

IBM Reliable Scalable Cluster Technology for AIX 5L



LAPI Programming Guide

IBM Reliable Scalable Cluster Technology for AIX 5L



LAPI Programming Guide

Note

Before using this information and the product it supports, read the information in "Notices" on page 291.

Third Edition (April 2005)

This edition applies to:

- version 5, release 2 of IBM AIX 5L for POWER™ (product number 5765-E62) with the 5200-06 Recommended Maintenance package
- version 5, release 3 of IBM AIX 5L for POWER (product number 5765-G03)
- all subsequent releases and modifications until otherwise indicated in new editions

Vertical lines (|) in the left margin indicate technical changes to the previous edition of this book.

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

IBM welcomes your comments. A form for your comments appears at the back of this publication. If the form has been removed, address your comments to:

IBM Corporation, Department 55JA, Mail Station P384
2455 South Road
Poughkeepsie, NY 12601-5400
United States of America

FAX (United States and Canada): 1+845+432-9405
FAX (Other Countries)
Your International Access Code +1+845+432-9405

IBMLink™ (United States customers only): IBMUSM10(MHVRCFS)
Internet: mhvrcfs@us.ibm.com

If you would like a reply, be sure to include your name, address, telephone number, or FAX number.

Make sure to include the following in your comment or note:

- Title and order number of this book
- Page number or topic related to your comment

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.

© Copyright International Business Machines Corporation 2003, 2005. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Figures	ix
Tables	xi
About this book	xiii
Who should use this book	xiii
Conventions and terminology used in this book	xiii
Conventions	xiii
Terminology	xiv
Prerequisite and related information	xiv
Using LookAt to find message explanations	xv
How to send your comments.	xvi

Part 1. LAPI concepts 1

Chapter 1. What is the low-level application programming interface (LAPI)?	3
Why use LAPI?	7
Chapter 2. An overview of LAPI	9
Initialization and termination	10
Querying and setting up the runtime environment	10
Address-related functions	10
Put and get functions	11
Active messages	12
Non-contiguous data transfer.	12
Remote read-modify-write functions	13
Generic data transfer functions	14
Progress-monitoring functions	15
Message ordering functions	16
Utility functions	16
Error message functions	16
Recovery-related functions	16
Chapter 3. What's new in LAPI?	19
Tips for LAPI users	21

Part 2. Basic LAPI tasks 23

Chapter 4. Installing RSCT LAPI	25
Requirements	25
Hardware	25
Software	25
How is RSCT LAPI packaged?	25
RSCT LAPI filesets	26
Installation steps	27
Uninstallation steps	27
Migration and coexistence.	28
Chapter 5. Setting up, initializing, and terminating LAPI	29
Setting and querying the LAPI environment	29
Setting environment variables	29
Initializing LAPI.	31

	Terminating LAPI	33
	Chapter 6. Transferring data	35
	Data transfer operations	35
	Flow of "put" operations	35
	Flow of "get" operations	36
	Flow of read-modify-write operations	37
	Non-contiguous data transfer.	37
	Using vectors	37
	Using data gather/scatter programs (DGSPs).	43
	Detecting completion.	47
	LAPI handlers	47
	LAPI counters	52
	Specifying target-side addresses	53
	Additional progress functions.	53
	Chapter 7. Active messaging	55
	Flow of active message operations	55
	Using LAPI_Amsend: a complete LAPI program.	56
	Chapter 8. Collecting statistics	63
	Printing data transfer statistics	63
	Querying US and UDP/IP statistics	63
	Querying local send statistics	63
	Querying shared memory statistics	64
	Chapter 9. Using LAPI's profiling interface	65
	Performing name-shift profiling	66
	A sample profiling program	68
	Chapter 10. Compiling and running LAPI programs	71
	<hr/> Part 3. Advanced LAPI tasks	73
	Chapter 11. Advanced programming	75
	The enhanced header handler interface.	75
	Inline completion handlers.	77
	LAPI performance considerations	78
	Use of handlers	78
	Running in interrupt mode	78
	Running in UDP/IP mode	78
	User header data	79
	Send-side copy of small messages	79
	Receive-side optimization for single-packet messages	79
	Tunable environment variables	80
	32-bit and 64-bit interoperability.	81
	The lapi_long_t datatype	81
	The LAPI_Address_init64 subroutine	81
	The LAPI_Xfer interface	81
	Chapter 12. Lock sharing	83
	Scenarios without lock sharing	83
	Scenarios with lock sharing	87
	Correctness of lock sharing	91
	Implications and restrictions	93
	Initialization and termination	93

Other LAPI calls	94
Callbacks	94
Long critical sections.	94
Lock preemption	94
Receive/timer interrupts.	94
Performance of multi-threaded programs	95
Compatibility.	95
A sample lock sharing program	95
Chapter 13. Bulk transfer of messages	99
Chapter 14. Striping, failover, and recovery	103
Using failover and recovery	103
Monitoring adapter status	103
Requesting the use of multiple adapters	104
Failover and recovery restrictions.	106
Data striping	106
Communication and memory considerations.	108
IP communication	108
US communication	109
Chapter 15. Threaded programming	111
General guidelines	111
Using LAPI_Address_init	111
Making global fence calls.	112
Making "wait on counter" calls.	112
Synchronizing threads across tasks	112
Using handlers	112
LAPI threads	113
Chapter 16. Using LAPI on a standalone system	115
Standalone setup	115
Standalone initialization	117
Using UDP/IP mode	117
Using US mode	117
Compiling LAPI programs on a standalone system	118

Part 4. LAPI reference. 119

Chapter 17. LAPI man pages.	121
lapi_subroutines	122
Chapter 18. Subroutines for all systems (PE and standalone).	125
LAPI_Addr_get	126
LAPI_Addr_set	128
LAPI_Address.	130
LAPI_Address_init	132
LAPI_Address_init64	134
LAPI_Amsend.	136
LAPI_Amsendv	143
LAPI_Fence	149
LAPI_Get	151
LAPI_Getcptr	154
LAPI_Getv	156
LAPI_Gfence	161
LAPI_Init.	163

LAPI_Msg_string	169
LAPI_Msgpoll	171
LAPI_Probe	174
LAPI_Put	176
LAPI_Putv	179
LAPI_Qenv	184
LAPI_Rmw	188
LAPI_Rmw64	192
LAPI_Senv	196
LAPI_Setcptr	198
LAPI_Term	201
LAPI_Util	203
LAPI_Waitcptr	217
LAPI_Xfer	219
Chapter 19. Subroutines for standalone systems	235
LAPI_Nopoll_wait	236
LAPI_Purge_totask	238
LAPI_Resume_totask	240
LAPI_Setcptr_wstatus	242
Chapter 20. LAPI sample programs	245
Sample program directory structure	245
Using the LAPI sample programs.	250
Summary of constructs and techniques for LAPI programming	250
Appendix A. Product-related information	253
RSCT version	253
ISO 9000	253
Product-related feedback.	254
Appendix B. LAPI execution models.	255
The IP/US execution model.	255
The shared memory execution model	257
Cross memory kernel extension	258
LAPI shared memory: functional flow	259
LAPI shared memory: requirements and restrictions	259
Appendix C. LAPI messages, return codes, and return values	261
LAPI attention messages.	261
LAPI return codes	261
LAPI error codes.	261
LAPI return values	264
Appendix D. LAPI environment variables and runtime attributes.	269
Environment variables.	269
Variables for communication	269
Variables for data transfer	269
Variables for diagnostics	270
Variables for performance tuning	270
Variables for POE	271
Variables for shared memory	274
Variables for standalone systems.	274
Runtime attributes	275
Attributes you can query or set	275
Attributes you can query	276

	Appendix E. LAPI datatypes	279
I	Appendix F. LAPI constants and size limits	283
	Appendix G. LAPI restrictions	285
	General restrictions	285
	Use of segment registers (32-bit applications only)	285
	Other restrictions.	285
	Glossary	287
	Notices	291
	Trademarks.	292
	Index	295

Figures

1.	How LAPI sets up the mode of communication	31
2.	The sequence of events for a "put" operation	36
3.	The sequence of events for a "get" operation	36
4.	The sequence of events for a read-modify-write operation	37
5.	Transferring data with type LAPI_GEN_IOVECTOR	40
6.	Transferring data with type LAPI_GEN_STRIDED_XFER	41
7.	The completion handler queue	51
8.	The sequence of events for an active message operation	56
9.	Inline completion handler flow	77
10.	A program initiates a call to LAPI, without lock sharing	84
11.	LAPI initiates a callback, without lock sharing	85
12.	A program initiates a call to LAPI, with embedded up- and down- calls	86
13.	LAPI initiates a callback, with embedded up- and down- calls	87
14.	A program initiates a call to LAPI, with one lock acquisition and release pair	88
15.	LAPI initiates a callback, with one lock acquisition and release pair	89
16.	A program initiates a call to LAPI, with embedded up- and down- calls and one lock acquisition and release pair	90
17.	LAPI initiates a callback, with embedded up- and down- calls and one lock acquisition and release pair	91
18.	Critical sections under lock sharing	93
19.	LAPI packet data flow	100
20.	LAPI bulk data flow	101
21.	Execution sequence of the accumulate_and_return.Am sample	248
22.	Execution sequence of the accumulate_and_return.Put sample	248
23.	A LAPI thread model	256
24.	LAPI_Put without shared memory	258
25.	LAPI_Put with shared memory	258

Tables

1.	Typographic conventions	xiii
2.	Terminology	xiv
3.	Changes in this edition	19
4.	Changes in the second edition	19
5.	Changes in the first edition	20
6.	Differences between LAPI versions	21
7.	Rules for vector transfer	43
8.	LAPI handlers	52
9.	LAPI counters	53
10.	LAPI profiling interfaces	65
11.	Compiling LAPI programs on a system that is running PE	71
12.	Failover and recovery operations	106
13.	Compiling LAPI programs on a standalone system	118
14.	lapi_util_type_t types	204
15.	The lapi_reg_dgsp_t fields	205
16.	The lapi_resv_dgsp_t fields	205
17.	The lapi_dref_dgsp_t fields	206
18.	The lapi_reg_ddm_t fields	206
19.	The lapi_pack_dgsp_t fields	207
20.	The lapi_unpack_dgsp_t fields	207
21.	The lapi_add_udp_port_t fields	208
22.	The lapi_thread_func_t fields	208
23.	LAPI_Xfer structure types	220
24.	LAPI_Amsend and lapi_am_t equivalents	221
25.	LAPI_Amsendv and lapi_amv_t equivalents	222
26.	The lapi_amdgsp_t fields	223
27.	LAPI_Get and lapi_get_t equivalents	225
28.	LAPI_Getv and lapi_getv_t equivalents	225
29.	LAPI_Put and lapi_put_t equivalents	226
30.	LAPI_Putv and lapi_putv_t equivalents	227
31.	LAPI_Rmw and lapi_rmw_t equivalents	228
32.	Constructs and techniques for LAPI programming	250
33.	LAPI attention messages	261
34.	LAPI return codes	261
35.	LAPI error codes	261
36.	LAPI return values	264
37.	Environment variables for communication	269
38.	Environment variables for data transfer	269
39.	Environment variables for diagnostics	270
40.	Environment variables for performance tuning	270
41.	Environment variables for POE	272
42.	Environment variables for shared memory	274
43.	Environment variables for standalone systems	274
44.	Runtime attributes you can query or set	275
45.	Attributes that return integers	276
46.	Attributes that return multiple values	277
47.	LAPI datatypes	279

About this book

This book includes conceptual, procedural, and reference information about the *low-level application programming interface (LAPI)*.

Who should use this book

This book is intended for programmers who want to write and run LAPI programs on the AIX[®] operating system. The programmer should be experienced with UNIX[®], networked systems, and the C or FORTRAN programming language.

Conventions and terminology used in this book

Conventions

This book uses the typographic conventions shown in Table 1:

Table 1. Typographic conventions

Convention	Usage
bold	Bold words or characters represent system elements that you must use literally, such as the names of commands, constants, datatypes, directories, environment variables, files, flags, paths, return values, structures, and subroutines.
constant width	Examples and information that the system displays appear in constant-width typeface.
<i>italic</i>	<i>Italicized</i> words or characters represent the values of programming variables or parameters that you must supply. <i>Italics</i> are also used for book titles, for the first use of a glossary term, and for general emphasis in text.
<u>underlined</u>	<ol style="list-style-type: none">When used to show the size of a parameter, a comparison of values, or a range of values, valid values for the <i>query</i> parameter of the LAPI_Qenv subroutine are <u>underlined</u>. For example: <u>NUM_TASKS</u> is a shorthand notation for: LAPI_Qenv(hndl, <u>NUM_TASKS</u>, ret_val) For a list of the <i>query</i> parameter's valid values, see "LAPI_Qenv" on page 184.<u>Underlined</u> characters are also a shorthand notation for: LAPI_Xfer with transfer type LAPI_xfer-type_XFER The eight valid values for <i>xfer-type</i> are: LAPI_AM_XFER, LAPI_AMV_XFER, LAPI_DGSP_XFER, LAPI_GET_XFER, LAPI_GETV_XFER, LAPI_PUT_XFER, LAPI_PUTV_XFER, and LAPI_RMW_XFER So, for example, this document would refer to: LAPI_Xfer with transfer type LAPI_AM_XFER as: <u>AM</u> For more information about these transfer types, see "LAPI_Xfer" on page 219.

Table 1. Typographic conventions (continued)

Convention	Usage
	<ol style="list-style-type: none"> 1. In the left margin of the book, vertical lines indicate technical changes to the information. 2. In syntax statements, vertical lines are used as <i>pipe</i> characters.
[item/element]	Brackets indicate an optional item or an element in an array.
...	Ellipses indicate items that can be repeated.

Terminology

This book uses the terminology conventions shown in Table 2:

Table 2. Terminology

Term	Usage
HPS	A shorthand notation for the <i>High Performance Switch</i> , which works in conjunction with IBM® @server® p5 servers (575, 595)
pSeries® HPS	A shorthand notation for the <i>pSeries High Performance Switch</i> , which works in conjunction with IBM @server pSeries servers (655, 690)

See the “Glossary” on page 287 for definitions of some of the other terms that are used in this book.

Prerequisite and related information

The core Reliable Scalable Cluster Technology (RSCT) publications are:

- *RSCT: Administration Guide*, SA22-7889, provides an overview of the RSCT components and describes how to:
 - Create and administer RSCT peer domains.
 - Manage and monitor resources using the resource monitoring and control (RMC) subsystem.
 - Administer cluster security services for RSCT peer domains and CSM management domains.
- *RSCT: Diagnosis Guide*, SA23-2202, describes how to diagnose and resolve problems related to the various components of RSCT. This book is a companion volume to *RSCT: Messages*, which lists the error messages that may be generated by each RSCT component. While *RSCT: Messages* describes the appropriate user responses to messages that are generated by RSCT components, this book contains additional and more detailed diagnostic procedures.
- *RSCT: Messages*, GA22-7891, lists the error messages that may be generated by each RSCT component. For each message, this manual provides an explanation of the message, and describes how you should respond to it.
- *RSCT for AIX 5L™: Technical Reference*, SA22-7890, and *RSCT for Linux®: Technical Reference*, SA22-7893, provide detailed reference information about all of the RSCT commands, daemons, files, and scripts.

In addition to these core RSCT publications, the library contains the following publications of interest:

- *RSCT: Group Services Programming Guide and Reference*, SA22-7888, contains information for programmers who want to write new clients that use the group

services subsystem's application programming interface (GSAPI) or who want to add the use of group services to existing programs. This book is intended for programmers of system management applications who want to use group services to make their applications highly available.

- *RSCT for AIX 5L: LAPI Programming Guide*, SA22-7936, provides conceptual, procedural, and reference information about the low-level application programming interface (LAPI). LAPI is part of the AIX implementation of RSCT only; it is not available with RSCT for Linux. LAPI is a message-passing API that provides optimal communication performance on an IBM @server pSeries High Performance Switch (pSeries HPS) or an IBM @server High Performance Switch (HPS) for p5 servers.
- *RSCT for AIX 5L: Managing Shared Disks*, SA22-7937, describes the shared disk management facilities of IBM @server Cluster 1600 server processors — the optional virtual shared disk and recoverable virtual shared disk components of RSCT for AIX 5L. These components are part of the AIX implementation of RSCT only; they are not available with RSCT for Linux. This book describes how you can use these components to manage cluster disks to enable multiple nodes to share the information they hold. The book includes an overview of the components and explains how to plan for them, install them, and use them to add reliability and availability to your data storage.

An **RSCT Documentation Updates** file is maintained on the World Wide Web at the following URL:

<http://publib.boulder.ibm.com/clresctr/docs/rsct/docupdates.html>

This file contains updates to the RSCT documentation. These updates include documentation corrections and clarifications, as well as information (such as needed software patches) that was discovered after the RSCT books were published. Check the **RSCT Documentation Updates** file for pertinent information.

To access all RSCT documentation, refer to the **IBM @server Cluster Information Center**. This Web site, which is located at **<http://publib.boulder.ibm.com/infocenter/clresctr>**, contains the most recent RSCT documentation in PDF and HTML formats.

The current RSCT books and earlier versions of the library are also available in PDF format from the **IBM Publications Center** Web site, which is located at **<http://www.ibm.com/shop/publications/order>**. It is easiest to locate a manual in the **IBM Publications Center** by supplying the manual's publication number. The publication number for each of the RSCT books is listed after the book title in the preceding list.

Using LookAt to find message explanations

LookAt is an online facility that lets you look up explanations for most of the IBM messages you encounter, as well as for some system abends and codes. You can use LookAt from the following locations to find IBM message explanations:

- The Internet. You can access IBM message explanations directly from the LookAt Web site:

<http://www.ibm.com/eserver/zseries/zos/bkserv/lookat>

- Your wireless handheld device. You can use the LookAt Mobile Edition with a handheld device that has wireless access and an Internet browser (for example:

Internet Explorer for Pocket PCs, Blazer, Eudora for Palm OS, or Opera for Linux handheld devices). Link to the LookAt Mobile Edition from the LookAt Web site.

How to send your comments

Your feedback is important in helping to provide accurate, high-quality information. If you have any comments about this book or any other RSCT documentation:

- Go to the IBM @server Cluster Information Center home page at:

<http://publib.boulder.ibm.com/infocenter/clresctr>

Click on the **Contact us** link to go to our feedback page, where you can enter and submit your comments.

- Send your comments by e-mail to: mhvrcfs@us.ibm.com
Include the book title and order number, and, if applicable, the specific location of the information about which you have comments (for example, a page number, table number, or figure number).
- Fill out one of the forms at the back of this book and return it by mail, by fax, or by giving it to an IBM representative.

Part 1. LAPI concepts

Chapter 1. What is the low-level application programming interface (LAPI)?	3
Why use LAPI?	7
Chapter 2. An overview of LAPI	9
Initialization and termination	10
Querying and setting up the runtime environment	10
Address-related functions	10
Put and get functions	11
Active messages	12
Non-contiguous data transfer.	12
Remote read-modify-write functions	13
Generic data transfer functions	14
Progress-monitoring functions	15
Message ordering functions	16
Utility functions	16
Error message functions	16
Recovery-related functions	16
Chapter 3. What's new in LAPI?	19
Tips for LAPI users	21

Chapter 1. What is the low-level application programming interface (LAPI)?

The low-level application programming interface (LAPI) is a message-passing API that provides a *one-sided communication model*. In this model, one task initiates a communication operation to a second task. The completion of the communication does not require the second task to take a complementary action. RSCT LAPI provides optimal communication performance on an IBM @server pSeries High Performance Switch (pSeries HPS) or an IBM @server High Performance Switch (HPS) for p5 servers. PSSP LAPI provides optimal communication performance on the SP™ Switch2.

The LAPI library provides basic operations to "put" data to and "get" data from one or more virtual addresses of a remote task. LAPI also provides an *active message* infrastructure. With active messaging, programmers can install a set of handlers that are called and run in the address space of a target task on behalf of the task originating the active message. Among their other uses, these handlers can be used to dynamically determine the target address (or addresses) where data from the originating task must be stored. You can use this generic interface to customize LAPI functions for your environment.

Some of LAPI's other general characteristics include:

- Flow control
- Support for large messages
- Support for generic non-contiguous messages
- Non-blocking calls
- Interrupt and polling modes
- Efficient exploitation of switch functions
- Event monitoring support (to simulate blocking calls, for example) for various types of completion events

LAPI is meant to be used by programming libraries, and by power programmers for whom performance is more important than code portability.

To use LAPI, you need to understand the following basic concepts and the related LAPI characteristics:

- **LAPI handle**

You interact with LAPI through an opaque object called a *LAPI handle*. This object is also referred to as a *LAPI instance* or a *LAPI context*. Almost without exception, LAPI function calls take a LAPI handle as the first argument.

- **Data buffer**

You provide data to LAPI for reading and writing. For contiguous data transfer, a *data buffer* is defined by a base address and data length. LAPI also provides a number of methods for transferring non-contiguous data, such as multiple buffers, repeating block/stride descriptions, and data gather/scatter programs (DGSPs).

- **Origin and target**

LAPI communication operations usually involve two tasks. For these operations, *origin* (or *source*) denotes the task that initiates the LAPI operation and *target* (or *destination*) denotes the task where the address space is accessed during the

operation. The origin and target can be the same for any of the LAPI communication calls, but if the origin and target data areas overlap, the result of the communication is undefined.

- **Push and pull operations**

A *push* operation transfers data from the origin task to the address space of the target task. A *pull* operation transfers data from the address space of a target task into the (local) address space of the origin task.

- **Blocking and non-blocking calls**

A *blocking* procedure returns only after the operation is complete. All LAPI synchronization calls are blocking. There are no restrictions on the modification of user resources.

A *non-blocking* procedure might return before the operation is complete and before you can modify all of the resources that are specified in the call. All LAPI data transfer calls are non-blocking. A non-blocking operation is considered to be complete only after a function or an event that tests for completion indicates that the operation is complete. LAPI provides counters and handlers to signal completion of various events for non-blocking calls.

Completion of LAPI communication operations can be detected either by checking the values of counters associated with LAPI operations or by the completed execution of user-specified handlers associated with these operations. To obtain the semantics of blocking communication with LAPI, you can combine a LAPI communication operation with procedures that wait on LAPI completion events.

- **Counters and handlers**

LAPI uses counters and handlers to notify you about such events as the arrival of a message or the completion of a message. A *counter* is an opaque object; only its value is of interest. A *handler* is a callback routine that you provide. LAPI updates a counter, calls a handler, or both to notify you about an event. In terms of notification latency, handlers are generally more efficient than counters. When a handler is called, your program takes control immediately. On the other hand, your program has to poll on a counter to know about any updates.

With a few notable exceptions, the use of counters and handlers is optional in LAPI communication calls. For any counters you specify, LAPI increments the counter at certain points in the message delivery sequence. Similarly, LAPI invokes any optional callback handlers you specify at the appropriate point in the operation.

- **Completion of communication operation**

A communication operation is considered to be *complete*, (with respect to the buffer) when the buffer is reusable.

With respect to the *origin buffer*:

- a push operation is complete when the data has been copied out of the buffer at the origin task and can be overwritten
- a pull operation is complete when the origin buffer holds the new data that was obtained by the pull operation

With respect to the *target buffer*:

- a push operation is complete when the new data is available at the target buffer
- a pull operation is complete when the data has been copied out of the target buffer and the target task can overwrite that buffer

Two types of communication behavior support two different definitions of *completion*:

1. In *standard* behavior, a communication operation is complete:
 - at the origin task when it is complete with respect to the origin buffer
 - at the target task when it is complete with respect to the target buffer
2. In *synchronous* behavior, a communication operation is complete:
 - at the origin task when it is complete with respect to both the origin buffer and the target buffer
 - at the target task when it is complete with respect to the target buffer

Both standard and synchronous behaviors can be obtained for LAPI push operations; however, only synchronous behavior can be obtained for LAPI pull operations. When using send completion handlers for notification of message completion, it is important to note that this only applies to the standard behavior as defined above, for push operations.

- **Message ordering and atomicity**

Two LAPI operations that have the same origin task are considered to be *ordered with respect to the origin* if one of the operations starts after the other operation has completed at the origin task. Similarly, two LAPI operations that have the same target task are considered to be *ordered with respect to the target* if one of the operations starts after the other operation has completed at the target task. If two operations are not ordered, they are considered *concurrent*. LAPI provides no guarantees of ordering for concurrent communication operations. However, LAPI does provide mechanisms that an application can use to guarantee order.

As an example, consider the case where a node issues two standard behavior push operations to the same target node, where the target buffer regions overlap. These two operations may complete in any order, including the possibility of the first push operation overlapping in time with the second push operation. The contents of the overlapping region will be undefined, even after both push operations complete. Using synchronous behavior for both push operations (waiting for the first to complete before starting the second) will ensure that the overlapping region contains the result of the second after both push operations have completed.

- **Error handling**

LAPI provides you with the option of registering an error handler during LAPI initialization. LAPI calls the error handler with an error code that is passed as a parameter to the handler when it encounters a fatal error that would normally cause LAPI to terminate. If no error handler is registered, LAPI terminates the job when such fatal errors occur. LAPI also provides functions to translate a LAPI error code — an integer value — that is passed in to the registered error handler into a more explanatory message string.

If an error occurs during a communication operation, the error may be signaled at the origin of the operation, the target of the operation, or both. Some errors may be caught before the communication operation begins; these are signaled at the origin. However, some errors will not occur until the communication is in progress (a segmentation violation at the target, for example); these may be signaled at either end (or at both ends) of the communication.

- **Progress**

Most LAPI communication calls are non-blocking and control may thus be returned to you without the communication completing. Other LAPI calls are therefore needed to make progress with these pending communications and to drive them to completion. Various LAPI subroutines drive the progress of LAPI communication explicitly or implicitly by invoking a communication *dispatcher* that is internal to LAPI.

- **Polling mode and interrupt mode**

You can run LAPI in either polling mode or interrupt mode. In *polling mode*, the sending and receiving of messages only happens when you explicitly call a LAPI function. In *interrupt mode*, a receive interrupt is generated for incoming messages when your program is not in any LAPI function call. An extra thread, which LAPI creates at initialization, is called to handle the interrupt.

- **Statistics collection**

Using LAPI's query function, you can query statistics related to data that is transferred using the user space (US) protocol or User Datagram Protocol/Internet Protocol (UDP/IP), intra-task local copy, and shared memory. In addition, you can print data transfer statistics.

- **Profiling**

LAPI's profiling interface includes wrappers for each LAPI function, so you can collect data about each of the LAPI calls. For example, you can write a program that records the message size that is used in each call.

- **Lock sharing**

Sharing locks with LAPI provides increased efficiency in protocol layering and user programming. When you need to use a locking mechanism to protect your programs' data structures, you can use the same locking mechanism that is employed by LAPI through its lock sharing interface. This way, your program is more tightly coupled with LAPI in terms of locking. Using a shared lock in your program may result in improved latency and throughput, when compared to using a separate lock.

- **Failover and recovery**

These LAPI functions are supported on HPS and pSeries HPS systems that have multiple adapters per node. All of the nodes in the system must be configured as part of a single RSCT peer domain. Using LoadLeveler[®] command file settings or POE environment variables, you can request that LAPI use more than one adapter for each of the job tasks. For jobs that are run using this configuration, if one of the adapters allocated to the tasks of a job fails during the course of the job run, the job will continue and LAPI will use the available adapters for communication. If the failed adapter recovers the ability to communicate while the job is running, LAPI recovers the use of this failed adapter for future communication during the remainder of the job run.

- **Striping**

LAPI can manage the distribution of bulk transfer data among the various communication adapters that are assigned for communication, thereby providing LAPI clients with improved performance with regard to communication bandwidth. By using striping in conjunction with the bulk transfer transport mechanism, LAPI clients can experience gains in communication performance that scale linearly with the number of adapters (up to a limit of 8) for sufficiently-large messages. Messages that do not use the bulk transfer communication mode cannot benefit practically from striping over multiple adapters, so LAPI uses only one of the assigned adapters for communication when bulk transfer is turned off with environment variable settings or for small messages, which do not use bulk transfer. Even when striping is not done, the other assigned adapters serve as backups for LAPI's failover and recovery function.

