim;           /* Extension List Delimiter    */
};
```

WINDOWS

```
struct SLI_OPEN
{
  unsigned char  lua_init_type;             /* Type of Session Initiation  */
  unsigned char  lua_session_type;          /* How to process host UNBIND  */
  AP_UINT16      lua_wait;                  /* Secondary Retry Wait Time   */

  struct LUA_EXT_ENTRY
  {
    unsigned char lua_routine_type;         /* Extension Routine Type      */
    unsigned char lua_module_name[9];         /* Extension DLL module name   */
    unsigned char lua_procedure_name[33];     /* Procedure name to call      */
  } lua_open_extension[MAX_EXTENSIONS];

  char reserved[93];                          /* Padding                     */
  unsigned char lua_ending_delim;             /* Extension List Delimiter    */
};
```

```
struct RUI_INIT
{
unsigned char    rui_init_format;
unsigned char    lua_puname[8];
unsigned char    lua_lunumber;
unsigned char    wait_for_link;
};
```

For RUI_BID and SLI_BID, this data structure contains the following field:

lua_peek_data
: Up to 12 bytes of the data waiting to be read.

AIX, LINUX

For the extended form of the RUI_INIT verb, this data structure contains the following fields. For more information about the extended form of RUI_INIT, see "RUI_INIT" on page 61.

rui_init_format
: Reserved—this parameter must be set to `0` (zero).

lua_puname
: The name of the local PU that owns the LU to be used for this session. The PU name must be specified in the definition of an LS or of an internal PU in the Communications Server configuration.

lua_lunumber
: The LU number of the LU to be used for this session. This must match the LU number of a type 0–3 LU that is configured for the PU name specified by *lua_puname*.

wait_for_link
: Normally, if the application issues RUI_INIT for an LU that cannot currently be used because the underlying communications link is inactive, the RUI_INIT verb fails. Set this parameter to 1 to override this default behavior so that LUA waits for the link and LU to become active before RUI_INIT completes, or `0` (zero) to use the default behavior.

WINDOWS

The extended form of the RUI_INIT verb does not apply to Windows. The RUI_INIT data structure is not used.

For SLI_OPEN, this data structure contains the following fields. See “SLI_OPEN” on page 112 for detailed information about these parameters.

lua_init_type
: Specifies how LUA initiates the session (whether the primary or secondary is responsible for session initiation, and the sequence of SNA messages required).

lua_session_type
: Specifies how LUA should process an UNBIND type X’01’ (normal): whether this is a normal or dedicated session.

lua_wait
: Retry timeout (in seconds) for secondary-initiated session startup.

lua_open_extension
: Structure containing information about the application’s SLI_OPEN extension routines, if any.

  lua_open_extension.lua_routine_type
  : Type of extension routine (BIND, SDT, or STSN).

  AIX, LINUX

  lua_open_extension.lua_routine_ptr
  : Pointer to the extension routine entry point.

  WINDOWS

  lua_open_extension.lua_module_name
  : Name of the DLL containing the extension module.

  lua_open_extension.lua_procedure_name
  : Procedure name to call within the extension module DLL.

lua_ending_delim
: The Communications Server SLI interface does not use this parameter; it is provided for compatibility with applications originally written for other SLI implementations.

For SLI_SEND, this data structure contains the following field.

lua_sequence_number
: The sequence number of the RU that LUA uses to send the data (or of the first RU, if the data requires a chain of RUs). This is stored in line format.

Chapter 4. RUI Verbs

This chapter contains a description of each LUA RUI verb. The following information is provided for each verb:

- Purpose of the verb.
- Parameters (VCB fields) supplied to and returned by LUA. The description of each parameter includes information about the valid values for that parameter, and any additional information necessary.
- Interaction with other verbs.
- Additional information describing the use of the verb.

For details of the Verb Control Block (VCB) used for all verbs, see Chapter 3, “LUA VCB Structure,” on page 45.

Symbolic constants are defined in the header files **lua_c.h** and **values_c.h** (AIX / Linux operating system) or **winlua.h** (Windows operating system)for many parameter values. For portability, use the symbolic constant and not the numeric value when setting values for supplied parameters, or when testing values of returned parameters. The file **values_c.h** also includes definitions of parameter types such as `AP_UINT16` that are used in the LUA VCBs.

Parameters marked as “reserved” should always be set to `0` (zero).

RUI_BID

The RUI_BID verb is used by the application to determine when a received message is waiting to be read. This enables the application to determine what data, if any, is available before issuing the RUI_READ verb.

When a message is available, the RUI_BID verb returns with details of the message flow on which it was received, the message type, the TH and RH of the message, and up to 12 bytes of message data.

The main difference between RUI_BID and RUI_READ is that RUI_BID enables the application to check the data without removing it from the incoming message queue, so it can be left and accessed at a later stage. The RUI_READ verb removes the message from the queue, so once the application has read the data it must process it.

Supplied Parameters

The application supplies the following parameters:

lua_verb
: `LUA_VERB_RUI`

lua_verb_length
: The length in bytes of the LUA verb record.

  Set this to `sizeof(LUA_VERB_RECORD)`.

lua_opcode
: `LUA_OPCODE_RUI_BID`

lua_correlator
: Optional. A four-byte value that you can use to correlate this verb with other processing within your application. LUA does not use or change this information.

lua_luname
: The name in ASCII of the LU used by the session. This must match the LU name of an active LUA session (as returned on the RUI_INIT or RUI_INIT_PRIMARY verb).

  This parameter is required only if the *lua_sid* parameter is `0` (zero). If a session ID is supplied in *lua_sid*, LUA does not use this parameter.

  This parameter must be eight bytes long; pad on the right with spaces, `0x20`, if the name is shorter than eight characters.

lua_sid
: The session ID of the session. This must match a session ID returned on a previous RUI_INIT or RUI_INIT_PRIMARY verb.

  This parameter is optional; if you do not specify the session ID, you must specify the LU name for the session in the *lua_luname* parameter.

lua_post_handle
: AIX, LINUX

  A pointer to a callback routine that LUA will call to indicate completion if the verb completes asynchronously.

  WINDOWS

  If the VCB is used in an `RUI` function call, set this field to an event handle. If the VCB is used in a `WinRUI` function call, this field is reserved.

For more information, see Chapter 2, “Designing and Writing LUA Applications,” on page 13.

Returned Parameters

LUA always returns the following parameter:

lua_flag2.async
: This flag is set to `1` if the verb completed asynchronously, or `0` (zero) if the verb completed synchronously.

Other returned parameters depend on whether the verb completed successfully; see the following sections.

Successful Execution

If the verb completed successfully, LUA returns the following parameters:

lua_prim_rc
: `LUA_OK`

lua_sid
: If the application specified the *lua_luname* parameter when issuing this verb, rather than specifying the session ID, LUA supplies the session ID.

lua_max_length
The total number of bytes in the waiting message.

lua_data_length
The number of bytes of data returned in the *lua_peek_data* parameter; from `0` to `12`.

lua_th The TH of the received message.

lua_rh The RH of the received message.

lua_message_type
Message type of the received message that will be one of the following:

`LUA_MESSAGE_TYPE_LU_DATA`

`LUA_MESSAGE_TYPE_SSCP_DATA`

`LUA_MESSAGE_TYPE_RSP`

`LUA_MESSAGE_TYPE_BID`

`LUA_MESSAGE_TYPE_BIND`

`LUA_MESSAGE_TYPE_BIS`

`LUA_MESSAGE_TYPE_CANCEL`

`LUA_MESSAGE_TYPE_CHASE`

`LUA_MESSAGE_TYPE_CLEAR`

`LUA_MESSAGE_TYPE_CRV`

`LUA_MESSAGE_TYPE_LUSTAT_LU`

`LUA_MESSAGE_TYPE_LUSTAT_SSCP`

`LUA_MESSAGE_TYPE_QC`

`LUA_MESSAGE_TYPE_QEC`

`LUA_MESSAGE_TYPE_RELQ`

`LUA_MESSAGE_TYPE_RTR`

`LUA_MESSAGE_TYPE_SBI`

`LUA_MESSAGE_TYPE_SHUTD`

`LUA_MESSAGE_TYPE_SIGNAL`

`LUA_MESSAGE_TYPE_SDT`

`LUA_MESSAGE_TYPE_STSN`

`LUA_MESSAGE_TYPE_UNBIND`

AIX, LINUX

The following values can be returned only to an RUI primary application (one that started the session using RUI_INIT_PRIMARY):

`LUA_MESSAGE_TYPE_INIT_SELF`

`LUA_MESSAGE_TYPE_NOTIFY`

`LUA_MESSAGE_TYPE_TERM_SELF`

lua_flag2
One of the following flags will be set to 1 to indicate on which message flow the data was received:

lua_flag2.sscp_exp

lua_flag2.lu_exp

lua_flag2.sscp_norm

lua_flag2.lu_norm

lua_peek_data
The first 12 bytes of the message data (or all of the message data if it is shorter than 12 bytes)

Unsuccessful Execution

If a verb does not complete successfully, LUA returns a primary return code to indicate the type of error and a secondary return code to provide specific details about the reason for unsuccessful execution.

Verb Canceled: The following return codes indicate that the verb did not complete successfully because it was canceled by another verb:

lua_prim_rc
`LUA_CANCELLED`

lua_sec_rc

`LUA_TERMINATED`
An RUI_TERM verb was issued while this verb was pending.

Parameter Check: The following return codes indicate that the verb did not complete successfully because a supplied parameter was in error:

lua_prim_rc
`LUA_PARAMETER_CHECK`

lua_sec_rc
Possible values are:

`LUA_BAD_SESSION_ID`
The *lua_sid* parameter did not match the session ID of any active LUA LU session.

`LUA_BID_ALREADY_ENABLED`
The RUI_BID verb was rejected because a previous RUI_BID verb was already outstanding for this session. Only one RUI_BID can be outstanding for each session at any time.

AIX, LINUX

`LUA_INVALID_POST_HANDLE`
The *lua_post_handle* parameter was not a valid pointer to a callback routine.

`LUA_RESERVED_FIELD_NOT_ZERO`
A reserved field in the verb record, or a parameter that is not used by this verb, was set to a nonzero value.

`LUA_VERB_LENGTH_INVALID`
The value of the *lua_verb_length* parameter was less than the length of the verb record required for this verb.

State Check: The following return codes indicate that the verb was issued in a session state in which it was not valid:

lua_prim_rc
`LUA_STATE_CHECK`

lua_sec_rc

`LUA_NO_RUI_SESSION`
An RUI_INIT or RUI_INIT_PRIMARY verb has not yet completed successfully for the LU name specified on this verb, or the session has failed.

Negative Response Sent to Host: The following return code indicates that Communications Server detected an error in the data received from the host. Instead of passing the received message to the application on an RUI_READ verb, Communications Server discards the message (and the rest of the chain if it is in a chain), and sends a negative response to the host. LUA informs the application on a subsequent RUI_READ or RUI_BID verb that a negative response was sent.

lua_prim_rc
`LUA_NEGATIVE_RSP`

lua_sec_rc
The secondary return code contains the sense code sent to the host on the negative response. See "SNA Information" on page 34, for information about interpreting the sense code values that can be returned.

A `0` (zero) secondary return code indicates that, following a previous RUI_WRITE of a negative response to a message in the middle of a chain, Communications Server has now received and discarded all messages from this chain.

Other Conditions: The following return codes indicate that the verb record supplied was valid, but the verb did not complete successfully:

lua_prim_rc
`LUA_UNSUCCESSFUL`

lua_sec_rc

`LUA_INVALID_PROCESS`
The operating system process that issued this verb was not the same process that issued the RUI_INIT or RUI_INIT_PRIMARY verb for this session. Only the process that started a session can issue verbs on that session.

The following return codes indicate that the verb did not complete successfully for other reasons:

lua_prim_rc

LUA_COMM_SUBSYSTEM_ABENDED
A required Communications Server software component (such as the node) has terminated. Contact your System Administrator if necessary.

lua_prim_rc

LUA_SESSION_FAILURE
The LUA session has failed.

If the session was started using RUI_INIT (not RUI_INIT_PRIMARY) and the secondary return code is not `LUA_RUI_LOGIC_ERROR`, then this LU can be reinitialized using an RUI_REINIT. If it is not reinitialized, then an RUI_TERM must be issued before an RUI_INIT or RUI_INIT_PRIMARY can be issued for the same LU.

lua_sec_rc
Possible values are:

LUA_LU_COMPONENT_DISCONNECTED
This return code indicates that the LUA session has failed because of a problem with the communications link or with the host LU.

LUA_RUI_LOGIC_ERROR
This return code indicates one of the following:

- The host system has violated SNA protocols
- An internal error was detected within LUA

Attempt to reproduce the problem with SNA tracing active (contact your System Administrator if necessary), and check that the host is sending correct data. If this does not solve the problem, contact your Communications Server support personnel.

lua_prim_rc

LUA_INVALID_VERB
Either the *lua_verb* parameter or the *lua_opcode* parameter was not valid. The verb did not execute.

WINDOWS

lua_prim_rc

LUA_STACK_TOO_SMALL
The stack size of the application is too small for LUA to complete the request. Increase the stack size of your application.

lua_prim_rc

LUA_COMM_SUBSYSTEM_NOT_LOADED
The Remote API Client software was not started, or the node was either not started or not configured properly for LUA applications. Check the Communications Server LUA configuration parameters and start the Remote API Client and the node before running your application.

lua_prim_rc

LUA_UNEXPECTED_DOS_ERROR
An operating system error occurred.

lua_sec_rc
This value is the operating system return code. Check your operating system documentation for the meaning of this return code.

Interaction with Other Verbs

The RUI_INIT or RUI_INIT_PRIMARY verb must complete successfully before this verb can be issued.

Only one RUI_BID for each session can be outstanding at any one time.

After the RUI_BID verb has completed successfully, it may be re-issued by setting the *lua_flag1.bid_enable* parameter on a subsequent RUI_READ verb. If the verb is to be re-issued in this way, the application program must not free or modify the storage associated with the RUI_BID verb record.

If a message arrives from the host when an RUI_READ and an RUI_BID are both outstanding, the RUI_READ completes and the RUI_BID is left in progress.

Usage and Restrictions

Each message that arrives will only be bid once. Once an RUI_BID verb has indicated that data is waiting on a particular session flow, the application should issue the RUI_READ verb to receive the data. Any subsequent RUI_BID will not report data arriving on that session flow until the message which was bid has been accepted by issuing an RUI_READ verb.

The following items describe the difference between the *lua_max_length* and *lua_data_length* parameters returned on this verb:

- The *lua_max_length* parameter indicates the length of the waiting message. When issuing the RUI_READ verb to accept the message, the application should supply a data buffer of at least this size, to ensure that the message can be received without truncation.
- The *lua_data_length* parameter indicates the length of data in *lua_peek_data*. If this is less than 12, indicating that the waiting message is shorter than 12 bytes, the remaining bytes in *lua_peek_data* are undefined and the application should not attempt to examine them.

RUI_INIT

The RUI_INIT verb establishes the SSCP-LU session for a given LU, or establishes an SSCP-LU session for the least recently used LU in a given LU pool.

AIX, LINUX

In general, the application specifies the name of an LU or an LU pool to be used for the session. Communications Server also provides an extended form of RUI_INIT, in which the application can identify the LU by specifying its PU name and LU number instead of its LU name; this function is not supported by other LUA implementations. The differences between the normal and extended versions of RUI_INIT are indicated where appropriate in the parameter descriptions in this section.

If the RUI application acts as an SNA primary for communications with a downstream LU, it must use RUI_INIT_PRIMARY instead of RUI_INIT.

Supplied Parameters

The application supplies the following parameters:

lua_verb
: `LUA_VERB_RUI`

lua_verb_length
: The length in bytes of the LUA verb record.

  AIX, LINUX

  Set this to `sizeof(LUA_VERB_RECORD)`.

  For compatibility with other LUA implementations, the value `sizeof(LUA_COMMON)` is also accepted if you are using the standard form of RUI_INIT and not the extended form.

  WINDOWS

  Set this to `sizeof(LUA_COMMON)`.

lua_opcode
: `LUA_OPCODE_RUI_INIT`

lua_correlator
: Optional. A four-byte value that you can use to correlate this verb with other processing within your application. LUA does not use or change this information.

lua_luname
: The name in ASCII of the LU or LU pool for which you want to start the session. This must match the name of an LU of type 0–3, or of an LU pool, configured for Communications Server. The name is used as follows:

  - If the name is the name of an LU that is not in a pool, Communications Server attempts to start the session using this LU. An application can start multiple sessions by using multiple RUI_INIT verbs with a different LU for each verb; it cannot start more than one session for the same LU.
  - If the name is the name of an LU pool, or the name of an LU within a pool, Communications Server attempts to start the session using the named LU, if it is available, or otherwise the least recently used LU from the pool. An application can start multiple sessions using the same pool; Communications Server will assign a different LU from the pool for each session. The name of the actual LU used for the session is a returned parameter on the RUI_INIT verb.

  This parameter must be eight bytes long; pad on the right with spaces, `0x20`, if the name is shorter than eight characters.

AIX, LINUX

The application can use the extended form of RUI_INIT to identify the LU by its PU name and LU number, instead of by its LU name. To do this, set *lua_luname* to eight binary zeros, and specify the PU name and LU number in the *lua_puname* and *lua_lunumber* parameters.

lua_post_handle

AIX, LINUX

A pointer to a callback routine. If the verb completes asynchronously, LUA will call this routine to indicate completion of the verb.

WINDOWS

If the VCB is used in an `RUI` function call, set this field to an event handle. If the VCB is used in a `WinRUI` function call, this field is reserved.

For more information, see Chapter 2, "Designing and Writing LUA Applications," on page 13.

lua_encr_decr_option

Session-level cryptography option. Communications Server accepts the following two values:

`0`	Session-level cryptography is not used.
`128`	Encryption and decryption are performed by the application program.

Any other value will result in the return code `LUA_ENCR_DECR_LOAD_ERROR`. (Values in the range `1` to `127`, indicating user-defined encryption and decryption routines, are supported by OS/2 Extended Edition's LUA implementation but not by Communications Server.)

AIX, LINUX

The following parameters are used only if the *lua_luname* parameter is set to eight binary zeros (the extended form of RUI_INIT). If *lua_luname* specifies the LU name (the standard form of RUI_INIT), these parameters are reserved.

lua_puname

The name of the PU that owns the LU to be used for the session. The name must be in ASCII, padded with spaces on the right (`0x20`). It must match a PU name defined in the Communications Server configuration.

lua_lunumber

The LU number of the LU to be used for the session. This must match the LU number of a type 0–3 LU configured to use the specified PU.

An application can start multiple sessions by using multiple RUI_INIT verbs with a different LU for each verb; it cannot start more than one session for the same LU.

wait_for_link
Normally, if the application issues RUI_INIT for an LU that cannot currently be used because the underlying communications link is inactive, the RUI_INIT verb fails. Set this parameter to `1` to override this default behavior so that LUA waits for the link and LU to become active before RUI_INIT completes, or `0` (zero) to use the default behavior.

[redacted]

Returned Parameters

LUA always returns the following parameter:

lua_flag2.async
This flag is set to `1` if the verb completed asynchronously, or `0` if the verb completed synchronously. (RUI_INIT will always complete asynchronously, unless it returns an error such as `LUA_PARAMETER_CHECK`.)

Other returned parameters depend on whether the verb completed successfully; see the following sections.

Successful Execution

If the verb executes successfully, LUA returns the following parameters.

lua_prim_rc
`LUA_OK`

lua_sid A session ID for the new session. This can be used by subsequent verbs to identify this session.

lua_luname
The name of the LU used by the new session. If the LU name in the request parameters specified an LU pool, or if the application used the extended form of RUI_INIT and specified the PU name and LU number instead of the LU name, Communications Server uses this parameter to return the name of the actual LU assigned to the session. Subsequent verbs must use this returned name (not the name specified in the request parameters) to identify the session.

Unsuccessful Execution

If a verb does not complete successfully, LUA returns a primary return code to indicate the type of error and a secondary return code to provide specific details about the reason for unsuccessful execution.

Verb Canceled: The following return codes indicate that the verb did not complete successfully because it was canceled by another verb:

lua_prim_rc
`LUA_CANCELLED`

lua_sec_rc

`LUA_TERMINATED`
An RUI_TERM verb was issued before the RUI_INIT had completed.

Parameter Check: The following return codes indicate that the verb did not complete successfully because a supplied parameter was in error:

lua_prim_rc
: `LUA_PARAMETER_CHECK`

lua_sec_rc
: Possible values are:

`LUA_INVALID_LUNAME`
The LU identified by the *lua_luname* parameter could not be found on any active nodes.Check that the LU name or LU pool name is defined in the configuration file and that the node on which it is configured has been started.

AIX, LINUX

`LUA_INVALID_POST_HANDLE`
The *lua_post_handle* parameter was not a valid pointer to a callback routine.

`LUA_RESERVED_FIELD_NOT_ZERO`
A reserved field in the verb record, or a parameter that is not used by this verb, was set to a nonzero value.

`LUA_VERB_LENGTH_INVALID`
The value of the *lua_verb_length* parameter was less than the length of the verb record required for this verb.

AIX, LINUX

The following parameters are used only if the *lua_luname* parameter is set to eight binary zeros (the extended form of RUI_INIT). If *lua_luname* specifies the LU name (the standard form of RUI_INIT), these parameters are reserved.

`LUA_INVALID_FORMAT`
The reserved parameter *rui_init_format* was set to a nonzero value.

`LUA_INVALID_PUNAME`
The *lua_puname* parameter did not match any PU name defined in the Communications Server configuration.

`LUA_INVALID_LUNUMBER`
The *lua_lunumber* parameter did not match the number of a type 0–3 LU defined to use the specified PU.

State Check: The following return codes indicate that the verb was issued in a session state in which it was not valid:

lua_prim_rc
: `LUA_STATE_CHECK`

lua_sec_rc

LUA_DUPLICATE_RUI_INIT
An RUI_INIT verb is currently being processed for this session.

Other Conditions: The following return codes indicate that the verb record supplied was valid, but the verb did not complete successfully:

lua_prim_rc
LUA_UNSUCCESSFUL

lua_sec_rc
Possible values are:

LUA_COMMAND_COUNT_ERROR
The verb specified the name of an LU pool, or the name of an LU in a pool, but all LUs in the pool are in use.

LUA_ENCR_DECR_LOAD_ERROR
The verb specified a value for *lua_encr_decr_option* other than 0 or 128.

LUA_INVALID_PROCESS
The LU specified by the *lua_luname* parameter is in use by another process.

LUA_LINK_NOT_STARTED
The connection to the host has not been started; none of the links it could use are active.

(any other value)
Any other secondary return code here is an SNA sense code. For information about interpreting the SNA sense codes that can be returned, see "SNA Information" on page 34.

The following sense code values are specific to Communications Server, and may indicate mismatches between the Communications Server configuration and the host configuration:

0x10020000
The host has not sent an activate physical unit (ACTPU) for the PU that owns the requested LU.

0x10110000
The host has not sent an ACTLU for the requested LU. This generally indicates that the LU is not configured at the host.

0x10120000
The host has not sent an ACTLU for the requested LU. The host supports DDDLU (Dynamic Definition of Dependent LUs), but DDDLU processing for this LU has failed.

The following return codes indicate that the verb did not complete successfully for other reasons:

lua_prim_rc

LUA_COMM_SUBSYSTEM_NOT_LOADED
This return code indicates one of the following conditions:

- The Remote API Client software was not started. Start the Remote API Client before running your application.
- There are no active Communications Server nodes. The local node that owns the requested LU, or a local node that owns one

or more LUs in the requested LU pool, must be started before you can use LUA verbs. Contact your System Administrator if necessary.

lua_prim_rc

LUA_COMM_SUBSYSTEM_ABENDED
A required Communications Server software component (such as the node) has terminated or has been stopped. Contact your System Administrator if necessary.

lua_prim_rc

LUA_SESSION_FAILURE
The LUA session has failed.

If the secondary return code is not LUA_RUI_LOGIC_ERROR, then this LU can be reinitialized using an RUI_REINIT. If it is not reinitialized, then an RUI_TERM must be issued before an RUI_INIT can be issued for the same LU.

lua_sec_rc

LUA_LU_COMPONENT_DISCONNECTED
The LUA session has failed because of a problem with the communications link or with the host LU.

lua_prim_rc

LUA_INVALID_VERB
Either the *lua_verb* parameter or the *lua_opcode* parameter was not valid. The verb did not execute.

WINDOWS

lua_prim_rc

LUA_STACK_TOO_SMALL
The stack size of the application is too small for LUA to complete the request. Increase the stack size of your application.

lua_prim_rc

LUA_UNEXPECTED_DOS_ERROR
An operating system error occurred.