- **Standalone operation**

You can use LAPI with or without the parallel operating environment (POE) component of the IBM Parallel Environment for AIX 5L (PE) licensed program. LAPI is referred to as operating in *standalone mode* if you use it without PE. In certain situations, such as failure recovery, you will have greater flexibility if you

use LAPI in standalone mode, as opposed to the enhanced usability that comes with using LAPI in conjunction with POE.

Why use LAPI?

LAPI provides the following advantages over other messaging layers:

- **Performance**

LAPI provides basic functions for optimal performance. In particular, LAPI provides *low latency* on short messages using the user space (US) protocol and high bandwidth on large messages.

- **Flexibility**

LAPI's one-sided communication model provides flexibility because the completion of an operation by one task does not require any other task to take a complementary action. This model is supported by the use of *virtual addressing*. Addresses in the target task address space are passed to LAPI communication calls by the origin task. These addresses can refer to remote data buffers, handlers, or counters. The use of target addresses allows data to be delivered without any need for explicit action by the target task.

Two-sided communication can be simulated using LAPI's communication calls in conjunction with various event-monitoring routines (such as waiting for counter values or execution of specified handlers, for example).

RSCT LAPI provides a lower-level interface to an HPS or a pSeries HPS (as does PSSP LAPI to an SP Switch2) than either the Message Passing Interface (MPI) or the Internet Protocol (IP), so you can choose how much additional communication protocol needs to be added.

With the addition of LAPI messaging support over UDP/IP, applications that are written using LAPI can be executed over any cluster of processors running AIX 5.2 or 5.3.

- **Reliability**

Using LAPI guarantees delivery of messages. Errors that are not directly related to the application are not propagated back to the application.

- **Availability**

In systems with multi-adapter nodes, if jobs are launched with settings to request use of multiple adapters within the job tasks, when an adapter fails, LAPI switches communication over to another, available adapter. If the original adapter regains the use of the connection during the course of the job run, LAPI recovers the use of this adapter for communication for the remainder of the job run. In addition, checkpoint and restart operations continue to be supported during these failures and recoveries.

- **Extendibility**

LAPI supports programmer-defined handlers that are called when a message arrives, so you can customize LAPI for your specific environment. For example, you can write completion handler functions to communicate data that helps maintain user program state. With such handlers, you can, in effect, extend LAPI's active message functionality to do your specific state update operations for you, in addition to the handlers' normal function of transferring data.

Chapter 2. An overview of LAPI

This chapter provides a functional overview of the various subroutines that constitute the low-level application programming interface. LAPI subroutines provide a wide variety of functions that can be used efficiently and flexibly to obtain most behaviors required from any parallel programming API.

In general, LAPI functions:

- Are non-blocking calls.
- Provide polling mode and interrupt mode.
- Indicate the completion of a message or operation either by incrementing counters or by running user-specified handlers. Counters and handlers are available at both the sending and receiving side, depending on the API call.
- Provide C and FORTRAN subroutine bindings.
- Provide **extern "C"** declarations for C++ programming.
- Provide profiling interfaces for C, C++, and FORTRAN programs.
- Do not guarantee order of message delivery.

Complementary functions provide for checking completion of operations and for enforcing relative ordering if required. Additionally, LAPI functions allow tasks to exchange addresses that will be used in LAPI operations.

LAPI functions (and related subroutines) include:

- Functions to initialize and terminate LAPI (**LAPI_Init**, **LAPI_Term**)
- Functions to query and set up the runtime environment (**LAPI_Qenv**, **LAPI_Senv**)
- Address-related functions (**LAPI_Address**, **LAPI_Address_init**, **LAPI_Address_init64**, **LAPI_Addr_get**, **LAPI_Addr_set**)
The **LAPI_Address_init64** subroutine treats all data as 64-bit values, so it can support communication between 32-bit and 64-bit tasks.
- Put and get functions (**LAPI_Put**, **LAPI_Get**, **LAPI_Xfer**)
- Active message functions (**LAPI_Amsend**, **LAPI_Xfer**)
- Non-contiguous data transfer functions (**LAPI_Amsendv**, **LAPI_Getv**, **LAPI_Putv**, **LAPI_Xfer**)
- Remote read-modify-write functions (**LAPI_Rmw**, **LAPI_Rmw64**, **LAPI_Xfer**)
The **LAPI_Rmw64** subroutine treats all data as 64-bit values, so it can support communication between 32-bit and 64-bit tasks.
- A wrapper function for all generic data transfer calls that includes support for 32-bit/64-bit interoperability and an optional send completion handler (**LAPI_Xfer**)
- Progress-monitoring functions (**LAPI_Getcntr**, **LAPI_Msgpoll**, **LAPI_Probe**, **LAPI_Setcntr**, **LAPI_Waitcntr**)
- Message ordering functions (**LAPI_Fence**, **LAPI_Gfence**)
- A wrapper function that provides additional utilities (**LAPI_Util**)
- Error message functions (**LAPI_Msg_string**)
- Recovery-related functions for standalone systems (**LAPI_Nopoll_wait**, **LAPI_Purge_totask**, **LAPI_Resume_totask**, **LAPI_Setcntr_wstatus**)

These functions are explained in more detail in the following sections.

Initialization and termination

LAPI uses a number of internal structures to enable it to perform message-passing operations on behalf of the user. Memory must be allocated for these internal structures, and the structures must be appropriately initialized before any LAPI communication is performed. Correspondingly, when all LAPI communication is done, memory used by LAPI structures must be freed and potentially reused by the user program.

The **LAPI_Init** subroutine is used to allocate memory for LAPI's communication structures and to initialize them. It returns a unique handle that represents a single LAPI communication context. This handle is subsequently passed as a parameter to each of the other LAPI functions. **LAPI_Init** takes in a parameter of type **lapi_info_t**. The fields in this structure are used to specify various initialization parameters. **LAPI_Init** reads in various environment variables and sets up various communication channels based on the values of these variables. For example, the user can set environment variables to indicate whether communication will take place using the user space (US) protocol or the user datagram protocol (UDP) and whether to use shared memory.

The **LAPI_Term** subroutine is used to free memory associated with LAPI's communication structures. It takes a LAPI handle as a parameter and uses it to terminate the corresponding communication context. Once **LAPI_Term** is called, no further LAPI communication can be performed on the handle that has been terminated. Typically, **LAPI_Init** is called once at the beginning of the user program and **LAPI_Term** is called just before the user program terminates. However, LAPI allows a handle to be initialized after it has been terminated.

Querying and setting up the runtime environment

A number of variables constitute LAPI's runtime state. Many of these variables can be queried at runtime. For example, it is often useful (if not always required) to know the number of tasks in a given job as well as the identity of the current task and to design the user program to take actions according to their values. Many of LAPI's runtime state variables can also be set to alter LAPI's behavior through the job execution and to tune LAPI's performance. For example, it may be useful to turn off interrupts to signal incoming packets when the user program explicitly makes a number of calls to various LAPI progress routines.

The **LAPI_Qenv** subroutine is used to query elements of LAPI's runtime state. The **LAPI_Senv** subroutine correspondingly allows the programmer to set the value of various elements of LAPI's runtime state. LAPI defines an enumeration of query types (**lapi_query_t**). A parameter of this type is passed to **LAPI_Qenv** or **LAPI_Senv** to indicate the value to query or set, respectively.

Address-related functions

Many of LAPI's communication operations take advantage of virtual addresses in the remote task's address space. An address might refer to a buffer location or a handler function on the target task. For example, LAPI's active message functions are passed the address of a header handler that executes on the target task upon the arrival of the first packet of a message.

The **LAPI_Address_init** collective operation allows tasks to exchange virtual addresses of mutual interest. Such a function is especially required in the typical

scenario where tasks do not have identical address maps. The **LAPI_Address_init** function takes in a virtual address and the address of a local buffer that is large enough to hold one address from each of the tasks in the job. When the function returns, the local buffer has virtual addresses from each of the tasks, at indices that correspond to each task's ID.

The **LAPI_Address_init64** collective operation is similar to **LAPI_Address_init**, but handles 64-bit operand addresses and 32-bit operand addresses.

LAPI also provides operations to register and retrieve active message header handlers as specific indices (referred to as *address handles*) in a table of handler functions that is maintained by LAPI. These user-defined address handles can be used instead of the virtual addresses of the header handler functions. The **LAPI_Addr_set** subroutine registers a header handler function at the index passed in by the caller. The **LAPI_Addr_get** subroutine retrieves the address of the header handler function stored at a given index. If all tasks of the job are programmed to register and use the same indices for specific header handler functions, a collective **LAPI_Address_init** call to set up the address table can be avoided

FORTRAN programmers:

Because there is no concept of address (&) in FORTRAN, LAPI provides the **LAPI_Address** subroutine, which FORTRAN programs can call when a value needs to be treated as an address.

Put and get functions

LAPI includes subroutines that provide the user with a Remote Memory Copy interface, allowing the direct transfer of contiguous data to or from the virtual address of a remote task. These subroutines support "pull" and "push" operations. The **LAPI_Get** subroutine copies (or "pulls") data from the address space of a target task into the address space of the origin task. The **LAPI_Put** subroutine copies (or "pushes") data into the address space of a target task from the address space of the origin task. Other primary characteristics of the **LAPI_Put** and **LAPI_Get** functions include the following:

- The basic data transfer operations are memory-to-memory copy operations that transfer data from one virtual address space to another virtual address space.
- The operations are unilateral. One task initiates an operation, but the completion of the operation does not require the other task to take some complementary action. (This model is different from the more common send-receive communication model, in which a send operation from one task requires a complementary receive operation with matching parameters at the target task to be posted for completion.)
- The initiating task specifies the virtual address of the source of the data and the virtual address of the destination of the data (as opposed to a send and receive task, in which each side specifies the address in its own address space). As described in "Address-related functions" on page 10, LAPI provides the **LAPI_Address_init** and **LAPI_Address_init64** subroutines to obtain virtual addresses in a remote task's address space to or from which data needs to be copied. These virtual addresses must be obtained before making the **LAPI_Put** and **LAPI_Get** calls.

- Because data transfer operations are unilateral and no synchronization between the two tasks is implied, additional primitives are provided for explicit task synchronization when such synchronization is necessary for program correctness.

Active messages

In many cases, it is impractical to exchange virtual addresses for all communication performed during a job. Instead, it is useful to decide on the buffer address to which incoming data must be stored when the data arrives at the target, based on information about the state of the target task. Such a decision would be taken, for example, by executing a function in the target task's address space that examines data in the incoming message and the state of the target task to decide where the data must be stored. It is also similarly useful to be able to define and execute a user-specified procedure at the target task as soon as a message is completely transferred into the target task's address space. Such a procedure can be used, for example, to automatically process the message data at the target and to send any necessary reply messages.

LAPI's active message subroutine, **LAPI_Amsend**, provides users with this ability to calculate the target buffer address at the destination by allowing the caller to specify the address of a header handler function in the target task's address space as one of its parameters. The header handler function is executed at the target as soon as the first packet of a message arrives and returns the address of the target task buffer where the message data must be stored. The header handler function may also change reference parameters passed into it to specify a completion handler function and a parameter that should be passed to it. The completion handler function, if specified, will be executed by LAPI as soon as the entire message is transferred into the address returned by the header handler. The **LAPI_Amsend** subroutine transfers contiguous messages only.

LAPI's underlying *active message* infrastructure has the following characteristics:

- It includes the address of a user-specified handler. When the active message arrives at the target task, the specified handler is called. This handler runs in the address space of the target task.
- Optionally, the active message might also bring with it user header data from the originating task that could aid in the computation of the target buffer address by the header handler function, or in the future processing of actual message data.
- Operations are unilateral in the sense that the LAPI client on the target task does not have to take explicit action for the active message to complete.
- The handler that is called must return the address of storage buffers for arriving data.
- If the **LAPI_Addr_set** subroutine (described in "Address-related functions" on page 10) is used to register the address of a header handler at a specific address handle, that address handle may be passed in to **LAPI_Amsend** in place of the header handler address.

Non-contiguous data transfer

The **LAPI_Put** and **LAPI_Get** subroutines (described in "Put and get functions" on page 11) and the **LAPI_Amsend** subroutine (described in "Active messages") can be used to transfer data that is laid out in *contiguous* portions of memory. Very often, though, data that is laid out in *non-contiguous* memory locations may need to be transferred between tasks. Such non-contiguous data is often referred to as a *vector*. Although it is possible to transfer a non-contiguous vector through multiple

LAPI_Put, **LAPI_Get**, or **LAPI_Amsend** calls, it is much more efficient to simply specify the layout of the data at the source or target task and make a single LAPI call that transfers data between the specified non-contiguous regions. At a minimum, having such a call reduces the locking overhead associated with every LAPI call that is incurred to maintain the integrity of LAPI's internal data structures.

LAPI provides a **lapi_vec_t** datatype that can be used to describe the layout of non-contiguous (vector) data and three subroutines (**LAPI_Putv**, **LAPI_Getv**, and **LAPI_Amsendv**) to transfer this data in a similar manner to **LAPI_Put**, **LAPI_Get**, and **LAPI_Amsend**. **LAPI_Putv** transfers vector data from the calling task to a target task, **LAPI_Getv** transfers vector data from a target task to the caller task, and **LAPI_Amsendv** specifies a header handler function that is executed at the message target to obtain a vector description of where data must be stored at the target. Descriptions of the source and target vectors, as pointers to the **lapi_vec_t** types, are passed in to the **LAPI_Putv** and **LAPI_Getv** subroutines. The **LAPI_Amsendv** subroutine only requires the source vector description. A target vector description is returned by the header handler in a manner similar to the way a single buffer is returned by the header handler in a **LAPI_Amsend** call. The vector description with the **lapi_vec_t** structure includes information that directly or indirectly indicates the starting buffer addresses and lengths of each of the non-contiguous source and target regions. LAPI does not require that non-contiguous data at the source and destination be laid out in identical fashion.

Remote read-modify-write functions

Multi-threaded, single-process programs often use atomic read-modify-write (RMW) operations to check and change the values of shared variables from multiple threads, thereby avoiding possible inconsistencies due to the effects of caching in the underlying architecture. To avoid more significant locking overheads, RMW operations are often used to synchronize among threads. With the **LAPI_Rmw** subroutine, a calling task in a parallel, multi-process program can atomically check and change the value of a remote task's variable. For example, you can use **LAPI_Rmw** to synchronize a pair of tasks in a job based on the value of a "shared" variable that is located on one of the tasks. **LAPI_Rmw** provides you with an alternative to using other LAPI calls that facilitate synchronization among all of the tasks in a job.

With **LAPI_Rmw**, the operation is performed at the target task on the remote address that is specified by the target variable. The operation takes in one or more input values from the origin and performs one of four selected operations on a variable from the target task using these origin task input values. Depending on the result of this operation, LAPI replaces the target task variable with a value dictated by the result of the operation. The previous value of the target variable is then returned to the origin by modifying an origin task variable. This variable's address is passed in as one of the parameters to the **LAPI_Rmw** subroutine. The four different read-modify-write operations are as follows:

1. **SWAP** replaces the target task variable with the value specified by the calling task and returns the original value of the target task variable by setting the specified origin task variable.
2. **COMPARE_AND_SWAP** compares an input value with the value of the target variable, and if they are equal, replaces the target task variable with another specified input value.

3. **FETCH_AND_ADD** adds the specified input value to the target task variable and stores the result in the target task variable. **FETCH_AND_ADD** returns the original value of the target task variable (before performing the addition) by setting the specified origin task variable.
4. **FETCH_AND_OR** does a bit-wise OR of the specified input value with the value in the target task variable and stores the result in the target task variable. Like **FETCH_AND_ADD**, **FETCH_AND_OR** also returns the original value of the target task variable.

Completion is signaled at the origin if an origin counter is specified with the operation. It is important to note that **LAPI_Rmw** operations are atomic with respect to the target, but not necessarily with respect to the origin. That is, two successive **LAPI_Rmw** calls from a task to the same origin variable at a target task may be executed in any order at the target task, with the only guarantee that each will execute atomically at the target.

The **LAPI_Rmw64** subroutine provides the same function as **LAPI_Rmw** for a 64-bit data structure.

Generic data transfer functions

The contiguous and non-contiguous LAPI subroutines described in the previous sections have some limitations. These limitations include: (1) there is no option to specify a send completion handler that can be executed when the source buffer is reusable, so the sending side must use counters to determine origin buffer availability, (2) there is no support for 32-bit and 64-bit interoperability, and (3) there is no support for flags that can be used to direct transfer-specific behavior.

These limitations are resolved by the **LAPI_Xfer** subroutine. This subroutine takes a LAPI handle and a pointer to union of structures as arguments. Each structure in the union corresponds to one of the data transfer functions described in the previous sections, with the first element of the structure specifying the type of transfer, such as **Put**, **Get**, or **Am**. Each structure contains fields that correspond to the parameters in the corresponding LAPI data transfer call (for example, the structure for the **Put** transfer has fields for each of the parameters in **LAPI_Put**), with type modifications and additional fields to achieve the enhanced functionality mentioned above. For example, each structure has a field for an optional send completion handler (or, in the case of **Get**, for a receive completion handler), and a **flags** field to control specific behavior relating to the data transfer. In addition, all address parameters that belong to a remote task (such as the target counter address) are specified as a new **lapi_long_t** type that allows remote 64-bit addresses to be represented on 32-bit tasks. There is a **LAPI_Xfer** call corresponding to each of the LAPI data transfer calls described in the previous sections, with the **LAPI_Xfer** call supporting a functional superset of the corresponding LAPI data transfer call. Data transfer between 32-bit and 64-bit tasks must use **LAPI_Xfer**.

The **LAPI_Xfer** subroutine provides a transfer type for non-contiguous messages that has no corresponding LAPI data transfer call. This **LAPI_Xfer** call is somewhat analogous to the **LAPI_Amsendv** call with one major difference: the layout of non-contiguous data is described for this transfer type using commands of a simple, assembly-type language. Such a description of the layout of non-contiguous data is called a *data gather/scatter program (DGSP)*. A DGSP description of non-contiguous data can be used to capture more powerful datatypes than can be captured by LAPI's vector description structures alone (for example, recursive datatypes can be more easily described with a DGSP). Before it is used, a DGSP

must be registered with the **LAPI_Util** subroutine. **LAPI_Util** returns a DGSP handle that can be used to refer to the corresponding DGSP. The **LAPI_Xfer** call to transfer non-contiguous data using DGSP takes in a DGSP handle describing the source data layout as input and transfers data to the destination by running the corresponding DGSP. At the destination, the data can be written either contiguously or written in a manner described by a different DGSP. A DGSP handle describing the data layout at the target must be returned through one of the parameters of an enhanced (but binary-compatible) header handler interface (see “The header handler” on page 48 for more information). This DGSP is then run at the target to store the non-contiguous message data into appropriate target buffers.

For more information, see “LAPI_Xfer” on page 219.

Progress-monitoring functions

As discussed in Chapter 1, progress operations are necessitated, when in polling mode, by the asynchronous nature of LAPI communication, and are provided by LAPI to drive pending communication operations to completion. These functions are used to explicitly invoke an internal LAPI communication dispatcher. The dispatcher will ensure that progress is made on any pending send or receive operations. If LAPI is running in interrupt mode, the progress operations are not strictly necessary.

The **LAPI_Probe** subroutine is used in polling mode to explicitly invoke LAPI’s internal communication dispatcher once. This subroutine is typically called in a loop that also checks for the occurrence of some completion event (for example, an update to a LAPI counter value or an internal variable updated by the execution of a handler function).

A LAPI user may want to monitor a number of events, such as availability of the source buffer, availability of the target buffer, or completion of message transmission. LAPI provides the origin counter, target counter, and completion counter, respectively, to monitor each of these three events. LAPI also provides methods to set and monitor the values of counters. In addition, LAPI provides users with the ability to specify a send completion handler that is executed at the source when the source buffer is reusable and a (receive) completion handler that is executed at the target when message data has completely arrived at the target buffer. (As discussed in “Generic data transfer functions” on page 14, send completion handlers are available only with the **LAPI_Xfer** calls.)

The following functions provide the means for a task to manage the completion state of LAPI operations using LAPI counters:

- **LAPI_Waitcntr** waits on a counter to reach a specified value and returns when the counter is equal to or greater than that value (blocking). This subroutine implicitly drives LAPI progress until the counter reaches the awaited value.
- **LAPI_Getcntr** gets the current value of a specified counter (non-blocking). This subroutine is typically used in conjunction with **LAPI_Probe** to drive progress while polling for counter values.
- **LAPI_Setcntr** sets the counter to a specified value.
- **LAPI_Nopoll_wait** waits for a counter update without polling (that is, without explicitly invoking LAPI’s internal communication dispatcher).

These subroutines also provide an efficient way to order the flow of LAPI operations or the use of such user-managed resources as buffers. For example, a series of **LAPI_Put** calls to a single target and buffer requires that the contents of the buffer

at the target remains in step with the order of execution of the **LAPI_Put** calls at the origin. Using the *cmpl_cntr* counter in the **LAPI_Put** subroutine in conjunction with the **LAPI_Waitcntr** subroutine provides the necessary ordering. The use of (receive) completion and send completion handlers allows a client to rely exclusively on asynchronous notification rather than polling for message completion. The send completion handler does not guarantee that data has arrived at the target; however, it does guarantee that the user header buffers (if any) and data buffers can be reused. The receive completion handler is called at the target after the data transfer has completed.

The **LAPI_Msgpoll** subroutine is provided as a mechanism to probe continuously for send or receive completions. It polls for a fixed number of iterations or until some message has completed. The number of iterations for polling and the type (or types) of completion — send, receive, or both — that the subroutine must monitor are specified as parameters to the subroutine. **LAPI_Msgpoll** is especially useful when multiple messages are outstanding and the client is waiting for any one of them to complete.

Message ordering functions

Because LAPI does not guarantee message order by default, it provides functions to enforce ordering among messages. The **LAPI_Fence** and **LAPI_Gfence** subroutines provide a way to enforce the order of execution of LAPI data transfer subroutines. LAPI subroutines that are initiated before these fencing operations are guaranteed to complete before LAPI functions initiated after the fencing functions. **LAPI_Fence** is a local operation used to guarantee that all LAPI operations initiated by the local task and the same task thread are complete. In contrast, **LAPI_Gfence** is a collective operation involving all tasks in the parallel program. A **LAPI_Gfence** operation combines local fencing on all tasks of the job with a barrier operation to synchronize the tasks in the parallel program. Both **LAPI_Fence** and **LAPI_Gfence** operations perform a data fence, only guaranteeing that data movement is complete. They do not perform an operation fence, which would need to include the completed execution of any pending active message completion handlers at the target.

Utility functions

The **LAPI_Util** subroutine serves as a wrapper function for such data gather/scatter operations as registration and reservation, and for obtaining pointers to locking and signaling functions that are associated with a shared LAPI lock.

Error message functions

The **LAPI_Msg_string** subroutine returns the message string that is associated with a LAPI return code.

Recovery-related functions

The following recovery-related functions are recommended for use on standalone systems only:

- **LAPI_Nopoll_wait** waits for a counter update without polling (that is, without explicitly invoking LAPI's internal communication dispatcher).
- **LAPI_Purge_totask** allows a task to cancel messages to a given destination.
- **LAPI_Resume_totask** re-enables the sending of messages to the task.

- **LAPI_Setcntr_wstatus** sets an associated destination list array and destination status array to a counter. A corresponding **LAPI_Nopoll_wait** call accesses these arrays.

Chapter 3. What's new in LAPI?

LAPI includes several enhancements. To take advantage of some of these enhancements, you might need to make changes to your existing LAPI programs (see “Tips for LAPI users” on page 21).

Major changes and additions to LAPI include:

Table 3. Changes in this edition

Change or addition	For more information, see:
For RSCT LAPI users, support for the IBM @server High Performance Switch (HPS) for IBM @server p5 servers. To use the HPS, you need to have version 1.4.1 (or later) of Cluster Systems Management (CSM) for AIX 5L (product number 5765-F67) installed on your system.	@server <i>Cluster 1600: High Performance Switch Planning, Installation, and Service for IBM eServer™ p5 Servers</i> <i>Cluster Systems Management for AIX 5L and Linux: Planning and Installation Guide</i>
For RSCT LAPI users, striping support.	“Data striping” on page 106
For RSCT LAPI users, bulk transfer of messages using the remote direct memory access (RDMA) protocol. This mechanism can only be used in conjunction with the HPS or the pSeries HPS. This support includes two new runtime attributes that let LAPI clients get information related to bulk transfer (BULK_XFER and BULK_MIN_MSG_SIZE).	Chapter 13, “Bulk transfer of messages,” on page 99
A new profiling interface.	Chapter 9, “Using LAPI’s profiling interface,” on page 65
Two additional runtime attributes that let LAPI clients get information about new statistics (QUERY_LOCAL_SEND_STATISTICS and QUERY_SHM_STATISTICS)	Chapter 8, “Collecting statistics,” on page 63
The maximum number of shared memory tasks per operating system image has changed from 32 to 128.	“LAPI shared memory: requirements and restrictions” on page 259
Support for 8192 user space (US) MPI tasks in a single job.	

Table 4. Changes in the second edition

Change or addition	For more information, see:
For RSCT LAPI users, failover and recovery support.	“Using failover and recovery” on page 103
Lock sharing support, which includes a new LAPI utility for shared locking and signalling functions.	<ul style="list-style-type: none"> Chapter 12, “Lock sharing,” on page 83 “LAPI_GET_THREAD_FUNC” on page 208
Environment variable changes. New: MP_INSTANCES , MP_LAPI_INET_ADDR , MP_RETRANSMIT_INTERVAL , MP_REXMIT_BUF_CNT , MP_REXMIT_BUF_SIZE . Deprecated: MP_COPY_SEND_BUF_SIZE .	“Variables for performance tuning” on page 270

Table 5. Changes in the first edition

Change or addition	For more information, see:
<p>For RSCT LAPI users, support for the IBM @server pSeries High Performance Switch (pSeries HPS).</p> <p>To use the pSeries HPS, you need to have version 1.3.2 (or later) of Cluster Systems Management (CSM) for AIX 5L (product number 5765-F67) installed on your system.</p>	<p>@server <i>Cluster 1600: pSeries High Performance Switch Planning, Installation, and Service</i></p> <p><i>Cluster Systems Management for AIX 5L and Linux: Planning and Installation Guide</i></p>
<p>The ability to run over the user datagram protocol (UDP) rather than the user space (US) protocol.</p>	<p>“LAPI communication modes” on page 30</p>
<p>Inline completion handlers, which allow your completion handler to be called from within the thread of execution that completed the data transfer. For applications that rely on completion handlers rather than counters, this can provide a significant performance enhancement.</p>	<p>“Inline completion handlers” on page 77</p>
<p>A generalized mechanism to transfer arbitrary portions of non-contiguous data using <i>data gather/scatter programs (DGSPs)</i>.</p>	<p>“Using data gather/scatter programs (DGSPs)” on page 43</p>
<p>Runtime attributes that let LAPI clients control the communication library and get information about new statistics (PRINT_STATISTICS and QUERY_STATISTICS)</p>	<p>“LAPI_Qenv” on page 184 and “Attributes that return multiple values” on page 277</p>
<p>A polling function that provides more flexible access to invoking the LAPI dispatcher explicitly and lets you poll for messages without polling for a specific counter.</p>	<p>“LAPI_Msgpoll” on page 171</p>
<p>Support for multi-threaded programs. LAPI calls can be made from multiple user threads.</p>	<p>Chapter 15, “Threaded programming,” on page 111</p>
<p>For RSCT LAPI users, an efficient bulk transfer mechanism for large data transfers. This mechanism can only be used in conjunction with the HPS or the pSeries HPS.</p>	<p>Chapter 13, “Bulk transfer of messages,” on page 99</p>
<p>An API call that performs various LAPI utility operations, most notably for user DGSPs.</p>	<p>“LAPI_Util” on page 203</p>
<p>A LAPI_Xfer transfer type to support DGSP transfers: LAPI_DGSP_XFER (DGSP).</p>	<p>“lapi_amdgspl_t details” on page 222</p>

Table 5. Changes in the first edition (continued)

Change or addition	For more information, see:
For AIX 5.2 and 5.3, several data structures.	<ul style="list-style-type: none"> • For extended header handler support: lapi_return_info_t (see “LAPI_Amsend” on page 136) • For message polling: lapi_msg_info_t (see “LAPI_Msgpoll” on page 171) • For IP/US statistics reporting: lapi_statistics_t (see “LAPI_Qenv” on page 184) • For UDP support: <ul style="list-style-type: none"> – lapi_add_udp_port_t (see “LAPI_Util” on page 203) – lapi_extend_t, lapi_udp_t, lapi_udpinfo_t (see “LAPI_Init” on page 163) • For utility functions and DGSM data transfers (see “LAPI_Util” on page 203): <ul style="list-style-type: none"> – ddm_func_t – lapi_add_udp_port_t – lapi_amdgsp_t – lapi_dg_handle_t – lapi_dgsm_block_t – lapi_dgsm_control_t – lapi_dgsm_copy_t – lapi_dgsm_gosub_t – lapi_dgsm_iterate_t – lapi_dgsm_mcopy_t – lapi_dgsp_descr_t – lapi_dref_dgsp_t – lapi_pack_dgsp_t – lapi_reg_ddm_t – lapi_reg_dgsp_t – lapi_resv_dgsp_t – lapi_unpack_dgsp_t – lapi_usr_fcall_t – lapi_util_t

Tips for LAPI users

RSCT LAPI and PSSP LAPI include almost all of the same features and functions. Here are the major differences:

Table 6. Differences between LAPI versions

RSCT LAPI	PSSP LAPI
Is part of the AIX operating system.	Is a component of the IBM Parallel System Support Programs licensed program.
Supports the IBM @server pSeries High Performance Switch (pSeries HPS) and the IBM @server High Performance Switch (HPS) for IBM @server p5 servers.	Supports the SP Switch2.
Provides support for bulk transfer of messages using the remote direct memory access (RDMA) protocol.	Does not provide this support.
Provides support for striping, failover, and recovery.	Does not provide this support.

The current version of LAPI maintains source and binary compatibility with previous versions of LAPI. Existing LAPI binaries will run under IBM's LoadLeveler for AIX 5L and Parallel Environment for AIX 5L (PE) licensed programs.

To take advantage of some of LAPI's latest features, you may need to make changes to your existing LAPI programs. For example: for the inline completion handler, the header handler needs to be modified to return more information to LAPI. See "The enhanced header handler interface" on page 75.

Your existing LAPI programs can also be run *standalone*. In this book, the term *standalone* refers to a system that is *not* running PE. See Chapter 16, "Using LAPI on a standalone system," on page 115.

For noncontiguous data transfers that are not easily described by the vector datatypes, *data gather/scatter programs (DGSPs)* may provide an alternative way to move data from one address space to another. See "Using data gather/scatter programs (DGSPs)" on page 43.

RSCT LAPI includes an updated set of error codes. See "LAPI error codes" on page 261.

Part 2. Basic LAPI tasks

Chapter 4. Installing RSCT LAPI	25
Requirements	25
Hardware	25
Software	25
How is RSCT LAPI packaged?	25
RSCT LAPI filesets	26
Installation steps	27
Uninstallation steps	27
Migration and coexistence	28
Chapter 5. Setting up, initializing, and terminating LAPI	29
Setting and querying the LAPI environment	29
Setting environment variables	29
LAPI communication modes	30
Initializing LAPI	31
Passing information to LAPI using <code>lapi_info_t</code>	31
Terminating LAPI	33
Chapter 6. Transferring data	35
Data transfer operations	35
Flow of "put" operations	35
Flow of "get" operations	36
Flow of read-modify-write operations	37
Non-contiguous data transfer	37
Using vectors	37
LAPI_GEN_GENERIC	39
LAPI_GEN_IOVECTOR	40
LAPI_GEN_STRIDED_XFER	41
Vector data transfer summary	42
Using data gather/scatter programs (DGSPs)	43
Detecting completion	47
LAPI handlers	47
The header handler	48
The completion handler	49
The send completion handler	51
LAPI handler summary	51
LAPI counters	52
Specifying target-side addresses	53
Additional progress functions	53
Chapter 7. Active messaging	55
Flow of active message operations	55
Using LAPI_Amsend: a complete LAPI program	56
Chapter 8. Collecting statistics	63
Printing data transfer statistics	63
Querying US and UDP/IP statistics	63
Querying local send statistics	63
Querying shared memory statistics	64
Chapter 9. Using LAPI's profiling interface	65
Performing name-shift profiling	66
A sample profiling program	68

I **Chapter 10. Compiling and running LAPI programs 71**

Chapter 4. Installing RSCT LAPI

This chapter includes information about installing the version of LAPI that is shipped as part of the Reliable Scalable Cluster Technology (RSCT) component of IBM's AIX 5.2 and 5.3 operating systems. For information about PSSP LAPI installation, see the *PSSP Read This First* document.

Requirements

To make use of all of RSCT LAPI's functions, you must make sure that the following hardware and software requirements are satisfied.

Hardware

In order to take advantage of all of RSCT LAPI's functions, you must be using the following hardware:

- An IBM @server pSeries server (655 or 690) with the pSeries High Performance Switch (pSeries HPS) or an IBM @server p5 server (575 or 595) with the High Performance Switch (HPS).
- A minimum of two links per logical partition (LPAR).

Software

To use all of RSCT LAPI's functions, you must have the following software installed on your system:

- The AIX 5L operating system — AIX 5L Version 5.3 (product number 5765-G03) or AIX 5L Version 5.2 with the 5200-06 Recommended Maintenance package (product number 5765-E62). In particular:
 - Version 1.1.1.0 with APAR IY59794 (or later) of the switch network interface (SNI) component.
 - For AIX 5.3: Version 2.4.0.0 (or later) of the group services subcomponent of the Reliable Scalable Cluster Technology (RSCT) component of AIX 5L.
For AIX 5.2: Version 2.3.3.2 (or later) of group services. The **rsct.basic.rte** fileset includes group services.
 - For AIX 5.3: Version 2.4.1.0 (or later) of the LAPI subcomponent of RSCT — specifically, the **rsct.lapi.rte** fileset for LAPI files and the **rsct.lapi.nam** fileset for Network Availability Matrix (NAM) files.
For AIX 5.2: Version 2.3.3.0 (or later) of LAPI.
- Version 1.3.2 or Version 1.4.1 (or later) of the Cluster Systems Management for AIX 5L licensed program (product number 5765-F67).
- Version 3.2.0.7 or Version 3.3 (or later) of the LoadLeveler for AIX 5L licensed program (product number 5765-E69).
- Version 4.1.1 or Version 4.2 (or later) of the Parallel Environment for AIX 5L licensed program (product number 5765-F83).

How is RSCT LAPI packaged?

The filesets for RSCT LAPI are packaged in a single **rsct.lapi** image. Most LAPI files are installed in the **/opt/rsct/lapi** directory. Exceptions are noted in “RSCT LAPI filesets” on page 26. Links are created within the **/usr** tree to such commonly-needed files as **liblapi_r.a**, **lapi.h**, and the 32-bit version of **lapif.h**.

RSCT LAPI filesets

RSCT LAPI consists of the following filesets:

rsct.lapi.nam Contains files for the Network Availability Matrix (NAM), which is associated with the HPS and the pSeries HPS. These files are required for LAPI users running over the HPS or the pSeries HPS who want to use LAPI's failover and recovery support. See "Using failover and recovery" on page 103 for more information.

The **rsct.lapi.rte** fileset is an installation prerequisite for this fileset.

rsct.lapi.rte Contains LAPI's runtime environment, including headers and libraries. It is a prerequisite for the other LAPI filesets.

The following files are installed from this fileset into **/opt/rsct/lapi**:

include/lapi.h LAPI's C header file, with a link in **/usr/include**

include/lapif.h

LAPI's 32-bit FORTRAN header file, with a link in **/usr/include**

include64/lapif.h

LAPI's 64-bit FORTRAN header file (no link)

lib/liblapi_r.a LAPI's library file, with a link in **/usr/lib**

libtrace/liblapi_r.a

LAPI's library file with tracing enabled

lib/lapisub.exp

Subroutine export file, with a link in **/usr/lib**

lib/lapisub64.exp

Subroutine export file (64-bit version), with a link in **/usr/lib**

dev/include/css_shared.h

Header file that contains communication subsystem (CSS) information (required by LAPI)

The following files are installed directly into the **/usr/lib** tree:

drivers/zcmem_ke

Contains the kernel services that LAPI uses for shared memory

methods/cfgzcmem

Loads the shared memory kernel extension (binary file). AIX is automatically updated so that the configuration is done at boot time.

methods/ucfgzcmem

Unloads the shared memory kernel extension (binary file). AIX is automatically updated so that the configuration is done at boot time.

These files provide LAPI's message catalog for AIX 5.2:

- **nls/msg/C/liblapi.cat**
- **nls/msg/en_US/liblapi.cat**
- **nls/msg/En_US/liblapi.cat**

These files provide LAPI's message catalog for AIX 5.3:

- **nls/msg/C/liblapi.cat**
- **nls/msg/ca_ES/liblapi.cat**
- **nls/msg/cs_CZ/liblapi.cat**
- **nls/msg/de_DE/liblapi.cat**
- **nls/msg/en_US/liblapi.cat**
- **nls/msg/En_US/liblapi.cat**
- **nls/msg/es_ES/liblapi.cat**
- **nls/msg/fr_FR/liblapi.cat**
- **nls/msg/hu_HU/liblapi.cat**
- **nls/msg/it_IT/liblapi.cat**
- **nls/msg/ja_JP/liblapi.cat**
- **nls/msg/ko_KR/liblapi.cat**
- **nls/msg/pl_PL/liblapi.cat**
- **nls/msg/pt_BR/liblapi.cat**
- **nls/msg/ru_RU/liblapi.cat**
- **nls/msg/sk_SK/liblapi.cat**
- **nls/msg/zh_CN/liblapi.cat**
- **nls/msg/zh_TW/liblapi.cat**

rsct.lapi.samp

Contains LAPI's sample files. These files are not required, but are recommended. LAPI has an extensive set of sample files that demonstrate various aspects of the API, including new features. See Chapter 20, "LAPI sample programs," on page 245 for more information.

Installation steps

1. If you are on a system that is running the IBM Parallel System Support Programs (PSSP) 3.5 licensed program and you want to use RSCT LAPI, you *must* uninstall the **ssp.css.lapi** fileset before you install the **rsct.lapi.rte** fileset. Use the AIX **installp** command to uninstall **ssp.css.lapi**:

```
installp -u ssp.css.lapi
```
2. Use the AIX **installp** command to install the LAPI filesets. For example, to install all of the LAPI filesets, enter:

```
installp -a -d rsct.lapi all
```
3. If you plan to use the **rsct.lapi.nam** fileset for failover and recovery, perform this step:
After the installation is finished, you need to reboot the system.

See *AIX 5L Version 5.2 Commands Reference* or *AIX 5L Version 5.3 Commands Reference* for more information on **installp**.

Uninstallation steps

1. Perform this step if you have the **rsct.lapi.nam** fileset installed on your system.
 - a. If you are running RSCT group services, use the **stopsrc** command to stop **cthagsglsm**:

```
stopsrc -s cthagsglsm
```

See *AIX 5L Version 5.2 Commands Reference* or *AIX 5L Version 5.3 Commands Reference* for more information on **stopsrc**.

- b. Run the following command to remove the NAM pseudo-device (**nampd0**):

```
/usr/bin/rmdev -d -l nampd0
```

For more information about NAM, see “Network Availability Matrix (NAM) overview” on page 103. For more information about **rmdev**, see *AIX 5L Version 5.2 Commands Reference* or *AIX 5L Version 5.3 Commands Reference*.

2. Use the **installp** command to uninstall the LAPI filesets. For example, to uninstall all of the LAPI filesets, enter:

```
installp -u rsct.lapi
```

See *AIX 5L Version 5.2 Commands Reference* or *AIX 5L Version 5.3 Commands Reference* for more information on **installp**.

Migration and coexistence

Parallel application programs that use LAPI must use the identical level of software.

For information about the binary compatibility issues of 32-bit applications that use striping, see “Communication and memory considerations” on page 108.

Descriptions and formats of the **MP_COMMON_TASKS**, **MP_LAPI_INET_ADDR**, **MP_LAPI_NETWORK** environment variables are provided in this book for informational purposes only. These environment variables are not intended to be used as external programming interfaces. IBM will not guarantee that the formats or values of these variables can continue to be used without change in future releases. Programmers and users who choose to develop applications that depend on these variables do so with the understanding that these variables may be subject to future change. IBM cannot guarantee that such applications can migrate or coexist with future releases without additional changes, nor will IBM ensure that there will be binary compatibility of these variables.

Chapter 5. Setting up, initializing, and terminating LAPI

This chapter explains how to set up, initialize, and terminate LAPI on systems running IBM's Parallel Environment for AIX 5L (PE) licensed program and on standalone systems (those not running PE). For specific information that applies only to the use of LAPI in standalone mode, see Chapter 16, "Using LAPI on a standalone system," on page 115.

Setting and querying the LAPI environment

LAPI allows users to configure the LAPI environment using two different types of variables: environment variables and runtime attributes.

Environment variables are set before job initialization. See "Environment variables" on page 269 for more information.

Runtime attributes can be retrieved using the **LAPI_Qenv** subroutine and set using the **LAPI_Senv** subroutine at runtime. See "LAPI_Qenv" on page 184, "LAPI_Senv" on page 196, and "Runtime attributes" on page 275 for more information.

Setting environment variables

You *must* set the following environment variable before LAPI is initialized:

```
MP_MSG_API=[ lapi | [ lapi,mpi | mpi,lapi ] mpi_lapi ]
```

In a LAPI environment, the valid settings for **MP_MSG_API** are:

- lapi** Sets up LAPI communication using an exclusive adapter window. Tasks can communicate using only LAPI calls.
- lapi,mpi** or **mpi,lapi** Sets up LAPI and MPI communication in the same job using separate adapter windows. Tasks can communicate using either LAPI calls or MPI calls.
- mpi_lapi** Sets up MPI and LAPI communication in the same job using a shared adapter window. This is a special setting that allows LAPI and MPI tasks to communicate using a shared handle and common adapter window. Currently, this shared handle is only supported for use by MPI.

For users running POE, the default setting for this environment variable is **mpi**. This setting applies to MPI communication only; it does not apply to LAPI communication. If you are using LAPI, you cannot use the default setting, as LAPI will not initialize if **MP_MSG_API=mpi** is set. You must set this environment variable explicitly to one of the settings that allows for LAPI communication: **lapi**, **lapi,mpi**, **mpi,lapi**, or **mpi_lapi**. See "Variables for communication" on page 269 for more information.

The following environment variables are also commonly used:

```
MP_EUILIB=[ ip | us ] (ip is the default)
```

```
MP_PROCS=number_of_tasks_in_job
```

```
LAPI_USE_SHM=[ yes | no | only ] (no is the default)
```

See “Environment variables” on page 269 for more information.

LAPI communication modes

LAPI communication takes place in one of the following modes:

- User space (US) over the HPS or the pSeries HPS (for RSCT LAPI users).
- User Datagram Protocol / Internet Protocol (UDP/IP) over the HPS or the pSeries HPS (for RSCT LAPI users) or any other device that supports IP communication (for all LAPI users).
- Shared memory, for tasks that are running on the same node (for all LAPI users).
This document refers to such tasks as *common tasks*.

For a given job, US and UDP/IP communication are mutually exclusive.

It is also possible to combine shared memory mode with one of the other two modes. There are a number of possibilities:

1. If **LAPI_USE_SHM=no** (or is not set, as **no** is the default) and:
 - a. **MP_EUILIB=ip** (or is not set, as **ip** is the default), LAPI sets up all tasks to communicate using UDP/IP.
 - b. **MP_EUILIB=us**, LAPI sets up all tasks to communicate using the user space (US) protocol.
2. If **LAPI_USE_SHM=yes**, an attempt is made to initialize shared memory among all common tasks. In addition, LAPI sets up communication based on the setting of **MP_EUILIB** as described in 1a and 1b above. This is the mode of communication for any tasks:
 - that are not on the same node
 - for which shared memory setup fails
3. If **LAPI_USE_SHM=only** and:
 - a. All tasks are common, LAPI attempts to set up shared memory communication among all tasks. If this attempt fails, **LAPI_Init** returns an error.
 - b. All tasks are not common (the job is spread across multiple nodes), initialization fails and **LAPI_Init** returns an error.

Figure 1 on page 31 illustrates LAPI’s sequence of events for setting up the mode of communication.

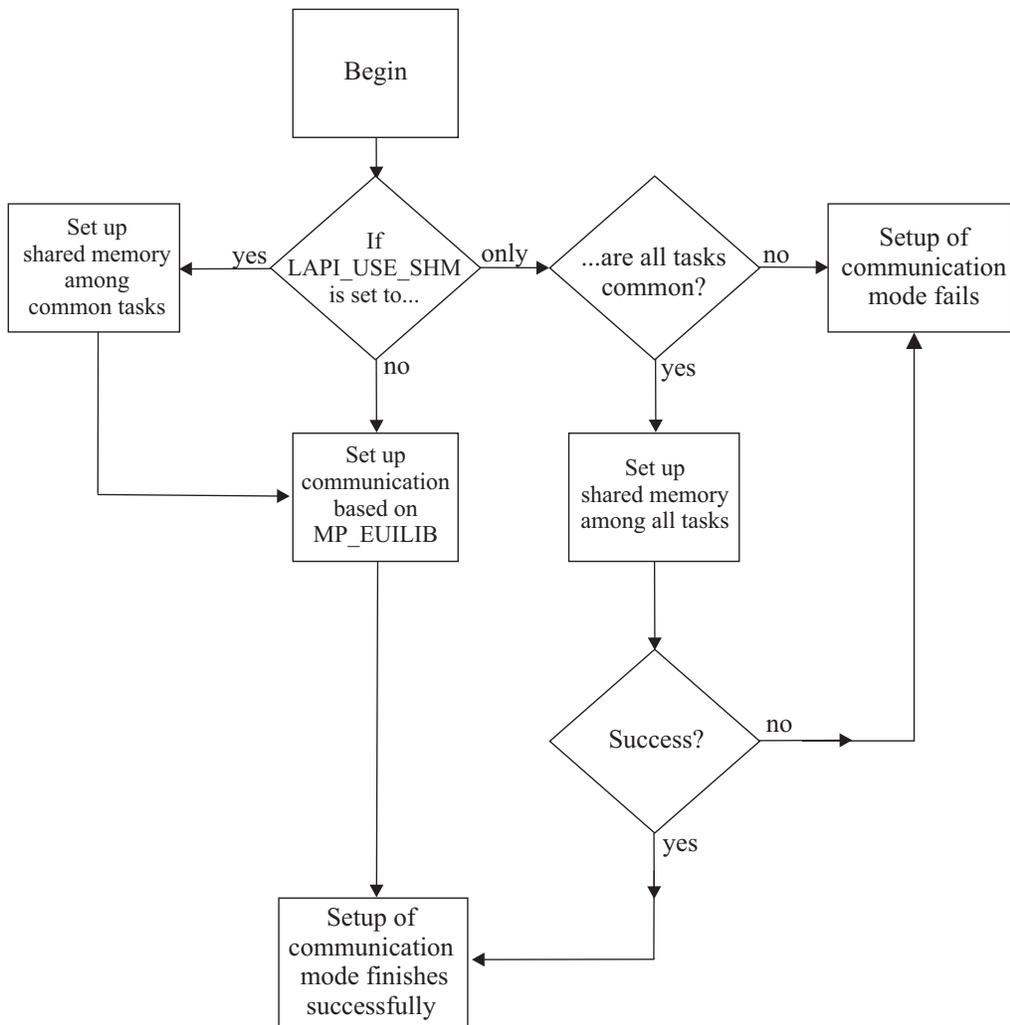


Figure 1. How LAPI sets up the mode of communication

Initializing LAPI

To initialize LAPI, follow these steps:

1. Set environment variables (as described in “Setting environment variables” on page 29) before the user application is invoked. The remaining steps are done in the user application.
2. Clear **lapi_info_t**, then set any fields.
See “LAPI_Init” on page 163 for the **lapi_info_t** structure. See “Passing information to LAPI using lapi_info_t” and “Registering an error handler” on page 32 for more information about using this structure.
3. Call **LAPI_Init**.
See “LAPI_Init” on page 163 for more information about using **LAPI_Init**, including various examples of LAPI initialization.

Passing information to LAPI using lapi_info_t

The second argument to **LAPI_Init** is the address of a **lapi_info_t** structure. You can use this structure to pass certain information to LAPI. Certain fields in the structure are reserved for future use and should be cleared before calling

LAPI_Init. It is strongly recommended that the entire structure be cleared, and then only desired fields get set. You can clear the memory of a **lapi_info_t** structure using the AIX **bzero** subroutine. For example, suppose you have a structure declared as follows:

```
lapi_info_t info_struct;
```

You would clear this structure by calling:

```
bzero(&info_struct, sizeof(lapi_info_t));
```

For more information about **lapi_info_t**, see “LAPI_Init” on page 163. For more information about **bzero**, see *AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions, Volume 1* or *AIX 5L Version 5.3 Technical Reference: Base Operating System and Extensions, Volume 1*.

Registering an error handler: Use the *err_hndlr* field in **lapi_info_t** to register an error handler that you provide. Your error handler is invoked by the occurrence of an asynchronous error. LAPI calls its own internal error handler, which then calls your error handler, if it is registered.

If your error handler is registered, you can control whether execution will continue, based on the type of error that is returned. You can choose to dump information, terminate the job from within its error handler, or both. If the user handler returns, LAPI continues execution. If the error handler you provided is not registered, LAPI’s internal error handler will terminate the job.

Normally, an error handler terminates. Your error handler is intended for the types of actions that are performed by the **atexit** subroutine, so you should do cleanup processing and exit. If you don’t exit, LAPI’s state is undefined. For example, there is no guarantee that outstanding messages will complete.

The format of a user-provided error handler follows:

```
void my_err_hndlr(lapi_handle_t *hdl, int *error_code, lapi_err_t *err_type,
                 int *task_ID, int *src)
{
    char errstring[100]; /* for error code translation */

    /* get the error code string to decipher the error */
    LAPI_Msg_string(*error_code, errstring);
    fprintf(stderr, "%s\n", errstring);

    if ( you want to terminate ) {
        LAPI_Term(*hdl); /* to terminate LAPI */
        exit(some_return_code);
    }

    /* any additional processing */

    return; /* signal to LAPI that the error is not */
           /* fatal and execution should continue */
}

{
    ...
    lapi_handle_t hndl;
    lapi_info_t info;

    bzero(&info, sizeof(lapi_info_t)); /* clear lapi_info */

    /* set the error handler pointer */
    info.err_hndlr = (LAPI_err_hndlr) my_err_hndlr;
```

```
LAPI_Init(&hndl, &info);  
:  
}
```

For information about **atexit**, **bzero**, **exit** and **fprintf**, see *AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions, Volume 1* or *AIX 5L Version 5.3 Technical Reference: Base Operating System and Extensions, Volume 1*.

Terminating LAPI

Use the **LAPI_Term** subroutine to terminate a LAPI context that is specified by a particular LAPI handle. For example:

```
LAPI_Term(hndl);
```

Any LAPI notification threads that are associated with this context are terminated. If any LAPI calls are made using *hndl* after **LAPI_Term** is called, an error occurs. See “LAPI_Term” on page 201 for more information.

Chapter 6. Transferring data

LAPI provides "put", "get", and read-modify-write (RMW) functions for data transfer.

LAPI's "put" and "get" functions are non-blocking calls. A "put" operation copies data from a specified region in the origin address space to the specified region in the target address space. A "get" operation copies data from a specified region in the target address space to a specified region in the origin address space. Completion of the operation is signaled if counters are specified.

For "get" functions, only synchronous operation is possible. For "put" functions, both standard and synchronous operations are supported. A standard "put" operation is provided by incrementing the origin counter (*org_cntr*) when the origin buffer can be reused. To guarantee the standard behavior of the LAPI "put" operations, you can use the **LAPI_Waitcntr** subroutine in conjunction with the origin counter (*org_cntr*). A synchronous "put" operation is provided by incrementing the completion counter (*cmpl_cntr*) after the data has been written into the target buffer. To guarantee the synchronous behavior of the LAPI "put" operations, you can use the **LAPI_Waitcntr** subroutine in conjunction with the origin counter (*org_cntr*). See "Flow of "put" operations" and "Flow of "get" operations" on page 36 for more information.

The **LAPI_Xfer** subroutine provides functions that are similar to **LAPI_Put** and **LAPI_Get**, with some enhancements. See "LAPI_Xfer" on page 219 for more information.

For RMW information, see "Flow of read-modify-write operations" on page 37.

For users of the HPS or the pSeries HPS, RSCT LAPI supports bulk message transfer using the remote direct memory access (RDMA) protocol. See Chapter 13, "Bulk transfer of messages," on page 99 for more information.

Data transfer operations

Information about the flow of LAPI's data transfer operations follows.

Flow of "put" operations

Figure 2 on page 36 illustrates the sequence of events for a **LAPI_Put**, **LAPI_Putv**, **PUT**, or **PUTV** operation. If an origin counter is specified, it is incremented by LAPI on the sending side when the send-side data buffer is available for reuse. If a target counter is specified, it is incremented by LAPI on the target side upon message completion. If a completion counter is specified, LAPI sends an internal message to the original sender. Upon receipt of this message, the completion counter is incremented by LAPI at the origin.

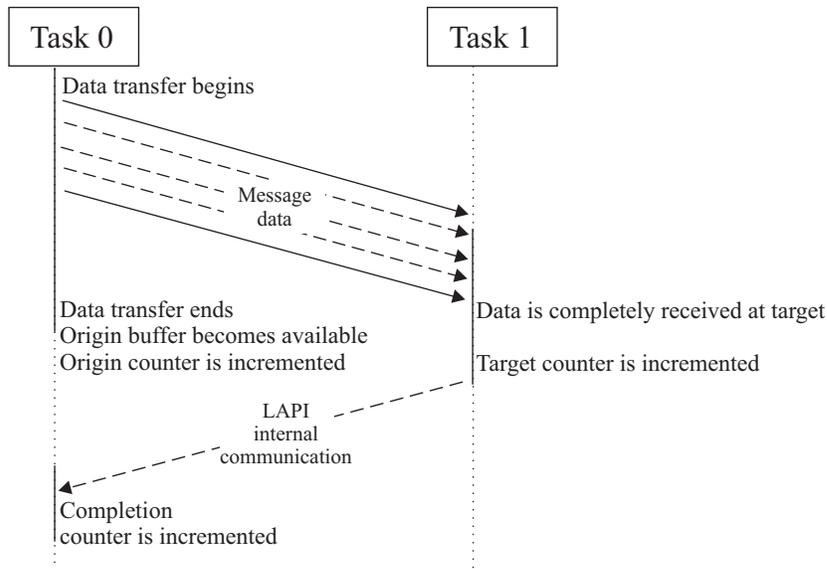


Figure 2. The sequence of events for a "put" operation

Flow of "get" operations

Figure 3 illustrates the sequence of events for a **LAPI_Get**, **LAPI_Getv**, **GET**, or **GETV** operation. In this case, the "source" is actually the receiver of the message, and the target can be thought of as the "sender". Because the direction of the data transfer is from the target to the source, it is helpful to think of the origin and target counters switching roles from the "put" case. If an origin counter is specified, it is incremented on the source task when the message completes. If a target counter is specified, it is incremented on the target side once the target data buffer is available (that is, when its contents can be modified without corrupting the message in transit). Because the origin counter actually signifies message completion, there is no completion counter for "get" operations.