lua_sec_rc
This value is the operating system return code. Check your operating system documentation for the meaning of this return code.

Interaction with Other Verbs

This verb must be the first LUA verb issued for the session.

Until this verb has completed successfully, the only other LUA verb that can be issued for this session is RUI_TERM (which will terminate a pending RUI_INIT).

All other verbs issued on this session must identify the session using one of the following returned parameters from this verb:

- The session ID, returned to the application in the *lua_sid* parameter
- The LU name, returned to the application in the *lua_luname* parameter

Usage and Restrictions

The RUI_INIT verb completes after an ACTLU is received from the host. If necessary, the verb waits indefinitely. If an ACTLU has already been received prior to the RUI_INIT verb, LUA sends a NOTIFY to the host to inform it that the LU is ready for use. Neither the ACTLU or NOTIFY is visible to the LUA application.

Once the RUI_INIT verb has completed successfully, this session uses the LU for which the session was started. No other LUA session (from this or any other application) can use the LU until the RUI_TERM verb is issued.

If the RUI_INIT verb returns with an `LUA_IN_PROGRESS` primary return code then the Session ID will be returned in the *lua_sid* parameter. This Session ID is the same as that returned when the verb completes successfully and can be used with the RUI_TERM verb to terminate an outstanding RUI_INIT verb.

RUI_INIT_PRIMARY

AIX, LINUX

The RUI_INIT_PRIMARY verb establishes the SSCP-LU session for an SNA Primary application that is communicating with a downstream LU. (If the RUI application acts as an SNA secondary and communicates with a host LU, it must use RUI_INIT instead of RUI_INIT_PRIMARY.)

Supplied Parameters

The application supplies the following parameters:

lua_verb
: `LUA_VERB_RUI`

lua_verb_length
: The length in bytes of the LUA verb record.

  Set this to `sizeof(LUA_COMMON)`.

lua_opcode
: `LUA_OPCODE_RUI_INIT_PRIMARY`

lua_correlator
: Optional. A four-byte value that you can use to correlate this verb with other processing within your application. LUA does not use or change this information.

lua_luname
: The name in ASCII of the LU for which you want to start the session. This must match the name of a downstream LU configured for use with SNA Gateway, or an LU created implicitly from a downstream LU template.

  This parameter must be eight bytes long; pad on the right with spaces, `0x20`, if the name is shorter than eight characters.

lua_max_length
: The length of a buffer supplied to receive a copy of the ACTLU(+RSP) RU received from the downstream PU. If the application does not need to

receive this information, it can specify a null pointer in the *lua_data_ptr* parameter, in which case it does not need to provide a data buffer.

lua_data_ptr
A pointer to the buffer supplied to receive a copy of the ACTLU(+RSP) RU received from the downstream PU. If the application does not need to receive this information, it can specify a null pointer, and the information will not be returned.

lua_post_handle
A pointer to a callback routine. If the verb completes asynchronously, LUA will call this routine to indicate completion of the verb. For more information, see Chapter 2, “Designing and Writing LUA Applications,” on page 13.

lua_encr_decr_option
Session-level cryptography option. Communications Server accepts the following two values:

`0`	Session-level cryptography is not used.
`128`	Encryption and decryption are performed by the application program.

Any other value will result in the return code `LUA_ENCR_DECR_LOAD_ERROR`. (Values in the range 1 to 127, indicating user-defined encryption and decryption routines, are supported by OS/2 Extended Edition’s LUA implementation but not by Communications Server.)

Returned Parameters

LUA always returns the following parameter:

lua_flag2.async
This flag is set to `1` if the verb completed asynchronously, or `0` if the verb completed synchronously. (RUI_INIT_PRIMARY will always complete asynchronously, unless it returns an error such as `LUA_PARAMETER_CHECK`.)

Other returned parameters depend on whether the verb completed successfully; see the following sections.

Successful Execution

If the verb executes successfully, LUA returns the following parameters.

lua_prim_rc
`LUA_OK`

lua_sid A session ID for the new session. This can be used by subsequent verbs to identify this session.

lua_data_length
The length of the ACTLU(+RSP) RU received from the downstream PU. LUA places the data in the buffer specified by *lua_data_ptr*.

Unsuccessful Execution

If a verb does not complete successfully, LUA returns a primary return code to indicate the type of error and a secondary return code to provide specific details about the reason for unsuccessful execution.

Verb Canceled: The following return codes indicate that the verb did not complete successfully because it was canceled by another verb:

lua_prim_rc
`LUA_CANCELLED`

lua_sec_rc

`LUA_TERMINATED`
An RUI_TERM verb was issued before the RUI_INIT_PRIMARY had completed.

Parameter Check: The following return codes indicate that the verb did not complete successfully because a supplied parameter was in error:

lua_prim_rc
`LUA_PARAMETER_CHECK`

lua_sec_rc
Possible values are:

`LUA_INVALID_LUNAME`
The LU identified by the *lua_luname* parameter could not be found on any active nodes.Check that the LU name is defined in the configuration file and that the nodeon which it is configured has been started.

`LUA_INVALID_POST_HANDLE`
The *lua_post_handle* parameter was not a valid pointer to a callback routine.

`LUA_RESERVED_FIELD_NOT_ZERO`
A reserved field in the verb record, or a parameter that is not used by this verb, was set to a nonzero value.

`LUA_VERB_LENGTH_INVALID`
The value of the *lua_verb_length* parameter was less than the length of the verb record required for this verb.

State Check: The following return codes indicate that the verb was issued in a session state in which it was not valid:

lua_prim_rc
`LUA_STATE_CHECK`

lua_sec_rc

`LUA_DUPLICATE_RUI_INIT_PRIMARY`
An RUI_INIT_PRIMARY verb is currently being processed for this session.

Other Conditions: The following return codes indicate that the verb record supplied was valid, but the verb did not complete successfully:

lua_prim_rc
`LUA_UNSUCCESSFUL`

lua_sec_rc
Possible values are:

`LUA_ENCR_DECR_LOAD_ERROR`
The verb specified a value for *lua_encr_decr_option* other than `0` or `128`.

`LUA_INVALID_PROCESS`
The LU specified by the *lua_luname* parameter is in use by another process.

(any other value)
Any other secondary return code here is an SNA sense code. For information about interpreting the SNA sense codes that can be returned, see “SNA Information” on page 34.

The following return codes indicate that the verb did not complete successfully for other reasons:

lua_prim_rc

LUA_COMM_SUBSYSTEM_NOT_LOADED
This return code indicates one of the following conditions:

- The Remote API Client software was not started. Start the Remote API Client before running your application.
- There are no active Communications Server nodes. The local node that owns the requested downstream LU must be started before you can use LUA verbs. Contact your System Administrator if necessary.

lua_prim_rc

LUA_COMM_SUBSYSTEM_ABENDED
A required Communications Server software component (such as the node) has terminated or has been stopped. Contact your System Administrator if necessary.

lua_prim_rc

LUA_SESSION_FAILURE
The LUA session has failed.

An RUI_TERM must be issued before another RUI_INIT_PRIMARY can be issued for the same LU.

lua_sec_rc

LUA_LU_COMPONENT_DISCONNECTED
The LUA session has failed because of a problem with the communications link or with the host LU.

lua_prim_rc

LUA_INVALID_VERB
Either the *lua_verb* parameter or the *lua_opcode* parameter was not valid. The verb did not execute.

lua_prim_rc

LUA_UNEXPECTED_DOS_ERROR
An operating system error occurred.

lua_sec_rc
This value is the operating system return code. Check your operating system documentation for the meaning of this return code.

Interaction with Other Verbs

This verb must be the first LUA verb issued for the session.

Until this verb has completed successfully, the only other LUA verb that can be issued for this session is RUI_TERM (which will terminate a pending RUI_INIT_PRIMARY).

All other verbs issued on this session must identify the session using one of the following parameters from this verb:

- The session ID, returned to the application in the *lua_sid* parameter
- The LU name, supplied bythe application in the *lua_luname* parameter

Usage and Restrictions

The RUI_INIT_PRIMARY verb completes after an ACTLU positive response is received from the downstream LU. If necessary, the verb waits indefinitely. If the application needs to check the contents of this ACTLU positive response, it can do so by supplying a data buffer on RUI_INIT_PRIMARY (using the *lua_max_length* and *lua_data_ptr* parameters) in which Communications Server returns the contents of the received message.

Once the RUI_INIT_PRIMARY verb has completed successfully, this session uses the LU for which the session was started. No other LUA session (from this or any other application) can use the LU until the RUI_TERM verb is issued, or until an `LUA_SESSION_FAILURE` primary return code is received.

If the RUI_INIT_PRIMARY verb returns with an `LUA_IN_PROGRESS` primary return code then the Session ID will be returned in the *lua_sid* parameter. This Session ID is the same as that returned when the verb completes successfully and can be used with the RUI_TERM verb to terminate an outstanding RUI_INIT_PRIMARY verb.

[redacted]

RUI_PURGE

The RUI_PURGE verb cancels a previous RUI_READ. An RUI_READ may wait indefinitely if it is sent without using the *lua_flag1.nowait* (immediate return) option, and no data is available on the specified flow; RUI_PURGE forces the waiting verb to return (with the primary return code `LUA_CANCELLED`).

Supplied Parameters

The application supplies the following parameters:

lua_verb
: `LUA_VERB_RUI`

lua_verb_length
: The length in bytes of the LUA verb record.

  Set this to `sizeof(LUA_COMMON)`.

lua_opcode
: `LUA_OPCODE_RUI_PURGE`

lua_correlator
: Optional. A four-byte value that you can use to correlate this verb with other processing within your application. LUA does not use or change this information.

lua_luname
: The name in ASCII of the LU used by the session. This must match the LU name of an active LUA session, as returned on the RUI_INIT or RUI_INIT_PRIMARY verb.

This parameter is required only if the *lua_sid* parameter is `0` (zero). If a session ID is supplied in *lua_sid*, LUA does not use this parameter.

This parameter must be eight bytes long; pad on the right with spaces, `0x20`, if the name is shorter than eight characters.

lua_sid The session ID of the session. This must match a session ID returned on a previous RUI_INIT or RUI_INIT_PRIMARY verb.

This parameter is optional; if you do not specify the session ID, you must specify the LU name for the session in the *lua_luname* parameter.

lua_data_ptr
A pointer to the RUI_READ VCB that is to be purged.

lua_post_handle

AIX, LINUX

A pointer to a callback routine that LUA will call to indicate completion if the verb completes asynchronously.

WINDOWS

If the VCB is used in an `RUI` function call, set this field to an event handle. If the VCB is used in a `WinRUI` function call, this field is reserved.

For more information, see Chapter 2, "Designing and Writing LUA Applications," on page 13.

Returned Parameters

LUA always returns the following parameter:

lua_flag2.async
This flag is set to 1 if the verb completed asynchronously, or `0` (zero) if the verb completed synchronously.

Other returned parameters depend on whether the verb completed successfully; see the following sections.

Successful Execution

If the verb completed successfully, the following parameters are returned:

lua_prim_rc
`LUA_OK`

lua_sid If the application specified the *lua_luname* parameter when issuing this verb, rather than specifying the session ID, LUA supplies the session ID.

Unsuccessful Execution

If a verb does not complete successfully, LUA returns a primary return code to indicate the type of error and a secondary return code to provide specific details about the reason for unsuccessful execution.

Verb Canceled: The following return codes indicate that the verb did not complete successfully because it was canceled by another verb:

lua_prim_rc
    LUA_CANCELLED

lua_sec_rc

LUA_TERMINATED
    An RUI_TERM verb was issued while this verb was pending.

Parameter Check: The following return codes indicate that the verb did not complete successfully because a supplied parameter was in error:

lua_prim_rc
    LUA_PARAMETER_CHECK

lua_sec_rc
    Possible values are:

LUA_BAD_DATA_PTR
    The *lua_data_ptr* parameter was set to 0 (zero).

LUA_BAD_SESSION_ID
    The *lua_sid* parameter did not match the session ID of any active LUA LU session.

AIX, LINUX

LUA_INVALID_POST_HANDLE
    The *lua_post_handle* parameter was not a valid pointer to a callback routine.

LUA_RESERVED_FIELD_NOT_ZERO
    A reserved field in the verb record, or a parameter that is not used by this verb, was set to a nonzero value.

LUA_VERB_LENGTH_INVALID
    The value of the *lua_verb_length* parameter was less than the length of the verb record required for this verb.

State Check: The following return codes indicate that the verb was issued in a session state in which it was not valid:

lua_prim_rc
    LUA_STATE_CHECK

lua_sec_rc

LUA_NO_RUI_SESSION
    An RUI_INIT or RUI_INIT_PRIMARY verb has not yet completed successfully for the LU name specified on this verb, or the session has failed.

Other Conditions: The following return codes indicate that the verb record supplied was valid, but the verb did not complete successfully:

lua_prim_rc
    LUA_UNSUCCESSFUL

lua_sec_rc
    Possible values are:

LUA_INVALID_PROCESS
The operating system process that issued this verb was not the same process that issued the RUI_INIT or RUI_INIT_PRIMARY verb for this session. Only the process that started a session can issue verbs on that session.

LUA_NO_READ_TO_PURGE
Either the *lua_data_ptr* parameter did not contain a pointer to an RUI_READ VCB, or the RUI_READ verb completed before the RUI_PURGE verb was issued.

The following return codes indicate that the verb did not complete successfully for other reasons:

lua_prim_rc

LUA_COMM_SUBSYSTEM_ABENDED
A required Communications Server software component (such as the node) has terminated or has been stopped. Contact your System Administrator if necessary.

lua_prim_rc

LUA_SESSION_FAILURE
The LUA session has failed.

If the session was started using RUI_INIT (not RUI_INIT_PRIMARY) and the secondary return code is not `LUA_RUI_LOGIC_ERROR`, then this LU can be reinitialized using an RUI_REINIT. If it is not reinitialized, then an RUI_TERM must be issued before an RUI_INIT or RUI_INIT_PRIMARY can be issued for the same LU.

lua_sec_rc
Possible values are:

LUA_LU_COMPONENT_DISCONNECTED
This return code indicates that the LUA session has failed because of a problem with the communications link or with the host LU.

LUA_RUI_LOGIC_ERROR
This return code indicates one of the following:

- The host system has violated SNA protocols
- An internal error was detected within LUA

Attempt to reproduce the problem with SNA tracing active (contact your System Administrator if necessary), and check that the host is sending correct data. If this does not solve the problem, contact your Communications Server support personnel.

lua_prim_rc

LUA_INVALID_VERB
Either the *lua_verb* parameter or the *lua_opcode* parameter was not valid. The verb did not execute.

WINDOWS

lua_prim_rc

LUA_STACK_TOO_SMALL
The stack size of the application is too small for LUA to complete the request. Increase the stack size of your application.

[redacted]

lua_prim_rc

LUA_UNEXPECTED_DOS_ERROR
An operating system error occurred.

lua_sec_rc
This value is the operating system return code. Check your operating system documentation for the meaning of this return code.

lua_prim_rc

LUA_COMM_SUBSYSTEM_NOT_LOADED
The Remote API Client software was not started, or the node was either not started or not configured properly for LUA applications. Check the Communications Server LUA configuration parameters and start the Remote API Client and the node before running your application.

Interaction with Other Verbs

This verb can only be used when an RUI_READ has been issued and is pending completion (that is, the primary return code is `IN_PROGRESS`).

RUI_READ

The RUI_READ verb receives data or status information sent from the host to the application's LU.

You can specify a particular message flow (LU normal, LU expedited, SSCP normal, or SSCP expedited) from which to read data, or you can specify more than one message flow. You can have multiple RUI_READ verbs outstanding, provided that no two of them specify the same flow.

Supplied Parameters

The application supplies the following parameters:

lua_verb
`LUA_VERB_RUI`

lua_verb_length
The length in bytes of the LUA verb record.

Set this to `sizeof(LUA_COMMON)`.

lua_opcode
`LUA_OPCODE_RUI_READ`

lua_correlator
Optional. A four-byte value that you can use to correlate this verb with other processing within your application. LUA does not use or change this information.

lua_luname
: The name in ASCII of the LU used by the session. This must match the LU name of an active LUA session, as returned on the RUI_INIT or RUI_INIT_PRIMARY verb.

  This parameter is required only if the *lua_sid* parameter is `0` (zero). If a session ID is supplied in *lua_sid*, LUA does not use this parameter.

  This parameter must be eight bytes long; pad on the right with spaces, `0x20`, if the name is shorter than eight characters.

lua_sid
: The session ID of the session. This must match a session ID returned on a previous RUI_INIT or RUI_INIT_PRIMARY verb.

  This parameter is optional; if you do not specify the session ID, you must specify the LU name for the session in the *lua_luname* parameter.

lua_max_length
: The length of the buffer supplied to receive the data.

lua_data_ptr
: A pointer to the buffer supplied to receive the data.

lua_post_handle

  AIX, LINUX

  A pointer to a callback routine that LUA will call to indicate completion if the verb completes asynchronously.

  WINDOWS

  If the VCB is used in an `RUI` function call, set this field to an event handle. If the VCB is used in a `WinRUI` function call, this field is reserved.

For more information, see Chapter 2, “Designing and Writing LUA Applications,” on page 13.

lua_flag1 **parameters**
: Set the *lua_flag1.nowait* parameter to 1 if you want the RUI_READ verb to return as soon as possible whether or not data is available to be read, or set it to `0` (zero) if you want the verb to wait for data before returning.

  Note: Setting the *lua_flag1.nowait* parameter to 1 does not mean that the verb will complete synchronously. The LUA library needs to communicate with the local node to determine whether or not any data is available, and this normally requires an asynchronous verb return to avoid blocking the application. The parameter means that, if there is no data available immediately, the asynchronous verb return will occur as soon as possible to indicate this.

  Set the *lua_flag1.bid_enable* parameter to 1 to re-enable the most recent RUI_BID verb (equivalent to issuing RUI_BID again with exactly the same parameters as before), or set it to `0` (zero) if you do not want to re-enable RUI_BID. Re-enabling the previous RUI_BID re-uses the VCB originally allocated for it, so this VCB must not have been freed or modified. (For more information, see “Interaction with Other Verbs” on page 83.)

Set one or more of the following flags to 1 to indicate which message flow to read data from:

lua_flag1.sscp_exp

lua_flag1.lu_exp

lua_flag1.sscp_norm

lua_flag1.lu_norm

If more than one flag is set, the highest-priority data available will be returned. The order of priorities (highest first) is: SSCP expedited, LU expedited, SSCP normal, LU normal. The equivalent flag in the *lua_flag2* group will be set to indicate which flow the data was read from (see "Returned Parameters").

The Communications Server implementation of LUA does not return data on the SSCP expedited flow. The application can set the *sscp_exp* flag, for compatibility with other LUA implementations, but data will never be returned on this flow.

Returned Parameters

LUA always returns the following parameters:

lua_flag2.async
This parameter is set to 1 if the verb completed asynchronously, or 0 if the verb completed synchronously.

lua_flag2.bid_enable
This parameter is set to 1 if an RUI_BID was successfully re-enabled, or to 0 if it was not re-enabled.

Other returned parameters depend on whether the verb completed successfully; see the following sections.

Successful Execution or Truncated Data

If the verb executes successfully, LUA returns the following parameters:

lua_prim_rc
`LUA_OK`

The following parameters are returned if the verb completes successfully. They are also returned if the verb returns with truncated data because the *lua_data_length* parameter supplied was too small (see "Other Conditions" on page 81).

lua_sid If the application specified the *lua_luname* parameter when issuing this verb, rather than specifying the session ID, LUA supplies the session ID.

lua_data_length
The length of the data received. LUA places the data in the buffer specified by *lua_data_ptr*.

lua_th Information from the transmission header (TH) of the received message.

lua_rh Information from the request/response header (RH) of the received message.

lua_message_type
Message type of the received message that will be one of the following:

`LUA_MESSAGE_TYPE_LU_DATA`

`LUA_MESSAGE_TYPE_SSCP_DATA`

LUA_MESSAGE_TYPE_RSP

LUA_MESSAGE_TYPE_BID

LUA_MESSAGE_TYPE_BIND

LUA_MESSAGE_TYPE_BIS

LUA_MESSAGE_TYPE_CANCEL

LUA_MESSAGE_TYPE_CHASE

LUA_MESSAGE_TYPE_CLEAR

LUA_MESSAGE_TYPE_CRV

LUA_MESSAGE_TYPE_LUSTAT_LU

LUA_MESSAGE_TYPE_LUSTAT_SSCP

LUA_MESSAGE_TYPE_QC

LUA_MESSAGE_TYPE_QEC

LUA_MESSAGE_TYPE_RELQ

LUA_MESSAGE_TYPE_RTR

LUA_MESSAGE_TYPE_SBI

LUA_MESSAGE_TYPE_SHUTD

LUA_MESSAGE_TYPE_SIGNAL

LUA_MESSAGE_TYPE_SDT

LUA_MESSAGE_TYPE_STSN

LUA_MESSAGE_TYPE_UNBIND

AIX, LINUX

The following values can be returned only to an RUI primary application (one that started the session using RUI_INIT_PRIMARY):

LUA_MESSAGE_TYPE_INIT_SELF

LUA_MESSAGE_TYPE_NOTIFY

LUA_MESSAGE_TYPE_TERM_SELF

lua_flag2 **parameters**

One of the following flags will be set to 1, to indicate on which message flow the data was received:

lua_flag2.lu_exp

lua_flag2.sscp_norm

lua_flag2.lu_norm

The Communications Server implementation of LUA does not return data on the SSCP expedited flow, and so the *sscp_exp* flag will never be set (although it may be set by other LUA implementations).

Unsuccessful Execution

If a verb does not complete successfully, LUA returns a primary return code to indicate the type of error and a secondary return code to provide specific details about the reason for unsuccessful execution.

Verb Canceled: The following return codes indicate that the verb did not complete successfully because it was canceled by another verb or by an internal error:

lua_prim_rc
`LUA_CANCELLED`

lua_sec_rc
Possible values are:

`LUA_PURGED`
This RUI_READ verb has been canceled by an RUI_PURGE verb.

`LUA_TERMINATED`
An RUI_TERM verb was issued while this verb was pending.

Parameter Check: The following return codes indicate that the verb did not complete successfully because a supplied parameter was in error:

lua_prim_rc
`LUA_PARAMETER_CHECK`

lua_sec_rc
Possible values are:

`LUA_BAD_DATA_PTR`
The *lua_data_ptr* parameter contained a value that was not valid.

`LUA_BAD_SESSION_ID`
The *lua_sid* parameter did not match the session ID of any active LUA LU session.

`LUA_BID_ALREADY_ENABLED`
The *lua_flag1.bid_enable* parameter was set to re-enable an RUI_BID verb, but the previous RUI_BID verb was still in progress.

`LUA_DUPLICATE_READ_FLOW`
The flow flags in the *lua_flag1* group specified one or more session flows for which an RUI_READ verb was already outstanding. Only one RUI_READ at a time can be waiting on each session flow.

`LUA_INVALID_FLOW`
None of the *lua_flag1* flow flags was set. At least one of these flags must be set to 1 to indicate which flow or flows to read from.

AIX, LINUX

`LUA_INVALID_POST_HANDLE`
The *lua_post_handle* parameter was not a valid pointer to a callback routine.

`LUA_NO_PREVIOUS_BID_ENABLED`
The *lua_flag1.bid_enable* parameter was set to re-enable an RUI_BID

verb, but there was no previous RUI_BID verb that could be enabled. (For more information, see "Interaction with Other Verbs" on page 83.)

LUA_RESERVED_FIELD_NOT_ZERO
A reserved field in the verb record, or a parameter that is not used by this verb, was set to a nonzero value.

LUA_VERB_LENGTH_INVALID
The value of the *lua_verb_length* parameter was less than the length of the verb record required for this verb.

State Check: The following return codes indicate that the verb was issued in a session state in which it was not valid:

lua_prim_rc
LUA_STATE_CHECK

lua_sec_rc

LUA_NO_RUI_SESSION
An RUI_INIT or RUI_INIT_PRIMARY verb has not yet completed successfully for the LU name specified on this verb, or the session has failed.

Negative Response Sent to Host: The following primary return code indicates one of the following two cases, which can be distinguished by the secondary return code:

- Communications Server detected an error in the data received from the host. Instead of passing the received message to the application on an RUI_READ verb, Communications Server discards the message (and the rest of the chain if it is in a chain), and sends a negative response to the host. LUA informs the application on a subsequent RUI_READ or RUI_BID verb that a negative response was sent.
- The LUA application previously sent a negative response to a message in the middle of a chain. Communications Server has purged subsequent messages in this chain, and is now reporting to the application that all messages from the chain have been received and purged.