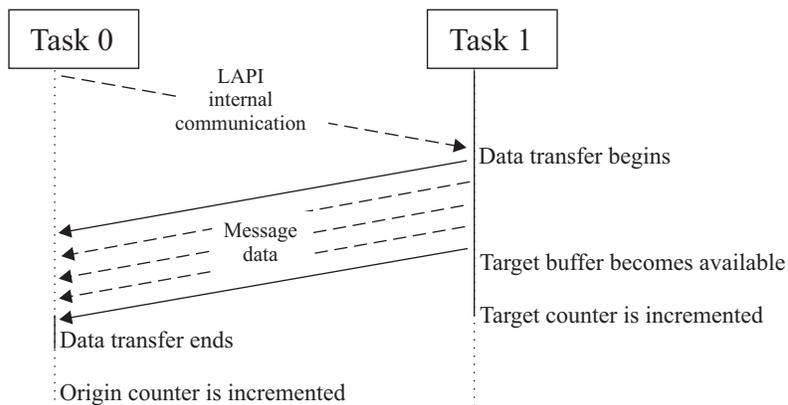


Figure 3. The sequence of events for a "get" operation

Flow of read-modify-write operations

Figure 4 illustrates the sequence of events for a **LAPI_Rmw** operation or a **LAPI_Rmw64** operation. Only the origin counter is supported, and its behavior is similar to that of the origin counter in the **LAPI_Get** case.

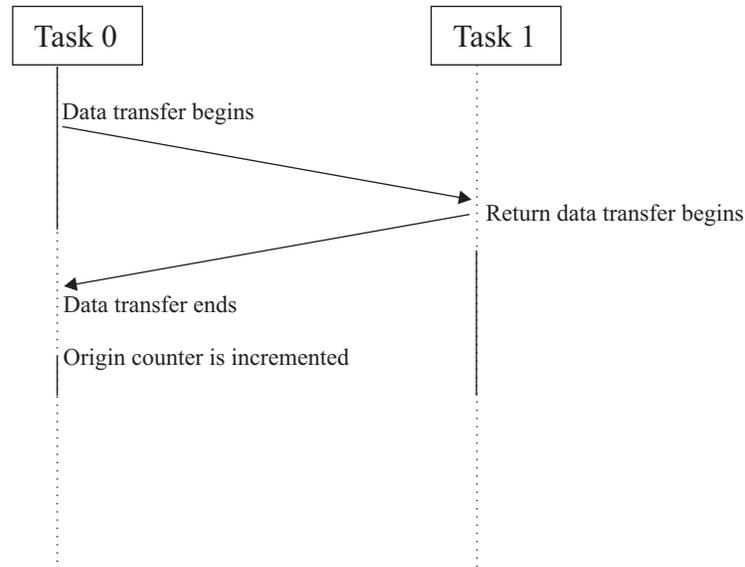


Figure 4. The sequence of events for a read-modify-write operation

Non-contiguous data transfer

LAPI supports two types of non-contiguous data transfer: LAPI vectors and data gather/scatter programs (DGSPs). Use LAPI vectors when the data to be transferred can be described simply, such as through the explicit specification of multiple buffer address/length pairs, or through the repetition of a data template that is described by a block size and stride. For more complex transfers, such as through recursive data descriptions, use one or more DGSPs.

Using vectors

LAPI vector transfers involve the specification of an origin vector description and a target vector description, each of type **lapi_vec_t**. The API calls for vector data transfer are: **LAPI_Amsendv**, **LAPI_Getv**, **LAPI_Putv**, **AMV**, **GETV**, and **PUTV**. These APIs are vector-based versions of the basic API calls of the same name: **LAPI_Amsend**, **LAPI_Get**, **LAPI_Put**, **AM**, **GET**, and **PUT**. For instance, **LAPI_Amsendv** can be thought of as "**LAPI_Amsend** with vectors." This document refers to the three non-vector calls (**LAPI_Amsend**, **LAPI_Get**, and **LAPI_Put**) as *basic calls* and the type of transfer that is associated with these basic calls as *scalar transfer*. The corresponding LAPI vector calls (**LAPI_Amsendv**, **LAPI_Getv**, and **LAPI_Putv**) are referred to as *vector calls* and the type of transfer that is associated with these calls as *vector transfer*.

LAPI vector transfers are set up similarly to scalar transfers. The difference lies in the data specification. For a basic transfer, an origin address, target address, and data length are required. For a vector transfer, two vector descriptions are required — one for the origin and one for the target. Consider the basic **LAPI_Amsend** call. The origin task provides an origin buffer (base address and data length) and the address or index of a header handler on the target task. The target header handler is expected to return a buffer address on the target task, using the origin data

length as part of the target buffer definition. A data buffer in LAPI is defined by a base address and a data length. For **LAPI_Amsendv**, the origin task is required to pass in an origin vector description (instead of buffer address and data length) and a header handler address or index. Similarly, the target header handler is expected to return the address of a target vector description, rather than a single data buffer.

As another example, when using **LAPI_Get**, you need to specify an origin address, a target address, and a data length. When using **LAPI_Getv**, you need to specify an origin vector description and a target vector description. Using vector descriptions gives you additional flexibility in how the data is collected on the origin side and distributed on the target side. In all vector transfers, the type of vector to be transferred determines the requirements on the origin and target vectors.

LAPI vectors are structures of type **lapi_vec_t**, defined as follows:

```
typedef struct {
    lapi_vectype_t  vec_type;
    uint           num_vecs;
    void           **info;
    ulong          *len;
} lapi_vec_t;
```

vec_type is an enumeration that describes the type of vector transfer, which can be one of the following:

- | | |
|------------------------------|--|
| LAPI_GEN_GENERIC | Supports explicit specification of the data buffers. The number of buffers and the buffer lengths between the origin side and the target side do not need to match. You can only use this datatype with LAPI_Amsendv and AMV . LAPI_GEN_GENERIC <i>cannot</i> be used with LAPI_Getv , LAPI_Putv , GETV , or PUTV . |
| LAPI_GEN_IOVECTOR | Supports explicit specification of the data buffers on the origin side and the target side. The number of buffers in the origin vector description and in the target vector description must be the same. The lengths of the origin vector's data buffers must equal the lengths of the target vector's data buffers. |
| LAPI_GEN_STRIDED_XFER | Supports the transfer of data through one or more iterations of sending a block and offsetting by its stride. The data is described by a block size, stride, and number of invocations. The origin vector description and the target vector description must be the same, though the strides can differ. |

If *vec_type* is **LAPI_GEN_GENERIC** or **LAPI_GEN_IOVECTOR**, the fields are used as follows:

- | | |
|-----------------|--|
| <i>num_vecs</i> | indicates the number of data vectors to transfer. Each data vector is defined by a base address and data length. |
| <i>info</i> | is the array of addresses. |
| <i>len</i> | is the array of data lengths. |

For example, consider the following vector description:

```

vec_type = LAPI_GEN_IOVECTOR
num_vecs = 3
info     = {addr_0, addr_1, addr_2}
len      = {len_0, len_1, len_2}

```

For an origin vector, LAPI would read *len_0* bytes from **addr_0**, *len_1* bytes from **addr_1**, and *len_2* bytes from **addr_2**. For a target vector, LAPI would write *len_0* bytes to **addr_0**, *len_1* bytes to **addr_1**, and *len_2* bytes to **addr_2**.

Recall that vector transfers require an origin vector and a target vector. For **LAPI_Amsendv** calls, the origin vector is passed to the API call on the origin task. The address of the target vector is returned by the target header handler.

LAPI_GEN_GENERIC

For transfers of type **LAPI_GEN_GENERIC**, the target vector description must also have type **LAPI_GEN_GENERIC**. Use this datatype only with **LAPI_Amsendv** and **AMV**. The contents of the *info* and *len* arrays are unrestricted in the generic case; the number of vectors and the length of vectors on the origin and target do not need to match. In this case, LAPI transfers a given number of bytes in noncontiguous buffers specified by the origin vector to a set of noncontiguous buffers specified by the target vector.

If the sum of target vector data lengths (say **TGT_LEN**) is less than the sum of origin vector data lengths (say **ORG_LEN**), only the first **TGT_LEN** bytes from the origin buffers are transferred and the remaining bytes are discarded. If **TGT_LEN** is greater than **ORG_LEN**, all **ORG_LEN** bytes are transferred. The rest of the target area is left alone. Consider the following example:

```

Origin_vector: {
    num_vecs = 3;
    info     = {orgaddr_0, orgaddr_1, orgaddr_2};
    len      = {5, 10, 5}
}

Target_vector: {
    num_vecs = 4;
    info     = {tgtaddr_0, tgtaddr_1, tgtaddr_2, tgtaddr_3};
    len      = {12, 2, 4, 2}
}

```

LAPI copies data as follows:

1. 5 bytes from **orgaddr_0** to **tgtaddr_0** (leaves 7 bytes of space at a 5-byte offset from **tgtaddr_0**)
2. 7 bytes from **orgaddr_1** to remaining space in **tgtaddr_0**, which leaves 3 bytes of data to transfer from **orgaddr_1**
3. 2 bytes from **orgaddr_1** to **tgtaddr_1**, which leaves 1 byte to transfer from **orgaddr_1**
4. 1 byte from **orgaddr_1** followed by 3 bytes from **orgaddr_2** to **tgt_addr_2**, which leaves 2 bytes to transfer from **orgaddr_2**
5. 2 bytes from **orgaddr_2** to **tgtaddr_3**

LAPI copies data from the origin until the space described by the target is filled. For example:

```

Origin_vector: {
    num_vecs = 1;
    info     = {orgaddr_0};
    len      = {20}
}

```

```

Target_vector: {
    num_vecs = 2;
    info     = {tgtaddr_0, tgtaddr_1};
    len      = {5, 10}
}

```

LAPI copies 5 bytes from **orgaddr_0** to **tgtaddr_0** and the next 10 bytes from **orgaddr_0** to **tgtaddr_1**. The remaining 5 bytes from **orgaddr_0** are not copied.

LAPI_GEN_IOVECTOR

For transfers of type **LAPI_GEN_IOVECTOR**, the lengths of the vectors must match and the target vector description must match the origin vector description. More specifically, the target vector description must:

- Also have type **LAPI_GEN_IOVECTOR**.
- Have the same *num_vecs* as the origin vector.
- Initialize the *info* array with *num_vecs* addresses in the target address space.
- Initialize the *len* array with *num_vecs* lengths. The values of those lengths must be the same as the values in the *len* array of the origin vector. In other words, for origin vector **o_vec** and target vector **t_vec**, **o_vec.len[i]** must equal **t_vec.len[i]** for all *i* ($0 \leq i < num_vecs$).

For LAPI vectors **origin_vector** and **target_vector** described similarly to the example above, data is copied as follows: for all *i*, *origin_vector.len[i]* bytes are transferred from the address at *origin_vector.info[i]* to the address at *target_vector.info[i]*.

Figure 5 shows how LAPI transfers data using type **LAPI_GEN_IOVECTOR**: In this case, four transfers take place:

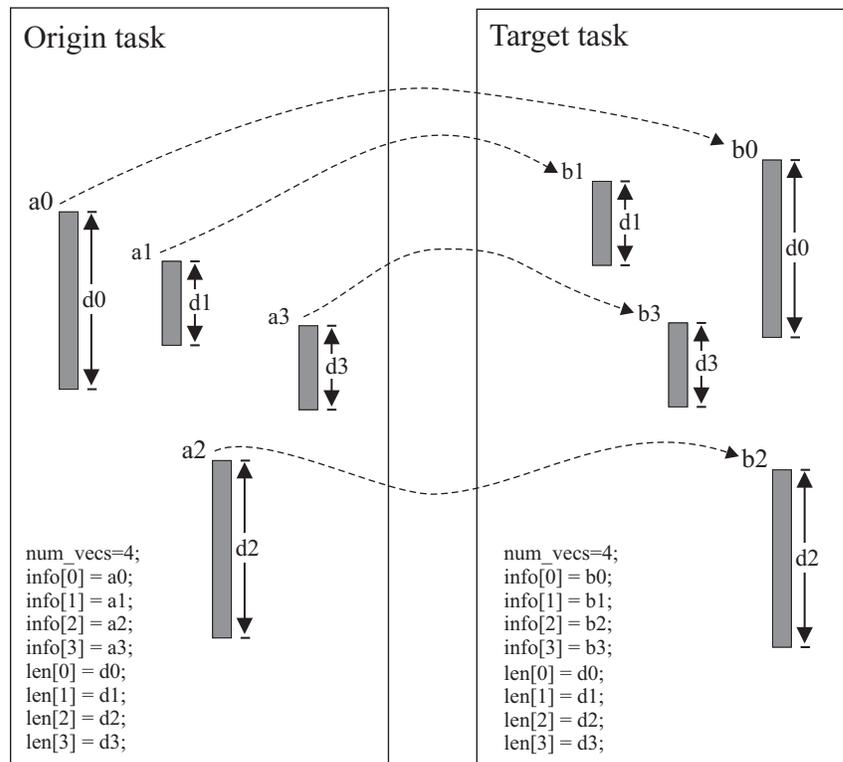


Figure 5. Transferring data with type **LAPI_GEN_IOVECTOR**

1. *d0* bytes are transferred from **a0** on the origin task to **b0** on the target task

2. $d1$ bytes are transferred from **a1** on the origin task to **b1** on the target task
3. $d2$ bytes are transferred from **a2** on the origin task to **b2** on the target task
4. $d3$ bytes are transferred from **a3** on the origin task to **b3** on the target task

LAPI_GEN_STRIDED_XFER

For transfers of type **LAPI_GEN_STRIDED_XFER**:

- the target vector description must match the origin vector description (though the strides can differ)
- the *info* array of the origin and target vectors is used differently than it is for non-strided vector data transfer:

Rather than specifying the set of addresses, the *info* array of the origin and target vectors is used to specify a data block "template", consisting of a base address, block size, and stride. LAPI thus expects the *info* array to contain three integers. The first integer contains the base address, the second integer contains the block size to copy, and the third integer contains the byte stride. In this case, *num_vecs* indicates the number of blocks of data that LAPI should copy, where the first block begins at the base address. The number of bytes to copy in each block is given by the block size and the starting address for all but the first block is given by previous address + stride. The total amount of data to be copied will be $num_vecs * block_size$. The *len* field of the vector description structure is not used for **LAPI_GEN_STRIDED_XFER**, so any values it contains are ignored. Consider the following example:

```
Origin_vector: {
    num_vecs = 3;
    info     = {orgaddr, 5, 8}
}
```

Based on this description, LAPI will transfer 5 bytes from *orgaddr*, 5 bytes from *orgaddr+8* and 5 bytes from *orgaddr+16*.

Figure 6 shows how LAPI transfers data using type **LAPI_GEN_STRIDED_XFER**: From the vector descriptions represented in Figure 6:

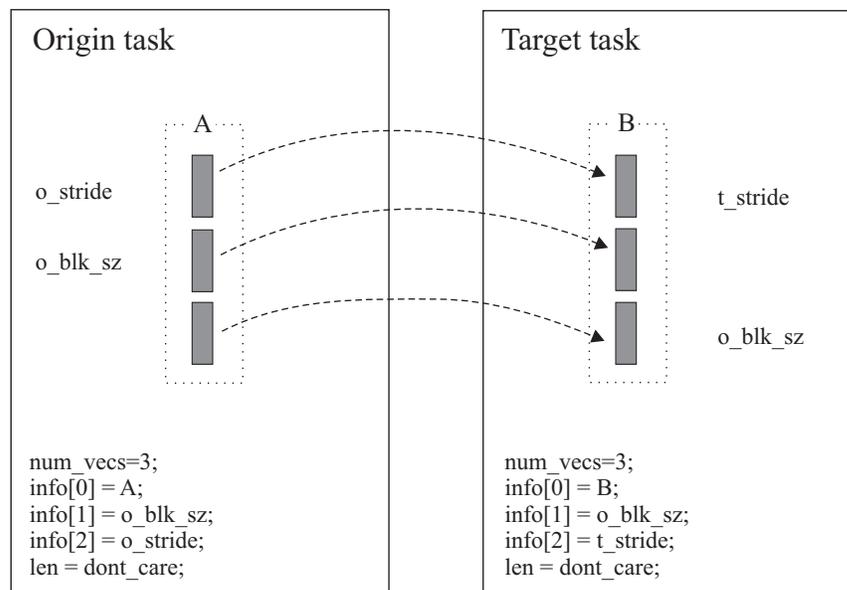


Figure 6. Transferring data with type **LAPI_GEN_STRIDED_XFER**

1. o_blk_sz bytes are transferred from **A** on the origin task to **B** on the origin task
2. o_blk_sz bytes are transferred from **A**+ o_stride on the origin task to **B**+ t_stride on the target task
3. o_blk_sz bytes are transferred from **A**+ $(2 * o_stride)$ on the origin task to **B**+ $(2 * t_stride)$ on the target task

Vector data transfer summary

Table 7 on page 43 summarizes the rules for transferring data using vectors.

Table 7. Rules for vector transfer

Type of transfer	Data is specified...	Target vector rules	The origin vector resides:	The target vector resides:
LAPI_GEN_GENERIC	By explicit lists of origin and target buffers.	The <i>vec_type</i> field must match the origin vector type.	On the origin for LAPI_Amsendv only.	On the target for LAPI_Amsendv only. The addresses in the <i>info</i> array should be in the target address space.
LAPI_GEN_IOVECTOR	By explicit lists of origin and target buffers.	The <i>vec_type</i> and <i>num_vecs</i> fields and values in the <i>len</i> array must match the origin vector type.	On the origin for all vector API calls (LAPI_Amsendv , LAPI_Getv , and LAPI_Putv).	On the origin for LAPI_Getv and LAPI_Putv . On the target for LAPI_Amsendv . The addresses in the <i>info</i> array should be in the target address space.
LAPI_GEN_STRIDED_XFER	With a block size, stride size, and the number of blocks to transfer.	The <i>vec_type</i> field and the contents of the <i>info</i> array must match the origin vector type (though the strides can differ).	On the origin for all vector API calls (LAPI_Amsendv , LAPI_Getv , and LAPI_Putv).	On the origin for LAPI_Getv and LAPI_Putv . On the target for LAPI_Amsendv .

Using data gather/scatter programs (DGSPs)

LAPI supports the *data gather/scatter program (DGSP)* as a mechanism for describing data layouts in memory so certain LAPI functions can operate directly on non-contiguous user buffers. This support is motivated by the need to support MPI datatype constructors. The MPI standard defines an API for specifying any possible data layout with a user-defined datatype. **MPI_Send** gathers data according to the datatype definition and transmits this data over the network. Likewise, an **MPI_Recv** can accept data from the network and scatter it to any layout in memory under control of an MPI datatype. The DGSP provides a compact and complete method of implementing any data layout an MPI user can specify using the MPI datatype constructors. The MPI library incorporates a "compiler" that is run at an **MPI_Type** commit call to construct the DGSP that corresponds to the user-defined datatype. Like computer object code, the DGSP is not intended as a human-readable specification.

It is possible for LAPI users to construct a DGSP by hand, just as it is possible for someone to write a computer program directly in object code. For anything with significant complexity, it is expected that the LAPI application that uses DGSP will incorporate some form of "code generator" that can generate DGSP code from some layout specification appropriate to the application data structures. This section documents the data gather/scatter instruction set with which LAPI applications can create DGSP codes. A DGSP consists of a descriptor and a "code" array of integers. The instruction counter (IC) of the DGSM, which is the DGSP interpreter within LAPI, is an index into this array of integers. Therefore, the addressing unit of the DGSM is one integer, just as the addressing unit of most computer hardware is one byte.

LAPI provides the **LAPI_VERIFY_DGSP** environment variable as a validation option to help LAPI users who create their own DGSPs and want to validate them as they are registered. Validating these DGSPs will catch many, but not all, possible errors. Validation may especially be needed because multiple DGSP transfers could be in progress concurrently and the DGSM runs to handle these data transfers need to report their failures asynchronously, which could make it difficult for a LAPI user who creates several DGSPs to know which DGSP was running at the failure point. The most common result of an uncatchable error in a user-created DGSP is a segmentation fault, which occurs asynchronously. Debugging a faulty DGSP can be challenging.

Each instruction for the DGSM is encoded as a structure that can be mapped into an integer "code" array. LAPI's header files provide a macro for each instruction that can be used to update the IC as instructions are added to a DGSP. In the example below, a COPY instruction is built and macro **LAPI_DGSM_COPY_SIZE** is used to advance the IC to the proper location for building the next instruction, which will be an ITERATE. You *must* include an ITERATE instruction in every LAPI DGSP, even if you don't think you'll run the program more than once. The suggested technique for building instructions is to declare a set of pointers, one per instruction type and map the instruction structure into the code array as follows:

```
lapi_dgsm_copy_t pCopy;
lapi_dgsm_iterate_t pIterate;
:
:
ic = 0;
pCopy = (lapi_dgsm_copy_t*) &code[ic];
pCopy->opcode = LAPI_DGSM_COPY;
pCopy->bytes = 5;
pCopy->offset = 0;
ic += LAPI_DGSM_COPY_SIZE;
pIterate = (lapi_dgsm_iterate_t*) &code[ic];
pIterate->opcode = LAPI_DGSM_ITERATE;
:
:
```

Each complete DGSP represents the gather or scatter of a specific number of bytes of data. It can be helpful to visualize a DGSP as a sieve or template that can be positioned on a memory buffer to let some byte ranges show through while masking others. A LAPI operation that uses a DGSP also specifies the address of a user buffer and a number of bytes to be processed. If the number of bytes to be processed is greater than the DGSP represents, the DGSP is reinterpreted as many times as needed. The DGSP template is first positioned at offset 0 in the user buffer and is advanced by some stride each additional time it is interpreted. The DGSP descriptor guides this process, so it is essential that the content of the descriptor be understood. This descriptor is represented by the following structure:

```
typedef enum {LAPI_DGSM_SPARSE=0, LAPI_DGSM_CONTIG,
             LAPI_DGSM_UNIT} lapi_dgsp_density_t;

typedef struct {
    int          *code;      /* array containing the DGSP code      */
    int          code_size; /* size of the DGSP code array        */
    int          depth;     /* required DGSM stack depth          */
    lapi_dgsp_density_t density; /* lapi_dgsp_density_t datatype      */
    long         size;      /* datatype packed size               */
    long         extent;    /* datatype extent (stride between reps) */
    long         lext;      /* leftmost data byte position        */
    long         rext;      /* rightmost data byte position       */
    int          atom_size; /* 0 or packet filling unit           */
} lapi_dgsp_descr_t;
```

where:

code is a pointer to the integer array in which the DGSP has been built.

code_size is the number of integers in the code array.

depth specifies the maximum subroutine activation depth in the DGSP. At a minimum, a DGSP has a "main" routine. It can also have "subroutines". The "machine" that interprets the DGSP uses a stack. For a DGSP that has only a "main" routine (because it does not use a GOSUB instruction), *depth* is equal to 1.

density is one of three enumerated values:

LAPI_DGSM_SPARSE

indicates that the DGSP data layout has internal gaps.

LAPI_DGSM_CONTIG

indicates that the DGSP data layout is contiguous, but there is a gap either before or after the contiguous section. That is, if the DGSP must be interpreted more than once, the DGSM must deal with a gap.

LAPI_DGSM_UNIT

indicates that the DGSP data layout is contiguous and interpreting the DGSP more than once still represents a contiguous layout.

size is the number of bytes represented by a single application of the DGSP.

extent is the stride for repeated interpretation of the DGSP, that is, the number of bytes to advance the template within the user buffer for the next application.

left is the offset of the leftmost byte (toward the low address) represented by the DGSP.

right is the the offset just past the rightmost byte (toward the high address) represented by the DGSP.

atom_size indicates that LAPI can choose the number of bytes per packet (if *atom_size* is set to 0, which is the norm). If *atom_size* is set to a non-zero value, every packet will contain an integral number of "atoms".

Consider a C structure that contains one integer followed by one character. The C compiler will put the integer at offset 0 in the structure and the character at offset 4. If an array of such structures is declared, the compiler word-aligns each array element so that there is a 3-byte gap. An efficient DGSP for this structure has two instructions: a COPY and an ITERATE. The code will be discussed in more detail later.

A DGSP descriptor for this structure would be constructed as follows:

```
int code[LAPI_DGSM_COPY_SIZE+LAPI_DGSM_ITERATE_SIZE];
lapi_dgsp_descr_t dgsp_d;
dgsp_d.code = &code[0];
dgsp_d.code_size = LAPI_DGSM_COPY_SIZE+LAPI_DGSM_ITERATE_SIZE;
dgsp_d.depth = 1;
dgsp_d.density = LAPI_DGSM_CONTIG;
dgsp_d.size = 5;
```

```
dgsp_d.extent = 8;
dgsp_d.ltext = 0;
dgsp_d.rext = 5;
dgsp_d.atom_size = 0;
```

A **LAPI_Xfer** call that used this DGSP to send the first three elements from an array of such structures would specify **array[0]** as the buffer address and **15** as the number of bytes. The DGSP would be interpreted 3 times, first at byte offset **0**, then at byte offset **8**, and finally at byte offset **16**. The instruction set of the DGSM contains five instructions, each of which is discussed in some detail below. The first field in any instruction is the *opcode*. The other fields are different for each instruction and are discussed below.

LAPI_DGSM_COPY

is the first of two instructions to move data. A COPY instruction represents a single contiguous block of bytes to be transferred. The instruction fields are *bytes* and *offset*.

LAPI_DGSM_MCOPY

is the second of two instructions to move data. An MCOPY instruction is a variable-length instruction that defines one or more contiguous blocks to copy. The MCOPY *count* field specifies how many block descriptions it contains. Each block description is a block displacement (*block_disp*, block length (*block_len*) tuple. If pMcopy is a pointer to an MCOPY instruction, block description fields can be identified as:

```
pMcopy->block[i].block_len and pMcopy->block[i].block_disp.
```

Because the first block is included in **LAPI_DGSM_MCOPY_SIZE**, the size of the instruction is:

```
LAPI_DGSM_MCOPY_SIZE + (count-1) * LAPI_DGSM_MCOPY_BLOCK_SIZE
```

LAPI_DGSM_GOSUB

calls a DGSP subroutine. The subroutine runs with its own stack frame and does not change the state of the calling stack frame. The subroutine can be visualized as a subordinate template to be applied one or more times within a single application of the containing template. The *reps* field specifies how many times, *offset* specifies the position relative to the current template base in bytes, and *stride* specifies how far to advance the subordinate template after each repetition. The *to_loc* and *ret_loc* are both IC-relative jumps that are based on the instruction counter at the GOSUB instruction. For a normal return to the instruction that follows the GOSUB instruction, the *ret_loc* field is set to **LAPI_DGSM_GOSUB_SIZE**. The *to_loc* field is always set to the distance from the GOSUB *opcode* to the *opcode* of the first instruction of the subroutine.

LAPI_DGSM_ITERATE

must terminate each DGSP "main" program as well as any subroutine within a DGSP. A DGSP with a *depth* of **1** has a "main" routine, but no subroutines. ITERATE will decrement the repetitions counter in the current stack frame. If the counter is not yet zero, ITERATE will branch to its *iter_loc* target to interpret the DGSP or subroutine again at its new position within the buffer. If the counter is **0**, the ITERATE pops the stack and branches to the subroutine return point. The *iter_loc* value is the distance from the ITERATE opcode to the first opcode of the DGSP or subroutine. The values are normally negative.

LAPI_DGSM_CONTROL

is only used in a scatter-side DGSP and in conjunction with a data distribution manager (DDM) function. Most DGSPs will have no CONTROL

instruction and the DGSM mode is non-CONTROL by default. A CONTROL puts the scatter DGSM in a mode where every copy operation is done by a call to the registered DDM function. The operation and operand field values are saved by the DGSM when a CONTROL instruction is encountered and are passed as parameters to every DDM function call. An example of using a DDM function is to construct an accumulate reduction operation for arriving data. A DDM (reduction) function could be written that does more than one kind of arithmetic operation and can handle more than one kind of operand. These parameters allow the CONTROL instruction to tell the reduction function what kind of operand and operation it is working on. A DDM function which does not require this versatility may ignore these parameters. Any non-negative value for the operation field puts the DGSM into CONTROL mode. An operation field with value **LAPI_DGSM_NO_CTL** takes the machine out of CONTROL mode. Because a reduction function must operate on full operands, the sender side DGSP must specify an atom size that matches the operand size. If the DDM expects to operate on 16-byte REALs, *atom_size* must be **16** so that all transmitted packets contain a multiple of 16 bytes. Also, because LAPI may bypass the scatter DGSM when the data is contiguous, any DGSP that contains a CONTROL instruction must be labeled **LAPI_DGSM_SPARSE** to force the data through the DGSM, which will honor the CONTROL instruction.

See “LAPI_Util” on page 203 for a sample program that creates and registers a DGSP.

Detecting completion

LAPI gives users two main approaches for notification of event occurrence during data transmission: handlers and counters. Handlers are registered with LAPI as callback pointers to be invoked at certain well-defined points during message transfer. Counters are implemented as abstract data types and managed through a set of LAPI API calls.

LAPI handlers

LAPI uses an active messaging infrastructure. Message retrieval involves the invocation of an asynchronous, active-message handler to process the incoming message. LAPI provides a way for users to register their own receive-side handlers through the **LAPI_Amsend** and **LAPI_Amsendv** subroutines, as well as through **LAPI_Xfer** with transfer types **LAPI_AM_XFER (AM)**, **LAPI_AMV_XFER (AMV)**, and **LAPI_DGSP_XFER (DGSP)**. For certain transfer types, **LAPI_Xfer** also provides a send-side handler to run once the send-side data buffer is available for re-use.

LAPI provides three types of handlers that you can register to run at certain points during data transmission:

Header handler

Runs on the receiving side upon the arrival of the first message packet. You need to provide a callback pointer to a header handler function when using the API calls that support it, namely **LAPI_Amsend**, **LAPI_Amsendv**, **AM**, **AMV**, and **DGSP**. See “The header handler” on page 48 for more information.

In addition to the callback pointer, you can optionally provide an index into the header handler function table. See “LAPI_Addr_set” on page 128 for more information.

Completion handler (or receive completion handler)

Runs at the target task upon message completion. Its use is optional, and is supported for the same set of API calls as the header handler. LAPI provides a completion handler parameter pointer so you can pass data from the header to the completion handler. The receive completion handler can also be specified in the call to **LAPI_Xfer** for the **LAPI_GET_XFER** and **LAPI_GETV_XFER** transfer types. Recall that in these "get" calls, the receiver is same as the original task that makes the "get" **LAPI_Xfer** call. See "The completion handler" on page 49 for more information.

Send completion handler

Is available only when you use **LAPI_Xfer** for all transfer types, except **LAPI_GET_XFER** and **LAPI_GETV_XFER**. As previously indicated, LAPI only supports synchronous behavior with its "get" calls, so send completion notification is not supported. This handler is invoked on the sending side when the sending data buffer is available for modification. As with the receive completion handler, LAPI provides a parameter pointer with which you can pass information in to the send completion handler, and its use is optional. See "The send completion handler" on page 51 for more information.

The header handler

LAPI header handlers are defined to be of the following type:

```
typedef void * (hdr_hdlr_t)(lapi_handle_t *hdl, void *uhdr, uint *uhdr_len,
                           ulong *msg_len, compl_hdlr_t **comp_h,
                           void **uinfo);
```

The header handler runs on the target task upon the arrival of the first message packet. As shown above, the header handler must return an address value to LAPI. This value is interpreted by LAPI as the base address of the data buffer where it must write the message on the target. Because of this, a header handler must be provided by the user for API calls that support its use. For the **LAPI_Amsendv** call, the header handler returns a pointer to a **lapi_vec_t** structure.

Additional information related to the arriving message at the target can be passed back to LAPI through reference parameters of the header handler. For example, if you want to use a completion handler for post-processing of the message after it has completely arrived at the target, a pointer to the completion handler must be returned to LAPI through one of the header handler parameters (denoted by *comp_h* in the header handler type definition above). Similarly, a parameter to the completion handler is also set and returned by way of another of the header handler parameters (denoted by *uinfo* in the header handler type definition). Note that the use of a completion handler is optional, so if either of the pointers is not going to be used, you need to set these header handler parameters to NULL within the header handler. Otherwise, LAPI will interpret the value as a valid function callback address and attempt to invoke it on message completion, which will almost certainly lead to a memory fault. You can use a completion handler and still set the parameter pointer to NULL (LAPI will simply invoke the completion handler without any parameters). If you set the completion handler as NULL and assign some non-zero value for the parameter pointer, the parameter pointer will be ignored.

To define a header handler:

1. Without using a completion handler:

```
void *my_hdr_hdlr(lapi_handle_t *hdl, void *uhdr, uint *uhdr_len,
                 ulong *msg_len, compl_hdlr_t **comp_h, void **uinfo) {

    /* some user code */
```

```

    *comp_h = NULL;
    *uinfo = NULL;

    return data_buffer;
}

```

2. Using a completion handler without a parameter:

```

void *my_hdr_hdlr(lapi_handle_t *hdl, void *uhdr, uint *uhdr_len,
                 ulong *msg_len, compl_hdlr_t **comp_h, void **uinfo) {

    /* pass the completion handler back by reference */
    *comp_h = compl_hdlr;
    *uinfo = NULL;

    return data_buffer;
}

```

For an example that illustrates the use of a completion handler and a completion handler parameter, see “The completion handler.”

For advanced programming information, see “The enhanced header handler interface” on page 75.

The completion handler

LAPI calls the completion handler after the final packet of the message has been completely received into the target buffers. Unlike header handlers, completion handlers are optional. LAPI completion handlers are defined to be of the following type:

```
typedef void (compl_hdlr_t)(lapi_handle_t *hdl, void *completion_param);
```

LAPI provides a facility with which you can pass information from the header handler to the completion handler. You set both the completion handler pointer and completion handler parameter pointer in the header handler (returned to LAPI by reference parameters). An example of using a completion handler and completion handler parameter follows. Note that this example uses a user-defined type for the completion parameter. It is possible through C type-casting to send in a single **long** or **int** value through this parameter instead of sending in a pointer to a complex datatype.

To use a completion handler with a completion handler parameter:

```

/* user-defined structure for data to pass */
typedef struct {
    int a;
    char b;
} user_compl_t;

/* user's completion handler */
void compl_hdlr(lapi_handle_t *hdl, void *completion_param)
{
    /*
    ** LAPI passes the uinfo parameter that was returned through the
    ** header handler in to the completion_param argument. We recast
    ** the parameter to be our defined type.
    */
    user_compl_t * compl_t = (user_compl_t *) completion_param;

    /* use data from the parameter, execute other statements, */
    /* then free the parameter */
}

```

```

/* user's header handler */
void *my_hdr_hdlr(lapi_handle_t *hdl, void *uhdr, uint *uhdr_len,
                 ulong *msg_len, compl_hdlr_t **comp_h,
                 void **uinfo) {

    user_compl_t *compl_t;

    /* malloc and store values in compl_t here; completion handler should free */

    /* this is how reference parameters are used to pass the pointers to LAPI */
    *comp_h = compl_hdlr;
    *uinfo = (void *) compl_t;

    return data_buffer;
}

```

A common use of the completion handler is to set a user-managed (as opposed to LAPI-managed) counter on which another thread is waiting. For example:

```

volatile int user_cntr;
:

/* user's completion handler */
void compl_hdlr(lapi_handle_t *hdl, void * completion_param)
{
    user_cntr++;
}

/* some other routine */
{

    /* some LAPI calls */

:

    while ( user_cntr < some_threshold ) {
        /* do other work */
    }

}

```

Completion handler execution: To ensure progress on other messages during completion handler execution, LAPI maintains a queue of completion handlers, which is serviced by a dedicated completion handler thread. Under normal execution, completion handler pointers are enqueued by the dispatcher at message completion and then executed within this separate thread. See Figure 7 on page 51.

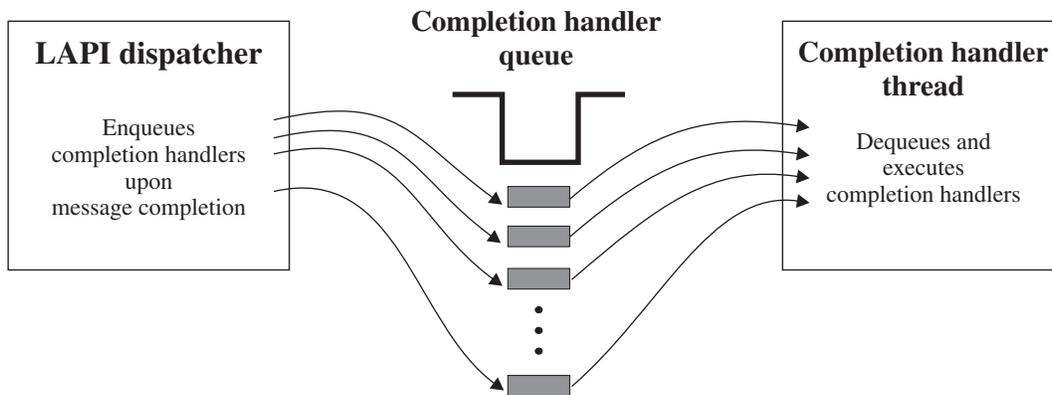


Figure 7. The completion handler queue

The use of a separate thread allows the dispatcher to make progress on other messages while the completion handler executes. If you set a completion handler using the header handler reference parameter, the LAPI dispatcher will enqueue the completion handler to the completion handler queue upon message completion.

If a number of handlers pile up, there may be some lag in completion handler execution. LAPI provides users with a mechanism for requesting that this process be short-circuited on a per-message basis and that completion handlers be run inline if possible. As discussed in “The enhanced header handler interface” on page 75, this can be done by setting the *ret_flags* field in the `lapi_return_info_t` structure to `LAPI_LOCAL_STATE` or `LAPI_SEND_REPLY`. See “Inline completion handlers” on page 77 for more information.

The send completion handler

LAPI provides users with the ability to provide a callback for send completion. This handler runs on the sending side. As with the (receive) completion handler, the use of a send completion handler is optional. If a send completion handler is specified by setting the corresponding field in the structure that is passed in to `LAPI_Xfer` to a non-NULL value, LAPI calls this handler when the send-side data buffer is available for reuse. Note that the send completion handler is only available through the `LAPI_Xfer` interface to the API calls. `LAPI_Xfer` also provides a parameter pointer for passing information in to the send completion handler. The handler’s signature is as follows:

```
void send_compl_hdlr(lapi_handle_t *hdl, void *completion_param, lapi_sh_info_t *info);
```

To use the send completion handler in C, set the *shdlr* pointer (and optionally, the *sinfo* pointer) of the `lapi_xfer_t` union of `LAPI_Xfer` to a non-NULL value. For example:

```
lapi_xfer_t xfer_struct;
some_user_type *info_ptr;

:

xfer_struct.Am.shdlr = (scompl_hdlr_t *) send_compl_hdlr;
xfer_struct.Am.sinfo = (void *) info_ptr;

:
LAPI_Xfer(hndl, &xfer_struct);
```

LAPI handler summary

Table 8 on page 52 provides summary information about the various handlers.

Table 8. LAPI handlers

Type of handler	Resides on:	Which subroutines use it?	How do I specify it?	How is it used?	When does it run?
Header	The target task	Required for LAPI_Amsend , LAPI_Amsendv , AM , AMV , and DGSP	Pass it as a callback pointer to the API call or (in C) pass it as a member of a lapi_xfer_t union for LAPI_Xfer . In addition to the callback pointer, optionally provide an index in to the header handler function table.	Must return the target-side data buffer address to LAPI. Optionally, sets completion handler and completion handler parameter pointers through reference parameters.	Upon arrival of the first data packet at the target task
Completion	The origin task	Optional for GET and GETV	Callback pointer set as a member of a lapi_xfer_t union for LAPI_Xfer (in C)	Data can be passed through a parameter pointer set as a member of a lapi_xfer_t union for LAPI_Xfer (in C)	Upon arrival of the last packet of the GET or GETV transfer
Completion	The target task	Optional for LAPI_Amsend , LAPI_Amsendv , AM , AMV , and DGSP	Callback pointer set in the header handler (user's option)	Data can be passed through a parameter pointer set in the header handler	Upon arrival of final data packet at the target task (message completion)
Send completion	The origin task	Optional for AM , AMV , DGSP , PUT , PUTV , and RMW	Callback pointer set as a member of a lapi_xfer_t union for LAPI_Xfer (in C). User's option.	Data can be passed through a parameter pointer set as a member of a lapi_xfer_t union for LAPI_Xfer (in C)	Upon availability of the send-side data buffer

LAPI counters

LAPI also provides a number of internally-managed counters, with a set of API calls for setting, getting and waiting on counter values. Three types of counters are provided:

Origin counter

Resides in the address space of the origin task. LAPI increments this counter when the send-side data buffer is available for reuse.

Target counter

Resides in the address space of the target task. LAPI increments this counter upon message completion at the target.

Completion counter

Resides in the address space of the sending side. LAPI increments this counter upon message completion at the target. LAPI sends an internal message back to the sender to indicate message completion.

LAPI provides three subroutines for counter manipulation:

LAPI_Getcntr(lapi_handle_t hndl, lapi_cntr_t *cntr, int *val)

Sets *val* to the value currently stored by *cntr*.

LAPI_Setcntr(lapi_handle_t hndl, lapi_cntr_t *cntr, int val)

Sets the value of the counter (*cntr*) to *val*.

LAPI_Waitcntr(lapi_handle_t hndl, lapi_cntr_t *cntr, int val, int *cur_cntr_val)

Blocks until the value of the counter (*cntr*) reaches *val*. **LAPI_Waitcntr** stores the current value of the counter in *cur_cntr_val* and decrements the counter's value by *val* before returning. The final value of the counter after this call does not need to be 0. (Consider, for example, a call to this subroutine after the value of the counter has already reached a value greater than *val*.)

Table 9 provides summary information about the various counters.

Table 9. LAPI counters

Type of counter	Resides on:	Which subroutines use it?	How do I specify it?	When is the counter incremented?
Origin	The origin task	LAPI_Amsend , LAPI_Amsendv , LAPI_Get , LAPI_Getv , LAPI_Put , LAPI_Putv , GET , GETV PUT , and PUTV	Pass it as an argument to an API call	Upon availability of the origin data buffer. For LAPI_Get , LAPI_Getv GET , and GETV : upon completion of message delivery. For LAPI_Rmw , LAPI_Rmw64 , and RMW : upon completion of return message delivery.
Target	The target task	LAPI_Amsend , LAPI_Amsendv , LAPI_Get , LAPI_Put , LAPI_Putv , GET , and PUT	Pass it as an argument to an API call	Upon completion of message delivery. For LAPI_Get , LAPI_Getv , GET , or GETV : upon availability of the target data buffer.
Completion	The origin task	LAPI_Amsend , LAPI_Amsendv , LAPI_Put , LAPI_Putv , and PUT	Pass it as an argument to an API call	Upon return of completion handler (if used), or on completion of message delivery (an internal message notifies the origin task in either case)

Specifying target-side addresses

When working with any target-side addresses for handlers, counters, or data buffers, a value in the target task's address space must be specified, often by the origin task. In those instances, it is important to remember that the address must be obtained from the target task. The normal mechanism for doing so is to use **LAPI_Address_init** or **LAPI_Address_init64**. If you are using the **LAPI_Xfer** interface for data communication, remote-side addresses are of type **lapi_long_t**, and must be obtained using **LAPI_Address_init64**.

Additional progress functions

LAPI provides functions for checking status while awaiting message delivery. Recall that progress is made in the LAPI dispatcher. When LAPI is used in polling mode (interrupts are turned off), it is possible that message packets have arrived, but the dispatcher does not get called (if nothing is being sent, for example). The user can invoke the dispatcher explicitly by calling **LAPI_Probe** on a LAPI handle. A call to **LAPI_Probe** will cause the dispatcher to check for the arrival of any message packets. See "LAPI_Probe" on page 174 for more information.

With the **LAPI_Msgpoll** subroutine, LAPI provides a means of running the dispatcher several times until either progress is made or a specified maximum

number of dispatcher loops have executed. Here, *progress* is defined as the completion of either a message send operation or a message receive operation. See “LAPI_Msgpoll” on page 171 for more information.

Chapter 7. Active messaging

The active message function (**LAPI_Amsend**) is a non-blocking call that causes the specified active message handler to be invoked and executed in the address space of the target process. Completion of the operation is signaled if counters are specified. Both standard and synchronous behaviors are supported. The **LAPI_Amsend** subroutine provides two counters: the origin counter (*org_cntr*) and the completion counter (*cmpl_cntr*), which can be used to provide the two behaviors. With standard behavior, LAPI increments the origin counter (*org_cntr*) when the origin buffer can be reused. With synchronous behavior, LAPI increments the completion counter (*cmpl_cntr*) after the completion handler has completed execution.

The **LAPI_Xfer** subroutine provides functions that are similar to **LAPI_Amsend**, with some enhancements. See “LAPI_Xfer” on page 219 for more information.

Flow of active message operations

Figure 8 on page 56 illustrates the sequence of events for a **LAPI_Amsend** operation or a **LAPI_Amsendv** operation. Because they use a header handler and a completion handler, active message operations are more complex than transfer operations. Upon arrival of the first packet, the user header handler is called. This handler is always run inline. For this reason, it is important to keep the body of the header handler small so that progress on other messages will not be blocked for long. The header handler returns a data buffer address to LAPI, which writes the data starting at that address. Once the final packet arrives, LAPI increments the target counter and either directly invokes or enqueues the completion handler for later execution in a completion handler thread. See “The completion handler” on page 49 for more information.

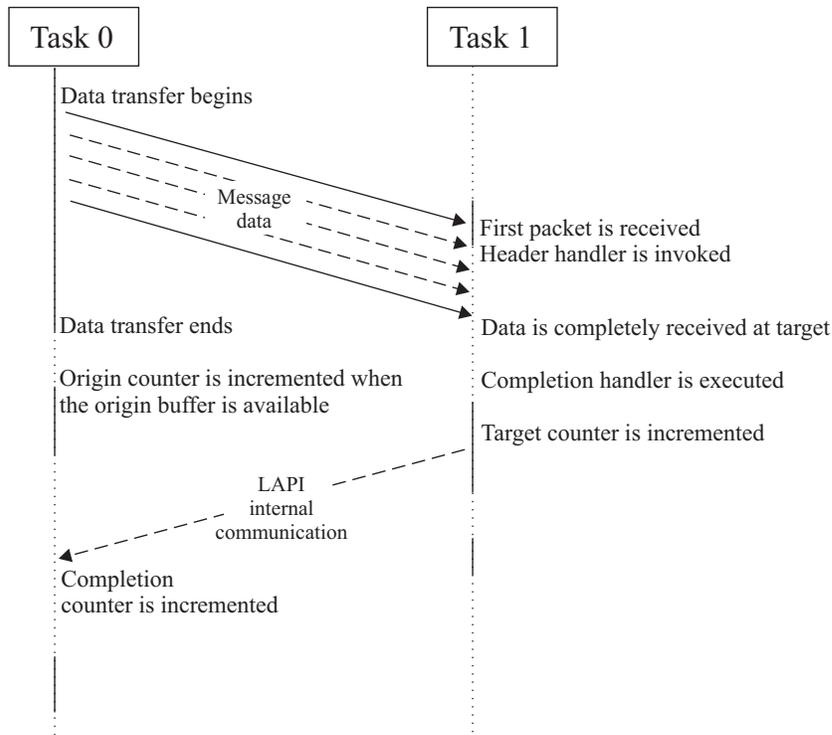


Figure 8. The sequence of events for an active message operation

Using LAPI_Amsend: a complete LAPI program

This section contains a complete listing of a LAPI program. A line-by-line description of the program follows this listing. The source code shown demonstrates a single **LAPI_Amsend** call, complete with header handler and completion handler definitions. This listing is a modified version of the sample program **Am.c**, which is available in the LAPI samples directory (`/opt/rsct/lapi/samples/lapi_api`). For more information, see Chapter 20, “LAPI sample programs,” on page 245.

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <lapi.h>
6
7 /* for any error messages returned by LAPI */
8 char err_msg_buf[LAPI_MAX_ERR_STRING];
9
10 /* constant for array lengths */
11 #define ARRAYLEN 10
12
13 /* list of header handler addresses */
14 void **hdr_hdlr_list;
15
16 /* stores initial value on src and final value on tgt */
17 int data_buffer[ARRAYLEN];
18
19 /* updates on src at msg completion */
20 lapi_cntr_t compl_cntr;
21
22 /*
23 ** completion handler that runs on the target after completion
  
```

```

24 ** of message delivery. prints the contents of the data
25 ** buffer that is the target of the LAPI_Amsend call.
26 */
27 void compl_hdlr(lapi_handle_t *hdl, void *completion_param)
28 {
29     int i;                /* loop counter */
30
31     printf("Greetings from the completion handler...\n");
32     for( i = 0; i < ARRAYLEN; i++ ) {
33         printf("final buffer[%d]: %d\n",i, data_buffer[i]);
34     }
35
36 }
37
38 /* header handler that runs on target when first packet arrives.
39 ** sets the completion_handler pointer and returns the address
40 ** of the data buffer for message delivery. LAPI writes the
41 ** Amsend data at this address.
42 */
43 void *header_handler(lapi_handle_t *hdl, void *uhdr, uint *uhdr_len,
44                     ulong *msg_len, compl_hdlr_t **comp_h,
45                     void **info) {
46
47     /* pass the completion handler back by reference */
48     *comp_h = compl_hdlr;
49
50     return data_buffer;
51 }
52
53
54 int main( int argc, char **argv )
55 {
56
57     lapi_handle_t hndl;          /* LAPI handle          */
58     lapi_info_t   info;         /* Info to pass to     */
59                               /* LAPI_Init           */
60     int          task_id;      /* Our LAPI task ID    */
61     int          num_tasks;    /* Total number of tasks */
62     int          i;           /* Loop counter        */
63     int          val;         /* Needed for waitcntr call */
64     int          buddy;      /* Our communication   */
65                               /* partner             */
66
67     /*
68     ** Clear the structure. Not passing any information
69     ** to initialize through this structure.
70     */
71     bzero(&info, sizeof(lapi_info_t));
72
73     /* Initialize the LAPI handle */
74     LAPI_Init(&hdl, &info);
75
76     /* Query LAPI for our task ID */
77     LAPI_Qenv(hndl, TASK_ID, &task_id);
78
79     /* Query LAPI for the total number of tasks in the job */
80     LAPI_Qenv(hndl, NUM_TASKS, &num_tasks);
81
82     /* This example only supports even numbers of tasks */
83     if ( (num_tasks < 2) || ((num_tasks % 2) != 0) ) {
84         fprintf(stderr,"ERROR: this example requires an even"
85                 "number of tasks, but has been invoked with"
86                 " %d\n", num_tasks);
87         exit(1);
88     }
89
90     /* get address of header handler on target */

```