lua_prim_rc
LUA_NEGATIVE_RSP

lua_sec_rc
A nonzero secondary return code contains the sense code sent to the host on the negative response. This indicates that Communications Server detected an error in the host data, and sent a negative response to the host. For information about interpreting the sense code values that can be returned, see "SNA Information" on page 34.

A 0 (zero) secondary return code indicates that, following a previous RUI_WRITE of a negative response to a message in the middle of a chain, Communications Server has now received and discarded all messages from this chain.

Other Conditions: The following return codes indicate that the verb record supplied was valid, but the verb did not complete successfully:

lua_prim_rc
LUA_UNSUCCESSFUL

lua_sec_rc
Possible values are:

LUA_DATA_TRUNCATED
The *lua_data_length* parameter was smaller than the actual length of data received on the message. Only *lua_data_length* bytes of data were returned to the verb; the remaining data was discarded. Additional parameters are also returned if this secondary return code is obtained; see "Successful Execution or Truncated Data" on page 78.

LUA_NO_DATA
The *lua_flag1.nowait* parameter was set to indicate immediate return without waiting for data, and no data was currently available on the specified session flow or flows.

LUA_INVALID_PROCESS
The operating system process that issued this verb was not the same process that issued the RUI_INIT or RUI_INIT_PRIMARY verb for this session. Only the process that started a session can issue verbs on that session.

The following return codes indicate that the verb did not complete successfully for other reasons:

lua_prim_rc

LUA_COMM_SUBSYSTEM_ABENDED
A required Communications Server software component (such as the node) has terminated or has been stopped. Contact your System Administrator if necessary.

lua_prim_rc

LUA_SESSION_FAILURE
The LUA session has failed.

If the session was started using RUI_INIT (not RUI_INIT_PRIMARY) and the secondary return code is not `LUA_RUI_LOGIC_ERROR`, then this LU can be reinitialized using an RUI_REINIT. If it is not reinitialized, then an RUI_TERM must be issued before an RUI_INIT or RUI_INIT_PRIMARY can be issued for the same LU.

lua_sec_rc
Possible values are:

LUA_LU_COMPONENT_DISCONNECTED
This return code indicates that the LUA session has failed because of a problem with the communications link or with the host LU.

LUA_RUI_LOGIC_ERROR
This return code indicates one of the following:
- The host system has violated SNA protocols
- An internal error was detected within LUA

Attempt to reproduce the problem with SNA tracing active (contact your System Administrator if necessary), and check that the host is sending correct data. If this does not solve the problem, contact your Communications Server support personnel.

lua_prim_rc

`LUA_INVALID_VERB`
: Either the *lua_verb* parameter or the *lua_opcode* parameter was not valid. The verb did not execute.

WINDOWS

lua_prim_rc

`LUA_STACK_TOO_SMALL`
: The stack size of the application is too small for LUA to complete the request. Increase the stack size of your application.

lua_prim_rc

`LUA_UNEXPECTED_DOS_ERROR`
: An operating system error occurred.

lua_sec_rc
: This value is the operating system return code. Check your operating system documentation for the meaning of this return code.

lua_prim_rc
: `LUA_COMM_SUBSYSTEM_NOT_LOADED`

The Remote API Client software was not started, or the node was either not started or not configured properly for LUA applications. Check the Communications Server LUA configuration parameters and start the Remote API Client and the node before running your application.

Interaction with Other Verbs

The RUI_INIT or RUI_INIT_PRIMARY verb must have completed successfully before this verb can be issued.

While an existing RUI_READ is pending, you can issue another RUI_READ only if it specifies a different session flow or flows from pending RUI_READs; you cannot have more than one RUI_READ outstanding for the same session flow.

The *lua_flag1.bid_enable* parameter can only be used if the following are true:
- RUI_BID has already been issued successfully and has completed
- The storage allocated for the RUI_BID verb has not been freed or modified
- No other RUI_BID is pending

If you use this parameter to re-enable a previous RUI_BID, at least one of the message flow flags on RUI_READ must still be set, to indicate the flow or flows on which the application will accept data. If the first data to be received is on a flow accepted by the RUI_READ verb, RUI_READ will return with this data, and RUI_BID will not return. Otherwise, RUI_BID will return to indicate that there is data to be read (since RUI_BID accepts data on all flows, it will always accept the data if RUI_READ does not). The application must then issue another RUI_READ on the appropriate flow to obtain the data.

If you want to use RUI_BID to handle data on all flows, rather than having the data on a particular flow handled by RUI_READ in preference to RUI_BID, you need to re-issue RUI_BID explicitly instead of using RUI_READ to re-enable the previous RUI_BID.

Usage and Restrictions

If the data received is longer than the *lua_max_length* parameter, it will be truncated; only *lua_max_length* bytes of data will be returned. The primary and secondary return codes `LUA_UNSUCCESSFUL` and `LUA_DATA_TRUNCATED` will also be returned.

Once a message has been read using the RUI_READ verb, it is removed from the incoming message queue, and cannot be accessed again. (The RUI_BID verb may be used as a non-destructive read; the application can use it to check the type of data available, but the data remains on the incoming queue and need not be used immediately.)

Pacing may be used on the primary-to-secondary half-session (this is specified in the host configuration), in order to protect the LUA application from being flooded with messages. If the LUA application is slow to read messages, Communications Server delays the sending of pacing responses to the host in order to slow it down.

RUI_REINIT

AIX, LINUX

The RUI_REINIT verb re-establishes the SSCP-LU session after a session failure. It is intended for use by an application that was using an LU from a pool, and needs to ensure that it accesses the same LU in order to continue processing. (Normally, an application recovers from a session failure by issuing RUI_TERM followed by a second RUI_INIT; however, if the application was using an LU from a pool, the second RUI_INIT will not necessarily get the same LU as the original one.)

This verb cannot be used to restart a Primary RUI session (one that was started using RUI_INIT_PRIMARY).

Supplied Parameters

The application supplies the following parameters:

lua_verb
: `LUA_VERB_RUI`

lua_verb_length
: The length in bytes of the LUA verb record. Set this to `sizeof(LUA_COMMON)`.

lua_opcode
: `LUA_OPCODE_RUI_REINIT`

lua_correlator
: Optional. A four-byte value that you can use to correlate this verb with other processing within your application. LUA does not use or change this information.

lua_luname
: The name in ASCII of the LU that was being used by the failed session.

This must match the name returned on the original RUI_INIT verb (not necessarily the same as the name that was supplied to the verb).

This parameter is required only if the *lua_sid* parameter is `0` (zero). If a session ID is supplied in *lua_sid*, LUA does not use this parameter.

This parameter must be eight bytes long; pad on the right with spaces, `0x20`, if the name is shorter than eight characters.

lua_sid
The session ID of the session. This must match a session ID returned on the previous RUI_INIT verb for the failed session.

This parameter is optional; if you do not specify the session ID, you must specify the LU name for the session in the *lua_luname* parameter.

lua_post_handle
A pointer to a callback routine. If the verb completes asynchronously, LUA will call this routine to indicate completion of the verb. For more information, see Chapter 2, "Designing and Writing LUA Applications," on page 13.

Returned Parameters

LUA always returns the following parameter:

lua_flag2.async
This flag is set to `1` if the verb completed asynchronously, or `0` if the verb completed synchronously. (RUI_REINIT will always complete asynchronously, unless it returns an error such as `LUA_PARAMETER_CHECK`.)

Other returned parameters depend on whether the verb completed successfully; see the following sections.

Successful Execution

If the verb executes successfully, LUA returns the following parameter:

lua_prim_rc
`LUA_OK`

Unsuccessful Execution

If a verb does not complete successfully, LUA returns a primary return code to indicate the type of error and a secondary return code to provide specific details about the reason for unsuccessful execution.

Verb Canceled: The following return codes indicate that the verb did not complete successfully because it was canceled by another verb:

lua_prim_rc
`LUA_CANCELLED`

lua_sec_rc

`LUA_TERMINATED`
An RUI_TERM verb was issued before the RUI_REINIT had completed.

Parameter Check: The following return codes indicate that the verb did not complete successfully because a supplied parameter was in error:

lua_prim_rc
`LUA_PARAMETER_CHECK`

lua_sec_rc
Possible values are:

LUA_INVALID_POST_HANDLE
The *lua_post_handle* parameter was not a valid pointer to a callback routine.

LUA_RESERVED_FIELD_NOT_ZERO
A reserved field in the verb record, or a parameter that is not used by this verb, was set to a nonzero value.

LUA_VERB_LENGTH_INVALID
The value of the *lua_verb_length* parameter was less than the length of the verb record required for this verb.

State Check: The following return codes indicate that the verb was issued in a session state in which it was not valid:

lua_prim_rc
LUA_STATE_CHECK

lua_sec_rc
Possible values are:

LUA_NO_RUI_SESSION
An RUI_INIT verb has not previously completed successfully for the specified LU name or session ID.

LUA_DUPLICATE_RUI_INIT
An RUI_REINIT verb is currently being processed for this session.

LUA_REINIT_INVALID
Session failure has not occurred for this session.

Other Conditions: The following return codes indicate that the verb record supplied was valid, but the verb did not complete successfully:

lua_prim_rc
LUA_UNSUCCESSFUL

lua_sec_rc

LUA_INVALID_PROCESS
The original RUI_INIT verb was issued from a different operating system process.

The following return codes indicate that the verb did not complete successfully for other reasons:

lua_prim_rc

LUA_COMM_SUBSYSTEM_NOT_LOADED
This return code indicates one of the following conditions:

- The Remote API Client software was not started. Start the Remote API Client before running your application.
- There are no active Communications Server nodes. The local node that owns the requested LU, or a local node that owns one or more LUs in the requested LU pool, must be started before you can use LUA verbs. Contact your System Administrator if necessary.

lua_prim_rc

LUA_COMM_SUBSYSTEM_ABENDED
A required Communications Server software component (such as the node) has terminated or has been stopped. Contact your System Administrator if necessary.

lua_prim_rc

LUA_SESSION_FAILURE
The LUA session has failed.

If the secondary return code is not `LUA_RUI_LOGIC_ERROR`, then this LU can be reinitialized using an RUI_REINIT. If it is not reinitialized, then an RUI_TERM must be issued before an RUI_INIT can be issued for the same LU.

lua_sec_rc
Possible values are:

LUA_LU_COMPONENT_DISCONNECTED
The LUA session has failed because of a problem with the communications link or with the host LU.

LUA_RUI_LOGIC_ERROR
This return code indicates one of the following:

- The host system has violated SNA protocols
- An internal error was detected within LUA

Attempt to reproduce the problem with SNA tracing active (contact your System Administrator if necessary), and check that the host is sending correct data. If this does not solve the problem, contact your Communications Server support personnel.

lua_prim_rc

LUA_INVALID_VERB
Either the *lua_verb* parameter or the *lua_opcode* parameter was not valid. The verb did not execute.

lua_prim_rc

LUA_UNEXPECTED_DOS_ERROR
An operating system error occurred.

lua_sec_rc
This value is the operating system return code. Check your operating system documentation for the meaning of this return code.

Interaction with Other Verbs

This verb can only be issued if a previous LUA verb has returned with primary return code `LUA_SESSION_FAILURE`, and with a secondary return code other than `LUA_RUI_LOGIC_ERROR`.

Until this verb has completed successfully, the only other LUA verb that can be issued for this session is RUI_TERM (which will terminate a pending RUI_REINIT).

Usage and Restrictions

The RUI_REINIT verb completes after an ACTLU is received from the host. If necessary, the verb waits indefinitely. If an ACTLU has already been received prior to the RUI_REINIT verb, the verb returns immediately with primary return code LUA_OK.

Once the RUI_REINIT verb has completed successfully, this session uses the LU for which the session was started. No other LUA session (from this or any other application) can use the LU until the RUI_TERM verb is issued, or until an `LUA_SESSION_FAILURE` primary return code is received.

If the secondary return code is not `LUA_RUI_LOGIC_ERROR`, then this LU can be reinitialized using an RUI_REINIT. If it is not reinitialized, then an RUI_TERM must be issued before an RUI_INIT can be issued for the same LU.

The session ID of the restarted session is the same as the session ID before the failure. Unlike RUI_INIT, RUI_REINIT does not return this session ID; the application should either use the session ID that was returned to the original RUI_INIT verb, or access the session using its LU name.

[redacted]

RUI_TERM

The RUI_TERM verb ends both the LU session and the SSCP session for a given LU.

Supplied Parameters

The application supplies the following parameters:

lua_verb
    `LUA_VERB_RUI`

lua_verb_length
    The length in bytes of the LUA verb record.

    Set this to `sizeof(LUA_COMMON)`.

lua_opcode
    `LUA_OPCODE_RUI_TERM`

lua_correlator
    Optional. A four-byte value that you can use to correlate this verb with other processing within your application. LUA does not use or change this information.

lua_luname
    The name in ASCII of the LU used by the session. This must match the LU name of an active LUA session, as returned on the RUI_INIT or RUI_INIT_PRIMARY verb (or the LU name that was specified on an outstanding RUI_INIT, RUI_INIT_PRIMARY, or RUI_REINIT verb).

    This parameter is required only if the *lua_sid* parameter is `0` (zero). If a session ID is supplied in *lua_sid*, LUA does not use this parameter.

    This parameter must be eight bytes long; pad on the right with spaces, `0x20`, if the name is shorter than eight characters.

lua_sid The session ID of the session. This must match a session ID returned on a previous RUI_INIT or RUI_INIT_PRIMARY verb.

This parameter is optional; if you do not specify the session ID, you must specify the LU name for the session in the *lua_luname* parameter.

lua_post_handle

AIX, LINUX

A pointer to a callback routine that LUA will call to indicate completion if the verb completes asynchronously.

WINDOWS

If the VCB is used in an `RUI` function call, set this field to an event handle. If the VCB is used in a `WinRUI` function call, this field is reserved.

For more information, see Chapter 2, "Designing and Writing LUA Applications," on page 13.

Returned Parameters

LUA always returns the following parameter:

lua_flag2.async
This flag is set to 1 if the verb completed asynchronously, or `0` if the verb completed synchronously.

Other returned parameters depend on whether the verb completed successfully; see the following sections.

Successful Execution

If the verb executes successfully, LUA returns the following parameters:

lua_prim_rc
`LUA_OK`

Unsuccessful Execution

If a verb does not complete successfully, LUA returns a primary return code to indicate the type of error and a secondary return code to provide specific details about the reason for unsuccessful execution.

Parameter Check: The following return codes indicate that the verb did not complete successfully because a supplied parameter was in error:

lua_prim_rc
`LUA_PARAMETER_CHECK`

lua_sec_rc
Possible values are:

`LUA_BAD_SESSION_ID`
The *lua_sid* parameter did not match the session ID of any active LUA LU session.

AIX, LINUX

LUA_INVALID_POST_HANDLE
The *lua_post_handle* parameter was not a valid pointer to a callback routine.

LUA_RESERVED_FIELD_NOT_ZERO
A reserved field in the verb record, or a parameter that is not used by this verb, was set to a nonzero value.

LUA_VERB_LENGTH_INVALID
The value of the *lua_verb_length* parameter was less than the length of the verb record required for this verb.

State Check: The following return codes indicate that the verb was issued in a session state in which it was not valid:

lua_prim_rc
`LUA_STATE_CHECK`

lua_sec_rc

LUA_NO_RUI_SESSION
Either there is no LUA session with the LU name specified on this verb, or the session has failed.

If the RUI_TERM verb was issued to cancel an outstanding RUI_INIT, RUI_INIT_PRIMARY, or RUI_REINIT verb, using the *lua_luname* parameter supplied to the outstanding verb, this return code may indicate that the RUI_INIT, RUI_INIT_PRIMARY, or RUI_REINIT completed before this verb was processed. The verb may have completed unsuccessfully (and so there is no session), or RUI_INIT may have completed successfully using a different LU from the pool specified by *lua_luname* (and so there is no session for the specified LU name).

Other Conditions: The following return codes indicate that the verb record supplied was valid, but the verb did not complete successfully:

lua_prim_rc
`LUA_UNSUCCESSFUL`

lua_sec_rc
Possible values are:

LUA_COMMAND_COUNT_ERROR
An RUI_TERM was already pending when the verb was issued.

LUA_INVALID_PROCESS
The operating system process that issued this verb was not the same process that issued the RUI_INIT or RUI_INIT_PRIMARY verb for this session. Only the process that started a session can issue verbs on that session.

The following return codes indicate that the verb did not complete successfully for other reasons:

lua_prim_rc

LUA_COMM_SUBSYSTEM_ABENDED
A required Communications Server software component (such as the node) has terminated or has been stopped. Contact your System Administrator if necessary.

lua_prim_rc

LUA_SESSION_FAILURE
The LUA session has failed.

lua_sec_rc
Possible values are:

LUA_LU_COMPONENT_DISCONNECTED
This return code indicates that the LUA session has failed because of a problem with the communications link or with the host LU.

LUA_RUI_LOGIC_ERROR
This return code indicates one of the following:

- The host system has violated SNA protocols
- An internal error was detected within LUA

Attempt to reproduce the problem with SNA tracing active (contact your System Administrator if necessary), and check that the host is sending correct data. If this does not solve the problem, contact your Communications Server support personnel.

lua_prim_rc

LUA_INVALID_VERB
Either the *lua_verb* parameter or the *lua_opcode* parameter was not valid. The verb did not execute.

WINDOWS

lua_prim_rc

LUA_STACK_TOO_SMALL
The stack size of the application is too small for LUA to complete the request. Increase the stack size of your application.

lua_prim_rc

LUA_UNEXPECTED_DOS_ERROR
An operating system error occurred.

lua_sec_rc
This value is the operating system return code. Check your operating system documentation for the meaning of this return code.

lua_prim_rc

LUA_COMM_SUBSYSTEM_NOT_LOADED
The Remote API Client software was not started, or the node was either not started or not configured properly for LUA applications. Check the Communications Server LUA configuration parameters and start the Remote API Client and the node before running your application.

Interaction with Other Verbs

This verb may be issued at any time after the RUI_INIT, RUI_INIT_PRIMARY, or RUI_REINIT verb has been issued (whether or not it has completed).

If any other LUA verb is pending when RUI_TERM is issued, no further processing on the pending verb will take place, and it will return with a primary return code of `LUA_CANCELLED`.

After this verb has completed, no other LUA verb can be issued for this session.

AIX, LINUX

If the session was started using RUI_INIT_PRIMARY, Communications Server terminates the session by sending DACTLU to the downstream LU. RUI_TERM does not wait for the DACTLU response before returning. The application can reissue RUI_INIT_PRIMARY as soon as RUI_TERM has finished, to start a new session with the downstream LU; however, Communications Server cannot process this RUI_INIT_PRIMARY until it has received the DACTLU response, and so the RUI_INIT_PRIMARY may take some time to complete.

RUI_WRITE

The RUI_WRITE verb sends an SNA request or response unit from the LUA application to the host, over either the LU session or the SSCP session.

Supplied Parameters

The application supplies the following parameters:

lua_verb
: `LUA_VERB_RUI`

lua_verb_length
: The length in bytes of the LUA verb record.

  Set this to `sizeof(LUA_COMMON)`.

lua_opcode
: `LUA_OPCODE_RUI_WRITE`

lua_correlator
: Optional. A four-byte value that you can use to correlate this verb with other processing within your application. LUA does not use or change this information.

lua_luname
: The name in ASCII of the LU used by the session. This must match the LU name of an active LUA session, as returned on the RUI_INIT or RUI_INIT_PRIMARY verb.

  This parameter is required only if the *lua_sid* parameter is `0` (zero). If a session ID is supplied in *lua_sid*, LUA does not use this parameter.

  This parameter must be eight bytes long; pad on the right with spaces, `0x20`, if the name is shorter than eight characters.

lua_sid
The session ID of the session. This must match a session ID returned on a previous RUI_INIT or RUI_INIT_PRIMARY verb.

This parameter is optional; if you do not specify the session ID, you must specify the LU name for the session in the *lua_luname* parameter.

lua_data_length
The length of the supplied data. When sending data on the LU normal flow, the maximum length is as specified in the BIND received from the host; for all other flows the maximum length is 256 bytes.

When sending a positive response, this parameter is normally set to `0` (zero). LUA will complete the response based on the supplied sequence number. In the case of a positive response to a BIND or STSN, an extended response is allowed, so a nonzero value may be used.

When sending a negative response, set this parameter to the length of the SNA sense code (four bytes), which is supplied in the data buffer.

lua_data_ptr
A pointer to the buffer containing the supplied data.

For a request, or a positive response that requires data, the buffer should contain the entire RU. The length of the RU must be specified in *lua_data_length*.

For a negative response, the buffer should contain the SNA sense code.

lua_post_handle

AIX, LINUX

A pointer to a callback routine that LUA will call to indicate completion if the verb completes asynchronously.

WINDOWS

If the VCB is used in an `RUI` function call, set this field to an event handle. If the VCB is used in a `WinRUI` function call, this field is reserved.

For more information, see Chapter 2, "Designing and Writing LUA Applications," on page 13.

lua_th.snf
Required only when sending a response. The sequence number of the request to which this is the response.

lua_rh
When sending a request, most of the *lua_rh* bits must be set to correspond to the RH (request header) of the message to be sent. Do not set *lua_rh.pi* and *lua_rh.qri*; these will be set by LUA.

When sending a response, only the following two *lua_rh* bits are used. The others must be `0` (zero). The *lua_rh* bits are:

lua_rh.rri
Set to `1` to indicate a response

lua_rh.ri
Set to `0` for a positive response, or `1` for a negative response

lua_flag1 **parameters**
Set one of the following flags to 1 to indicate which message flow the data is to be sent on:

lua_flag1.lu_exp

lua_flag1.sscp_norm

lua_flag1.lu_norm

One and only one of the flags must be set to 1. Communications Server does not allow applications to send data on the SSCP expedited flow (the *lua_flag1.sscp_exp* flag).

Returned Parameters

LUA always returns the following parameter:

lua_flag2.async
This flag is set to 1 if the verb completed asynchronously, or `0` (zero) if the verb completed synchronously.

Other returned parameters depend on whether the verb completed successfully; see the following sections.

Successful Execution

If the verb executes successfully, LUA returns the following parameters:

lua_prim_rc
`LUA_OK`

lua_sid If the application specified the *lua_luname* parameter when issuing this verb, rather than specifying the session ID, LUA supplies the session ID.

lua_th The completed TH of the message written, including the fields filled in by LUA. You may need to save the value of *lua_th.snf* (the sequence number) for correlation with responses from the host.

lua_rh The completed RH of the message written, including the fields filled in by LUA.

lua_flag2 **parameters**
One of the following flags will be set to 1 to indicate which message flow the data was sent on:

lua_flag2.lu_exp

lua_flag2.sscp_norm

lua_flag2.lu_norm

The Communications Server implementation of LUA does not allow applications to send data on the SSCP expedited flow, and so will never set the *sscp_exp* flag (although other LUA implementations may set it).

Unsuccessful Execution

If a verb does not complete successfully, LUA returns a primary return code to indicate the type of error and a secondary return code to provide specific details about the reason for unsuccessful execution.

Verb Canceled: The following return codes indicate that the verb did not complete successfully because it was canceled by another verb:

lua_prim_rc
LUA_CANCELLED

lua_sec_rc

LUA_TERMINATED
The verb was canceled because an RUI_TERM verb was issued for this session.

Parameter Check: The following return codes indicate that the verb did not complete successfully because a supplied parameter was in error:

lua_prim_rc
LUA_PARAMETER_CHECK

lua_sec_rc
Possible values are:

LUA_BAD_DATA_PTR
The *lua_data_ptr* parameter contained a value that was not valid.

LUA_BAD_SESSION_ID
The *lua_sid* parameter did not match the session ID of any active LUA LU session.

LUA_DUPLICATE_WRITE_FLOW
An RUI_WRITE was already outstanding for the session flow specified on this verb (the session flow is specified by setting one of the *lua_flag1* flow flags to 1). Only one RUI_WRITE at a time can be outstanding on each session flow.

LUA_INVALID_FLOW
The *lua_flag1.sscp_exp* flow flag was set, indicating that the message should be sent on the SSCP expedited flow. Communications Server does not allow applications to send data on this flow.

AIX, LINUX

LUA_INVALID_POST_HANDLE
The *lua_post_handle* parameter was not a valid pointer to a callback routine.

LUA_MULTIPLE_WRITE_FLOWS
More than one of the *lua_flag1* flow flags was set to 1. One and only one of these flags must be set to 1, to indicate which session flow the data is to be sent on.

LUA_REQUIRED_FIELD_MISSING
This return code indicates one of the following cases:

- None of the *lua_flag1* flow flags was set. One and only one of these flags must be set to 1.
- The RUI_WRITE verb was used to send a response, and the response required more data than was supplied.

LUA_RESERVED_FIELD_NOT_ZERO
A reserved field in the verb record, or a parameter that is not used by this verb, was set to a nonzero value.

LUA_VERB_LENGTH_INVALID
The value of the *lua_verb_length* parameter was less than the length of the verb record required for this verb.

State Check: The following return codes indicate that the verb was issued in a session state in which it was not valid:

lua_prim_rc
LUA_STATE_CHECK

lua_sec_rc
Possible values are:

LUA_MODE_INCONSISTENCY
The SNA message sent on the RUI_WRITE was not valid at this time. This is caused by trying to send data on the LU session before the session is bound. Check the sequence of SNA messages sent.

LUA_NO_RUI_SESSION
An RUI_INIT or RUI_INIT_PRIMARY verb has not yet completed successfully for the LU name specified on this verb, or the session has failed.

Other Conditions: The following return codes indicate that the verb record supplied was valid, but the verb did not complete successfully:

lua_prim_rc
LUA_UNSUCCESSFUL

lua_sec_rc
Possible values are:

LUA_FUNCTION_NOT_SUPPORTED
This return code indicates one of the following cases:

- The *lua_rh.fi* bit (Format Indicator) was set to 1, but the first byte of the supplied RU was not a recognized request code.
- The *lua_rh.ruc* parameter (RU category) specified the Network Control (NC) category; Communications Server does not allow applications to send requests in this category.

LUA_INVALID_PROCESS
The operating system process that issued this verb was not the same process that issued the RUI_INIT or RUI_INIT_PRIMARY verb for this session. Only the process that started a session can issue verbs on that session.

LUA_INVALID_SESSION_PARAMETERS
The application used RUI_WRITE to send a positive response to a BIND message received from the host. However, the Communications Server node cannot accept the BIND parameters as specified, and has sent a negative response to the host. For more information about the BIND profiles accepted by Communications Server, see “SNA Information” on page 34.

LUA_RSP_CORRELATION_ERROR
When using RUI_WRITE to send a response, the *lua_th.snf* parameter (which indicates the sequence number of the received message being responded to) did not contain a valid value.

LUA_RU_LENGTH_ERROR
The *lua_data_length* parameter contained a value that was not valid. When sending data on the LU normal flow, the maximum length is as specified in the BIND received from the host; for all other flows the maximum length is 256 bytes.

(any other value)
Any other secondary return code here is an SNA sense code indicating that the supplied SNA data was not valid or could not be sent. For information about interpreting the SNA sense codes that can be returned, see "SNA Information" on page 34.

The following return codes indicate that the verb did not complete successfully for other reasons:

lua_prim_rc

LUA_COMM_SUBSYSTEM_ABENDED
A required Communications Server software component (such as the node) has terminated or has been stopped. Contact your System Administrator if necessary.

lua_prim_rc

LUA_SESSION_FAILURE
The LUA session has failed.

If the session was started using RUI_INIT (not RUI_INIT_PRIMARY) and the secondary return code is not LUA_RUI_LOGIC_ERROR, then this LU can be reinitialized using an RUI_REINIT. If it is not reinitialized, then an RUI_TERM must be issued before an RUI_INIT or RUI_INIT_PRIMARY can be issued for the same LU.

lua_sec_rc
Possible values are:

LUA_LU_COMPONENT_DISCONNECTED
This return code indicates that the LUA session has failed because of a problem with the communications link or with the host LU.

LUA_RUI_LOGIC_ERROR
This return code indicates one of the following:
- The host system has violated SNA protocols
- An internal error was detected within LUA

Attempt to reproduce the problem with SNA tracing active (contact your System Administrator if necessary), and check that the host is sending correct data. If this does not solve the problem, contact your Communications Server support personnel.

lua_prim_rc

LUA_INVALID_VERB
Either the *lua_verb* parameter or the *lua_opcode* parameter was not valid. The verb did not execute.

WINDOWS

lua_prim_rc

LUA_STACK_TOO_SMALL
The stack size of the application is too small for LUA to complete the request. Increase the stack size of your application.

[redacted]

lua_prim_rc

LUA_UNEXPECTED_DOS_ERROR
An operating system error occurred.

lua_sec_rc
This value is the operating system return code. Check your operating system documentation for the meaning of this return code.

lua_prim_rc

LUA_COMM_SUBSYSTEM_NOT_LOADED
The Remote API Client software was not started, or the node was either not started or not configured properly for LUA applications. Check the Communications Server LUA configuration parameters and start the Remote API Client and the node before running your application.

Interaction with Other Verbs

The RUI_INIT or RUI_INIT_PRIMARY verb must be issued successfully before this verb can be issued.

While an existing RUI_WRITE is pending, you can issue a second RUI_WRITE only if it specifies a different session flow from the pending RUI_WRITE; that is, you cannot have more than one RUI_WRITE outstanding for the same session flow.

The RUI_WRITE verb can be issued on the SSCP normal flow at any time after a successful RUI_INIT or RUI_INIT_PRIMARY verb. RUI_WRITE verbs on the LU expedited or LU normal flows are permitted only after a BIND has been received, and must abide by the protocols specified on the BIND.

Usage and Restrictions

Successful completion of RUI_WRITE indicates that the message was queued successfully to the data link; it does not necessarily indicate that the message was sent successfully, or that the host accepted it.

Pacing may be used on the secondary-to-primary half-session (this is specified on the BIND), in order to prevent the LUA application from sending more data than the Communications Server LU or the host LU can handle. If this is the case, an RUI_WRITE on the LU normal flow may be delayed by LUA and may take some time to complete.

Chapter 5. SLI Verbs

This chapter contains a description of each LUA SLI verb. The following information is provided for each verb:

- Purpose of the verb.
- Parameters (VCB fields) supplied to and returned by LUA. The description of each parameter includes information about the valid values for that parameter, and any additional information necessary.
- Interaction with other verbs.
- Additional information describing the use of the verb.

For details of the Verb Control Block (VCB) used for all verbs, see Chapter 3, "LUA VCB Structure," on page 45.

Symbolic constants are defined in the header files **lua_c.h** and **values_c.h** (AIX / Linux operating system) or **winlua.h** (Windows operating system)for many parameter values. For portability, use the symbolic constant and not the numeric value when setting values for supplied parameters, or when testing values of returned parameters. The file **values_c.h** also includes definitions of parameter types such as `AP_UINT16` that are used in the LUA VCBs.

Parameters marked as "reserved" should always be set to `0` (zero).

SLI_BID

The SLI_BID verb is used by the application to determine when a received message is waiting to be read. This enables the application to determine what data, if any, is available before issuing the SLI_RECEIVE verb.

When a message is available, the SLI_BID verb returns with details of the message flow on which it was received, the message type, the TH and RH of the message, and up to 12 bytes of message data.

The main difference between SLI_BID and SLI_RECEIVE is that SLI_BID enables the application to check the data without removing it from the incoming message queue, so it can be left and accessed at a later stage. The SLI_RECEIVE verb removes the message from the queue, so once the application has read the data it must process it.

Supplied Parameters

The application supplies the following parameters:

lua_verb
  `LUA_VERB_SLI`

lua_verb_length
  The length in bytes of the LUA verb record.

  Set this to `sizeof(LUA_VERB_RECORD)`.

lua_opcode
  `LUA_OPCODE_SLI_BID`

lua_correlator
: Optional. A four-byte value that you can use to correlate this verb with other processing within your application. LUA does not use or change this information.

lua_luname
: The name in ASCII of the LU used by the session. This must match the LU name of an active LUA session (as returned on the SLI_OPEN verb).

  This parameter is required only if the *lua_sid* parameter is `0` (zero). If a session ID is supplied in *lua_sid*, LUA does not use this parameter.

  This parameter must be eight bytes long; pad on the right with spaces, `0x20`, if the name is shorter than eight characters.

lua_sid
: The session ID of the session. This must match a session ID returned on a previous SLI_OPEN verb.

  This parameter is optional; if you do not specify the session ID, you must specify the LU name for the session in the *lua_luname* parameter.

lua_post_handle

  AIX, LINUX

  A pointer to a callback routine that LUA will call to indicate completion if the verb completes asynchronously.

  WINDOWS

  If the VCB is used in an `SLI` function call, set this field to an event handle. If the VCB is used in a `WinSLI` function call, this field is reserved.

For more information, see Chapter 2, "Designing and Writing LUA Applications," on page 13.

Returned Parameters

LUA always returns the following parameter:

lua_flag2.async
: This flag is set to 1 if the verb completed asynchronously, or `0` (zero) if the verb completed synchronously.

Other returned parameters depend on whether the verb completed successfully; see the following sections.

Successful Execution

If the verb completed successfully, LUA returns the following parameters:

lua_prim_rc
: `LUA_OK`

lua_sid
: If the application specified the *lua_luname* parameter when issuing this verb, rather than specifying the session ID, LUA supplies the session ID.

lua_data_length
: The number of bytes of data returned in the *lua_peek_data* parameter; from `0` to 12.

lua_th The TH of the received message.

lua_rh The RH of the received message.

lua_message_type
: Message type of the received message, which is one of the following:

`LUA_MESSAGE_TYPE_LU_DATA`

`LUA_MESSAGE_TYPE_SSCP_DATA`

`LUA_MESSAGE_TYPE_RSP`

`LUA_MESSAGE_TYPE_BID`

`LUA_MESSAGE_TYPE_BIND`

`LUA_MESSAGE_TYPE_BIS`

`LUA_MESSAGE_TYPE_CANCEL`

`LUA_MESSAGE_TYPE_CHASE`

`LUA_MESSAGE_TYPE_LUSTAT_LU`

`LUA_MESSAGE_TYPE_LUSTAT_SSCP`

`LUA_MESSAGE_TYPE_QC`

`LUA_MESSAGE_TYPE_QEC`

`LUA_MESSAGE_TYPE_RELQ`

`LUA_MESSAGE_TYPE_RTR`

`LUA_MESSAGE_TYPE_SBI`

`LUA_MESSAGE_TYPE_SIGNAL`

`LUA_MESSAGE_TYPE_STSN`

The SLI uses the application's LUA interface extension routines to receive and respond to the BIND and STSN requests.

lua_flag2
: One of the following flags will be set to 1 to indicate on which message flow the data was received:

lua_flag2.sscp_exp

lua_flag2.lu_exp

lua_flag2.sscp_norm

lua_flag2.lu_norm

lua_peek_data
: The first 12 bytes of the message data (or all of the message data if it is shorter than 12 bytes)

If *lua_rh.rri* is off (request unit) and *lua_rh.sdi* is on (sense data included), this indicates that LUA has converted a request unit sent by the host into an exception request (EXR). In this case, bytes 0–3 of *lua_peek_data* contain the sense data associated with the exception, and bytes 4–6 contain up to the first 3 bytes of the original request unit.

Successful Execution: Status Information

If the verb returned LUA status information instead of data, LUA returns the following parameters:

lua_prim_rc
: `LUA_STATUS`

lua_sec_rc

`LUA_READY`
: The SLI session is now ready to process additional commands. This status is used after a previous `LUA_NOT_READY` status was reported, or after an SLI_CLOSE verb completed with *lua_prim_rc* set to `LUA_CANCELLED` and *lua_sec_rc* set to `RECEIVED_UNBIND_HOLD` or `RECEIVED_UNBIND_NORMAL`.

`LUA_NOT_READY`
: The SLI session has been temporarily suspended for one of the following reasons:

- A CLEAR command was received. The session resumes when an SDT command is received.
- An UNBIND command type X'02' (BIND forthcoming) was received. The session is suspended until a BIND, optional CRV and STSN, and SDT commands are received; it resumes after the SDT. Any user extension routines that were supplied by the original SLI_OPEN verb will be called again.
- An UNBIND command type X'01' (normal) was received, and the SLI_OPEN verb for this session specified *lua_session_type* `LUA_SESSION_TYPE_DEDICATED`. The session is suspended until a BIND, optional CRV and STSN, and SDT commands are received; it resumes after the SDT. Any user extension routines that were supplied by the original SLI_OPEN verb will be called again.

The application should issue another SLI_BID or SLI_RECEIVE to receive the `READY` status when the session resumes. It can continue to issue SLI_SEND and SLI_RECEIVE verbs for SSCP normal-flow data even though the session status is `LUA_NOT_READY`.

`LUA_INIT_COMPLETE`
: The application issued SLI_OPEN with type `LUA_OPEN_TYPE_PRIM_SSCP`, and the underlying RUI_INIT verb has now completed. The application can now issue SLI_SEND and SLI_RECEIVE verbs for SSCP normal-flow data.

`LUA_SESSION_END_REQUESTED`
: The host has sent a SHUTD command, requesting the application to shut down the session. The application should issue SLI_CLOSE as soon as it is ready to close the session.

lua_sid If the application specified the *lua_luname* parameter when issuing this verb, rather than specifying the session ID, LUA supplies the session ID.

Unsuccessful Execution

If a verb does not complete successfully, LUA returns a primary return code to indicate the type of error and a secondary return code to provide specific details about the reason for unsuccessful execution.

Verb Canceled: The following return codes indicate that the verb did not complete successfully because it was canceled by another verb:

lua_prim_rc
: `LUA_CANCELLED`

lua_sec_rc

`LUA_TERMINATED`
: An SLI_CLOSE verb was issued while this verb was pending.

Parameter Check: The following return codes indicate that the verb did not complete successfully because a supplied parameter was in error:

lua_prim_rc
: `LUA_PARAMETER_CHECK`

lua_sec_rc
: Possible values are:

`LUA_BAD_SESSION_ID`
: The *lua_sid* parameter did not match the session ID of any active LUA LU session.

`LUA_INVALID_LUNAME`
: The LU identified by the *lua_luname* parameter could not be found on any active nodes.Check that the LU name or LU pool name is defined in the configuration file and that the node on which it is configured has been started.

AIX, LINUX

`LUA_INVALID_POST_HANDLE`
: The *lua_post_handle* parameter was not a valid pointer to a callback routine.

`LUA_RESERVED_FIELD_NOT_ZERO`
: A reserved field in the verb record, or a parameter that is not used by this verb, was set to a nonzero value.

`LUA_VERB_LENGTH_INVALID`
: The value of the *lua_verb_length* parameter was less than the length of the verb record required for this verb.

State Check: The following return codes indicate that the verb was issued in a session state in which it was not valid:

lua_prim_rc
: `LUA_STATE_CHECK`

lua_sec_rc

`LUA_NO_SLI_SESSION`
: An SLI_OPEN verb has not yet completed successfully for the LU specified on this verb, or the session has failed.

`LUA_SLI_BID_PENDING`
: The SLI_BID verb was rejected because a previous SLI_BID verb

was already outstanding for this session. Only one SLI_BID can be outstanding for each session at any time.

Negative Response Sent to Host: The following return code indicates that Communications Server detected an error in the data received from the host. Instead of passing the received message to the application on an SLI_RECEIVE verb, Communications Server discards the message (and the rest of the chain if it is in a chain), and sends a negative response to the host. LUA informs the application on a subsequent SLI_RECEIVE or SLI_BID verb that a negative response was sent.

lua_prim_rc
: `LUA_NEGATIVE_RSP`

lua_sec_rc
: The secondary return code contains the sense code sent to the host on the negative response. See "SNA Information" on page 34, for information about interpreting the sense code values that can be returned.

  A `0` (zero) secondary return code indicates that, following a previous SLI_SEND of a negative response to a message in the middle of a chain, Communications Server has now received and discarded all messages from this chain.

Other Conditions: The following return codes indicate that the verb record supplied was valid, but the verb did not complete successfully:

lua_prim_rc
: `LUA_UNSUCCESSFUL`

lua_sec_rc

`LUA_INVALID_PROCESS`
: The operating system process that issued this verb was not the same process that issued the SLI_OPEN verb for this session. Only the process that started a session can issue verbs on that session.

The following return codes indicate that the verb did not complete successfully for other reasons:

lua_prim_rc

`LUA_COMM_SUBSYSTEM_ABENDED`
: A required Communications Server software component (such as the node) has terminated. Contact your System Administrator if necessary.

lua_prim_rc

`LUA_SESSION_FAILURE`
: The LUA session has failed. To restart it, the application can reissue SLI_OPEN.

lua_sec_rc
: Possible values are:

`LUA_LU_COMPONENT_DISCONNECTED`
: This return code indicates that the LUA session has failed because of a problem with the communications link or with the host LU.

`LUA_RECEIVED_UNBIND`
: This return code indicates that the host sent an UNBIND command

to end the session. This value can occur only if the SLI_OPEN verb for this session specified *lua_session_type* `LUA_SESSION_TYPE_DEDICATED`.

`LUA_RUI_LOGIC_ERROR`
This return code indicates one of the following:

- The host system has violated SNA protocols
- An internal error was detected within LUA

Attempt to reproduce the problem with SNA tracing active (contact your System Administrator if necessary), and check that the host is sending correct data. If this does not solve the problem, contact your Communications Server support personnel.

lua_prim_rc

`LUA_INVALID_VERB`
Either the *lua_verb* parameter or the *lua_opcode* parameter was not valid. The verb did not execute.

WINDOWS

lua_prim_rc

`LUA_STACK_TOO_SMALL`
The stack size of the application is too small for LUA to complete the request. Increase the stack size of your application.

lua_prim_rc

`LUA_UNEXPECTED_DOS_ERROR`
An operating system error occurred.

lua_sec_rc
This value is the operating system return code. Check your operating system documentation for the meaning of this return code.

Interaction with Other Verbs

The SLI_OPEN verb must complete successfully before this verb can be issued.

Only one SLI_BID for each session can be outstanding at any one time.

After the SLI_BID verb has completed successfully, it may be re-issued by setting the *lua_flag1.bid_enable* parameter on a subsequent SLI_RECEIVE verb. If the verb is to be re-issued in this way, the application program must not free or modify the storage associated with the SLI_BID verb record.

If a message arrives from the host when an SLI_RECEIVE and an SLI_BID are both outstanding, the SLI_RECEIVE completes and the SLI_BID is left in progress.

Usage and Restrictions

Each message that arrives will only be bid once. Once an SLI_BID verb has indicated that data is waiting on a particular session flow, the application should issue the SLI_RECEIVE verb to receive the data. Any subsequent SLI_BID will not report data arriving on that session flow until the message which was bid has been accepted by issuing an SLI_RECEIVE verb.

If there is data available on more than one session flow, the data on the highest-priority flow will be returned to the application. The flow priorities are as follows (highest to lowest):

- SSCP expedited
- LU expedited
- SSCP normal
- LU normal

Once a message has been read using the SLI_RECEIVE verb, it is removed from the incoming message queue, and cannot be accessed again. The application can use SLI_BID as a non-destructive read to check the type of data available and determine how to process it, and then issue a subsequent SLI_RECEIVE to collect the data. However, if it issues the SLI_RECEIVE with multiple *lua_flag1* flags set to accept data on more than one flow, it may receive a different message from the one identified in the SLI_BID, if data arrived on a higher-priority flow between the SLI_BID and SLI_RECEIVE verbs. To ensure that it receives the same message that was identified in the SLI_BID, it should set the *lua_flag1* flags on SLI_RECEIVE to accept data only on the flow identified in the SLI_BID response.

The *lua_data_length* parameter indicates the length of data in *lua_peek_data*. If this is less than 12, indicating that the waiting message is shorter than 12 bytes, the remaining bytes in *lua_peek_data* are undefined and the application should not attempt to examine them.

SLI_CLOSE

The SLI_CLOSE verb ends both the LU session and the SSCP session for a given LU.

Supplied Parameters

The application supplies the following parameters:

lua_verb
: `LUA_VERB_SLI`

lua_verb_length
: The length in bytes of the LUA verb record.

  Set this to `sizeof(LUA_VERB_RECORD)`.

lua_opcode
: `LUA_OPCODE_SLI_CLOSE`

lua_correlator
: Optional. A four-byte value that you can use to correlate this verb with other processing within your application. LUA does not use or change this information.

lua_luname
: The name in ASCII of the LU used by the session. This must match the LU name of an active LUA session, as returned on the SLI_OPEN verb (or the LU name that was specified on an outstanding SLI_OPEN verb).

  This parameter is required only if the *lua_sid* parameter is `0` (zero). If a session ID is supplied in *lua_sid*, LUA does not use this parameter.

This parameter must be eight bytes long; pad on the right with spaces, `0x20`, if the name is shorter than eight characters.

lua_sid The session ID of the session. This must match a session ID returned on a previous SLI_OPEN verb.

This parameter is optional; if you do not specify the session ID, you must specify the LU name for the session in the *lua_luname* parameter.

lua_post_handle

AIX, LINUX

A pointer to a callback routine that LUA will call to indicate completion if the verb completes asynchronously.

WINDOWS

If the VCB is used in an `SLI` function call, set this field to an event handle. If the VCB is used in a `WinSLI` function call, this field is reserved.

lua_flag1 **parameters**
Set the *lua_flag1.close_abend* parameter to `1` if you want the session to be closed immediately, or set it to `0` (zero) if you want the SLI to go through the normal exchange of SNA messages with the host to close the session gracefully. For more details of normal or abend close processing, see "Usage and Restrictions" on page 111.

Returned Parameters

LUA always returns the following parameter:

lua_flag2.async
This flag is set to `1` if the verb completed asynchronously, or `0` (zero) if the verb completed synchronously.

Other returned parameters depend on whether the verb completed successfully; see the following sections.

Successful Execution

If the verb executes successfully, LUA returns the following parameters:

lua_prim_rc
`LUA_OK`

Unsuccessful Execution

If a verb does not complete successfully, LUA returns a primary return code to indicate the type of error and a secondary return code to provide specific details about the reason for unsuccessful execution.

Parameter Check: The following return codes indicate that the verb did not complete successfully because a supplied parameter was in error:

lua_prim_rc
`LUA_PARAMETER_CHECK`

lua_sec_rc

Possible values are:

LUA_BAD_SESSION_ID
The *lua_sid* parameter did not match the session ID of any active LUA LU session.

LUA_INVALID_LUNAME
The LU identified by the *lua_luname* parameter could not be found on any active nodes.Check that the LU name or LU pool name is defined in the configuration file and that the node on which it is configured has been started.

AIX, LINUX

LUA_INVALID_POST_HANDLE
The *lua_post_handle* parameter was not a valid pointer to a callback routine.

LUA_RESERVED_FIELD_NOT_ZERO
A reserved field in the verb record, or a parameter that is not used by this verb, was set to a nonzero value.

LUA_VERB_LENGTH_INVALID
The value of the *lua_verb_length* parameter was less than the length of the verb record required for this verb.

State Check: The following return codes indicate that the verb was issued in a session state in which it was not valid:

lua_prim_rc

LUA_STATE_CHECK

lua_sec_rc

Possible values are:

LUA_CLOSE_PENDING
The application issued SLI_CLOSE (normal) when an SLI_CLOSE (either normal or abend) was already in progress, or issued SLI_CLOSE (abend) when an SLI_CLOSE (abend) was already in progress. A second SLI_CLOSE is valid only if it is an an SLI_CLOSE (abend) following an earlier SLI_CLOSE (normal).

LUA_NO_SLI_SESSION
Either there is no LUA session with the LU name specified on this verb, or the session has failed.

If the SLI_CLOSE verb was issued to cancel an outstanding SLI_OPEN verb, using the *lua_luname* parameter supplied to the outstanding verb, this return code may indicate that the SLI_OPEN completed before this verb was processed. The verb may have completed unsuccessfully (and so there is no session), or SLI_OPEN may have completed successfully using a different LU from the pool specified by *lua_luname* (and so there is no session for the specified LU name).

Verb Canceled: The following return codes indicate that the verb did not complete successfully because it was canceled by a message sent from the host:

lua_prim_rc
: `LUA_CANCELLED`

lua_sec_rc
: Possible values are:

  `LUA_RECEIVED_UNBIND_HOLD`
  : This SLI_CLOSE verb has been canceled by an UNBIND type 0x02 (UNBIND with BIND forthcoming) from the host. The session is not closed; the application should issue SLI_BID or SLI_RECEIVE to get status information. Any user extension routines specified by the application on the SLI_OPEN verb will be called again when the host sends the new BIND.

  `LUA_RECEIVED_UNBIND_NORMAL`
  : This SLI_CLOSE verb has been canceled by an UNBIND type 0x01 (normal UNBIND) from the host, and the *lua_session_type* parameter on the SLI_OPEN that started the session was set to `LUA_SESSION_TYPE_DEDICATED`. The session is not closed; the application should issue SLI_BID or SLI_RECEIVE to get status information. Any user extension routines specified by the application on the SLI_OPEN verb will be called again when the host sends the new BIND. If the application wants to end the session without waiting for a new BIND, it should issue SLI_CLOSE (abend).

Other Conditions: The following return codes indicate that the verb record supplied was valid, but the verb did not complete successfully:

lua_prim_rc
: `LUA_UNSUCCESSFUL`

lua_sec_rc

  `LUA_INVALID_PROCESS`
  : The operating system process that issued this verb was not the same process that issued the SLI_OPEN verb for this session. Only the process that started a session can issue verbs on that session.

  `LUA_NAU_INOPERATIVE`
  : A required SNA component (such as the LUA LU) is not active or is in an abnormal state.

  `LUA_NO_SESSION`
  : The SNA session to the remote LU is not active.

  `LUA_SLI_LOGIC_ERROR`
  : This return code indicates one of the following:

  - The host system has violated SNA protocols
  - An internal error was detected within LUA

  Attempt to reproduce the problem with SNA tracing active (contact your System Administrator if necessary), and check that the host is sending correct data. If this does not solve the problem, contact your Communications Server support personnel.

The following return codes indicate that the verb did not complete successfully for other reasons:

lua_prim_rc

LUA_COMM_SUBSYSTEM_ABENDED
A required Communications Server software component (such as the node) has terminated or has been stopped. Contact your System Administrator if necessary.

lua_prim_rc

LUA_COMM_SUBSYSTEM_NOT_LOADED
The Remote API Client software was not started, or the node was either not started or not configured properly for LUA applications. Check the Communications Server LUA configuration parameters and start the Remote API Client and the node before running your application.

lua_prim_rc

LUA_SESSION_FAILURE
The LUA session has failed. To restart it, the application can reissue SLI_OPEN.

lua_sec_rc
Possible values are:

LUA_LU_COMPONENT_DISCONNECTED
This return code indicates that the LUA session has failed because of a problem with the communications link or with the host LU.

LUA_NEGATIVE_RSP_CHASE
This return code indicates that the LUA session has been closed because SLI received a negative response to a CHASE command.

LUA_NEGATIVE_RSP_SHUTD
This return code indicates that the LUA session has been closed because SLI received a negative response to a SHUTD command.

LUA_NEGATIVE_RSP_RSHUTD
This return code indicates that the LUA session has been closed because SLI received a negative response to an RSHUTD command.

LUA_RECEIVED_UNBIND
This return code indicates that the host sent an UNBIND command to end the session. This value can occur only if the SLI_OPEN verb for this session specified *lua_session_type* `LUA_SESSION_TYPE_DEDICATED`.

LUA_UNEXPECTED_SNA_SEQUENCE
This return code indicates that the LUA session has been closed because SLI received an unexpected SNA message from the host.

lua_prim_rc

LUA_STACK_TOO_SMALL
The stack size of the application is too small for LUA to complete the request. Increase the stack size of your application.

lua_prim_rc

LUA_UNEXPECTED_DOS_ERROR
An operating system error occurred.

lua_sec_rc
: This value is the operating system return code. Check your operating system documentation for the meaning of this return code.

lua_prim_rc

`LUA_INVALID_VERB`
: Either the *lua_verb* parameter or the *lua_opcode* parameter was not valid. The verb did not execute.

Interaction with Other Verbs

This verb may be issued at any time after the SLI_OPEN verb has been issued. If SLI_OPEN has not yet completed and the application wants to cancel it, it should do so by issuing SLI_CLOSE with *lua_flag1.close_abend* set to `1` (indicating an abnormal close).

While an SLI_CLOSE (normal) is pending, the application can issue an SLI_CLOSE (abend) if it determines that it needs to end the session quickly without waiting for normal close processing.

If any other LUA verb is pending when SLI_CLOSE is issued, no further processing on the pending verb will take place, and it will return with a primary return code of `LUA_CANCELLED`.

After this verb has completed, no other LUA verb can be issued for this session. The application can issue SLI_OPEN for the same LU or a different LU, to start a new session.

Usage and Restrictions

Session close processing may be initiated either by the host (primary-initiated close) or by the LUA application (secondary-initiated close), as follows. In both cases the application normally sets *lua_flag1.close_abend* to `0` (zero), indicating a normal close in which LUA and the host exchange the usual sequence of messages to end the session.

Primary-initiated close
: The host initiates close processing by sending a SHUTD command, which is returned to the application as a status value of `LUA_SESSION_END_REQUESTED` on an SLI_BID or SLI_RECEIVE verb.

  When the application is ready to close the session, it responds by issuing SLI_CLOSE. This results in the following sequence of messages between LUA and the host.

  - LUA sends CHASE to the host and receives the response.
  - LUA sends Shutdown Complete (SHUTC) to the host and receives the response.
  - Optionally, the host sends CLEAR; LUA receives this and sends the response.
  - The host sends UNBIND; LUA receives this and sends the response.
  - LUA stops the RUI session, and the SLI_CLOSE verb returns.

Secondary-initiated close
: The application initiates close processing by issuing SLI_CLOSE. This results in the following sequence of messages between LUA and the host.

  - LUA sends RSHUTD to the host and receives the response.

- Optionally, the host sends CLEAR; LUA receives this and sends the response.
- The host sends UNBIND; LUA receives this and sends the response.
- LUA stops the RUI session, and the SLI_CLOSE verb returns.

While an SLI_CLOSE (normal) is in progress, the host may interrupt it by sending one of the following messages:

- UNBIND type 0x02 (UNBIND with BIND forthcoming)
- UNBIND type 0x01 (normal UNBIND), if the *lua_session_type* parameter on the SLI_OPEN that started the session was set to `LUA_SESSION_TYPE_DEDICATED`

In either of these cases, the SLI_CLOSE verb returns with the primary return code `CANCELLED`. The session is not closed; the application should issue SLI_BID or SLI_RECEIVE to get status information. Any user extension routines specified by the application on the SLI_OPEN verb will be called again when the host sends the new BIND.

If the application needs to end the session quickly without waiting for the usual message sequence, or to close a dedicated session without waiting for a new BIND after the host has send UNBIND (normal), it does this by issuing SLI_CLOSE with *lua_flag1.close_abend* set to 1. This ends the SLI session; LUA will do all the required cleanup processing to inform the host that the session has ended.

Before issuing SLI_CLOSE (normal), with *lua_flag1.close_abend* set to `0` (zero), the application should ensure that it has received all outstanding messages from the host and sent all the required responses. If a response is required and has not been sent, LUA automatically changes the close type and performs CLOSE (abend) processing as above.

SLI_OPEN

The SLI_OPEN verb establishes the SNA session for a given LU, or for the least recently used LU in a given LU pool.

Supplied Parameters

The application supplies the following parameters:

lua_verb
: `LUA_VERB_SLI`

lua_verb_length
: The length in bytes of the LUA verb record.

: Set this to `sizeof(LUA_VERB_RECORD)`.

lua_opcode
: `LUA_OPCODE_SLI_OPEN`

lua_correlator
: Optional. A four-byte value that you can use to correlate this verb with other processing within your application. LUA does not use or change this information.

lua_luname
: The name in ASCII of the LU or LU pool for which you want to start the session. This must match the name of an LU of type 0–3, or of an LU pool, configured for Communications Server. The name is used as follows:

- If the name is the name of an LU that is not in a pool, Communications Server attempts to start the session using this LU. An application can start multiple sessions by using multiple SLI_OPEN verbs with a different LU for each verb; it cannot start more than one session for the same LU.
- If the name is the name of an LU pool, or the name of an LU within a pool, Communications Server attempts to start the session using the named LU, if it is available, or otherwise the least recently used LU from the pool. An application can start multiple sessions using the same pool; Communications Server will assign a different LU from the pool for each session. The name of the actual LU used for the session is a returned parameter on the SLI_OPEN verb.

This parameter must be eight bytes long; pad on the right with spaces, `0x20`, if the name is shorter than eight characters.

lua_data_length
The length of the unformatted LOGON or INITSELF data supplied in the *lua_data_ptr* parameter, or zero if no data is to be supplied.

lua_data_ptr
A pointer to the message, if any, that must be sent to the host to start the session. This depends on the *lua_init_type* parameter, as follows.

- If *lua_init_type* is `LUA_INIT_TYPE_SEC_IS`, the application must provide an INITSELF request unit containing the required user information such as the mode name and PLU name.
- If *lua_init_type* is `LUA_INIT_TYPE_SEC_LOG`, the application must provide an unformatted LOGON message to be sent on the SSCP normal flow.
- If *lua_init_type* is `LUA_INIT_TYPE_PRIM` or `LUA_INIT_TYPE_PRIM_SSCP`, this parameter is not used and the application must supply a null pointer.

lua_post_handle

AIX, LINUX

A pointer to a callback routine that LUA will call to indicate completion if the verb completes asynchronously. (If the verb fails LUA's initial checks and the SLI entry point returns zero, LUA will not call this routine.)

WINDOWS

If the VCB is used in an `SLI` function call, set this field to an event handle. If the VCB is used in a `WinSLI` function call, this field is reserved.

For more information, see Chapter 2, "Designing and Writing LUA Applications," on page 13.

lua_encr_decr_option
This parameter is reserved and must be set to zero.

lua_init_type
Specifies how LUA should initiate the session. Possible values are:

LUA_INIT_TYPE_SEC_IS
: Secondary-initiated: send the application's INITSELF message (indicated by *lua_data_ptr*) to the host.

LUA_INIT_TYPE_SEC_LOG
: Secondary-initiated: send the application's unformatted LOGON message (indicated by *lua_data_ptr*) to the host.

LUA_INIT_TYPE_PRIM
: Primary-initiated: wait for a BIND from the host.

LUA_INIT_TYPE_PRIM_SSCP
: Primary-initiated with SSCP access: allow the application to issue SLI_SEND and SLI_RECEIVE verbs on the SSCP normal flow, so that it can provide its own INITSELF or LOGON messages and receive their responses. After issuing SLI_OPEN, the application can issue SLI_BID or SLI_RECEIVE to get the status indication `INIT_COMPLETE`, and can then use SLI_SEND and SLI_RECEIVE to send INITSELF or LOGON messages and receive their responses.

lua_session_type
: Specifies how LUA should process an UNBIND type X'01' (normal). Possible values are:

LUA_SESSION_TYPE_NORMAL
: Send a positive response, and issue RUI_TERM so that a NOTIFY(disabled) is sent to the SSCP. The SSCP-LU flow is disabled.

LUA_SESSION_TYPE_DEDICATED
: Send a positive response, and suspend the SLI session until BIND, optional CRV and STSN, and SDT commands are received. NOTIFY(disabled) is not sent to the SSCP. In this case the application can end the suspended session, without waiting for a new BIND from the host, by issuing SLI_CLOSE (abend).

lua_wait
: Timeout (in seconds) for retrying a secondary-initiated session initiation. This parameter is ignored if *lua_init_type* is `LUA_INIT_TYPE_PRIM` or `LUA_INIT_TYPE_PRIM_SSCP`.

LUA retries the session initiation after this timeout (by resending the application's INITSELF or LOGON message) if the host responds to the initial attempt with one of the following messages.

- A negative response to the INITSELF or LOGON with a secondary return code of `RESOURCE_NOT_AVAILABLE`, `SESSION_LIMIT_EXCEEDED`, `SSCP_LU_SESS_NOT_ACTIVE`, or `SESSION_SERVICE_PATH_ERROR`.
- A Network Services Procedure Error (NSPE) message.
- A NOTIFY command, which indicates a procedure error.

If this parameter is set to zero, LUA does not retry the session initiation.

lua_open_extension
: Information about the application's SLI_OPEN extension routines, if any. This parameter is an array of structures, each of which holds information about a specific extension routine.

The application can specify 0–3 extension routines, each of which identifies the application's routine for handling a specific SNA message during session initialization (as indicated by the *lua_routine_type* parameter). These must be specified in consecutive elements in the array, starting with the first; the supplied entries must end with one in which *lua_open_extension.lua_routine_type* is set to `LUA_ROUTINE_TYPE_END`, indicating the end of the list.

lua_open_extension.lua_routine_type
Type of extension routine. Possible values are:

`LUA_ROUTINE_TYPE_BIND`
Routine for checking and responding to a BIND message from the host.

`LUA_ROUTINE_TYPE_SDT`
Routine for checking and responding to an SDT message from the host.

`LUA_ROUTINE_TYPE_STSN`
Routine for checking and responding to an STSN message from the host.

`LUA_ROUTINE_TYPE_END`
This value indicates the end of the list of extension routines. It must be used in the array element immediately following the other routines (or in the first array element if the application is not specifying any extension routines).

AIX, LINUX

lua_open_extension.lua_routine_ptr
Pointer to the extension routine entry point. This parameter is not used in the last array entry, in which *lua_open_extension.lua_routine_type* is set to `LUA_ROUTINE_TYPE_END`.

LUA calls this entry point with the SLI_BIND_ROUTINE, SLI_SDT_ROUTINE, or SLI_STSN_ROUTINE verb, according to the value of the *lua_routine_type* parameter.

WINDOWS

lua_open_extension.lua_module_name
Name of the DLL containing the extension module. This parameter is not used in the last array entry, in which *lua_open_extension.lua_routine_type* is set to `LUA_ROUTINE_TYPE_END`.

lua_open_extension.lua_procedure_name
Procedure name to call within the extension module DLL. This parameter is not used in the last array entry, in which *lua_open_extension.lua_routine_type* is set to `LUA_ROUTINE_TYPE_END`.

LUA calls this entry point with the SLI_BIND_ROUTINE, SLI_SDT_ROUTINE, or SLI_STSN_ROUTINE verb, according to the value of the *lua_routine_type* parameter.

lua_ending_delim
The Communications Server SLI interface does not use this parameter; it is provided for compatibility with applications originally written for other SLI implementations.

Return Value from SLI Entry Point

The SLI_OPEN verb is the only verb for which the SLI entry point returns a value.

- If the verb fails LUA's initial checks (for example because the application supplied incorrect parameters), the SLI function call returns a value of zero to indicate this. The application should check the *lua_prim_rc* and *lua_sec_rc* parameters to determine the cause of the failure. Communications Server does not call the application-supplied callback routine.
- If the initial checks succeed, the SLI function call returns a non-zero value representing the session ID of the new session. If *lua_init_type* was set to `LUA_INIT_TYPE_PRIM_SSCP`, the application can use this session ID for subsequent SLI_BID or SLI_RECEIVE verbs on the SSCP normal flow (to receive the `INIT_COMPLETE` status indicator), and then for SLI_SEND and SLI_RECEIVE verbs on this flow.

  Communications Server then uses the application-supplied callback routine in the same way as for other SLI verbs.

Returned Parameters

LUA always returns the following parameter:

lua_flag2.async
: This flag is set to `1` if the verb completed asynchronously, or `0` (zero) if the verb completed synchronously.

Other returned parameters depend on whether the verb completed successfully; see the following sections.

Successful Execution

If the verb executes successfully, LUA returns the following parameters.

lua_prim_rc
: `LUA_OK`

lua_sid
: A session ID for the new session. This is the same as the return value from the `SLI` entry point for this verb, and can be used by subsequent verbs to identify this session.

lua_luname
: The name of the LU used by the new session. If the LU name in the request parameters specified an LU pool, Communications Server uses this parameter to return the name of the actual LU assigned to the session. Subsequent verbs must use this returned name (not the name specified in the request parameters) to identify the session.

Unsuccessful Execution

If a verb does not complete successfully, LUA returns a primary return code to indicate the type of error and a secondary return code to provide specific details about the reason for unsuccessful execution.

Verb Canceled: The following return codes indicate that the verb did not complete successfully because it was canceled by another verb:

lua_prim_rc
: `LUA_CANCELLED`

lua_sec_rc

`LUA_TERMINATED`
An SLI_CLOSE verb was issued before the SLI_OPEN had completed.

Parameter Check: The following return codes indicate that the verb did not complete successfully because a supplied parameter was in error:

lua_prim_rc
`LUA_PARAMETER_CHECK`

lua_sec_rc
Possible values are:

`LUA_DATA_LENGTH_ERROR`
The *lua_init_type* parameter specified a secondary-initiated session, but the application did not supply the required data to be sent to the host.

`LUA_INVALID_LUNAME`
The LU identified by the *lua_luname* parameter could not be found on any active nodes.Check that the LU name or LU pool name is defined in the configuration file and that the node on which it is configured has been started.

`LUA_INVALID_OPEN_DATA`
The *lua_init_type* parameter was set to `LUA_INIT_TYPE_SEC_IS`, but the data buffer indicated by *lua_data_ptr* did not contain a valid INITSELF command.

`LUA_INVALID_OPEN_INIT_TYPE`
The *lua_init_type* parameter was not set to a valid value.

`LUA_INVALID_OPEN_ROUTINE_TYPE`
The *lua_routine_type* parameter was not set to a valid value.

AIX, LINUX

`LUA_INVALID_POST_HANDLE`
The *lua_post_handle* parameter was not a valid pointer to a callback routine.

`LUA_INVALID_SESSION_TYPE`
The *lua_session_type* parameter was not set to a valid value.

`LUA_INVALID_SLI_ENCR_OPTION`
The *lua_encr_decr_option* parameter was not set to a valid value. For Communications Server, this parameter must be set to `0` (zero).

`LUA_RESERVED_FIELD_NOT_ZERO`
A reserved field in the verb record, or a parameter that is not used by this verb, was set to a nonzero value.

`LUA_VERB_LENGTH_INVALID`
The value of the *lua_verb_length* parameter was less than the length of the verb record required for this verb.

`LUA_BAD_DATA_PTR`
The *lua_data_ptr* parameter contained a value that was not valid.

LUA_BAD_SESSION_ID
: The *lua_sid* parameter did not match the session ID of any active LUA LU session.

State Check: The following return codes indicate that the verb was issued in a session state in which it was not valid:

lua_prim_rc
: LUA_STATE_CHECK

lua_sec_rc

LUA_DUPLICATE_RUI_INIT
: An SLI_OPEN verb is currently being processed for this session.

Other Conditions: The following return codes indicate that the verb record supplied was valid, but the verb did not complete successfully:

lua_prim_rc
: LUA_UNSUCCESSFUL

lua_sec_rc
: Possible values are:

LUA_COMMAND_COUNT_ERROR
: The verb specified the name of an LU pool, or the name of an LU in a pool, but all LUs in the pool are in use.

LUA_INVALID_PROCESS
: The LU specified by the *lua_luname* parameter is in use by another process.

LUA_LINK_NOT_STARTED
: The connection to the host has not been started; none of the links it could use are active.

LUA_SESSION_ALREADY_OPEN
: The application supplied an LU name for which a session has already been started.

LUA_NAU_INOPERATIVE
: A required SNA component (such as the LUA LU) is not active or is in an abnormal state.

LUA_NO_SESSION
: The SNA session to the remote LU is not active.

LUA_SLI_LOGIC_ERROR
: This return code indicates one of the following:

- The host system has violated SNA protocols
- An internal error was detected within LUA

Attempt to reproduce the problem with SNA tracing active (contact your System Administrator if necessary), and check that the host is sending correct data. If this does not solve the problem, contact your Communications Server support personnel.

(any other value)
: Any other secondary return code here is an SNA sense code. For information about interpreting the SNA sense codes that can be returned, see “SNA Information” on page 34.

The following sense code values are specific to Communications Server, and may indicate mismatches between the Communications Server configuration and the host configuration:

0x10020000
: The host has not sent an activate physical unit (ACTPU) for the PU that owns the requested LU.

0x10110000
: The host has not sent an ACTLU for the requested LU. This generally indicates that the LU is not configured at the host.

0x10120000
: The host has not sent an ACTLU for the requested LU. The host supports DDDLU (Dynamic Definition of Dependent LUs), but DDDLU processing for this LU has failed.

The following return codes indicate that the verb did not complete successfully for other reasons:

lua_prim_rc

LUA_COMM_SUBSYSTEM_NOT_LOADED
: This return code indicates one of the following conditions:
- The Remote API Client software was not started. Start the Remote API Client before running your application.
- There are no active Communications Server nodes. The local node that owns the requested LU, or a local node that owns one or more LUs in the requested LU pool, must be started before you can use LUA verbs. Contact your System Administrator if necessary.

lua_prim_rc

LUA_COMM_SUBSYSTEM_ABENDED
: A required Communications Server software component (such as the node) has terminated or has been stopped. Contact your System Administrator if necessary.

lua_prim_rc

LUA_SESSION_FAILURE
: The LUA session has failed. To restart it, the application can reissue SLI_OPEN.

lua_sec_rc
: Possible values are:

LUA_LU_COMPONENT_DISCONNECTED
: The LUA session has failed because of a problem with the communications link or with the host LU.

LUA_SLI_LOGIC_ERROR
: This return code indicates one of the following:
- The host system has violated SNA protocols
- An internal error was detected within LUA

Attempt to reproduce the problem with SNA tracing active (contact your System Administrator if necessary), and check that the host is sending correct data. If this does not solve the problem, contact your Communications Server support personnel.

lua_prim_rc

LUA_INVALID_VERB
Either the *lua_verb* parameter or the *lua_opcode* parameter was not valid. The verb did not execute.

lua_prim_rc

LUA_STACK_TOO_SMALL
The stack size of the application is too small for LUA to complete the request. Increase the stack size of your application.

lua_prim_rc

LUA_UNEXPECTED_DOS_ERROR
An operating system error occurred.

lua_sec_rc
This value is the operating system return code. Check your operating system documentation for the meaning of this return code.

Interaction with Other Verbs

The SLI_OPEN verb must be the first LUA verb issued for the session.

Until this verb has completed successfully, the only other LUA verbs that can be issued for this session are:

- SLI_CLOSE with *lua_flag1.close_abend* set to 1 (indicating an abnormal close), which will cancel the pending SLI_OPEN
- If *lua_init_type* was set to `LUA_INIT_TYPE_PRIM_SSCP`:
  - SLI_BID or SLI_RECEIVE to get the `INIT_COMPLETE` status indication
  - SLI_SEND and SLI_RECEIVE for SSCP normal-flow data, to send INITSELF or LOGON messages and receive their responses.

All other verbs issued on this session must identify the session using one of the following returned parameters from this verb:

- The session ID, returned to the application in the *lua_sid* parameter (and as the return value from the SLI entry point)
- The LU name, returned to the application in the *lua_luname* parameter

Usage and Restrictions

Once the SLI_OPEN verb has completed successfully, this session uses the LU for which the session was started. No other LUA session (from this or any other application) can use the LU until the SLI_CLOSE verb is issued, or until an `LUA_SESSION_FAILURE` primary return code is received.

If the SLI_OPEN verb returns with an `LUA_IN_PROGRESS` primary return code, the Session ID will be returned in the *lua_sid* parameter. This Session ID is the same as that returned when the verb completes successfully, and can be used to issue other verbs on the session.

SLI_PURGE

The SLI_PURGE verb cancels a previous SLI_RECEIVE. An SLI_RECEIVE may wait indefinitely if it is sent without using the *lua_flag1.nowait* (immediate return) option, and no data is available on the specified flow; SLI_PURGE forces the waiting verb to return (with the primary return code `LUA_CANCELLED`).

Supplied Parameters

The application supplies the following parameters:

lua_verb
: `LUA_VERB_SLI`

lua_verb_length
: The length in bytes of the LUA verb record.

  Set this to `sizeof(LUA_VERB_RECORD)`.

lua_opcode
: `LUA_OPCODE_SLI_PURGE`

lua_correlator
: Optional. A four-byte value that you can use to correlate this verb with other processing within your application. LUA does not use or change this information.

lua_luname
: The name in ASCII of the LU used by the session. This must match the LU name of an active LUA session, as returned on the SLI_OPEN verb.

  This parameter is required only if the *lua_sid* parameter is `0` (zero). If a session ID is supplied in *lua_sid*, LUA does not use this parameter.

  This parameter must be eight bytes long; pad on the right with spaces, `0x20`, if the name is shorter than eight characters.

lua_sid
: The session ID of the session. This must match a session ID returned on a previous SLI_OPEN verb.

  This parameter is optional; if you do not specify the session ID, you must specify the LU name for the session in the *lua_luname* parameter.

lua_data_ptr
: A pointer to the SLI_RECEIVE VCB that is to be purged.

lua_post_handle
: AIX, LINUX

  A pointer to a callback routine that LUA will call to indicate completion if the verb completes asynchronously.

  WINDOWS

  If the VCB is used in an `SLI` function call, set this field to an event handle. If the VCB is used in a `WinSLI` function call, this field is reserved.

For more information, see Chapter 2, “Designing and Writing LUA Applications,” on page 13.

Returned Parameters

LUA always returns the following parameter:

lua_flag2.async
: This flag is set to `1` if the verb completed asynchronously, or `0` (zero) if the verb completed synchronously.

Other returned parameters depend on whether the verb completed successfully; see the following sections.

Successful Execution

If the verb completed successfully, the following parameters are returned:

lua_prim_rc
: `LUA_OK`

lua_sid
: If the application specified the *lua_luname* parameter when issuing this verb, rather than specifying the session ID, LUA supplies the session ID.

Unsuccessful Execution

If a verb does not complete successfully, LUA returns a primary return code to indicate the type of error and a secondary return code to provide specific details about the reason for unsuccessful execution.

Verb Canceled: The following return codes indicate that the verb did not complete successfully because it was canceled by another verb:

lua_prim_rc
: `LUA_CANCELLED`

lua_sec_rc

`LUA_TERMINATED`
: An SLI_CLOSE verb was issued while this verb was pending.

Parameter Check: The following return codes indicate that the verb did not complete successfully because a supplied parameter was in error:

lua_prim_rc
: `LUA_PARAMETER_CHECK`

lua_sec_rc
: Possible values are:

`LUA_BAD_DATA_PTR`
: The *lua_data_ptr* parameter was set to `0` (zero).

`LUA_BAD_SESSION_ID`
: The *lua_sid* parameter did not match the session ID of any active LUA LU session.

AIX, LINUX

`LUA_INVALID_POST_HANDLE`
: The *lua_post_handle* parameter was not a valid pointer to a callback routine.

`LUA_RESERVED_FIELD_NOT_ZERO`
: A reserved field in the verb record, or a parameter that is not used by this verb, was set to a nonzero value.

LUA_VERB_LENGTH_INVALID
The value of the *lua_verb_length* parameter was less than the length of the verb record required for this verb.

State Check: The following return codes indicate that the verb was issued in a session state in which it was not valid:

lua_prim_rc
LUA_STATE_CHECK

lua_sec_rc
Possible values are:

LUA_NO_RECEIVE_TO_PURGE
The *lua_data_ptr* parameter was not set to the address of a previous SLI_RECEIVE VCB.

LUA_NO_SLI_SESSION
An SLI_OPEN verb has not yet completed successfully for the LU name specified on this verb, or the session has failed.

LUA_SLI_PURGE_PENDING
An SLI_PURGE verb was already pending when this verb was issued. Only one SLI_PURGE can be outstanding at a time.

Other Conditions: The following return codes indicate that the verb record supplied was valid, but the verb did not complete successfully:

lua_prim_rc
LUA_UNSUCCESSFUL

lua_sec_rc
Possible values are:

LUA_INVALID_PROCESS
The operating system process that issued this verb was not the same process that issued the SLI_OPEN verb for this session. Only the process that started a session can issue verbs on that session.

LUA_NO_RECEIVE_TO_PURGE
The previous SLI_RECEIVE verb completed before the application issued SLI_PURGE. This is not an error condition, so the application program should be designed to handle this without reporting errors.

LUA_NAU_INOPERATIVE
A required SNA component (such as the LUA LU) is not active or is in an abnormal state.

LUA_NO_SESSION
The SNA session to the remote LU is not active.

LUA_SLI_LOGIC_ERROR
This return code indicates one of the following:

- The host system has violated SNA protocols
- An internal error was detected within LUA

Attempt to reproduce the problem with SNA tracing active (contact your System Administrator if necessary), and check that the host is sending correct data. If this does not solve the problem, contact your Communications Server support personnel.

The following return codes indicate that the verb did not complete successfully for other reasons:

lua_prim_rc

LUA_COMM_SUBSYSTEM_ABENDED
A required Communications Server software component (such as the node) has terminated or has been stopped. Contact your System Administrator if necessary.

lua_prim_rc

LUA_COMM_SUBSYSTEM_NOT_LOADED
The Remote API Client software was not started, or the node was either not started or not configured properly for LUA applications. Check the Communications Server LUA configuration parameters and start the Remote API Client and the node before running your application.

lua_prim_rc

LUA_SESSION_FAILURE
The LUA session has failed. To restart it, the application can reissue SLI_OPEN.

lua_sec_rc
Possible values are:

LUA_LU_COMPONENT_DISCONNECTED
This return code indicates that the LUA session has failed because of a problem with the communications link or with the host LU.

LUA_RECEIVED_UNBIND
This return code indicates that the host sent an UNBIND command to end the session. This value can occur only if the SLI_OPEN verb for this session specified *lua_session_type* `LUA_SESSION_TYPE_DEDICATED`.

lua_prim_rc

LUA_INVALID_VERB
Either the *lua_verb* parameter or the *lua_opcode* parameter was not valid. The verb did not execute.

lua_prim_rc

LUA_STACK_TOO_SMALL
The stack size of the application is too small for LUA to complete the request. Increase the stack size of your application.

lua_prim_rc

LUA_UNEXPECTED_DOS_ERROR
An operating system error occurred.

lua_sec_rc
This value is the operating system return code. Check your operating system documentation for the meaning of this return code.

Interaction with Other Verbs

This verb can only be used when an SLI_RECEIVE has been issued and is pending completion (that is, the primary return code is `IN_PROGRESS`).

SLI_RECEIVE

The SLI_RECEIVE verb receives a complete chain of data, or status information, sent from the host to the application's LU.

You can specify a particular message flow (LU normal, LU expedited, SSCP normal, or SSCP expedited) from which to read data, or you can specify more than one message flow. You can have multiple SLI_RECEIVE verbs outstanding, provided that no two of them specify the same flow.

Supplied Parameters

The application supplies the following parameters:

lua_verb
`LUA_VERB_SLI`

lua_verb_length
The length in bytes of the LUA verb record.

Set this to `sizeof(LUA_VERB_RECORD)`.

lua_opcode
`LUA_OPCODE_SLI_RECEIVE`

lua_correlator
Optional. A four-byte value that you can use to correlate this verb with other processing within your application. LUA does not use or change this information.

lua_luname
The name in ASCII of the LU used by the session. This must match the LU name of an active LUA session, as returned on the SLI_OPEN verb.

This parameter is required only if the *lua_sid* parameter is `0` (zero). If a session ID is supplied in *lua_sid*, LUA does not use this parameter.

This parameter must be eight bytes long; pad on the right with spaces, `0x20`, if the name is shorter than eight characters.

lua_sid
The session ID of the session. This must match a session ID returned on a previous SLI_OPEN verb.

This parameter is optional; if you do not specify the session ID, you must specify the LU name for the session in the *lua_luname* parameter.

lua_max_length
The length of the buffer supplied to receive the data.

lua_data_ptr
A pointer to the buffer supplied to receive the data.

lua_post_handle

AIX, LINUX

A pointer to a callback routine that LUA will call to indicate completion if the verb completes asynchronously.

WINDOWS

If the VCB is used in an `SLI` function call, set this field to an event handle. If the VCB is used in a `WinSLI` function call, this field is reserved.

[redacted]

For more information, see Chapter 2, "Designing and Writing LUA Applications," on page 13.

lua_flag1 **parameters**

Set the *lua_flag1.nowait* parameter to 1 if you want the SLI_RECEIVE verb to return as soon as possible whether or not data is available to be read, or set it to `0` (zero) if you want the verb to wait for data before returning.

Note:

1. Setting the *lua_flag1.nowait* parameter to 1 does not mean that the verb will complete synchronously. The LUA library needs to communicate with the local node to determine whether or not any data is available, and this requires an asynchronous verb return to avoid blocking the application. The parameter means that, if there is no data available immediately, the asynchronous verb return will occur as soon as possible to indicate this.
2. If the first RU of a multiple-RU chain is available when the application issues SLI_RECEIVE, the *lua_flag1.nowait* parameter is ignored; SLI_RECEIVE waits until the complete chain of data has arrived before returning.

Set the *lua_flag1.bid_enable* parameter to 1 to re-enable the most recent SLI_BID verb (equivalent to issuing SLI_BID again with exactly the same parameters as before), or set it to `0` (zero) if you do not want to re-enable SLI_BID. Re-enabling the previous SLI_BID re-uses the VCB originally allocated for it, so this VCB must not have been freed or modified. (For more information, see "Interaction with Other Verbs" on page 132.)

Set one or more of the following flags to 1 to indicate which message flow to read data from:

lua_flag1.sscp_exp

lua_flag1.lu_exp

lua_flag1.sscp_norm

lua_flag1.lu_norm

If more than one flag is set, the highest-priority data available will be returned. The order of priorities (highest first) is: SSCP expedited, LU expedited, SSCP normal, LU normal. The equivalent flag in the *lua_flag2* group will be set to indicate which flow the data was read from (see "Returned Parameters").

The Communications Server implementation of LUA does not return data on the SSCP expedited flow. The application can set the *sscp_exp* flag, for compatibility with other LUA implementations, but data will never be returned on this flow.

Returned Parameters

LUA always returns the following parameters:

lua_flag2.async
: This flag is set to 1 if the verb completed asynchronously, or `0` (zero) if the verb completed synchronously.

lua_flag2.bid_enable
: This parameter is set to 1 if an SLI_BID was successfully re-enabled, or to `0` if it was not re-enabled.

Other returned parameters depend on whether the verb completed successfully; see the following sections.

Successful Execution or Truncated Data

If the verb executes successfully, LUA returns the following parameters:

lua_prim_rc
: `LUA_OK`

The following parameters are returned if the verb completes successfully. They are also returned if the verb returns with truncated data because the *lua_data_length* parameter supplied was too small (see "Other Conditions" on page 131).

lua_sid If the application specified the *lua_luname* parameter when issuing this verb, rather than specifying the session ID, LUA supplies the session ID.

lua_data_length
: The length of the data received. LUA places the data in the buffer specified by *lua_data_ptr*.

  If *lua_rh.rri* is off (request unit) and *lua_rh.sdi* is on (sense data included), this indicates that LUA has converted a request unit sent by the host into an exception request (EXR). In this case, bytes 0–3 of the data buffer contain the sense data associated with the exception, and bytes 4–6 contain up to the first 3 bytes of the original request unit.

lua_th Information from the transmission header (TH) of the received message.

lua_rh Information from the request/response header (RH) of the received message.

lua_message_type
: Message type of the received message, which is one of the following:

  `LUA_MESSAGE_TYPE_LU_DATA`

  `LUA_MESSAGE_TYPE_SSCP_DATA`

  `LUA_MESSAGE_TYPE_RSP`

  `LUA_MESSAGE_TYPE_BID`

  `LUA_MESSAGE_TYPE_BIS`

  `LUA_MESSAGE_TYPE_CANCEL`

  `LUA_MESSAGE_TYPE_CHASE`

  `LUA_MESSAGE_TYPE_LUSTAT_LU`

  `LUA_MESSAGE_TYPE_LUSTAT_SSCP`

  `LUA_MESSAGE_TYPE_QC`

  `LUA_MESSAGE_TYPE_QEC`

  `LUA_MESSAGE_TYPE_RELQ`

  `LUA_MESSAGE_TYPE_RTR`

`LUA_MESSAGE_TYPE_SBI`

`LUA_MESSAGE_TYPE_SIGNAL`

lua_flag2 **parameters**
One of the following flags will be set to 1, to indicate on which message flow the data was received:

lua_flag2.lu_exp

lua_flag2.sscp_norm

lua_flag2.lu_norm

The Communications Server implementation of LUA does not return data on the SSCP expedited flow, and so the *sscp_exp* flag will never be set (although it may be set by other LUA implementations).

Successful Execution: Status Information

Note: SLI_RECEIVE can return status information only if there is no SLI_BID verb outstanding. If both verbs are in progress when status information becomes available, the status is returned on the SLI_BID verb, and the SLI_RECEIVE remains in progress.

If the verb returned LUA status information instead of data, LUA returns the following parameters:

lua_prim_rc
`LUA_STATUS`

lua_sec_rc

`LUA_READY`
The SLI session is now ready to process additional commands. This status is used after a previous `LUA_NOT_READY` status was reported, or after an SLI_CLOSE verb completed with *lua_prim_rc* set to `LUA_CANCELLED` and *lua_sec_rc* set to `RECEIVED_UNBIND_HOLD` or `RECEIVED_UNBIND_NORMAL`.

`LUA_NOT_READY`
The SLI session has been temporarily suspended for one of the following reasons:

- A CLEAR command was received. The session resumes when an SDT command is received.
- An UNBIND command type X'02' (BIND forthcoming) was received. The session is suspended until a BIND, optional CRV and STSN, and SDT commands are received; it resumes after the SDT. Any user extension routines that were supplied by the original SLI_OPEN verb will be called again.
- An UNBIND command type X'01' (normal) was received, and the SLI_OPEN verb for this session specified *lua_session_type* `LUA_SESSION_TYPE_DEDICATED`. The session is suspended until a BIND, optional CRV and STSN, and SDT commands are received; it resumes after the SDT. Any user extension routines that were supplied by the original SLI_OPEN verb will be called again.

The application should issue another SLI_BID or SLI_RECEIVE to receive the `READY` status when the session resumes. It can continue to issue SLI_SEND and SLI_RECEIVE verbs for SSCP normal-flow data even though the session status is `LUA_NOT_READY`.

`LUA_INIT_COMPLETE`
The application issued SLI_OPEN with type `LUA_OPEN_TYPE_PRIM_SSCP`, and the underlying RUI_INIT verb has now completed. The application can now issue SLI_SEND and SLI_RECEIVE verbs for SSCP normal-flow data.

`LUA_SESSION_END_REQUESTED`
The host has sent a SHUTD command, requesting the application to shut down the session. The application should issue SLI_CLOSE as soon as it is ready to close the session.

lua_sid If the application specified the *lua_luname* parameter when issuing this verb, rather than specifying the session ID, LUA supplies the session ID.

Unsuccessful Execution

If a verb does not complete successfully, LUA returns a primary return code to indicate the type of error and a secondary return code to provide specific details about the reason for unsuccessful execution.

Verb Canceled: The following return codes indicate that the verb did not complete successfully because it was canceled by another verb or by a message from the host:

lua_prim_rc
`LUA_CANCELLED`

lua_sec_rc
Possible values are:

`LUA_PURGED`
This SLI_RECEIVE verb has been canceled by an SLI_PURGE verb.

`LUA_TERMINATED`
An SLI_CLOSE verb was issued while this verb was pending.

`LUA_CANCEL_COMMAND_RECEIVED`
The host sent a CANCEL command to cancel the remainder of the chain of data being received.

Parameter Check: The following return codes indicate that the verb did not complete successfully because a supplied parameter was in error:

lua_prim_rc
`LUA_PARAMETER_CHECK`

lua_sec_rc
Possible values are:

`LUA_BAD_DATA_PTR`
The *lua_data_ptr* parameter contained a value that was not valid.

`LUA_BAD_SESSION_ID`
The *lua_sid* parameter did not match the session ID of any active LUA LU session.

`LUA_BID_ALREADY_ENABLED`
The *lua_flag1.bid_enable* parameter was set to re-enable an SLI_BID verb, but the previous SLI_BID verb was still in progress.

LUA_INVALID_FLOW
: None of the *lua_flag1* flow flags was set. At least one of these flags must be set to 1 to indicate which flow or flows to read from.

AIX, LINUX

LUA_INVALID_POST_HANDLE
: The *lua_post_handle* parameter was not a valid pointer to a callback routine.

LUA_NO_PREVIOUS_BID_ENABLED
: The *lua_flag1.bid_enable* parameter was set to re-enable an SLI_BID verb, but there was no previous SLI_BID verb that could be enabled. (For more information, see "Interaction with Other Verbs" on page 132.)

LUA_RESERVED_FIELD_NOT_ZERO
: A reserved field in the verb record, or a parameter that is not used by this verb, was set to a nonzero value.

LUA_VERB_LENGTH_INVALID
: The value of the *lua_verb_length* parameter was less than the length of the verb record required for this verb.

State Check: The following return codes indicate that the verb was issued in a session state in which it was not valid:

lua_prim_rc
: `LUA_STATE_CHECK`

lua_sec_rc
: Possible values are:

LUA_NO_SLI_SESSION
: An SLI_OPEN verb has not yet completed successfully for the LU name specified on this verb, or the session has failed.

LUA_RECEIVE_ON_FLOW_PENDING
: The flow flags in the *lua_flag1* group specified one or more session flows for which an SLI_RECEIVE verb was already outstanding. Only one SLI_RECEIVE at a time can be waiting on each session flow.

Negative Response Sent to Host: The following primary return code indicates that Communications Server detected an error in the data received from the host. Instead of passing the received message to the application on an SLI_RECEIVE verb, Communications Server discards the message and sends a negative response to the host. LUA informs the application on a subsequent SLI_RECEIVE or SLI_BID verb that a negative response was sent.

lua_prim_rc
: `LUA_NEGATIVE_RSP`

lua_sec_rc
: The sense code sent to the host on the negative response. This indicates that Communications Server detected an error in the host data, and sent a

negative response to the host. For information about interpreting the sense code values that can be returned, see "SNA Information" on page 34.

Other Conditions: The following return codes indicate that the verb record supplied was valid, but the verb did not complete successfully:

lua_prim_rc
: `LUA_UNSUCCESSFUL`

lua_sec_rc
: Possible values are:

`LUA_DATA_TRUNCATED`
: The *lua_data_length* parameter was smaller than the actual length of data received on the message. Only *lua_data_length* bytes of data were returned to the verb; the remaining data was discarded. Additional parameters are also returned if this secondary return code is obtained; see "Successful Execution or Truncated Data" on page 127.

`LUA_NO_DATA`
: The *lua_flag1.nowait* parameter was set to indicate immediate return without waiting for data, and no data was currently available on the specified session flow or flows.

`LUA_INVALID_PROCESS`
: The operating system process that issued this verb was not the same process that issued the SLI_OPEN verb for this session. Only the process that started a session can issue verbs on that session.

`LUA_NAU_INOPERATIVE`
: A required SNA component (such as the LUA LU) is not active or is in an abnormal state.

`LUA_NO_SESSION`
: The SNA session to the remote LU is not active.

`LUA_SLI_LOGIC_ERROR`
: This return code indicates one of the following:

- The host system has violated SNA protocols
- An internal error was detected within LUA

Attempt to reproduce the problem with SNA tracing active (contact your System Administrator if necessary), and check that the host is sending correct data. If this does not solve the problem, contact your Communications Server support personnel.

The following return codes indicate that the verb did not complete successfully for other reasons:

lua_prim_rc

`LUA_COMM_SUBSYSTEM_ABENDED`
: A required Communications Server software component (such as the node) has terminated or has been stopped. Contact your System Administrator if necessary.

lua_prim_rc

`LUA_COMM_SUBSYSTEM_NOT_LOADED`
: The Remote API Client software was not started, or the node was either not started or not configured properly for LUA applications.

Check the Communications Server LUA configuration parameters and start the Remote API Client and the node before running your application.

lua_prim_rc

LUA_SESSION_FAILURE
The LUA session has failed. To restart it, the application can reissue SLI_OPEN.

lua_sec_rc
Possible values are:

LUA_LU_COMPONENT_DISCONNECTED
This return code indicates that the LUA session has failed because of a problem with the communications link or with the host LU.

LUA_RECEIVED_UNBIND
This return code indicates that the host sent an UNBIND command to end the session. This value can occur only if the SLI_OPEN verb for this session specified *lua_session_type* `LUA_SESSION_TYPE_DEDICATED`.

LUA_RUI_WRITE_FAILURE
An RUI_WRITE verb used in processing this SLI verb has failed with an unexpected error return code.

lua_prim_rc

LUA_INVALID_VERB
Either the *lua_verb* parameter or the *lua_opcode* parameter was not valid. The verb did not execute.

lua_prim_rc

LUA_STACK_TOO_SMALL
The stack size of the application is too small for LUA to complete the request. Increase the stack size of your application.

lua_prim_rc

LUA_UNEXPECTED_DOS_ERROR
An operating system error occurred.

lua_sec_rc
This value is the operating system return code. Check your operating system documentation for the meaning of this return code.

Interaction with Other Verbs

The SLI_OPEN verb must have completed successfully before this verb can be issued.

While an existing SLI_RECEIVE is pending, you can issue another SLI_RECEIVE only if it specifies a different session flow or flows from pending SLI_RECEIVEs; you cannot have more than one SLI_RECEIVE outstanding for the same session flow.

The *lua_flag1.bid_enable* parameter can only be used if the following are true:
- SLI_BID has already been issued successfully and has completed
- The storage allocated for the SLI_BID verb has not been freed or modified
- No other SLI_BID is pending

If you use this parameter to re-enable a previous SLI_BID, at least one of the message flow flags on SLI_RECEIVE must still be set, to indicate the flow or flows on which the application will accept data. If the first data to be received is on a flow accepted by the SLI_RECEIVE verb, SLI_RECEIVE will return with this data, and SLI_BID will not return. Otherwise, SLI_BID will return to indicate that there is data to be read (since SLI_BID accepts data on all flows, it will always accept the data if SLI_RECEIVE does not). The application must then issue another SLI_RECEIVE on the appropriate flow to obtain the data.

If you want to use SLI_BID to handle data on all flows, rather than having the data on a particular flow handled by SLI_RECEIVE in preference to SLI_BID, you need to re-issue SLI_BID explicitly instead of using SLI_RECEIVE to re-enable the previous SLI_BID.

Usage and Restrictions

If the data received is longer than the *lua_max_length* parameter, it will be truncated; only *lua_max_length* bytes of data will be returned. The primary and secondary return codes `LUA_UNSUCCESSFUL` and `LUA_DATA_TRUNCATED` will also be returned.

If the SLI_RECEIVE verb sets bits in *lua_flag1* to accept data on more than one flow, and there is data available on more than one of the specified flows, the data on the highest-priority flow will be returned to the application. The flow priorities are as follows (highest to lowest):
- SSCP expedited
- LU expedited
- SSCP normal
- LU normal

Once a message has been read using the SLI_RECEIVE verb, it is removed from the incoming message queue, and cannot be accessed again. The application can use SLI_BID as a non-destructive read to check the type of data available and determine how to process it, and then issue a subsequent SLI_RECEIVE to collect the data. However, if it issues the SLI_RECEIVE with multiple *lua_flag1* flags set to accept data on more than one flow, it may receive a different message from the one identified in the SLI_BID, if data arrived on a higher-priority flow between the SLI_BID and SLI_RECEIVE verbs. To ensure that it receives the same message that was identified in the SLI_BID, it should set the *lua_flag1* flags on SLI_RECEIVE to accept data only on the flow identified in the SLI_BID response.

Pacing may be used on the primary-to-secondary half-session (this is specified in the host configuration), in order to protect the LUA application from being flooded with messages. If the LUA application is slow to read messages, Communications Server delays the sending of pacing responses to the host in order to slow it down.

SLI_SEND

The SLI_SEND verb sends an SNA request or response unit from the LUA application to the host, over either the LU session or the SSCP session.

An application can have at most two SLI_SEND verbs outstanding at a time, which must be on different session flows.

Supplied Parameters

The application supplies the following parameters:

lua_verb
: `LUA_VERB_SLI`

lua_verb_length
: The length in bytes of the LUA verb record.

  Set this to `sizeof(LUA_VERB_RECORD)`.

lua_opcode
: `LUA_OPCODE_SLI_SEND`

lua_correlator
: Optional. A four-byte value that you can use to correlate this verb with other processing within your application. LUA does not use or change this information.

lua_luname
: The name in ASCII of the LU used by the session. This must match the LU name of an active LUA session, as returned on the SLI_OPEN verb.

  This parameter is required only if the *lua_sid* parameter is `0` (zero). If a session ID is supplied in *lua_sid*, LUA does not use this parameter.

  This parameter must be eight bytes long; pad on the right with spaces, `0x20`, if the name is shorter than eight characters.

lua_sid
: The session ID of the session. This must match a session ID returned on a previous SLI_OPEN verb.

  This parameter is optional; if you do not specify the session ID, you must specify the LU name for the session in the *lua_luname* parameter.

lua_data_length
: The length of the supplied data.

  When sending a positive response, this parameter is normally set to `0` (zero). LUA will complete the response based on the supplied sequence number. In the case of a positive response to a BIND or STSN, an extended response is allowed, so a nonzero value may be used.

  When sending a negative response, set this parameter to the length of the SNA sense code (four bytes), which is supplied in the data buffer.

lua_data_ptr
: A pointer to the buffer containing the supplied data.

  For a request, or a positive response that requires data, the buffer should contain the entire RU. The length of the RU must be specified in *lua_data_length*.

  For a negative response, the buffer should contain the SNA sense code.

lua_post_handle

AIX, LINUX

A pointer to a callback routine that LUA will call to indicate completion if the verb completes asynchronously.

WINDOWS

If the VCB is used in an `SLI` function call, set this field to an event handle. If the VCB is used in a `WinSLI` function call, this field is reserved.

[redacted]

For more information, see Chapter 2, “Designing and Writing LUA Applications,” on page 13.

lua_th.snf
Required only when sending a response. The sequence number of the request to which this is the response.

lua_rh When sending a request, most of the *lua_rh* bits must be set to correspond to the RH (request header) of the message to be sent. Do not set *lua_rh.pi* and *lua_rh.qri*; these will be set by LUA.

When sending a response, only the following two *lua_rh* bits are used. The others must be `0` (zero). The *lua_rh* bits are:

lua_rh.rri
Set to `1` to indicate a response

lua_rh.ri
Set to `0` for a positive response, or `1` for a negative response

lua_flag1 **parameters**
Set one of the following flags to `1` to indicate which message flow the data is to be sent on:

lua_flag1.lu_exp

lua_flag1.sscp_norm

lua_flag1.lu_norm

One and only one of the flags must be set to 1.Communications Server does not allow applications to send data on the SSCP expedited flow (the *lua_flag1.sscp_exp* flag).

lua_message_type
Message type of the message to be sent. Possible values are:

`LUA_MESSAGE_TYPE_LU_DATA`

`LUA_MESSAGE_TYPE_SSCP_DATA`

`LUA_MESSAGE_TYPE_RSP`

`LUA_MESSAGE_TYPE_BID`

`LUA_MESSAGE_TYPE_BIS`

`LUA_MESSAGE_TYPE_CANCEL`

`LUA_MESSAGE_TYPE_CHASE`

`LUA_MESSAGE_TYPE_LUSTAT_LU`

`LUA_MESSAGE_TYPE_LUSTAT_SSCP`

`LUA_MESSAGE_TYPE_QC`

`LUA_MESSAGE_TYPE_QEC`

`LUA_MESSAGE_TYPE_RELQ`

`LUA_MESSAGE_TYPE_RTR`

LUA_MESSAGE_TYPE_SBI

Returned Parameters

LUA always returns the following parameter:

lua_flag2.async
: This flag is set to 1 if the verb completed asynchronously, or 0 (zero) if the verb completed synchronously.

Other returned parameters depend on whether the verb completed successfully; see the following sections.

Successful Execution

If the verb executes successfully, LUA returns the following parameters:

lua_prim_rc
: LUA_OK

lua_sid
: If the application specified the *lua_luname* parameter when issuing this verb, rather than specifying the session ID, LUA supplies the session ID.

lua_th
: The completed TH of the message written, including the fields filled in by LUA. You may need to save the value of *lua_th.snf* (the sequence number) for correlation with responses from the host.

lua_flag2 **parameters**
: One of the following flags will be set to 1 to indicate which message flow the data was sent on:

lua_flag2.lu_exp

lua_flag2.sscp_norm

lua_flag2.lu_norm

The Communications Server implementation of LUA does not allow applications to send data on the SSCP expedited flow, and so will never set the *sscp_exp* flag (although other LUA implementations may set it).

lua_sequence_number
: The sequence number of the RU that LUA uses to send the data (or of the first RU, if the data requires a chain of RUs). This is stored in line format.

Successful Execution: Status Information

If the verb returned LUA status information, LUA returns the following parameters:

lua_prim_rc
: LUA_STATUS

lua_sec_rc

LUA_READY
: The SLI session is now ready to process additional commands. This status is used after a previous LUA_NOT_READY status was reported, or after an SLI_CLOSE verb completed with *lua_prim_rc* set to LUA_CANCELLED and *lua_sec_rc* set to RECEIVED_UNBIND_HOLD or RECEIVED_UNBIND_NORMAL.

LUA_NOT_READY
: The SLI session has been temporarily suspended for one of the following reasons:

- A CLEAR command was received. The session resumes when an SDT command is received.
- An UNBIND command type X'02' (BIND forthcoming) was received. The session is suspended until a BIND, optional CRV and STSN, and SDT commands are received; it resumes after the SDT. Any user extension routines that were supplied by the original SLI_OPEN verb will be called again.
- An UNBIND command type X'01' (normal) was received, and the SLI_OPEN verb for this session specified *lua_session_type* `LUA_SESSION_TYPE_DEDICATED`. The session is suspended until a BIND, optional CRV and STSN, and SDT commands are received; it resumes after the SDT. Any user extension routines that were supplied by the original SLI_OPEN verb will be called again.

The application should issue SLI_BID or SLI_RECEIVE to receive the `READY` status when the session resumes. It can continue to issue SLI_SEND and SLI_RECEIVE verbs for SSCP normal-flow data even though the session status is `LUA_NOT_READY`.

`LUA_INIT_COMPLETE`
The application issued SLI_OPEN with type `LUA_OPEN_TYPE_PRIM_SSCP`, and the underlying RUI_INIT verb has now completed. The application can now issue SLI_SEND and SLI_RECEIVE verbs for SSCP normal-flow data.

`LUA_SESSION_END_REQUESTED`
The host has sent a SHUTD command, requesting the application to shut down the session. The application should issue SLI_CLOSE as soon as it is ready to close the session.

Unsuccessful Execution

If a verb does not complete successfully, LUA returns a primary return code to indicate the type of error and a secondary return code to provide specific details about the reason for unsuccessful execution.

Verb Canceled: The following return codes indicate that the verb did not complete successfully because it was canceled by another verb:

lua_prim_rc
`LUA_CANCELLED`

lua_sec_rc

`LUA_TERMINATED`
The verb was canceled because an SLI_CLOSE verb was issued for this session.

Parameter Check: The following return codes indicate that the verb did not complete successfully because a supplied parameter was in error:

lua_prim_rc
`LUA_PARAMETER_CHECK`

lua_sec_rc
Possible values are:

`LUA_BAD_DATA_PTR`
The *lua_data_ptr* parameter contained a value that was not valid.

LUA_BAD_SESSION_ID
: The *lua_sid* parameter did not match the session ID of any active LUA LU session.

LUA_INVALID_FLOW
: More than one of the *lua_flag1* flow flags was set to 1. One and only one of these flags must be set to 1, to indicate which session flow the data is to be sent on.

: The *lua_flag1.sscp_exp* flow flag was set, indicating that the message should be sent on the SSCP expedited flow. Communications Server does not allow applications to send data on this flow.

LUA_INVALID_MESSAGE_TYPE
: The *lua_message_type* parameter was not set to a valid value.

AIX, LINUX

LUA_INVALID_POST_HANDLE
: The *lua_post_handle* parameter was not a valid pointer to a callback routine.

LUA_REQUIRED_FIELD_MISSING
: None of the *lua_flag1* flow flags was set. One and only one of these flags must be set to 1.

LUA_RESERVED_FIELD_NOT_ZERO
: A reserved field in the verb record, or a parameter that is not used by this verb, was set to a nonzero value.

LUA_VERB_LENGTH_INVALID
: The value of the *lua_verb_length* parameter was less than the length of the verb record required for this verb.

LUA_DATA_LENGTH_ERROR
: The application used SLI_SEND to send LUSTAT to the host, but did not provide the required 4 bytes of status information.

State Check: The following return codes indicate that the verb was issued in a session state in which it was not valid:

lua_prim_rc
: LUA_STATE_CHECK

lua_sec_rc
: Possible values are:

LUA_MAX_NUMBER_OF_SENDS
: The application already had two SLI_SEND verbs in progress when it issued this verb. An application can have at most two SLI_SEND verbs outstanding at a time, which must be on different session flows.

LUA_NO_SLI_SESSION
: An SLI_OPEN verb has not yet completed successfully for the LU name specified on this verb, or the session has failed.

LUA_SEND_ON_FLOW_PENDING
: An SLI_SEND was already outstanding for the session flow

specified on this verb (the session flow is specified by setting one of the *lua_flag1* flow flags to 1). Only one SLI_SEND at a time can be outstanding on each session flow.

Other Conditions: The following return codes indicate that the verb record supplied was valid, but the verb did not complete successfully:

lua_prim_rc
: `LUA_UNSUCCESSFUL`

lua_sec_rc
: Possible values are:

`LUA_INVALID_PROCESS`
: The operating system process that issued this verb was not the same process that issued the SLI_OPEN verb for this session. Only the process that started a session can issue verbs on that session.

`LUA_INVALID_SESSION_PARAMETERS`
: The application used SLI_SEND to send a positive response to a BIND message received from the host. However, the Communications Server node cannot accept the BIND parameters as specified, and has sent a negative response to the host. For more information about the BIND profiles accepted by Communications Server, see "SNA Information" on page 34.

`LUA_RSP_CORRELATION_ERROR`
: When using SLI_SEND to send a response, the *lua_th.snf* parameter (which indicates the sequence number of the received message being responded to) did not contain a valid value.

`LUA_RU_LENGTH_ERROR`
: The *lua_data_length* parameter contained a value that was not valid. When sending data on the LU normal flow, the maximum length is as specified in the BIND received from the host; for all other flows the maximum length is 256 bytes.

`LUA_NAU_INOPERATIVE`
: A required SNA component (such as the LUA LU) is not active or is in an abnormal state.

`LUA_NO_SESSION`
: The SNA session to the remote LU is not active.

`LUA_SLI_LOGIC_ERROR`
: This return code indicates one of the following:

- The host system has violated SNA protocols
- An internal error was detected within LUA

Attempt to reproduce the problem with SNA tracing active (contact your System Administrator if necessary), and check that the host is sending correct data. If this does not solve the problem, contact your Communications Server support personnel.

(any other value)
: Any other secondary return code here is an SNA sense code indicating that the supplied SNA data was not valid or could not be sent. For information about interpreting the SNA sense codes that can be returned, see "SNA Information" on page 34.

The following return codes indicate that the verb did not complete successfully for other reasons:

lua_prim_rc

`LUA_COMM_SUBSYSTEM_ABENDED`
A required Communications Server software component (such as the node) has terminated or has been stopped. Contact your System Administrator if necessary.

lua_prim_rc

`LUA_COMM_SUBSYSTEM_NOT_LOADED`
The Remote API Client software was not started, or the node was either not started or not configured properly for LUA applications. Check the Communications Server LUA configuration parameters and start the Remote API Client and the node before running your application.

lua_prim_rc

`LUA_SESSION_FAILURE`
The LUA session has failed. To restart it, the application can reissue SLI_OPEN.

lua_sec_rc
Possible values are:

`LUA_LU_COMPONENT_DISCONNECTED`
This return code indicates that the LUA session has failed because of a problem with the communications link or with the host LU.

`LUA_RECEIVED_UNBIND`
This return code indicates that the host sent an UNBIND command to end the session. This value can occur only if the SLI_OPEN verb for this session specified *lua_session_type* `LUA_SESSION_TYPE_DEDICATED`.

`LUA_SLI_LOGIC_ERROR`
This return code indicates one of the following:

- The host system has violated SNA protocols
- An internal error was detected within LUA

Attempt to reproduce the problem with SNA tracing active (contact your System Administrator if necessary), and check that the host is sending correct data. If this does not solve the problem, contact your Communications Server support personnel.

lua_prim_rc

`LUA_INVALID_VERB`
Either the *lua_verb* parameter or the *lua_opcode* parameter was not valid. The verb did not execute.

lua_prim_rc

`LUA_STACK_TOO_SMALL`
The stack size of the application is too small for LUA to complete the request. Increase the stack size of your application.

lua_prim_rc

`LUA_UNEXPECTED_DOS_ERROR`
An operating system error occurred.

lua_sec_rc
This value is the operating system return code. Check your operating system documentation for the meaning of this return code.

Interaction with Other Verbs

The SLI_OPEN verb must be issued successfully before this verb can be issued.

While an existing SLI_SEND is pending, you can issue a second SLI_SEND only if it specifies a different session flow from the pending SLI_SEND; that is, you cannot have more than one SLI_SEND outstanding for the same session flow. You cannot have more than two SLI_SENDs outstanding in total.

The SLI_SEND verb can be issued on the SSCP normal flow at any time after a successful SLI_OPEN verb that specifies primary-initiated session initiation with SSCP access. SLI_SEND verbs on other flows or for other session initiation types are permitted only after a BIND has been received, and must abide by the protocols specified on the BIND.

Usage and Restrictions

Table 2 shows the valid settings for various parameters on SLI_SEND, depending on the type of SNA message being sent.

Table 2. SLI_SEND Parameter Settings based on Message Type

SLI_SEND parameter	LU_DATA, SSCP_DATA	RSP	BID, BIS, RTR	CHASE QC	QEC, RELQ, SBL, SIG	RQR	LUSTAT_LU, LUSTAT_SSCP
lua_rh	FI, DR1I, DR2I, RI, BBI, EBI, CDI, CSI, EDI	RI	SDI, QRI	SDI, QRI, EBI, CDI	SDI	0	SDI, QRI, DR1I, DR2I, RI, BBI, EBI, CDI
lua_th	0	SNF	0	0	0	0	0
lua_data_ptr	Required (null if no data)	Required (null if no data)	null	null	null	null	Required
lua_data_length	Required	Required (0 if no data)	0	0	0	0	Required
lua_flag1 flow flags	0	Required (set one)	0	0	0	0	0

When the application sends an SNA response, it must do the following. LUA will fill in the appropriate request code based on the supplied sequence number.

- Set *lua_message_type* to `LUA_MESSAGE_TYPE_RSP`.
- Set *lua_th.snf* to the sequence number of the request to which this is a response.
- Set the appropriate *lua_flag1* flow flag.
- For a positive response that requires only the request code, set both *lua_rh.ri* and *lua_data_length* to `0` (zero).
- For a negative response:
  – Set *lua_rh.ri* to `1`.
  – Set *lua_data_ptr* to point to an appropriate SNA sense code.
  – Set *lua_data_length* to `4` (the length of the sense code).

Successful completion of SLI_SEND indicates that the message was queued successfully to the data link; it does not necessarily indicate that the message was sent successfully, or that the host accepted it.

Pacing may be used on the secondary-to-primary half-session (this is specified on the BIND), in order to prevent the LUA application from sending more data than the Communications Server LU or the host LU can handle. If this is the case, an SLI_SEND on the LU normal flow may be delayed by LUA and may take some time to complete.

SLI_BIND_ROUTINE

This verb is sent from LUA to the application (using the BIND extension routine entry point supplied by the application on the SLI_OPEN verb), and not from the application to LUA.

The SLI_BIND_ROUTINE verb passes a BIND request from the host to the LUA application. The application can accept the BIND as it is, modify it in an attempt to negotiate the BIND parameters, or reject it with an appropriate SNA sense code.

Supplied Parameters

LUA supplies the the following parameters to the application:

lua_verb
: `LUA_VERB_SLI`

lua_verb_length
: The length in bytes of the LUA verb record.

lua_opcode
: `LUA_OPCODE_SLI_BIND_ROUTINE`

lua_luname
: The name in ASCII of the LU used by the session.

lua_sid
: The session ID of the session.

lua_data_length
: The length of the supplied BIND RU.

lua_data_ptr
: A pointer to the buffer containing the supplied BIND RU.

lua_th
: The TH parameters from the BIND.

lua_rh
: The RH parameters from the BIND.

Returned Parameters

The parameters returned by the application depend on whether the verb completed successfully; see the following sections.

Successful Execution: BIND Accepted or Negotiated

If the application decides to accept or negotiate the BIND, it returns the following parameters:

lua_prim_rc
: `LUA_OK`

lua_data_ptr
: A pointer to the buffer containing the supplied BIND RU. If the application is accepting the BIND as is, it must not modify the contents of the buffer; if it is attempting to negotiate one or more parameters in the BIND, it must modify the data to set the appropriate parameters to its preferred values.

Unsuccessful Execution: BIND Rejected

If the application decides to reject the BIND, it returns the following parameters:

lua_prim_rc
: `LUA_NEGATIVE_RSP`

lua_data_length
: The length of the returned SNA sense code (in the *lua_data_ptr* parameter).

lua_data_ptr
: A pointer to the buffer containing the SNA sense code associated with the application's reason for rejecting the BIND.

Interaction with Other Verbs

LUA will call this routine from within its processing of the SLI_OPEN verb (after the application issues SLI_OPEN and before its asynchronous return).

Usage and Restrictions

There is no asynchronous return mechanism for the application's extension routines. The routine must return synchronously.

SLI_SDT_ROUTINE

This verb is sent from LUA to the application (using the SDT extension routine entry point supplied by the application on the SLI_OPEN verb), and not from the application to LUA.

The SLI_SDT_ROUTINE verb passes an SDT request from the host to the LUA application. The application can respond with an SDT response, or reject it with an appropriate SNA sense code.

Supplied Parameters

LUA supplies the the following parameters to the application:

lua_verb
: `LUA_VERB_SLI`

lua_verb_length
: The length in bytes of the LUA verb record.

lua_opcode
: `LUA_OPCODE_SLI_SDT_ROUTINE`

lua_luname
: The name in ASCII of the LU used by the session.

lua_sid The session ID of the session.

lua_data_length
: The length of the supplied SDT RU.

lua_data_ptr
: A pointer to the buffer containing the supplied SDT RU.

lua_th The TH parameters from the SDT.

lua_rh The RH parameters from the SDT.

Returned Parameters

The parameters returned by the application depend on whether the verb completed successfully; see the following sections.

Successful Execution: SDT Response

If the application decides to accept the SDT, it returns the following parameters:

lua_prim_rc
    `LUA_OK`

lua_data_ptr
    A pointer to the buffer containing the supplied SDT response RU.

Unsuccessful Execution: SDT Rejected

If the application decides to reject the SDT, it returns the following parameters:

lua_prim_rc
    `LUA_NEGATIVE_RSP`

lua_data_length
    The length of the returned SNA sense code (in the *lua_data_ptr* parameter).

lua_data_ptr
    A pointer to the buffer containing the SNA sense code associated with the application's reason for rejecting the SDT.

Interaction with Other Verbs

LUA will call this routine from within its processing of the SLI_OPEN verb (after the application issues SLI_OPEN and before its asynchronous return).

Usage and Restrictions

There is no asynchronous return mechanism for the application's extension routines. The routine must return synchronously.

SLI_STSN_ROUTINE

This verb is sent from LUA to the application (using the STSN extension routine entry point supplied by the application on the SLI_OPEN verb), and not from the application to LUA.

The SLI_STSN_ROUTINE verb passes an STSN request from the host to the LUA application. The application can respond with an STSN response, or reject it with an appropriate SNA sense code.

Supplied Parameters

LUA supplies the the following parameters to the application:

lua_verb
    `LUA_VERB_SLI`

lua_verb_length
    The length in bytes of the LUA verb record.

lua_opcode
    `LUA_OPCODE_SLI_STSN_ROUTINE`

lua_luname
    The name in ASCII of the LU used by the session.

lua_sid The session ID of the session.

lua_data_length
The length of the supplied STSN RU.

lua_data_ptr
A pointer to the buffer containing the supplied STSN RU.

lua_th The TH parameters from the STSN.

lua_rh The RH parameters from the STSN.

Returned Parameters

The parameters returned by the application depend on whether the verb completed successfully; see the following sections.

Successful Execution: STSN Response

If the application decides to accept the STSN, it returns the following parameters:

lua_prim_rc
`LUA_OK`

lua_data_ptr
A pointer to the buffer containing the supplied STSN response RU.

Unsuccessful Execution: STSN Rejected

If the application decides to reject the STSN, it returns the following parameters:

lua_prim_rc
`LUA_NEGATIVE_RSP`

lua_data_length
The length of the returned SNA sense code (in the *lua_data_ptr* parameter).

lua_data_ptr
A pointer to the buffer containing the SNA sense code associated with the application's reason for rejecting the STSN.

Interaction with Other Verbs

LUA will call this routine from within its processing of the SLI_OPEN verb (after the application issues SLI_OPEN and before its asynchronous return).

Usage and Restrictions

There is no asynchronous return mechanism for the application's extension routines. The routine must return synchronously.

Chapter 6. Sample LUA Application

This chapter describes the Communications Server sample LUA program **lsample.c**, written for the AIX / Linux operating system, which illustrates the use of LUA RUI verbs. This file is stored in the directory **/usr/lib/sna/samples** (AIX) or **/opt/ibm/sna/samples** (Linux).

The following information is provided:

- Processing overview of the application
- Instructions for compiling, linking, and running the application

Processing Overview

The application is a very simple 3270 emulation program. It provides an unformatted display of screen data sent from the host (on both the LU and SSCP sessions), together with status messages (indicating whether the application is connected to the LU session or the SSCP session). When a definite-response request is received from the host, a positive response is sent automatically. Data typed in by the user is sent to the host, with the exception of two special keystrokes:

[(left square bracket)
Toggles between the LU and SSCP sessions

] (right square bracket)
Terminates the application

Once it has completed some initialization processing, the program essentially consists of two main loops: one reading data from the host and one sending data supplied by the user to the host. These are implemented as follows:

The **read loop** uses recursive calls to the RUI_READ verb. The following processing is performed by the callback routine, which LUA calls when the verb completes asynchronously:

- Any screen data is written to the screen
- Any session status information is processed
- If a response is required, a positive response is built and sent
- The RUI_READ verb is then re-issued to continue the loop

If the verb completes synchronously, the same routine used as a callback routine is called explicitly on return. This ensures that the same processing is done whatever the type of return.

The **write loop** reads data from the keyboard. If either of the two special keystrokes is supplied, it is acted on; otherwise, the incoming data is translated to extended binary coded decimal interchange code (EBCDIC) (using the CSV CONVERT verb) to be sent to the host on either the LU session or the SSCP session, depending on which the application is currently connected to. The data is sent using the RUI_WRITE verb; again, the callback routine is used whether or not the verb returns asynchronously. The program waits for a semaphore to be cleared by the callback routine before continuing with the loop.

When the user types the] keystroke to terminate the application, the program breaks out of the write loop and issues the RUI_TERM verb to end the session. The session is also terminated if the read loop encounters a non-LUA_OK return code from the RUI_READ verb.

The program flow can be represented by the diagram shown in Figure 5:

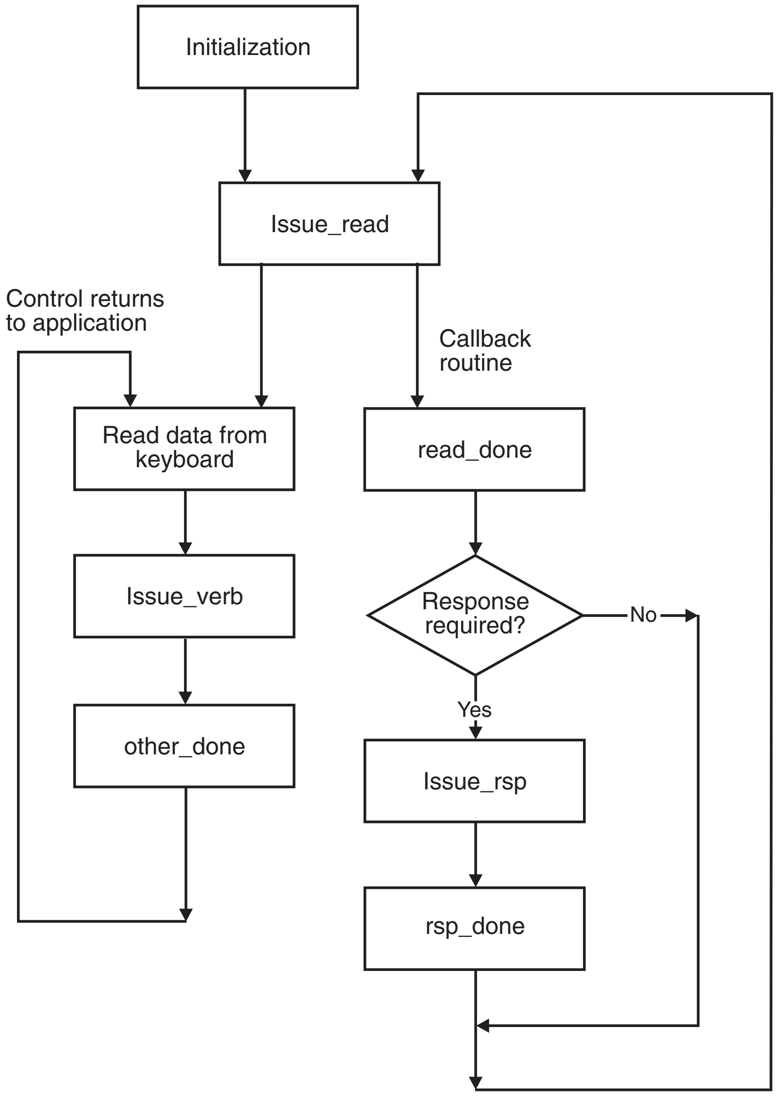

Figure 5. Program Flow for the Sample LUA Application

Testing the Application

After examining the source code for the sample application, you may want to test it. The following steps are required:

1. Ensure that you have access to a suitable host computer against which you can run the application
2. Compile and link the application
3. Configure Communications Server for use with LUA (this task will normally be performed by your System Administrator)
4. Run the application

These steps are explained in more detail in the following sections.

Host Requirements

To run the sample application, you will need an LU on the host computer. Because the sample application is emulating a 3270 display terminal, the LU must be configured at the host as a 3270 display LU (LU type 2) such as 3278 or 3279. The LU number assigned at the host must be used when configuring the LU on Communications Server.

Configuration for the Sample Application

Communications Server must be configured to include the required LU. This task is usually performed by the System Administrator. The following components are required:

- A DLC, port, and LS
- An LU of type 0–3, with an LU number which matches that of a suitable LU on the host

These components can be given any names you wish; the only information required by the application is the LU to be used for the session. This is passed to the application as a single command-line parameter (the LU name), or as two command-line parameters (the PU name and LU number). The following items also apply to LU configuration:

- The LU number configured for this LU in the Communications Server configuration must match the LU number assigned at the host.
- You can configure an LU pool for use with the application, containing one or more LUs. To access the pool, you can then supply either the name of the pool or the name of any LU within it; the least recently used LU from the pool will be used.

Compiling and Linking the Sample Application

To compile and link the program for an AIX or Linux system, take the following steps.

1. Copy the file **lsample.c** from **/usr/lib/sna/samples** (AIX) or **/opt/ibm/sna/samples** (Linux) to a private directory.
2. To compile and link the program for AIX, use the following command:

   ```
   cc -o lsample -I /usr/include/sna -bimport:/usr/lib/sna/lua.exp
   -bimport:/usr/lib/sna/csv.exp lsample.c
   ```

   To compile and link the program for Linux, use the following command:

   ```
   gcc -o lsample -I /opt/ibm/sna/include -L /opt/ibm/sna/lib -llua
   -lsna -lcsv -lpLiS -lpthread lsample.c
   ```

Running the Sample Application

This section assumes you have compiled and linked the sample application as described in “Compiling and Linking the Sample Application.”

The sample application uses the CSV interface as well as LUA; it includes calls to the CSV CONVERT verb to translate user-supplied data from ASCII to EBCDIC before sending it to the host, and to translate data received from the host into ASCII before displaying it on the screen. This translation uses a user-defined translation table (Table G), which is stored in a file on the Communications Server computer. A suitable file, **luatblg.dat**, is supplied with the LUA sample application program source, in the directory **/usr/lib/sna/samples** (AIX) or **/opt/ibm/sna/samples** (Linux).

To run the sample application, follow these steps:

1. Ensure that the Communications Server software is started, and that the LS to the host is active; contact your System Administrator if necessary.
2. Set the environment variable `SNATBLG` to the name of the file containing the Table G translation table. Include the full path of the file if it is not in the current directory.
3. Start the application by entering one of the following commands:

   lsample *luname*

   lsample *puname lunumber*

   In this example, *luname* is the name of the LU you configured for this application (or the name of the LU pool or any LU within it).

   In this example, *puname* is the name of the PU that owns the required LU, and *lunumber* is the LU number (specified as a decimal number).

   The application will display the message **LU active** when it has successfully established a session to the host.
4. Enter data as you would normally do to log on and access host applications.
5. To switch between the LU session and the SSCP session, press the [(left square bracket) key followed by **Enter**.

   The application will display the message **LU session** or **SSCP session** to indicate the session you are currently connected to. It also switches automatically when a BIND or UNBIND message is received.
6. When you have finished with host applications, follow any steps you would normally take to end the applications and log off.
7. To terminate the application, press the] (right square bracket) key followed by **Enter**.

   The application will display the message **Closedown** followed by **Terminated** to indicate that it has ended the session with the host. There may also be a “Read failed” message, with return codes that indicate that an outstanding RUI_READ verb was canceled by the RUI_TERM verb.

Appendix A. Return Code Values

This appendix lists all the possible return codes in the LUA interface in numerical order. The values are defined in the LUA header file **lua_c.h** (for AIX / Linux) or **winlua.h** (for Windows).

You can use this appendix as a reference to check the meaning of a return code received by your application.

Primary Return Codes

The following primary return codes are used in LUA applications.

LUA_OK	0x0100
LUA_STATE_CHECK	0x0200
LUA_COMM_SUBSYSTEM_ABENDED	0x03F0
LUA_COMM_SUBSYSTEM_NOT_LOADED	0x04F0
LUA_INVALID_VERB_SEGMENT	0x08F0
LUA_SESSION_FAILURE	0x0F00
LUA_UNEXPECTED_DOS_ERROR	0x11F0
LUA_UNSUCCESSFUL	0x1400
LUA_STACK_TOO_SMALL	0x15F0
LUA_NEGATIVE_RSP	0x1800
LUA_CANCELLED	0x2100
LUA_IN_PROGRESS	0x3000
LUA_STATUS	0x4000
LUA_INVALID_VERB	0xFFFF

Secondary Return Codes

The following secondary return codes are used in LUA applications.

LUA_SEC_RC_OK	0x00000000
LUA_INVALID_LUNAME	0x01000000
LUA_BAD_SESSION_ID	0x02000000
LUA_DATA_TRUNCATED	0x03000000
LUA_BAD_DATA_PTR	0x04000000
LUA_DATA_SEG_LENGTH_ERROR	0x05000000
LUA_RESERVED_FIELD_NOT_ZERO	0x06000000
LUA_INVALID_POST_HANDLE	0x07000000
LUA_PURGED	0x0C000000
LUA_BID_VERB_SEG_ERROR	0x0F000000
LUA_NO_PREVIOUS_BID_ENABLED	0x10000000
LUA_NO_DATA	0x11000000
LUA_BID_ALREADY_ENABLED	0x12000000
LUA_VERB_RECORD_SPANS_SEGMENTS	0x13000000
LUA_INVALID_FLOW	0x14000000
LUA_NOT_ACTIVE	0x15000000
LUA_VERB_LENGTH_INVALID	0x16000000
LUA_REQUIRED_FIELD_MISSING	0x19000000
LUA_READY	0x30000000
LUA_NOT_READY	0x31000000
LUA_INIT_COMPLETE	0x32000000
LUA_SESSION_END_REQUESTED	0x33000000
LUA_NO_SLI_SESSION	0x34000000
LUA_SESSION_ALREADY_OPEN	0x35000000
LUA_INVALID_OPEN_INIT_TYPE	0x36000000
LUA_INVALID_OPEN_DATA	0x37000000
LUA_UNEXPECTED_SNA_SEQUENCE	0x38000000
LUA_NEG_RSP_FROM_BIND_ROUTINE	0x39000000
LUA_NEG_RSP_FROM_CRV_ROUTINE	0x3A000000

```
LUA_NEG_RSP_FROM_STSN_ROUTINE   0x3B000000
LUA_CRV_ROUTINE_REQUIRED        0x3C000000
LUA_STSN_ROUTINE_REQUIRED       0x3D000000
LUA_INVALID_OPEN_ROUTINE_TYPE   0x3E000000
LUA_MAX_NUMBER_OF_SENDS         0x3F000000
LUA_SEND_ON_FLOW_PENDING        0x40000000
LUA_INVALID_MESSAGE_TYPE        0x41000000
LUA_RECEIVE_ON_FLOW_PENDING     0x42000000
LUA_DATA_LENGTH_ERROR           0x43000000
LUA_CLOSE_PENDING               0x44000000
LUA_NEGATIVE_RSP_CHASE          0x46000000
LUA_NEGATIVE_RSP_SHUTC          0x47000000
LUA_NEGATIVE_RSP_RSHUTD         0x48000000
LUA_NO_RECEIVE_TO_PURGE         0x4A000000
LUA_CANCEL_COMMAND_RECEIVED     0x4D000000
LUA_RUI_WRITE_FAILURE           0x4E000000
LUA_INVALID_SESSION_TYPE        0x4F000000
LUA_SLI_BID_PENDING             0x51000000
LUA_SLI_PURGE_PENDING           0x52000000
LUA_PROCEDURE_ERROR             0x53000000
LUA_INVALID_SLI_ENCR_OPTION     0x54000000
LUA_RECEIVED_UNBIND             0x55000000
LUA_RECEIVED_UNBIND_HOLD        0x56000000
LUA_RECEIVED_UNBIND_NORMAL      0x57000000
LUA_SLI_LOGIC_ERROR             0x7F000000
LUA_TERMINATED                  0x80000000
LUA_NO_RUI_SESSION              0x81000000
LUA_DUPLICATE_RUI_INIT          0x82000000
LUA_INVALID_PROCESS             0x83000000
LUA_API_MODE_CHANGE             0x85000000
LUA_COMMAND_COUNT_ERROR         0x87000000
LUA_NO_READ_TO_PURGE             0x88000000
LUA_MULTIPLE_WRITE_FLOWS        0x89000000
LUA_DUPLICATE_READ_FLOW         0x8A000000
LUA_DUPLICATE_WRITE_FLOW        0x8B000000
LUA_LINK_NOT_STARTED            0x8C000000
LUA_INVALID_ADAPTER             0x8D000000
LUA_ENCR_DECR_LOAD_ERROR        0x8E000000
LUA_ENCR_DECR_PROC_ERROR        0x8F000000
LUA_INVALID_PUNAME              0x90000000
LUA_UNAUTHORIZED_ACCESS         0x90020000
LUA_INVALID_LUNUMBER            0x91000000
LUA_INVALID_FORMAT              0x92000000
LUA_DUPLICATE_RUI_REINIT        0x93000000
LUA_REINIT_INVALID              0x94000000
LUA_TCPCV_LENGTH_INVALID        0x95000000
LUA_LINK_NOT_STARTED_RETRY      0x95FF0000
LUA_NEG_RSP_FROM_SDT_ROUTINE    0x96000000
LUA_NEG_NOTIFY_RSP              0xBE000000
LUA_RUI_LOGIC_ERROR             0xBF000000
LUA_COBOL_NOT_SUPPORTED         0xC0000000
LUA_DUPLICATE_RUI_INIT_PRIMARY  0xC2000000
LUA_LU_INOPERATIVE              0xFF000000
```

The following secondary return codes are SNA sense codes. They are listed both in the standard byte ordering used by LUA and in the byte ordering used for SNA sense codes in SNA reference manuals.

```
LUA_RESOURCE_NOT_AVAILABLE      0x00000108   (SNA sense 0801 0000)
LUA_RU_DATA_ERROR               0x00000110   (SNA sense 1001 0000)
LUA_INCORRECT_SEQUENCE_NUMBER   0x00000120   (SNA sense 2001 0000)
LUA_INVALID_SC_OR_NC_RH         0x00000140   (SNA sense 4001 0000)
LUA_RU_LENGTH_ERROR             0x00000210   (SNA sense 1002 0000)
LUA_CHAINING_ERROR              0x00000220   (SNA sense 2002 0000)
LUA_FUNCTION_NOT_SUPPORTED      0x00000310   (SNA sense 1003 0000)
LUA_BRACKET                     0x00000320   (SNA sense 2003 0000)
LUA_BB_NOT_ALLOWED              0x00000340   (SNA sense 4003 0000)
```

```
LUA_NAU_INOPERATIVE             0x00000380   (SNA sense 8003 0000)
LUA_DIRECTION                   0x00000420   (SNA sense 2004 0000)
LUA_EB_NOT_ALLOWED              0x00000440   (SNA sense 4004 0000)
LUA_SESSION_LIMIT_EXCEEDED      0x00000508   (SNA sense 0805 0000)
LUA_DATA_TRAFFIC_RESET          0x00000520   (SNA sense 2005 0000)
LUA_NO_SESSION                  0x00000580   (SNA sense 8005 0000)
LUA_DATA_TRAFFIC_QUIESCED       0x00000620   (SNA sense 2006 0000)
LUA_EXCEPTION_RSP_NOT_ALLOWED   0x00000640   (SNA sense 4006 0000)
LUA_CATEGORY_NOT_SUPPORTED      0x00000710   (SNA sense 1007 0000)
LUA_DATA_TRAFFIC_NOT_RESET      0x00000720   (SNA sense 2007 0000)
LUA_DEFINITE_RSP_NOT_ALLOWED    0x00000740   (SNA sense 4007 0000)
LUA_NO_BEGIN_BRACKET             0x00000820   (SNA sense 2008 0000)
LUA_PACING_NOT_SUPPORTED        0x00000840   (SNA sense 4008 0000)
LUA_MODE_INCONSISTENCY          0x00000908   (SNA sense 0809 0000)
LUA_SC_PROTOCOL_VIOLATION       0x00000920   (SNA sense 2009 0000)
LUA_CD_NOT_ALLOWED              0x00000940   (SNA sense 4009 0000)
LUA_IMMEDIATE_REQ_MODE_ERROR    0x00000A20   (SNA sense 200A 0000)
LUA_NO_RESPONSE_NOT_ALLOWED     0x00000A40   (SNA sense 400A 0000)
LUA_BRACKET_RACE_ERROR          0x00000B08   (SNA sense 800B 0000)
LUA_QUEUED_RESPONSE_ERROR       0x00000B20   (SNA sense 200B 0000)
LUA_CHAINING_NOT_SUPPORTED      0x00000B40   (SNA sense 400B 0000)
LUA_ERP_SYNC_EVENT_ERROR        0x00000C20   (SNA sense 200C 0000)
LUA_BRACKETS_NOT_SUPPORTED      0x00000C40   (SNA sense 400C 0000)
LUA_RSP_BEFORE_SENDING_REQ      0x00000D20   (SNA sense 200D 0000)
LUA_CD_NOT_SUPPORTED            0x00000D40   (SNA sense 400D 0000)
LUA_RSP_CORRELATION_ERROR       0x00000E20   (SNA sense 200E 0000)
LUA_RSP_PROTOCOL_ERROR          0x00000F20   (SNA sense 200F 0000)
LUA_INCORRECT_USE_OF_FI         0x00000F40   (SNA sense 400F 0000)
LUA_ALTERNATE_CODE_NOT_SUPPORT  0x00001040   (SNA sense 4001 0000)
LUA_INCORRECT_RU_CATEGORY       0x00001140   (SNA sense 4011 0000)
LUA_INSUFFICIENT_RESOURCES      0x00001208   (SNA sense 0812 0000)
LUA_INCORRECT_REQUEST_CODE      0x00001240   (SNA sense 4012 0000)
LUA_BB_REJECT_NO_RTR            0x00001308   (SNA sense 0813 0000)
LUA_INCORRECT_SPEC_OF_SDI_RTI   0x00001340   (SNA sense 4013 0000)
LUA_BB_REJECT_RTR               0x00001408   (SNA sense 0814 0000)
LUA_INCORRECT_DR1I_DR2I_ERI     0x00001440   (SNA sense 4014 0000)
LUA_INCORRECT_USE_OF_QRI        0x00001540   (SNA sense 4015 0000)
LUA_INCORRECT_USE_OF_EDI        0x00001640   (SNA sense 4016 0000)
LUA_INCORRECT_USE_OF_PDI        0x00001740   (SNA sense 4017 0000)
LUA_RECEIVER_IN_TRANSMIT_MODE   0x00001B08   (SNA sense 081B 0000)
LUA_REQUEST_NOT_EXECUTABLE      0x00001C08   (SNA sense 081C 0000)
LUA_INVALID_SESSION_PARAMETERS  0x00002108   (SNA sense 0821 0000)
LUA_UNIT_OF_WORK_ABORTED        0x00002408   (SNA sense 0824 0000)
LUA_FM_FUNCTION_NOT_SUPPORTED   0x00002608   (SNA sense 0826 0000)
LUA_LU_COMPONENT_DISCONNECTED   0x00003108   (SNA sense 0831 0000)
LUA_INVALID_PARAMETER_FLAGS     0x00003308   (SNA sense 0833 0000)
LUA_INVALID_PARAMETER           0x00003508   (SNA sense 0835 0000)
LUA_CRYPTOGRAPHY_INOPERATIVE    0x00004808   (SNA sense 0848 0000)
LUA_REQ_RESOURCES_NOT_AVAIL     0x00004B08   (SNA sense 084B 0000)
LUA_SSCP_LU_SESSION_NOT_ACTIVE  0x00005708   (SNA sense 0857 0000)
LUA_SYNC_EVENT_RESPONSE         0x00006708   (SNA sense 0867 0000)
LUA_SESSION_SERVICE_PATH_ERROR  0x00007D08   (SNA sense 087D 0000)
LUA_NEGOTIABLE_BIND_ERROR       0x01003508   (SNA sense 0835 0001)
LUA_REC_CORR_TABLE_FULL         0x01007808   (SNA sense 0878 0001)
LUA_NON_UNIQ_ID                 0x011000C0   (SNA sense C000 1001)
LUA_INV_NAU_ADDR                0x012000C0   (SNA sense C000 2001)
LUA_BIND_FM_PROFILE_ERROR       0x02003508   (SNA sense 0835 0002)
LUA_SSCP_PLU_SESS_NOT_ACTIVE    0x02005708   (SNA sense 0857 0002)
LUA_SEND_CORR_TABLE_FULL        0x02007808   (SNA sense 0878 0002)
LUA_NON_UNIQ_NAU_AD             0x021000C0   (SNA sense C000 1002)
LUA_INV_ADPT_NUM                0x022000C0   (SNA sense C000 2002)
LUA_BIND_TS_PROFILE_ERROR       0x03003508   (SNA sense 0835 0003)
LUA_SSCP_SLU_SESS_INACT         0x03005708   (SNA sense 0857 0003)
LUA_SLU_SESSION_LIMIT_EXCEEDED  0x0A000508   (SNA sense 0805 000A)
LUA_BIND_LU_TYPE_ERROR          0x0E003508   (SNA sense 0835 000E)
LUA_HDX_BRACKET_STATE_ERROR     0x21010510   (SNA sense 1005 0121)
LUA_RESPONSE_ALREADY_SENT       0x22010510   (SNA sense 1005 0122)
```

```
LUA_EXR_SENSE_INCORRECT         0x23010510   (SNA sense 1005 0123)
LUA_RESPONSE_OUT_OF_ORDER       0x24010510   (SNA sense 1005 0124)
LUA_CHASE_RESPONSE_REQUIRED     0x25010510   (SNA sense 1005 0125)
```

Appendix B. Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Trademarks

Bibliography

The following IBM publications provide information about the topics discussed in this library. The publications are divided into the following broad topic areas:

- IBM Communications Server for AIX
- IBM Communications Server for Linux
- Systems Network Architecture (SNA)
- Advanced Program-to-Program Communication (APPC)
- Programming

For IBM Communications Server for AIX and IBM Communications Server for Linux books, brief descriptions are provided. For other books, only the titles and order numbers are shown here.

IBM Communications Server for AIX Publications

The IBM Communications Server for AIX library comprises the following books. In addition, softcopy versions of these documents are provided on the CD-ROM. See *IBM Communications Server for AIX Quick Beginnings* for information about accessing the softcopy files on the CD-ROM. To install these softcopy books on your system, you require 9–15 MB of hard disk space (depending on which national language versions you install).

- *IBM Communications Server for AIX Migration Guide* (SC31-8585)

  This book explains how to migrate from Communications Server for AIX Version 4 Release 2 or earlier to IBM Communications Server for AIX Version 6.
- *IBM Communications Server for AIX Quick Beginnings* (GC31-8583)

  This book is a general introduction to IBM Communications Server for AIX, including information about supported network characteristics, installation, configuration, and operation.
- *IBM Communications Server for AIX Administration Guide* (SC31-8586)

  This book provides an overview of SNA and IBM Communications Server for AIX, and information about IBM Communications Server for AIX configuration and operation.
- *IBM Communications Server for AIX Administration Command Reference* (SC31-8587)

  This book provides information about SNA and IBM Communications Server for AIX commands.
- *IBM Communications Server for AIX or Linux CPI-C Programmer's Guide* (SC23-8591)

  This book provides information for experienced “C” or Java™programmers about writing SNA transaction programs using the IBM Communications Server CPI Communications API.
- *IBM Communications Server for AIX or Linux APPC Programmer's Guide* (SC23-8592)

  This book contains the information you need to write application programs using Advanced Program-to-Program Communication (APPC).
- *IBM Communications Server for AIX or Linux LUA Programmer's Guide* (SC23-8590)

  This book contains the information you need to write applications using the Conventional LU Application Programming Interface (LUA).

- *IBM Communications Server for AIX or Linux CSV Programmer's Guide* (SC23-8589)

  This book contains the information you need to write application programs using the Common Service Verbs (CSV) application program interface (API).
- *IBM Communications Server for AIX or Linux MS Programmer's Guide* (SC23-8596)

  This book contains the information you need to write applications using the Management Services (MS) API.
- *IBM Communications Server for AIX NOF Programmer's Guide* (SC31-8595)

  This book contains the information you need to write applications using the Node Operator Facility (NOF) API.
- *IBM Communications Server for AIX Diagnostics Guide* (SC31-8588)

  This book provides information about SNA network problem resolution.
- *IBM Communications Server for AIX or Linux APPC Application Suite User's Guide* (SC23-8595)

  This book provides information about APPC applications used with IBM Communications Server for AIX.
- *IBM Communications Server for AIX Glossary* (GC31-8589)

  This book provides a comprehensive list of terms and definitions used throughout the IBM Communications Server for AIX library.

IBM Communications Server for Linux Publications

The IBM Communications Server for Linux library comprises the following books. In addition, softcopy versions of these documents are provided on the CD-ROM. See *IBM Communications Server for Linux Quick Beginnings* for information about accessing the softcopy files on the CD-ROM. To install these softcopy books on your system, you require 9–15 MB of hard disk space (depending on which national language versions you install).

- *IBM Communications Server for Linux Quick Beginnings* (GC31-6768 and GC31-6769)

  This book is a general introduction to IBM Communications Server for Linux, including information about supported network characteristics, installation, configuration, and operation. There are two versions of this book:

  GC31-6768 is for IBM Communications Server for Linux on the i686, x86_64, and ppc64 platforms

  GC31-6769 is for IBM Communications Server for Linux on System z.
- *IBM Communications Server for Linux Administration Guide* (SC31-6771)

  This book provides an overview of SNA and IBM Communications Server for Linux, and information about IBM Communications Server for Linux configuration and operation.
- *IBM Communications Server for Linux Administration Command Reference* (SC31-6770)

  This book provides information about SNA and IBM Communications Server for Linux commands.
- *IBM Communications Server for AIX or Linux CPI-C Programmer's Guide* (SC23-8691)

  This book provides information for experienced "C" or Javaprogrammers about writing SNA transaction programs using the IBM Communications Server CPI Communications API.
- *IBM Communications Server for AIX or Linux APPC Programmer's Guide* (SC23-8692)

This book contains the information you need to write application programs using Advanced Program-to-Program Communication (APPC).

- *IBM Communications Server for AIX or Linux LUA Programmer's Guide* (SC23-8690)

  This book contains the information you need to write applications using the Conventional LU Application Programming Interface (LUA).
- *IBM Communications Server for AIX or Linux CSV Programmer's Guide* (SC23-8689)

  This book contains the information you need to write application programs using the Common Service Verbs (CSV) application program interface (API).
- *IBM Communications Server for AIX or Linux MS Programmer's Guide* (SC23-8596)

  This book contains the information you need to write applications using the Management Services (MS) API.
- *IBM Communications Server for Linux NOF Programmer's Guide* (SC31-6778)

  This book contains the information you need to write applications using the Node Operator Facility (NOF) API.
- *IBM Communications Server for Linux Diagnostics Guide* (SC31-6779)

  This book provides information about SNA network problem resolution.
- *IBM Communications Server for AIX or Linux APPC Application Suite User's Guide* (SC23-8595)

  This book provides information about APPC applications used with IBM Communications Server for Linux.
- *IBM Communications Server for Linux Glossary* (GC31-6780)

  This book provides a comprehensive list of terms and definitions used throughout the IBM Communications Server for Linux library.

Systems Network Architecture (SNA) Publications

The following books contain information about SNA networks:

- *Systems Network Architecture: Format and Protocol Reference Manual—Architecture Logic for LU Type 6.2* (SC30-3269)
- *Systems Network Architecture: Formats* (GA27-3136)
- *Systems Network Architecture: Guide to SNA Publications* (GC30-3438)
- *Systems Network Architecture: Network Product Formats* (LY43-0081)
- *Systems Network Architecture: Technical Overview* (GC30-3073)
- *Systems Network Architecture: APPN Architecture Reference* (SC30-3422)
- *Systems Network Architecture: Sessions between Logical Units* (GC20-1868)
- *Systems Network Architecture: LU 6.2 Reference—Peer Protocols* (SC31-6808)
- *Systems Network Architecture: Transaction Programmer's Reference Manual for LU Type 6.2* (GC30-3084)
- *Systems Network Architecture: 3270 Datastream Programmer's Reference* (GA23-0059)
- *Networking Blueprint Executive Overview* (GC31-7057)
- *Systems Network Architecture: Management Services Reference* (SC30-3346)

APPC Publications

The following books contain information about Advanced Program-to-Program Communication (APPC):

- *APPC Application Suite V1 User's Guide* (SC31-6532)
- *APPC Application Suite V1 Administration* (SC31-6533)
- *APPC Application Suite V1 Programming* (SC31-6534)

- *APPC Application Suite V1 Online Product Library* (SK2T-2680)
- *APPC Application Suite Licensed Program Specifications* (GC31-6535)
- *z/OS V1R2.0 Communications Server: APPC Application Suite User's Guide* (SC31-8809)

Programming Publications

The following books contain information about programming:

- *Common Programming Interface Communications CPI-C Reference* (SC26-4399)
- *Communications Server for OS/2 Version 4 Application Programming Guide* (SC31-8152)

Index

Program Number: 5765-E51 and 5724-i33

Printed in USA

SC23-8590-00