```

91     hdr_hdlr_list = (void **)malloc(num_tasks*(sizeof(void *)));
92     LAPI_Address_init(hndl, (void *)header_handler,
93                       hdr_hdlr_list);
94
95     /*
96     ** up to this point, all instructions have executed on all
97     ** tasks. we now begin differentiating tasks.
98     */
99     if ((task_id %2) == 0) {           /* message source */
100
101         /* set up buddy pairs as (0,1), (2,3), and so on */
102         buddy = task_id+1;
103
104         /* initialize data buffer */
105         for (i = 0; i < ARRAYLEN; i++ ) {
106             data_buffer[i] = i*buddy;
107         }
108
109         /*
110         ** initialize the completion counter. When it increments,
111         ** we know that the completion handler has returned
112         */
113         LAPI_Setcptr(hndl,&compl_cntr,0);
114
115         /* collective call. synchronize before starting data transfer */
116         LAPI_Gfence(hndl);
117
118         /*
119         ** execute the data transfer to our buddy task. send
120         ** ARRAYLEN integers, starting with data_buffer[0].
121         */
122         LAPI_Amsend(hndl, buddy,
123                    hdr_hdlr_list[buddy], NULL,
124                    0,&(data_buffer[0]),ARRAYLEN*(sizeof(int)),
125                    NULL, NULL, &compl_cntr);
126
127         /* wait for message completion before final termination */
128         LAPI_Waitcptr(hndl, &compl_cntr,1, &val);
129
130     } else {                           /* receiver */
131                                         /* Note: buddy = task_id-1; */
132
133         /* initialize the data buffer (not necessary, */
134         /* but provided for clarity) */
135         for (i = 0; i < ARRAYLEN; i++ ) {
136             data_buffer[i] = 0;
137         }
138
139         /* to match the source's synchronization before data transfer */
140         LAPI_Gfence(hndl);
141     }
142
143     /* all tasks will execute this before termination */
144     LAPI_Gfence(hndl);
145
146     /* clean up */
147     free(hdr_hdlr_list);
148
149     /* terminate the LAPI handle */
150     LAPI_Term(hndl);
151
152     return 0;
153 }

```

Lines 1 to 5 include standard header files for LAPI programs.

Lines 7 and 8 declare a string buffer for retrieving messages that are associated

with non-zero LAPI return codes. Calling `LAPI_Msg_string(rc, err_msg_buf)` will store the message associated with `rc` in `err_msg_buf`.

- Line 11** defines a fixed data length for this example.
- Line 14** declares a pointer to maintain a list of header handler addresses for each task. The header handler runs on the target task, so the address that is specified for the header handler pointer must be in the target address space.
- Line 17** declares a simple data buffer. Because it is declared in common code, each task will have this buffer allocated in its address space. This program initializes a set of values in the sender task to this buffer, clears this set of values in the receiver task, then transfers the original set of values from the sender to the receiver.
- Line 20** specifies the only LAPI counter that this program will require for synchronization.
- Lines 22 to 36** specifies the completion handler that runs at the target upon completion of message delivery. When executed, this guarantees that all data from the transfer has been written to the desired target address. In this example, the program simply prints the results. In practice, completion handlers can be used for many things, including additional data manipulation or synchronization (such as by incrementing counters that the user is managing internally). Completion handlers are not required for active messaging calls. If no completion handler is desired, simply pass NULL to the completion handler pointer in the user header handler.
- Lines 38 to 51** specifies the user header handler that LAPI will call when the first packet arrives at the target. At a minimum, it must return to LAPI the buffer address at which data is to be written. It can also return (by reference parameters) a pointer to the user's completion handler and a pointer to a structure containing any information to pass to the completion handler when it is invoked.
- Header handlers are always run inline. That is, progress on any other messages is blocked until the header handler completes. For this reason it is advised that the user keep execution time in the header handler to a minimum.
- Line 54** Main begins here. This example is short: only one LAPI communication operation is done before the program exits. For this reason, it is not a problem to keep all of the program flow in main. Recall that a separate application of this code will be executed for each task. As such, all code will execute on all tasks. This program branches on task ID at line 130 to ensure that certain code only executes on one set of tasks, and the other branch executes only on the other set of tasks. After the branched sections, control rejoins and the remaining code is executed on all tasks. It is helpful to think of this fork/join model as you write LAPI programs.
- Lines 57 to 65** include variable declarations. The handle is the user's abstraction of a LAPI instance. After initialization (line 74 in this case), the handle is used to interact with the LAPI subsystem. Users can use the

handle to query runtime parameters and execute data transfers. As part of cleanup, the handle should be properly terminated (line 150).

The `lapi_info_t` structure is used to pass certain information to `LAPI_Init`. Most often, it is used for passing a pointer to the user's error handler. Certain fields in the structure are not used; LAPI will return an error if these fields contain values. For this reason, it is important to clear the structure (line 71), and then set any desired values. This example does not use any of the fields, so the program leaves the memory cleared. See "Registering an error handler" on page 32 for an example of using the `lapi_info_t` structure to register a user error handler.

LAPI allows us to query our task ID. At line 99, the code stream splits. One branch will execute on all tasks with even-numbered IDs (`task_id %2 == 0`), and the other branch will execute on tasks with odd IDs.

In this example, task IDs are used to set up buddy pairs for data transfer. Buddy pairs are established as (0,1), (2,3), and so on, such that the lower task ID in the pair is the even number. The lower (even) task ID drives the communication by sending a message to the odd-numbered task.

The buddy-pair concept is a specific case of task grouping, where one task is the "master" for a group of tasks. The "master task" drives all communication and synchronization. It is possible to have master tasks for groups of larger cardinalities. One common model is to use task 0 as a master task and all other tasks as "slave tasks".

Similar to `task_id`, LAPI has a mechanism for querying the total number of tasks in the job. Because this example communicates in task pairs, it requires an even number of tasks. This program uses the value queried from LAPI to check the number of tasks on line 83 and exits if this value is not even.

The remaining, basic variables are needed for various tasks.

Line 71 clears the fields in the `lapi_info_t` structure before passing it to LAPI. See the description of lines 57 to 65 for information about `lapi_info_t`.

Line 74 initializes the LAPI handle. This call creates a LAPI instance and initializes the communication subsystem. After this call completes, any query and communication API calls can be performed on this handle, until this handle is terminated.

Lines 76 to 80 Lines 77 and 80 are executed in most programs that use LAPI, because the task ID and the total number of tasks in the job are commonly-used values. See the description of lines 57 to 65 for more information on how these values are used in this example.

Lines 82 to 88 check to make sure that the number of tasks is even. See the description of lines 57 to 65 for more information on task topology.

Lines 90 to 93 Recall that the header handler executes on the target, but the

address of the function must be known to the source task. This program uses the **LAPI_Address_init** subroutine to exchange header handler address data among all of the tasks.

LAPI_Address_init expects a buffer with enough space to hold an address for every task in the job, so this program does the required **malloc** first at line 91, then frees it at line 147. Note that since we are still in common code, the **malloc** and **LAPI_Address_init** call will be executed at this point on every task, which is what we want.

Line 99 Control forks here. This branch will only be entered by even-numbered tasks (the "drivers" in this example). The branch for odd-numbered tasks begins at line 130.

Line 102 Because this task will drive, it needs to keep track of its buddy's task ID.

Line 104 populates the buffer with data to be transferred. The values are relative to the ID of the buddy task so they can be easily and uniquely verified after being received by the buddy.

Lines 109 to 113

LAPI communication API calls are non-blocking; message delivery is asynchronous. LAPI provides counters that signify different events. In this example, we want to know when message delivery at our buddy is complete. For this purpose, we use a local completion counter, which LAPI will increment once the completion handler on the target returns. We will wait on this counter in a **LAPI_Waitcntr** call at line 128, immediately after the **LAPI_Amsend** call on line 122 returns.

Line 116 This is a collective data fence operation. All calls that are made before this point have no asynchronous side effects and thus are guaranteed to complete before we leave this fence. Our buddy has a matching **LAPI_Gfence** call at line 140.

Lines 118 to 125

This is the actual communication API call. As above, all LAPI calls are made with respect to the handle returned from **LAPI_Init**. We are sending data to our buddy task. For the header handler pointer, we pass the entry returned from the collective **LAPI_Address_init** call on line 92. The address passed into the call by our buddy task is stored at index `buddy` in the table.

We are not using a user header in this example, so the next two arguments (the user header and user header length) are `NULL` and `0`, respectively. See the sample program **vector/accumulate_and_return.Amv.c** for an example of using a user header.

The next argument is the base data address from which data is to be read (`&(data_buffer[0])`) followed by the number of bytes to transfer (`ARRAYLEN*(sizeof(int))`).

The final three arguments are pointers to counters that LAPI will manage. As mentioned above, we are only concerned in this case with the completion counter. So we pass `NULL` for both the target and origin counters. The target counter will increment on the target when message delivery is complete, so if used, requires a remote address to be passed. See the sample program **xfer/Put_xfer.c** for an example of using a target counter.

When used, the origin counter increments locally when the origin data buffer is no longer needed by LAPI and can thus be reused by the origin task. See the **vector/strided.c** example for demonstration of origin counter usage.

As mentioned above, we are using the completion counter in this case and so pass its address.

- Line 128** We wait here for LAPI to signal message completion. This completes the branch specific to even-numbered tasks.
- Line 130** This begins the branch that will only be executed by odd-numbered (receiver) tasks.
- Line 131** Our buddy is driving the communication, so we do not need its ID. This comment is just left in as a reminder.
- Lines 135 to 137** Similarly, we zero out the data space here so that we know that the data in the final output must have come from a remote task.
- Line 140** This call matches the sender's **LAPI_Gfence** call on line 116.
- Line 141** Code paths re-join here. All remaining code is executed on both tasks.
- Lines 143 to 150** represent a standard LAPI shutdown sequence. All tasks execute a final **LAPI_Gfence** call (line 144), then clean up any allocated data structures (line 147). Any frees should be done before executing **LAPI_Term**.
- Line 152** Finally, we return success. If POE is running, it will pass this code back to AIX for the process return code.

Chapter 8. Collecting statistics

You can use the **LAPI_Qenv** subroutine to print and query LAPI statistics. See “LAPI_Qenv” on page 184 for more information about this subroutine.

Printing data transfer statistics

When passed the **PRINT_STATISTICS** query type, **LAPI_Qenv** sends data transfer statistics to standard output. In this case, *ret_val* is unaffected. However, LAPI’s error checking requires that the value of *ret_val* is not NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN) for all **LAPI_Qenv** types (including **PRINT_STATISTICS**).

Querying US and UDP/IP statistics

When passed the **QUERY_STATISTICS** query type, **LAPI_Qenv** interprets *ret_val* as a pointer to type **lapi_statistics_t**. Upon function return, the fields of the structure contain LAPI’s data transfer statistics for data transferred using the user space (US) protocol or UDP/IP. **QUERY_STATISTICS** expects a pointer to type **lapi_statistics_t**. The address of **stats** is cast to **int ***, which is required to match the signature of **LAPI_Qenv**. For example:

```
{
lapi_statistics_t stats;

LAPI_Qenv(handle, QUERY_STATISTICS, (int *)&stats));
}
```

Querying local send statistics

When passed the **QUERY_LOCAL_SEND_STATISTICS** query type, **LAPI_Qenv** interprets *ret_val* as a pointer to type **lapi_statistics_t**. Upon function return, the fields of the structure contain LAPI’s data transfer statistics for data transferred through intra-task local copy. For example:

```
{
lapi_statistics_t stats;

LAPI_Qenv(handle, QUERY_LOCAL_SEND_STATISTICS, (int *)&stats));
}
```

With this query, you can obtain the byte count of the data that is transferred through the local copy path when the source and target are the same. The following statistics are reported: the number of bytes sent and the number of bytes received for the local copy path, the number of packets sent, and the number of packets received. The number of bytes sent for a message are counted only for data messages that are sent successfully. The number of bytes received for a message are counted only for data messages that complete successfully. As with adapter statistics, LAPI also provides separate sets of statistics for LAPI-only traffic and shared traffic in the local copy path. The packet count reported for the local copy path will always be **0** because data transfer in this path does not use packetization.

Querying shared memory statistics

When passed the **QUERY_SHM_STATISTICS** query type, **LAPI_Qenv** interprets *ret_val* as a pointer to type **lapi_statistics_t**. Upon function return, the fields of the structure contain LAPI's data transfer statistics for data transferred through shared memory. **QUERY_SHM_STATISTICS** expects a pointer to type **lapi_statistics_t**. For example:

```
{
lapi_statistics_t stats;

LAPI_Qenv(handle, QUERY_SHM_STATISTICS, (int *)&stats));
}
```

With this query, you can obtain the byte count of the data that is transferred through the shared memory path when the source and target are the same. The following statistics are reported: the number of bytes sent and the number of bytes received for the shared memory path, the number of packets sent, the number of packets received, and the number of retransmit packets. A packet in shared memory path refers to a shared memory slot when using slot transfer or a shared memory attachment when using attach transfer. The number of retransmit packets in the shared memory path refers to the number of times when attaching a shared memory segment fails. The number of bytes sent via shared memory is defined as all the data bytes sent through shared memory either by slot path or attach path. The number of bytes sent for a message are counted only for data messages that are sent successfully. If attaching a shared memory segment does not succeed for transferring a message and that message has been failed over to the slot path, only slot path data bytes are counted into the number of bytes sent. The number of bytes received for a message are counted only for data messages that complete successfully. As with adapter statistics, LAPI also provides separate sets of statistics for LAPI-only traffic and shared traffic in the shared memory path.

See “LAPI shared memory: functional flow” on page 259 for more information.

Chapter 9. Using LAPI's profiling interface

LAPI's profiling interface includes wrappers for each LAPI function, so you can collect data about each of the LAPI calls. For example, you can write a program that records the message size that is used in each call. This interface supports applications that are written in C, C++, and FORTRAN.

Each LAPI subroutine has a "name-shifted" interface. Suppose your program calls a LAPI subroutine, such as **LAPI_Get**. When you link your program with the LAPI library, the LAPI function call results in a call to the corresponding name-shifted LAPI subroutine, in this case **PLAPI_Get**, using the same parameters and returning the same function result. You can create a profiling library that contains function implementations for each LAPI subroutine that it will override. When you link your program with a profiling library, this library calls the corresponding name-shifted LAPI subroutine without modifying any parameters and returns the same function result. When LAPI function calls are profiled, a wrapper (for each LAPI function to be profiled) collects the profiling data and calls the name-shifted LAPI subroutine exactly as if it was called by your program.

Table 10 lists LAPI's profiling interfaces.

Table 10. LAPI profiling interfaces

LAPI subroutine	C and C++ profiling interface	FORTRAN profiling interfaces
LAPI_Addr_get	PLAPI_Addr_get	plapi_addr_get, plapi_addr_get_, PLAPI_ADDR_GET, PLAPI_ADDR_GET_
LAPI_Addr_set	PLAPI_Addr_set	plapi_addr_set, plapi_addr_set_, PLAPI_ADDR_SET, PLAPI_ADDR_SET_
LAPI_Address	PLAPI_Address	plapi_address, plapi_address_, PLAPI_ADDRESS, PLAPI_ADDRESS_
LAPI_Address_init	PLAPI_Address_init	plapi_address_init, plapi_address_init_, PLAPI_ADDRESS_INIT, PLAPI_ADDRESS_INIT_
LAPI_Address_init64	PLAPI_Address_init64	plapi_address_init64, plapi_address_init64_, PLAPI_ADDRESS_INIT64, PLAPI_ADDRESS_INIT64_
LAPI_Amsend	PLAPI_Amsend	plapi_amsend, plapi_amsend_, PLAPI_AMSEND, PLAPI_AMSEND_
LAPI_Amsendv	PLAPI_Amsendv	plapi_amsendv, plapi_amsendv_, PLAPI_AMSENDV, PLAPI_AMSENDV_
LAPI_Fence	PLAPI_Fence	plapi_fence, plapi_fence_, PLAPI_FENCE, PLAPI_FENCE_
LAPI_Get	PLAPI_Get	plapi_get, plapi_get_, PLAPI_GET, PLAPI_GET_
LAPI_Getcntr	PLAPI_Getcntr	plapi_getcntr, plapi_getcntr_, PLAPI_GETCNTR, PLAPI_GETCNTR_
LAPI_Getv	PLAPI_Getv	plapi_getv, plapi_getv_, PLAPI_GETV, PLAPI_GETV_
LAPI_Gfence	PLAPI_Gfence	plapi_gfence, plapi_gfence_, PLAPI_GFENCE, PLAPI_GFENCE_
LAPI_Init	PLAPI_Init	plapi_init, plapi_init_, PLAPI_INIT, PLAPI_INIT_
LAPI_Msg_string	PLAPI_Msg_string	plapi_msg_string, plapi_msg_string_, PLAPI_MSG_STRING, PLAPI_MSG_STRING_

Table 10. LAPI profiling interfaces (continued)

LAPI subroutine	C and C++ profiling interface	FORTTRAN profiling interfaces
LAPI_Msgpoll	PLAPI_Msgpoll	plapi_msgpoll, plapi_msgpoll_, PLAPI_MSGPOLL, PLAPI_MSGPOLL_
LAPI_Nopoll_wait	PLAPI_Nopoll_wait	plapi_nopoll_wait, plapi_nopoll_wait_, PLAPI_NOPOLL_WAIT, PLAPI_NOPOLL_WAIT_
LAPI_Probe	PLAPI_Probe	plapi_probe, plapi_probe_, PLAPI_PROBE, PLAPI_PROBE_
LAPI_Purge_totask	PLAPI_Purge_totask	plapi_purge_totask, plapi_purge_totask_, PLAPI_PURGE_TOTASK, PLAPI_PURGE_TOTASK_
LAPI_Put	PLAPI_Put	plapi_put, plapi_put_, PLAPI_PUT, PLAPI_PUT_
LAPI_Putv	PLAPI_Putv	plapi_putv, plapi_putv_, PLAPI_PUTV, PLAPI_PUTV_
LAPI_Qenv	PLAPI_Qenv	plapi_qenv, plapi_qenv_, PLAPI_QENV, PLAPI_QENV_
LAPI_Resume_totask	PLAPI_Resume_totask	plapi_resume_totask, plapi_resume_totask_, PLAPI_RESUME_TOTASK, PLAPI_RESUME_TOTASK_
LAPI_Rmw	PLAPI_Rmw	plapi_rmw, plapi_rmw_, PLAPI_RMW, PLAPI_RMW_
LAPI_Rmw64	PLAPI_Rmw64	plapi_rmw64, plapi_rmw64_, PLAPI_RMW64, PLAPI_RMW64_
LAPI_Senv	PLAPI_Senv	plapi_senv, plapi_senv_, PLAPI_SENV, PLAPI_SENV_
LAPI_Setcptr	PLAPI_Setcptr	plapi_setcptr, plapi_setcptr_, PLAPI_SETCPTR, PLAPI_SETCPTR_
LAPI_Setcptr_wstatus	PLAPI_Setcptr_wstatus	plapi_setcptr_wstatus, plapi_setcptr_wstatus_, PLAPI_SETCPTR_WSTATUS, PLAPI_SETCPTR_WSTATUS_
LAPI_Term	PLAPI_Term	plapi_term, plapi_term_, PLAPI_TERM, PLAPI_TERM_
LAPI_Util	PLAPI_Util	plapi_util, plapi_util_, PLAPI_UTIL, PLAPI_UTIL_
LAPI_Waitcptr	PLAPI_Waitcptr	plapi_waitcptr, plapi_waitcptr_, PLAPI_WAITCPTR, PLAPI_WAITCPTR_
LAPI_Xfer	PLAPI_Xfer	plapi_xfer, plapi_xfer_, PLAPI_XFER, PLAPI_XFER_

Performing name-shift profiling

To use name-shift profiling routines that are either written to the C bindings with a LAPI program written in C, or that are written to the FORTRAN bindings with a LAPI program written in FORTRAN, use the following steps.

Programs that use LAPI's C language bindings can create profiling libraries using the name-shifted interface.

- If you are both the creator and user of the profiling library and you are not using FORTRAN, follow steps 1 through 6. If you are using FORTRAN, follow steps 1 through 4, then steps 7 through 9.
- If you are the creator of the profiling library, follow steps 1 through 4. You also need to provide the user with the file created in step 3.
- If you are the user of the profiling library and you are not using FORTRAN, follow steps 5 and 6. If you are using FORTRAN, start at step 7. You will need to make sure that you have the file generated by the creator in step 3.

To perform LAPI name-shift profiling, follow the appropriate steps:

1. Create a source file that contains profiling versions of all of the LAPI subroutines you want to profile. For example, create a source file called **myprof_r.c** that contains the following code:

```
#include <pthread.h>
#include <stdio.h>
#include <lapi.h>
int LAPI_Init(lapi_handle_t *hdl, lapi_info_t *lapi_info) {
    int rc;
    printf("Hello, from profiling layer LAPI_Init ...\n");
    rc = PLAPI_Init(hndl, lapi_info);
    printf("goodbye, from profiling layer LAPI_Init ...\n");
    return rc;
}
int lapi_init(lapi_handle_t *hdl, lapi_info_t *lapi_info, int *err) {
    int rc;
    printf("Hello, from profiling layer LAPI_Init ...\n");
    rc = plapi_init(hndl, lapi_info, err);
    printf("goodbye, from profiling layer LAPI_Init ...\n");
    return rc;
}
```

2. Compile the source file that contains your profiling LAPI routines. For example, to compile the profiling source file you created in step 1:

```
cc_r -c myprof_r.c -I/usr/include
```

The **-I** flag defines the location of **lapi.h**.

3. Create an export file that contains all of the symbols your profiling library will export. Begin this file with the name of your profiling library and the name of the **.o** file that will contain the object code of your profiling routines. For example, to create an export file for the profiling source file that you created in step 1, create a file called **myprof_r.exp** that contains this statement:

```
LAPI_Init
```

4. Create a shared library called **libmyprof_r.a** that contains the profiled versions, exporting their symbols and linking with the LAPI library, using **myprof_r.exp** as shown. For example:

```
ld -o newmyprof_r.o myprof_r.o -bnoentry -bE:myprof_r.exp -lc_r -L/usr/lib \
    -llapi_r -lpthreads
ar rv libmyprof_r.a newmyprof_r.o
```

5. Link your user program:

```
mpcc_r -o test1 test1.c -L. -lmyprof_r
```

6. Run the resulting executable.

7. For a FORTRAN program using a LAPI function that you have profiled in step 1, create a file called **hwinit.f** that contains the following statements:

```
include 'lapif.h'
integer :: handle
integer :: info(25)
integer :: ierror, i
do i=1, 25
    info(i) = 0
enddo
call LAPI_INIT(handle, info, ierror)
write(*,*) ierror
call LAPI_TERM(handle, ierror)
stop
end
```

8. Compile your FORTRAN program (**hwinit.f**) using the new library (**libmyprof_r.a**):

```
mpxlf_r -o hwinit hwinit.f -L. -lmyprof_r
```

9. Run the resulting executable.

A sample profiling program

A sample profiling program follows. Before it is linked with the LAPI library, the user program should be linked with a profiling library that is formed from the following program, using the instructions in “Performing name-shift profiling” on page 66. Then, the time spent in **LAPI_Xfer** and **LAPI_Msgpoll** will be printed out when the user program terminates. Note that this sample program is not multithread-safe.

```
#include <stdio.h>
#include <sys/time.h>
#include <lapi.h>

typedef struct timeval prof_time_t;

prof_time_t xfer_time;
prof_time_t poll_time;

void accumulate(prof_time_t *sum, prof_time_t *start, prof_time_t *stop)
{
    sum->tv_sec += stop->tv_sec - start->tv_sec;
    sum->tv_usec += stop->tv_usec - start->tv_usec;
    if (sum->tv_usec >= 1000000) {
        sum->tv_sec += 1;
        sum->tv_usec -= 1000000;
    }
    if (sum->tv_usec < 0) {
        sum->tv_sec -= 1;
        sum->tv_usec += 1000000;
    }
}

int LAPI_Init(lapi_handle_t *hndl, lapi_info_t *lapi_info)
{
    bzero(&xfer_time, sizeof(xfer_time));
    bzero(&poll_time, sizeof(poll_time));
    return PLAPI_Init(hndl, lapi_info);
}

int LAPI_Xfer(lapi_handle_t hndl, lapi_xfer_t *xfer_cmd)
{
    int rc;
    prof_time_t start, stop;

    gettimeofday(&start, NULL);
    rc = PLAPI_Xfer(hndl, xfer_cmd);
    gettimeofday(&stop, NULL);
    accumulate(&xfer_time, &start, &stop);
    return rc;
}

int LAPI_Msgpoll(lapi_handle_t hndl, uint cnt, lapi_msg_info_t *info)
{
    int rc;
    prof_time_t start, stop;

    gettimeofday(&start, NULL);
    rc = PLAPI_Msgpoll(hndl, cnt, info);
    gettimeofday(&stop, NULL);
    accumulate(&poll_time, &start, &stop);
    return rc;
}

int LAPI_Term(lapi_handle_t hndl)
{
    printf("LAPI_Xfer time: %u h %u m %u s %06u us\n",
           xfer_time.tv_sec/3600, xfer_time.tv_sec%3600/60,
```

```
|         xfer_time.tv_sec%60, xfer_time.tv_usec);  
| printf("LAPI_Msgpoll time: %u h %u m %u s %06u us\n",  
|         poll_time.tv_sec/3600, poll_time.tv_sec%3600/60,  
|         poll_time.tv_sec%60, poll_time.tv_usec);  
| return PLAPI_Term(hndl);  
|     }  
  
|
```


Chapter 10. Compiling and running LAPI programs

As with a serial application, you must compile a parallel C, C++, or FORTRAN program before you can run it. The commands shown in Table 11 support programs that use the threaded LAPI library. These commands also link in the POE partition manager and AIX communication subsystem (CSS) interfaces.

Table 11 shows what commands to enter to compile a LAPI program on a system that is running PE.

Table 11. Compiling LAPI programs on a system that is running PE

To compile a C program	mpcc_r program.c -o program
To compile a C++ program	mpCC_r program.C -o program
To compile a FORTRAN program	mpxlf_r program.f -o program

See Chapter 16, “Using LAPI on a standalone system,” on page 115 for information about compiling LAPI programs on a system that is not running PE.

If you compiled your program using one of the commands shown in Table 11, the CSS libraries are dynamically linked with the executable when you run your program. Subroutines in these libraries enable POE’s home node (the node from which the parallel program is invoked) to communicate with the parallel tasks, and tasks with each other.

Part 3. Advanced LAPI tasks

Chapter 11. Advanced programming	75
The enhanced header handler interface	75
Inline completion handlers	77
LAPI performance considerations	78
Use of handlers	78
Running in interrupt mode	78
Running in UDP/IP mode	78
User header data	79
Send-side copy of small messages	79
Receive-side optimization for single-packet messages	79
Tunable environment variables	80
32-bit and 64-bit interoperability	81
The lapi_long_t datatype	81
The LAPI_Address_init64 subroutine	81
The LAPI_Xfer interface	81
Chapter 12. Lock sharing	83
Scenarios without lock sharing	83
Scenarios with lock sharing	87
Correctness of lock sharing	91
Implications and restrictions	93
Initialization and termination	93
Other LAPI calls	94
Callbacks	94
Long critical sections	94
Lock preemption	94
Receive/timer interrupts	94
Performance of multi-threaded programs	95
Compatibility	95
A sample lock sharing program	95
Chapter 13. Bulk transfer of messages	99
Chapter 14. Striping, failover, and recovery	103
Using failover and recovery	103
Monitoring adapter status	103
Network Availability Matrix (NAM) overview	103
RSCT peer domains and group services	104
Requesting the use of multiple adapters	104
Using POE environment variables	104
Using LoadLeveler JCF keywords	106
Failover and recovery restrictions	106
Data striping	106
Communication and memory considerations	108
IP communication	108
US communication	109
Chapter 15. Threaded programming	111
General guidelines	111
Using LAPI_Address_init	111
Making global fence calls	112
Making "wait on counter" calls	112
Synchronizing threads across tasks	112

Using handlers	112
LAPI threads	113
Chapter 16. Using LAPI on a standalone system	115
Standalone setup	115
Standalone initialization	117
Using UDP/IP mode	117
Using US mode	117
Compiling LAPI programs on a standalone system	118

Chapter 11. Advanced programming

The enhanced header handler interface

In PSSP LAPI, the header handler was originally implemented as follows:

```
typedef void *(hdr_hdlr_t)(lapi_handle_t *hdl, void *uhdr, uint *uhdr_len,
                          ulong *msg_len, compl_hdlr_t **compl_h, void **uinfo);
```

LAPI passes the LAPI context, user header address, user header length, and message length in to a header handler. From the header handler, you can pass back to LAPI a data buffer address, a completion handler, and any user-defined information for the completion handler.

In order to allow more information exchange between LAPI and the user program, the *msg_len* parameter type has been changed from **ulong *** to **lapi_return_info_t ***, which is a pointer to the following structure:

```
typedef struct {
    ulong      msg_len;
    int        MAGIC;
    lapi_ret_flags_t  ret_flags;
    lapi_ctl_flags_t  ctl_flags;
    lapi_dg_handle_t  dgsp_handle;
    ulong      bytes;
    int        src;
    void       *udata_one_pkt_ptr;
} lapi_return_info_t;
```

Note: This applies to new or modified LAPI programs only.

With this enhanced interface, LAPI has the following new capabilities:

- running the completion handler inline as opposed to in a separate thread, which significantly improves performance (see “Inline completion handlers” on page 77)
- options for the user to drop the message instead of delivering it (described in this section)
- transferring data in the layout described by data gather-scatter programs (DGSPs) (see “Using data gather/scatter programs (DGSPs)” on page 43)
- optimization for receiving one-packet messages (see “Receive-side optimization for single-packet messages” on page 79)

This extension does not require any change in existing LAPI programs that were coded and compiled with the original header handler interface. To these programs, the header handler interface remains the same because 1) *msg_len* is the first field in the **lapi_return_info_t** structure and 2) all other fields are set to default values by LAPI before calling a header handler, so LAPI behaves exactly the same as it does without the extension if none of the fields is changed by the header handler.

The prototype of the header handler remains the same in LAPI’s header files (as it is shown above), but you optionally use the following extended interface to define header handlers:

```
void *header_handler(lapi_handle_t *hdl, void *uhdr, uint *uhdr_len,
                    lapi_return_info *msg_len, compl_hdlr_t **compl_h, void **uinfo);
```

LAPI allocates the **lapi_return_info_t** structure, initializes it with appropriate values, and passes a pointer to the structure into the header handler. You can then alter the fields of this structure as needed to convey specific information back to LAPI.

The `lapi_return_info_t` structure includes the following fields:

Field (input or output)	Description
<code>msg_len</code> (IN)	Specifies the length of the incoming message (for compatibility with the original header handler interface).
MAGIC (IN)	Indicates the integrity of the structure. The user must not change this field.
<code>ret_flags</code> (OUT)	Tells LAPI how to run the completion handler (if any).
<code>ctl_flags</code> (OUT)	Tells LAPI to deliver, "bury", or drop the incoming message.
<code>dgsp_handle</code> (OUT)	Tells LAPI to deliver the message according to the specified DGSP. If this field is NULL (the default), LAPI delivers the message to the contiguous buffer returned by the header handler.
<code>bytes</code> (OUT)	Tells LAPI the number of bytes to deliver when a DGSP is used. This value can be less than the message length, but should not be greater.
<code>src</code> (IN)	Specifies the source task ID of the message.
<code>udata_one_pkt_ptr</code> (IN)	Specifies the pointer to the incoming data if this is a one-packet message. Otherwise, the value is set to NULL.

You can set `ret_flags` to the following values to instruct LAPI how to call the completion handler for the incoming message:

LAPI_NORMAL	Instructs LAPI to run the completion handler in the completion handler thread.
LAPI_LOCAL_STATE	Instructs LAPI to run the completion handler in the same thread that is receiving the message. (You guarantee that the completion handler does not make any LAPI calls.)
LAPI_SEND_REPLY	Instructs LAPI to run the completion handler in the same thread that is receiving the message. (You want to send a reply to the message by calling LAPI.)

In addition, setting `ctl_flags` tells LAPI how to handle the incoming message besides delivering it:

LAPI_DELIVER_MSG	delivers the message as normal (the default).
LAPI_BURY_MSG	does not copy data for the message. (From the perspective of the origin side, there is no difference between this setting and LAPI_DELIVER_MSG .)
LAPI_DROP_PKT	ignores the message as if it never arrives. The dropped packet will be retransmitted.

Warning: Every time a packet is dropped, it will eventually be retransmitted. If the packet is not accepted on some follow-up retransmit, the entire communication link will stall. Furthermore, the

packet acceptance cannot depend on the arrival of other messages, because these messages may be stalled by the dropped packet.

Note: You *cannot* make LAPI calls from within the header handler. For contiguous data, you can copy the data to the buffer you specified. For non-contiguous data, you must pass the DGSP handle and a buffer address to LAPI. LAPI will unpack the data to the specified buffer address.

Inline completion handlers

You can prioritize the completion of certain messages by requesting that your completion handler be run inline. LAPI runs the completion handler inline (if possible) if you set the *ret_flags* field in the `lapi_return_info_t` structure to `LAPI_SEND_REPLY` or `LAPI_LOCAL_STATE`. This flag is returned to LAPI as a reference parameter in the header handler you provided.

If you request that your completion handler be run inline, LAPI attempts to acquire the necessary send token before completion handler execution is attempted. If you specified `LAPI_SEND_REPLY`, a check is made for send tokens. Only in the case of `LAPI_SEND_REPLY` is it necessary to check for send tokens, because you must not call a LAPI routine in the completion handler in the case of `LAPI_LOCAL_STATE`.

If LAPI is successful in getting its internal send token without polling for a token to free up, or if you specified `LAPI_LOCAL_STATE`, the completion handler is run inline. Otherwise, LAPI enqueues the completion handler in the separate completion handler thread. See Figure 9:

See “LAPI_Amsend” on page 136 for more information and an example.

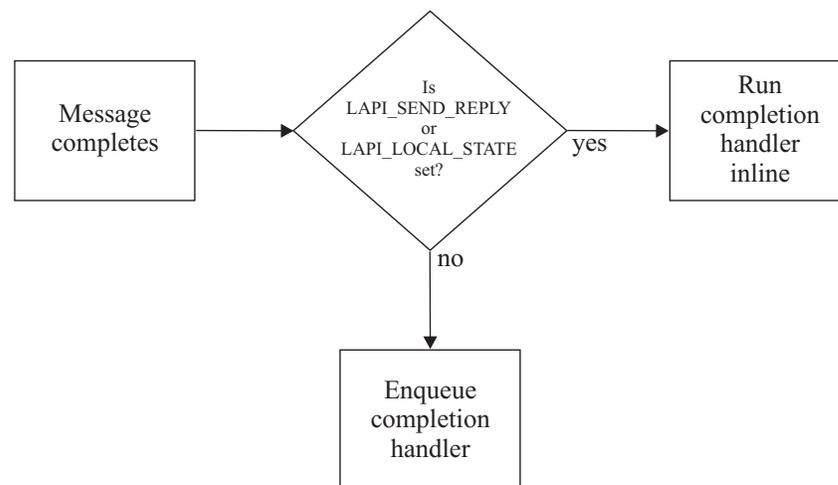


Figure 9. Inline completion handler flow

I/O operations and blocking calls, including blocking LAPI calls, should *not* be performed within an inline completion handler. Inline completion handlers should be short, because no progress can be made while the main thread is executing the handler. You must use caution with inline completion handlers so that LAPI’s internal queues do not fill up while waiting for the handler to complete. Note that LAPI places no restrictions on completion handlers that are run “normally” (that is, by the completion handler thread).

LAPI performance considerations

LAPI provides a one-sided programming model with a pseudo shared-memory view of multi-task operations on distributed servers. For this reason, it is expected that users will find the best performance when LAPI is used with interrupts turned on (the default value). Indeed, LAPI message-passing calls communicate with a remote task without the need for that task's active participation in the communication. The use of interrupts frees both the origin and target-side tasks from the need to poll for buffer availability and message completion, respectively.

Use of handlers

There are a few points in LAPI's message transfer sequence in which user notification can occur. In most instances, LAPI provides notification through the incrementing of a counter or the invocation of a user-supplied handler. LAPI's one-sided model tends to perform better through the use of handlers for notification, because they involve the actual execution of user code without any explicit action on the part of the task in question. The use of counters requires the task to poll on the counter value. This could needlessly consume CPU cycles, depending on what else the application has available for the CPU to work on and how counter checking is performed.

One counter-example to the philosophy that the use of handlers provides better performance is as follows. When running in a fully-loaded configuration (n tasks on n CPUs), LAPI may perform better in polling mode (interrupts off). One explanation for this anomaly has to do with the scheduling of the thread that handles interrupts.

Running in interrupt mode

LAPI runs in interrupt mode by default. Interrupts cause thread context switching, which can adversely affect performance. For optimal performance, the user program should turn off interrupt mode when entering a polling section and turn it on again when exiting the polling section. To turn interrupt mode off and on, call **LAPI_Senv** with the **INTERRUPT_SET** attribute. See "LAPI_Senv" on page 196 for more information. Whenever the user program transfers control to LAPI by calling a LAPI function, LAPI will drive communication traffic, so interrupts are not necessary. Interrupts will only introduce the overhead of thread context switching if LAPI is already polling.

For jobs running in interrupt mode, performance gains may be realized by setting environment variables **AIX_THREAD_SCOPE=S** and **RT_GRQ=ON**. There is also evidence to suggest that the performance of $n-1$ tasks on n CPUs is considerably better than that of a fully-loaded system.

Running in UDP/IP mode

When LAPI is running over UDP/IP, it requires a certain amount of "send space" to buffer packets to transmit and a certain amount of "receive space" to buffer packets to receive. You can change the amount of send space by modifying the **udp_sendspace** attribute of the **no** command. You can change the amount of receive space by modifying the **udp_recvspace** attribute of the **no** command. Send space and receive space are upper-bounded by the **sb_max** attribute of the **no** command. When there is not enough receive space to hold incoming packets, packets will be dropped, causing performance degradation. Therefore, the key tuning factor is to have a sufficiently large **sb_max** value. This value depends on the total number of tasks and the communication pattern in the job. In general, the value of **sb_max** should be equal to the maximum number of tasks that

simultaneously communicate with one task multiplied by the value of the **MP_UDP_PACKET_SIZE** environment variable. For more information about **MP_UDP_PACKET_SIZE**, see “Variables for performance tuning” on page 270. For more information about **no**, see *AIX 5L Version 5.2 Commands Reference* or *AIX 5L Version 5.3 Commands Reference*.

User header data

When you use the user header facility for LAPI message-passing calls, make sure the user header length is a multiple of 4 (that is, an integral number of words). Due to the layout of data within the lower-level first-in, first-out structures (FIFOs), you will see noticeable performance gains if header lengths are an integral number of doublewords.

Send-side copy of small messages

Because LAPI is a reliable protocol, it must handle retransmission if packets are dropped by the lower communication layers. To do this, LAPI must maintain user data uncorrupted until it receives acknowledgement that all packets have arrived. Since LAPI calls are non-blocking, return from a LAPI call does not mean that the send-side data buffer can be modified by the sending task.

LAPI provides the origin counter and send completion handler for this notification. For small messages however, LAPI will make a local copy of the message so that the origin buffer is reusable immediately upon return by the sending task. When sending such messages, the user can assume that the send-side data buffer is available for modification immediately upon return of the LAPI call. The maximum size for this local copy can be set using the **MP_REXMIT_BUF_SIZE** environment variable. The default is to perform the copy for messages of size 128 bytes or less, counting both message data and any user header.

Receive-side optimization for single-packet messages

For single-packet messages, you can optimize the copying of data out of the network FIFO on the receive side. LAPI provides a mechanism with which you can copy the data directly from the receive FIFO. For single packets, LAPI passes the data address to your header handler using the *msg_len* parameter of the **lapi_return_info_t** mechanism. You can then access the data directly in the receive FIFO. If you choose to copy data directly from the FIFO, it can inform LAPI that the message does not need to be delivered by returning NULL from the header handler without changing **lapi_return_info_t**. Recall that the header handler returns the base address of the target data buffer. A NULL value indicates to LAPI that no data is to be copied. An alternative way to indicate to LAPI to not copy the message data is to set the *ctl_flag* field in **lapi_return_info_t** to **LAPI_BURY_MSG**. You must contain the data from the receive FIFO within the header handler when using this optimization. Upon header handler return, LAPI informs the communication subsystem that the receive FIFO slot is available for modification.

Here is a sample header handler that uses this method for fast retrieval of a single-packet message:

```
void *header_handler(lapi_handle_t *hndl, void *uhdr, uint *uhdr_len,
                    ulong *msg_len, compl_hdlr_t **compl_h, void **uinfo);

lapi_return_info_t *ret_info_ptr; /* struct pointer */
void                *base_addr;   /* base address to pull from, could also be */
                                /* a pointer to a user-defined datatype */
long                msg_len;      /* first field of struct is the msg length */

/* grab struct */
```

```

ret_info_ptr = (lapi_return_info_t *) ret_info;
base_addr = (void *) (ret_info_ptr->udata_one_pkt_ptr);
msg_len = ret_info_ptr->msg_len;

/* process data at base_addr */

/* inform LAPI not to copy data */
return NULL;
}

```

Tunable environment variables

The following environment variables are considered user-tunable for performance:

MP_ACK_THRESH

Sets the number of packets that are received before LAPI returns a batch of acknowledgments to the sending task. If you do not set this variable, LAPI sets the default value according to the type of communication adapter that is being used.

Note: If you decide to set this variable, proceed with caution.

The value must be in the range **1 <= MP_ACK_THRESH <= 31**. A value that is too small will result in high-acknowledgment traffic generated by the job consisting of a large number of small packets. A value that is too large may impede progress on the sending side and slow down the entire job, because send-side flow control will prevent the transmission of additional packets until packets are acknowledged.

MP_BULK_MIN_MSG_SIZE

Changes the minimum message size for which LAPI will attempt to make bulk transfers. This environment variable is a hint that may or may not be honored by the communication library.

See Chapter 13, “Bulk transfer of messages,” on page 99 for more information.

MP_POLLING_INTERVAL

Controls the interval for LAPI timer pops (in microseconds). Timer pops cause LAPI to go through its acknowledgment and retransmit processing logic. The default is **400000** (400 milliseconds).

MP_RETRANSMIT_INTERVAL

Controls how often the communication subsystem library checks to see if it should retransmit packets that have not been acknowledged. The value specified is the number of polling loops between checks. The default is **1000000**.

MP_REXMIT_BUF_CNT

Specifies the number of buffers that LAPI must allocate. The size of each buffer is defined by **MP_REXMIT_BUF_SIZE**. This count indicates the number of in-flight messages smaller than **MP_REXMIT_BUF_SIZE** that LAPI can store in its local buffers in order to free up the user’s message buffers more quickly. The default is **128**.

MP_REXMIT_BUF_SIZE	Specifies the maximum message size, in bytes, that LAPI will store in its local buffers in order to more quickly free up the user buffer containing message data. This size indicates the size of the local buffers LAPI will allocate to store such messages, and will impact memory usage, while potentially improving performance. LAPI will use the buffer to store the user header and the user data. The default is 16384 .
MP_UDP_PACKET_SIZE	Controls the size of LAPI packets for UDP data transfer. For optimum performance, this variable must be set to the size of the maximum transfer unit (MTU) of IP that is in use on the system. Setting MP_UDP_PACKET_SIZE to a value that is larger than the IP MTU size will result in potential performance degradation due to packetization by the IP layer for messages larger than MP_UDP_PACKET_SIZE . Setting it to a value that is smaller than the IP MTU size will result in unnecessary packetization overhead in the LAPI layer.

See “Variables for performance tuning” on page 270 for more information.

32-bit and 64-bit interoperability

The `lapi_long_t` datatype

The `lapi_long_t` datatype is used by various LAPI functions to ensure 32-bit/64-bit interoperability.

The `LAPI_Address_init64` subroutine

The `LAPI_Address_init64` subroutine provides a mechanism for a 64-bit task to share addresses with 32-bit tasks. See “`LAPI_Address_init64`” on page 134 for more information.

The `LAPI_Xfer` interface

The `LAPI_Xfer` interface serves as a wrapper function for LAPI data transfer functions. Its remote address fields are expanded to be of type `lapi_long_t`, which is long enough for a 64-bit address. This allows a 32-bit task to send data to 64-bit addresses, which may be important in client/server programs. See “`LAPI_Xfer`” on page 219 for more information.

Chapter 12. Lock sharing

Sharing locks with LAPI provides increased efficiency in protocol layering and user programming. When you need to use a locking mechanism to protect your programs' data structures, you can use the same locking mechanism that is employed by LAPI through its lock sharing interface. This way, your program is more tightly coupled with LAPI in terms of locking. When compared to the use of a separate lock, the use of a shared lock in your program may result in improved latency and throughput.

This optional LAPI function is used mainly to improve performance, especially latency. You can share one lock between LAPI and your program. If you decide not to use a shared lock, your program will proceed as if this function does not exist. If you want to use a shared lock, you need to understand what LAPI does and what you need to do.

LAPI uses locks to protect its internal data structures. Typically, LAPI API calls involve acquisition and release of the LAPI lock. Multi-threaded user programs also often use locks to protect data structures of the user program. It is recommended that such locks are released before making a LAPI API call, because any (unrelated) LAPI message may be received or completed before the API call completes. Such a reception or completion can result in the invocation of the user-provided LAPI handler functions that can access user structures and therefore, will need the user lock to be free for acquisition from within the user-provided handlers.

Having separate locks to protect user structures and LAPI's internal data structures can result in significant inefficiencies due to locks needing to be released before LAPI calls, LAPI locks being acquired and released within the API call, and locks potentially being acquired and released within the body of user handler functions that are executed as a result of the API invocation. Such overhead can be significantly reduced by calling the **LAPI_GET_THREAD_FUNC** utility, which returns function pointers to various locking and signaling functions associated with LAPI's internal lock, and thereafter use LAPI's internal lock with these functions to protect data structures in your program. Thus, your program can share a lock with LAPI, using the lock to protect your data structures, while LAPI uses the same lock to protect its internal data structures. When you invoke LAPI API calls while holding the shared lock, LAPI will check to determine whether the lock is already held by the caller, and if so, continue with LAPI functions without acquiring the lock again. Similarly, when user handler functions are called, if they use the locking functions provided by **LAPI_GET_THREAD_FUNC**, LAPI can ensure that the lock is already held so the user doesn't need to get the lock again. Such lock sharing can result in significant performance gains.

For more information, see "LAPI_GET_THREAD_FUNC" on page 208.

Scenarios without lock sharing

To understand lock sharing better, let's consider the case without lock sharing, in which the user program uses its own lock to protect data structures. Figure 10 on page 84 and Figure 11 on page 85 illustrate the typical calling sequences.

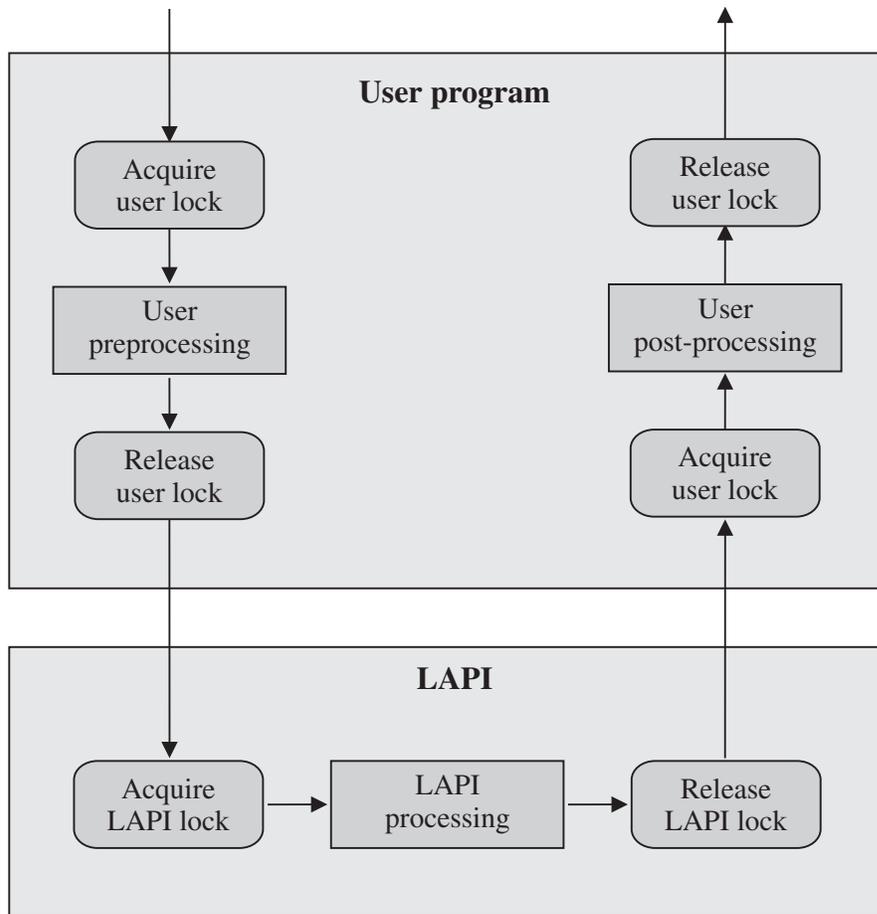


Figure 10. A program initiates a call to LAPI, without lock sharing

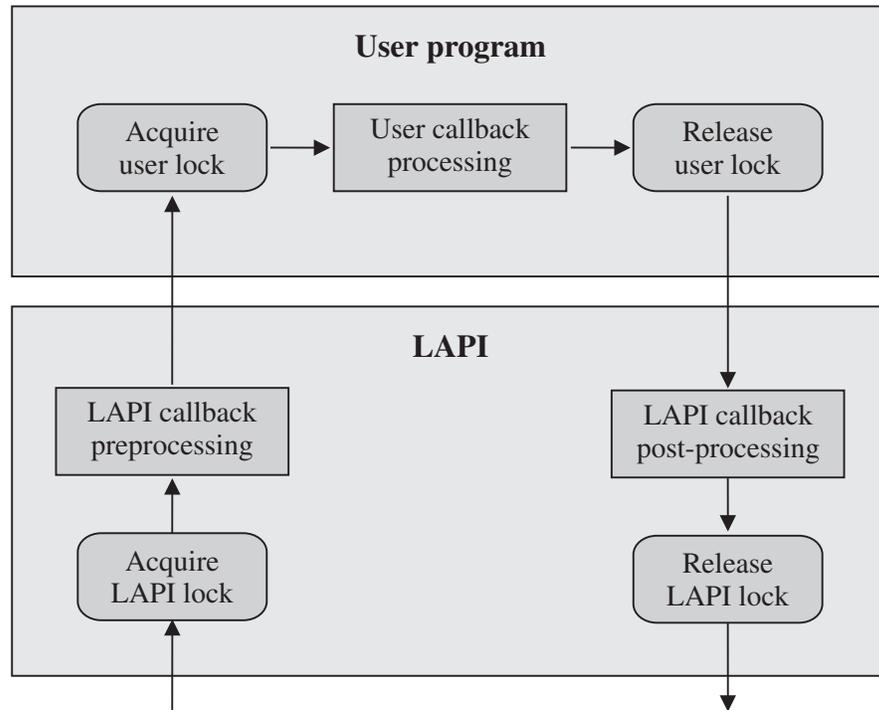


Figure 11. LAPI initiates a callback, without lock sharing

In Figure 10 on page 84, the user program initiates a call to LAPI during some processing. The user program releases its lock before calling LAPI and reacquires the lock after LAPI returns. In Figure 11, LAPI initiates a callback (a header handler callback, for example), into the user program, without releasing LAPI's lock. To embed the acquisition of one lock inside another will not cause any deadlock problems as long as the locking hierarchy is strictly maintained. In these examples, a hierarchy is forced, so the LAPI lock will never be acquired inside the user lock. The user lock must always be released before LAPI is called.

This book refers to a user-initiated call sequence as a "down-call" and a LAPI-initiated call sequence as an "up-call". Up-calls and down-calls can be embedded inside each other, as illustrated by Figure 12 on page 86 and Figure 13 on page 87.

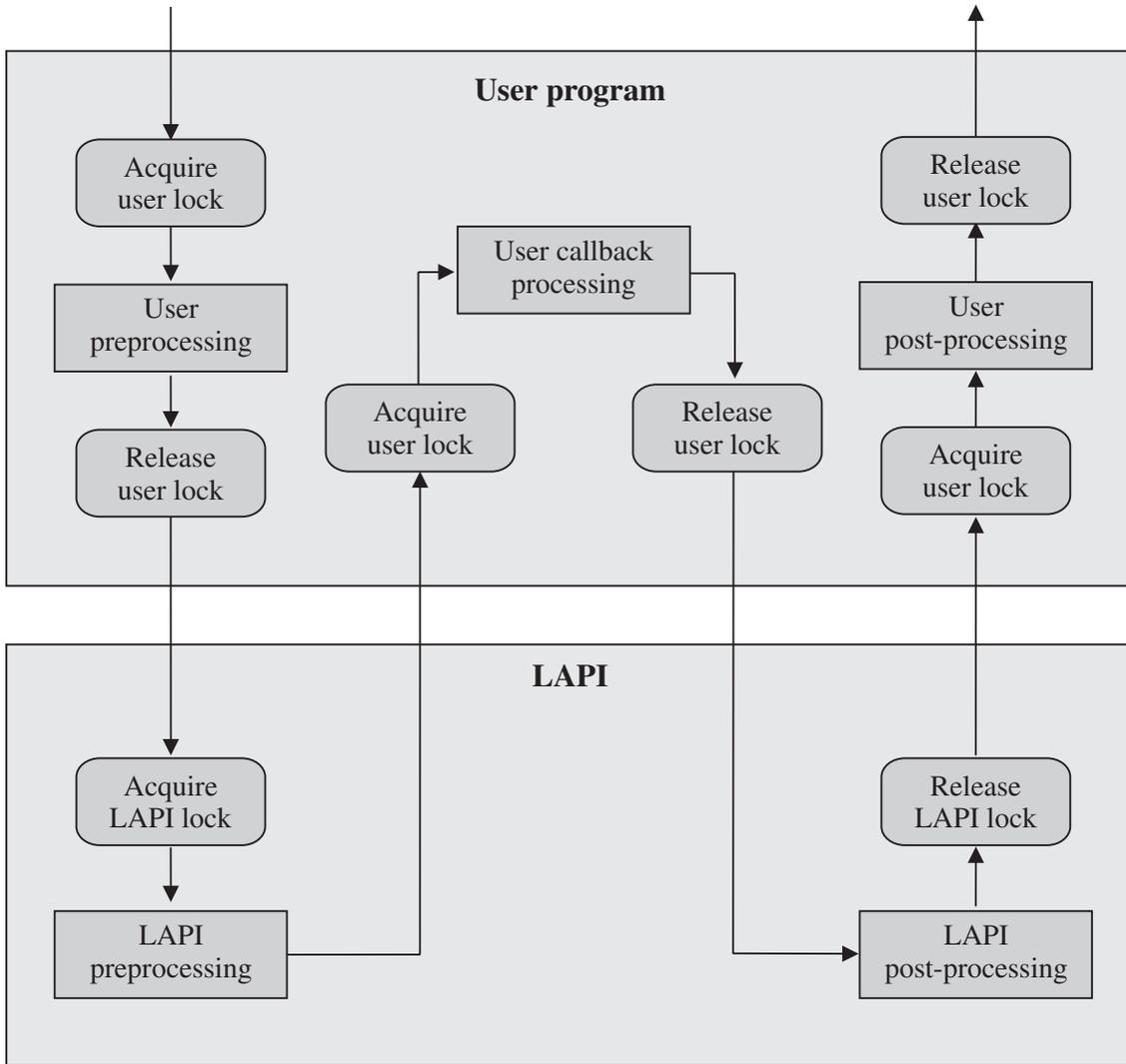


Figure 12. A program initiates a call to LAPI, with embedded up- and down- calls

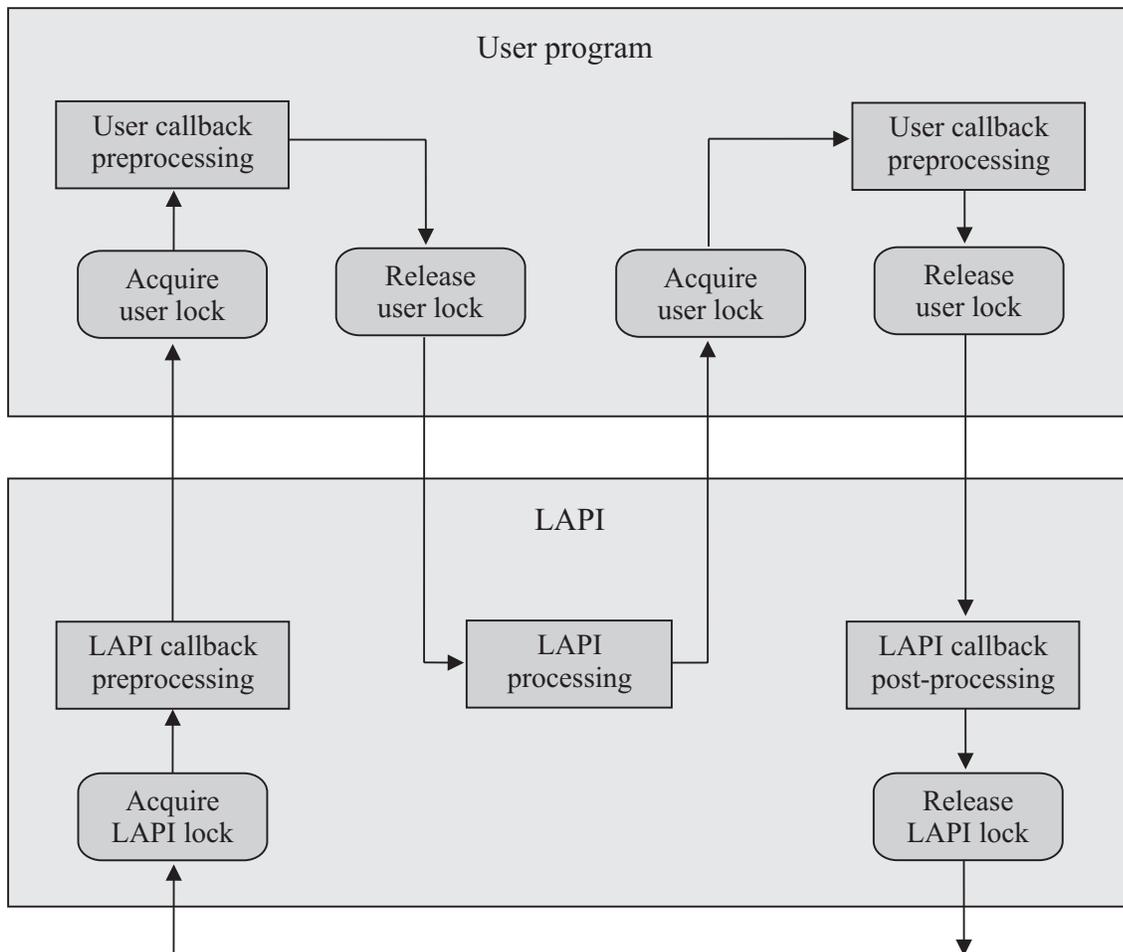


Figure 13. LAPI initiates a callback, with embedded up- and down-calls

Without lock sharing, there are many lock acquisition and release pairs involved: three pairs in Figure 10 on page 84, two pairs in Figure 11 on page 85, four pairs in Figure 12 on page 86, and three pairs in Figure 13. When the cost of locking turns out to be critical for latency-sensitive applications, it would be much more desirable to reduce locking operations using the lock sharing function provided by LAPI.

Scenarios with lock sharing

Lock sharing makes the user program more tightly coupled with LAPI because one shared lock is now protecting both the user data structures and LAPI data structures. When the user program calls LAPI, it is optional to release the shared lock and to reacquire the lock after LAPI returns, as depicted in the down-call in Figure 14 on page 88. However, for optimal performance, it is better not to release the shared lock if the user has already acquired it. Because LAPI does not make any assumptions about the lock ownership, it will check the ownership upon entry into any of its API calls. If a lock is not held, LAPI will acquire the lock and release it after processing. Checking for lock ownership is much less expensive than getting a lock, so the locking cost is essentially one pair of lock acquisition and release, as shown in Figure 14 on page 88. For an up-call, LAPI guarantees that the shared lock is held before the user callback is invoked so there is no need to check lock ownership in the user callback. Similarly, the locking cost in Figure 15 on page 89 is also only one lock acquisition and release pair.

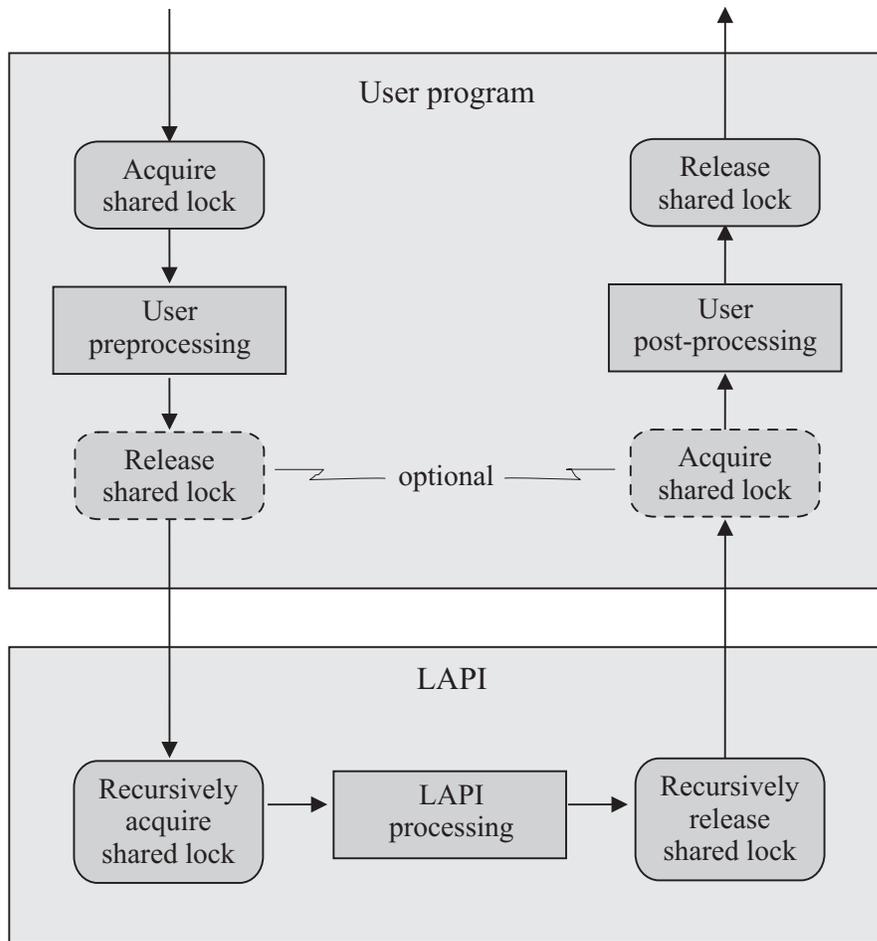


Figure 14. A program initiates a call to LAPI, with one lock acquisition and release pair

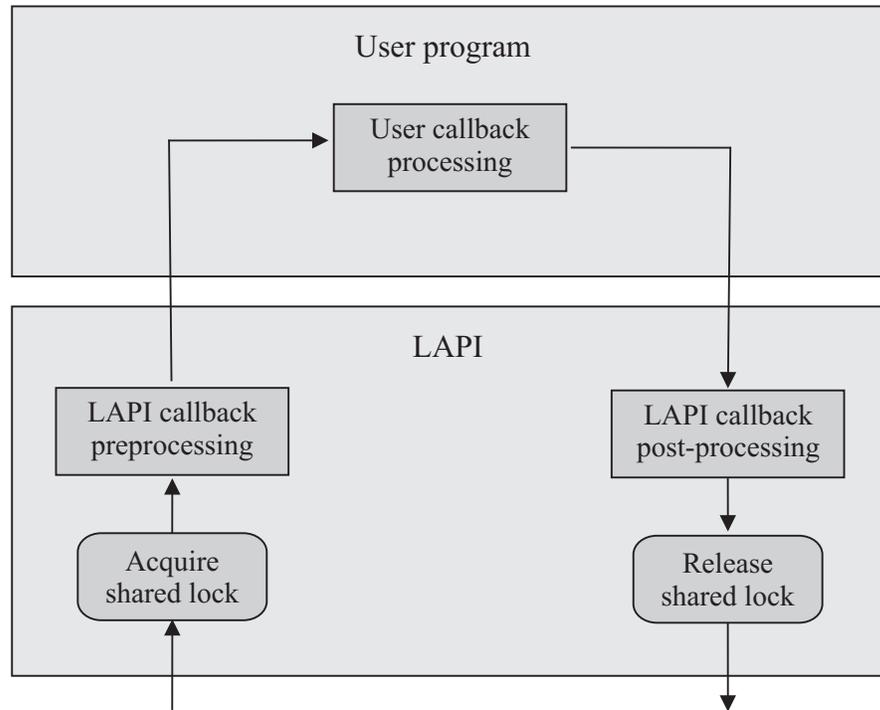


Figure 15. LAPI initiates a callback, with one lock acquisition and release pair

When up-calls and down-calls are embedded inside each other, the saving from lock sharing becomes more prominent. As shown in Figure 16 on page 90 and Figure 17 on page 91, only one lock acquisition and release pair is involved in both cases, compared to four pairs and three pairs without lock sharing, as shown in Figure 12 on page 86 and Figure 13 on page 87, respectively.

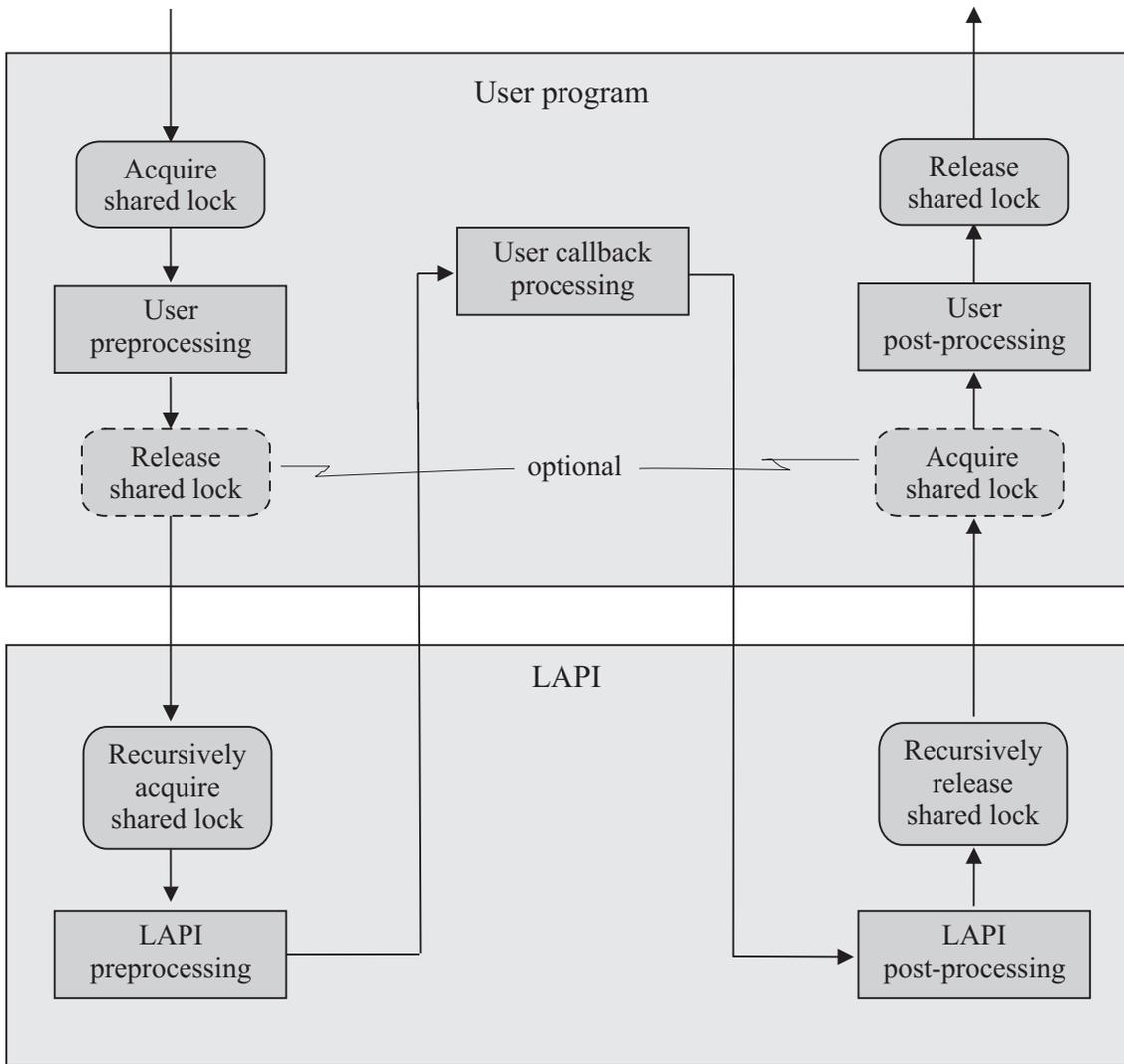


Figure 16. A program initiates a call to LAPI, with embedded up- and down- calls and one lock acquisition and release pair

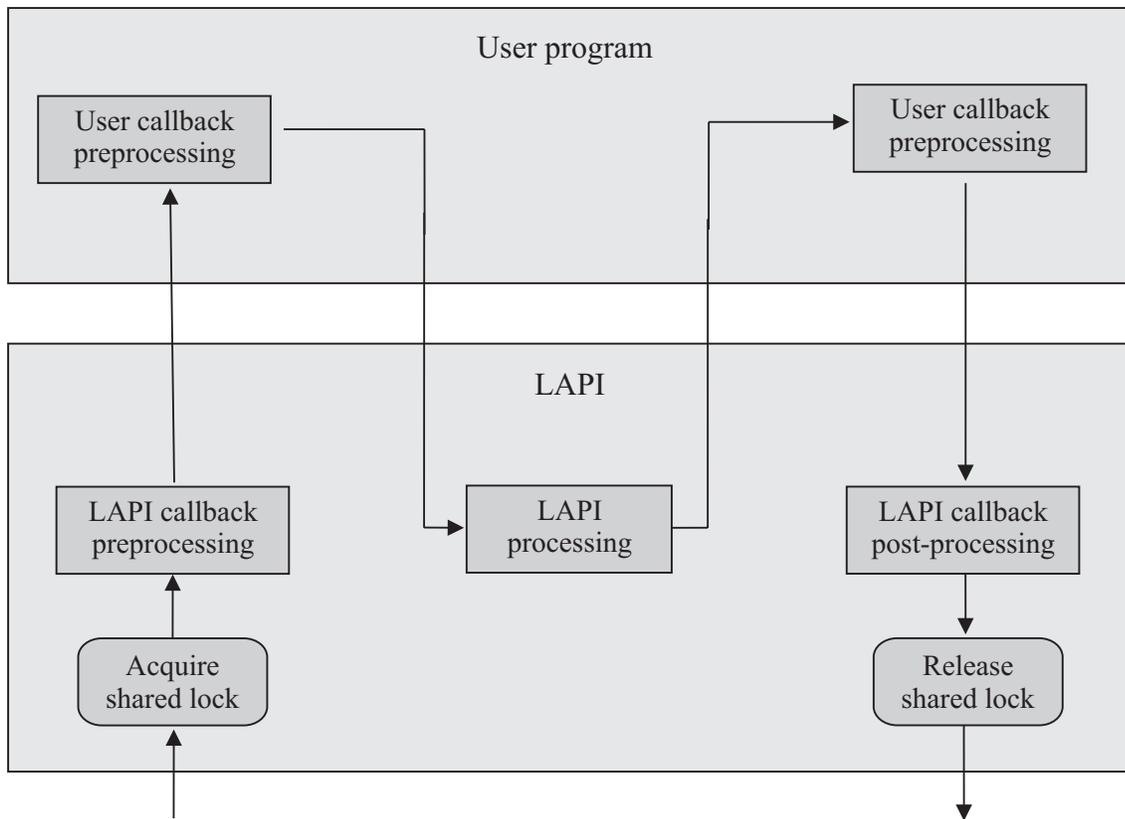


Figure 17. LAPI initiates a callback, with embedded up- and down- calls and one lock acquisition and release pair

Correctness of lock sharing

When a lock is shared between LAPI and a user program, but they do not share any data structures, the correctness of sharing needs to be addressed. Before proceeding, let's clarify the concepts of atomic operation and critical section. An *atomic operation* is an operation whose execution steps on data structures must be carried out without interruption from other operations on the same data structures. As an example, when a stack is shared by multiple threads, putting an element onto the stack and updating the stack top must be performed atomically to maintain the integrity of the stack. A *critical section* is a sequence of atomic and non-atomic operations protected by a lock.

If a critical section consists of multiple atomic operations in a sequence, it is correct to break the sequence into multiple critical sections so that each atomic operation lies entirely in one of the smaller critical sections. For example, there are two correct ways that a thread can push two logically-independent elements onto the stack:

method 1:

```
lock(stack)
push(stack, element1)
push(stack, element2)
unlock(stack)
```

method 2:

```
lock(stack)
push(stack, element1)
unlock(stack)
lock(stack)
push(stack, element2)
unlock(stack)
```

The first method is more efficient, but it doesn't mean that the two push operations must be done together in one critical section. It is correct for the second method to release the lock after the first push operation and reacquire the lock before the second push operation. The same analogy can be applied to the meaning of critical section under lock sharing. Even if there is one big critical section protected by the shared lock (analogous to method 1) from the user's view, it is possible that LAPI breaks up the big critical section into multiple smaller ones by releasing and reacquiring the shared lock. Therefore, one requirement is placed upon the user to guarantee the correctness of lock sharing: even if you create a big critical section protected by the shared lock, you must still structure the program so that any of its atomic operations complete before control is transferred to LAPI.

This requirement is useful because:

- LAPI can generate callbacks into your program. If there are any incomplete atomic operations, you must be very careful not to break the atomicity of the incomplete operations in the callbacks.
- It lets your program easily switch from using a shared lock to using a separate lock because using a separate lock requires you to complete atomic operations before the user lock is released and the control is transferred to LAPI.
- LAPI can safely release the shared lock to allow processing in a different thread and callbacks into your program from that thread.

To meet this requirement, you would structure your program so that it functions even if the shared lock is always released before calling LAPI and reacquired after LAPI returns, as illustrated by Program B versus Program A in Figure 18 on page 93 below. Such a thinking process always assumes that LAPI would release the shared lock and reacquire it, which is what LAPI could do to the shared lock.

a.

Program A

```

user_function()
{
    lock(shared_lock)
    processing 1
    LAPI_Put
    processing 2
    unlock(shared_lock)
}
send_completion_callback()
{
    processing 3
}

```

b.

Program B

```

user_function()
{
    lock(shared_lock)
    processing 1
    unlock(shared_lock)
    LAPI_Put
    lock(shared_lock)
    processing 2
    unlock(shared_lock)
}
send_completion_callback()
{
    processing 3
}

```

c.

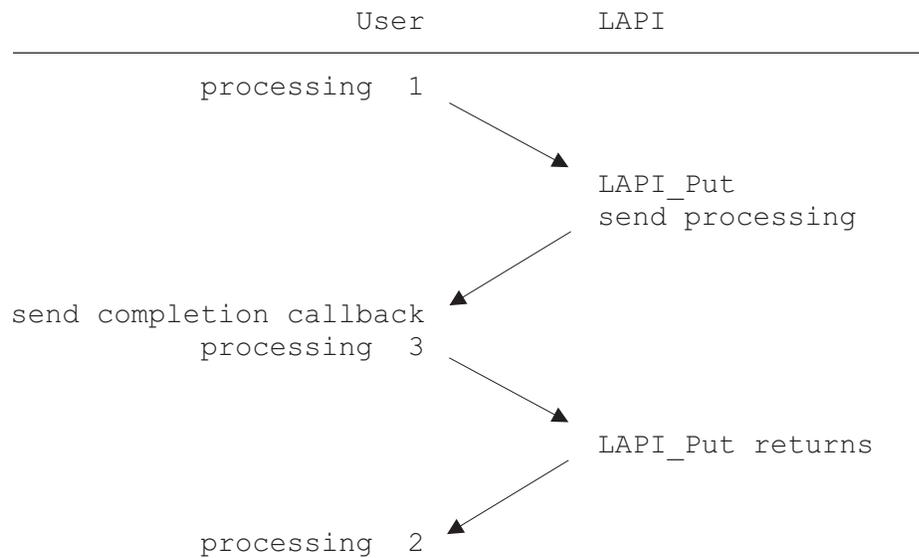


Figure 18. Critical sections under lock sharing. (a) Program with a large critical section, (b) Program with multiple smaller critical sections, (c) Execution flow of the two programs

Implications and restrictions

Initialization and termination

Your program must not hold a shared lock when calling **LAPI_Init** and **LAPI_Term**. In the case of **LAPI_Init**, the lock is not available yet. In the case of **LAPI_Term**, it is not on the performance path and it is better for LAPI to have the lock released before the call.

Other LAPI calls

For all LAPI calls other than **LAPI_Init** and **LAPI_Term**, LAPI detects whether a shared lock is already held by the running thread. If it is held, LAPI does not acquire the lock again and keeps the lock upon return; otherwise, LAPI acquires the lock and releases it upon return. You can decide whether your program will acquire the shared lock before calling LAPI. For better performance, your program should keep the already-acquired lock before calling LAPI.

Callbacks

The shared lock is guaranteed to be held by the running thread when the following user callbacks are invoked. You should not try to acquire the shared lock and you must not release the lock in these callbacks:

- Send completion handler: invoked when the user header and data buffers for sending a message can be reused.
- Header handler: invoked when the first user header of a message arrives at the receive side.
- Inline completion handler: invoked when a message is completely received, upon the user's request specified by the header handler.

All of the callbacks can be on the latency path, so avoiding extra locking improves latency.

The out-of-line completion handler is invoked from LAPI's completion handler thread. It is not on the latency path and the thread will not be holding the shared lock upon the entry of the handler. If your program has any critical sections to process, the shared lock must be acquired first. Calls to LAPI can be made with the lock held.

The error handler will always be invoked without the thread holding the shared lock. Your program should terminate the job upon error, possibly without returning control to LAPI at all.

Long critical sections

Use a shared lock to protect critical sections in communication. You should not use it for long computation, because this could slow down progress in communication. If you want to hold the lock for a long period, your program must poll LAPI by calling **LAPI_Probe** or **LAPI_Msgpoll** in a timely manner so that the communication layer can make progress. Such timely polling is required not only for communication but also for the out-of-line completion handler, checkpoint handler, and error handler to acquire the lock.

Lock preemption

In order to support out-of-line completion, checkpoint/restart, and gang scheduling, in which the handlers must acquire the shared lock first, LAPI will release the lock for the handlers even if it is acquired by your program. The release will only be done when your program calls LAPI. It will never happen that one thread releases a lock acquired by another thread. Because of this, any of your program's long critical sections must poll LAPI in a timely manner in order to give the handlers a chance to run.

Receive/timer interrupts

Suppose your program initializes using the following sequence:

```

call LAPI_Init
call LAPI_Util to retrieve the shared lock
lock(shared lock)
initialize global data structure
unlock(shared lock)

```

A receive/timer interrupt could start before the global data structure is initialized. Your program must not operate on uninitialized data. There are two ways to be sure of this:

1. Check to see if the data structure has been initialized in callbacks from LAPI.
2. Use a different lock to protect the initialization of data and call **LAPI_Init** only after the global data structure is initialized.

Performance of multi-threaded programs

Using one shared lock makes locking granularity coarse and there may be more contentions on the lock between threads. In case there is performance degradation, you can change your program's locking structure to reduce situations in which one thread is blocking another thread. There are two possible ways to do this:

1. A polling thread that is holding the shared lock yields the lock to other threads.
2. Other threads hand off their work to the polling thread.

Compatibility

Existing LAPI programs that don't exploit lock sharing will run without any change. To test for the presence of the lock sharing function, you can call **LAPI_Util** with **LAPI_GET_THREAD_FUNC** and check for a return code. If any error is returned, lock sharing must not be used.

A sample lock sharing program

A sample lock sharing program follows. This program measures LAPI latency with and without lock sharing. It illustrates the following aspects of lock sharing:

- how to detect whether lock sharing is available
- how to retrieve functions for lock sharing
- how to use the basic lock and unlock functions on the shared lock

```

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <lapi.h>

#define RC(statement) \
{ \
    int rc = statement; \
    if (rc != 0) { \
        printf(#statement " rc = %d, line %d\n", rc, __LINE__); \
        exit(-1); \
    } \
}

lapi_handle_t    hndl;
int              my_ID;

/* Function to get current time in microseconds */
double
microseconds()

```

```

    {
        struct timeval time_v;
        gettimeofday ( &time_v, NULL );
        return ( (double)time_v.tv_sec * uS_PER_SECOND + time_v.tv_usec );
    }

/* A dummy header handler to consume the incoming message */
void *
am_hdlr(lapi_handle_t *hdl, int *u_hdr, uint *hdr_len,
        lapi_return_info_t *ret_info,
        compl_hdlr_t **chndl, void **saved_info)
{
    *chndl = NULL;
    *saved_info = NULL;
    ret_info->ctl_flags = LAPI_BURY_MSG;
    return NULL;
}

/* Function to measure LAPI latency */
double
lapi_latency(int num_pongs, int share_lock)
{
    int rc, i;
    lapi_xfer_t xfer;
    lapi_msg_info_t msg_info;
    double t1, t2;
    lapi_thread_func_t tf;

    /* Retrieve function for lock sharing */
    if (share_lock) {
        tf.Util_type = LAPI_GET_THREAD_FUNC;
        rc = LAPI_Util(hdl, (lapi_util_t *)&tf);
        if (rc != LAPI_SUCCESS) {
            printf("Lock sharing is not supported by this library.\n");
            share_lock = 0;
        }
    }

    /* Clear structures */
    bzero(&msg_info, sizeof(msg_info));
    bzero(&xfer, sizeof(xfer));

    /* Setup command structure for transfer */
    xfer.Xfer_type=LAPI_AM_XFER;
    xfer.Am.hdr_hdl=1;
    xfer.Am.tgt=1^my_ID;
    xfer.Am.uhdr=NULL;
    xfer.Am.uhdr_len=0;
    xfer.Am.udata=NULL;
    xfer.Am.udata_len=0;
    xfer.Am.shdlr=NULL;
    xfer.Am.sinfo=NULL;
    xfer.Am.tgt_cntr=NULL;
    xfer.Am.org_cntr=NULL;
    xfer.Am.cmpl_cntr=NULL;

    /* Acquire shared lock */
    if (share_lock)
        tf.mutex_lock(hdl);

    RC( LAPI_Gfence(hdl) );

    t1 = microseconds();
    /*
     * Ping-pong test for latency. Count 1 is used in LAPI_Msgpoll
     * to magnify the effect of lock sharing.

```

```

    */
    if (my_ID == 0)    {
        for (i=0; i<num_pongs; i++)    {
            /* send then receive */
            RC( LAPI_Xfer(hndl, &xfer) );
            msg_info.status = 0;
            while (!(msg_info.status & LAPI_RECV_COMPLETE)) {
                RC( LAPI_Msgpoll(hndl, 1, &msg_info) );
            }
        }
    } else {
        for (i=0; i<num_pongs; i++)    {
            /* Receive then send */
            msg_info.status = 0;
            while (!(msg_info.status & LAPI_RECV_COMPLETE)) {
                RC( LAPI_Msgpoll(hndl, 1, &msg_info) );
            }
            RC( LAPI_Xfer(hndl, &xfer) );
        }
    }
    t2 = microseconds();

    RC( LAPI_Gfence(hndl) );

    /* Release shared lock */
    if (share_lock)
        tf.mutex_unlock(hndl);

    /* Calculate latency */
    return (t2 - t1)/num_pongs/2;
}

/*
 * This testcase is to be invoked with 2 optional arguments.
 * 1st arg: number of ping-pongs to measure latency
 * 2nd arg: number of times to measure latency repeatedly.
 */
main(int argc, char *argv[])
{
    int            share_lock = 1;
    int            num_pongs = 10000;
    int            times = 10, i;
    lapi_info_t    lapi_info;
    double         10, 11;

    /* Read arguments */
    if (argc > 1)
        num_pongs = atoi(argv[1]);
    if (argc > 2)
        times = atoi(argv[2]);

    /*
     * Improvement of lock sharing is more significant
     * when checkpoint is enabled
     */
    putenv("CHECKPOINT=yes");

    /* Initialize LAPI */
    bzero(&lapi_info, sizeof(lapi_info));
    lapi_info.lib_vers = LAST_LIB;
    RC( LAPI_Init(&hndl, &lapi_info) );

    /* Query/set LAPI settings */
    RC( LAPI_Qenv(hndl, TASK_ID, &my_ID) );
    RC( LAPI_Senv(hndl, INTERRUPT_SET, 0) );
    RC( LAPI_Senv(hndl, ERROR_CHK, 0) );

```

```

/* Set address index for header handler */
RC( LAPI_Addr_set(hndl, (void*)&am_hndl, 1) );

/* Performance latency tests */
for (i=0; i<times; i++) {
    l0 = lapi_latency(num_pongs, 0);
    l1 = lapi_latency(num_pongs, 1);
    if (my_ID == 0)
        printf( "Latency without lock sharing: %.2f us, with: %.2f us\n",
                l0, l1);
}

/* Terminate LAPI */
RC( LAPI_Term(hndl) );
}

```

Use the following command to compile your program:

```
mpcc_r program.c -o program
```

where *program* is the name of your lock sharing program.

The program should be run with two tasks. The output will look like this:

```
Latency without lock sharing: 13.75 us, with: 12.89 us
```

Chapter 13. Bulk transfer of messages

For users of the HPS or the pSeries HPS, RSCT LAPI supports bulk message transfer using the remote direct memory access (RDMA) protocol. Bulk transfer is especially useful for applications that transfer relatively large amounts of data — more than 150 kilobytes (KB) — in a single call, or that overlap computation and communication, because the CPU is no longer required to copy data.

By default, bulk transfer is disabled in RSCT LAPI. To enable bulk transfer for interactive POE jobs, set the environment variable **MP_USE_BULK_XFER=yes**. This transparently causes portions of the user's virtual address space to be pinned and mapped to a communication adapter. LAPI then uses RDMA to move data from the send buffer to the receive buffer. Note that not all communication adapters support RDMA.

To change the minimum message size for which LAPI will attempt to make bulk transfers, modify the setting of the **MP_BULK_MIN_MSG_SIZE** environment variable.

For more information about **MP_USE_BULK_XFER** and **MP_BULK_MIN_MSG_SIZE**, see “Variables for data transfer” on page 269.

These environment variables are hints that may or may not be honored by the communication library. For the communication library to honor these variables, the system administrator must:

1. Enable the RDMA protocol in the Switch Network Interface (SNI) device driver. To do this, set the **rdma_xlat_limit** attribute of the SNI devices to an appropriate value. For more information, see the appropriate SNI documentation for the specific server type, for example: *Switch Network Interface for eServer pSeries High Performance Switch: Guide and Reference*.
2. Follow the instructions in the “Using bulk data transfer” section of *LoadLeveler for AIX 5L and Linux: Using and Administering*.

You can use the **LAPI_Qenv** subroutine to find out if bulk transfer is enabled and to query the minimum message size for bulk transfer. See “LAPI_Qenv” on page 184 for more information.

The maximum message size for bulk transfer is 32 megabytes (MB). Transparently to the user, LAPI delivers messages that are larger than 32MB in 32MB chunks. The performance of bulk message transfer may be enhanced by using technical large pages. RDMA operations are considerably more efficient when large (16 MB) pages are used rather than small (4 KB) pages, especially for large transfers.

Normal LAPI message passing involves packetization of messages for transfer in sizes that can be handled by the lower-level communication subsystem. LAPI also allows for bulk transfer of messages using the adapter's direct memory access (DMA) capability. To illustrate the use of bulk transfer, let's start with an examination of the packet-mode approach to message transfer. Figure 19 on page 100 illustrates the flow of LAPI packet-level message passing. Each vertical arrow represents a data copy by LAPI through calls to the communication subsystem layer. There is one send-side copy into a network FIFO from a user data buffer and one receive-side copy out of a network FIFO into a user data buffer.

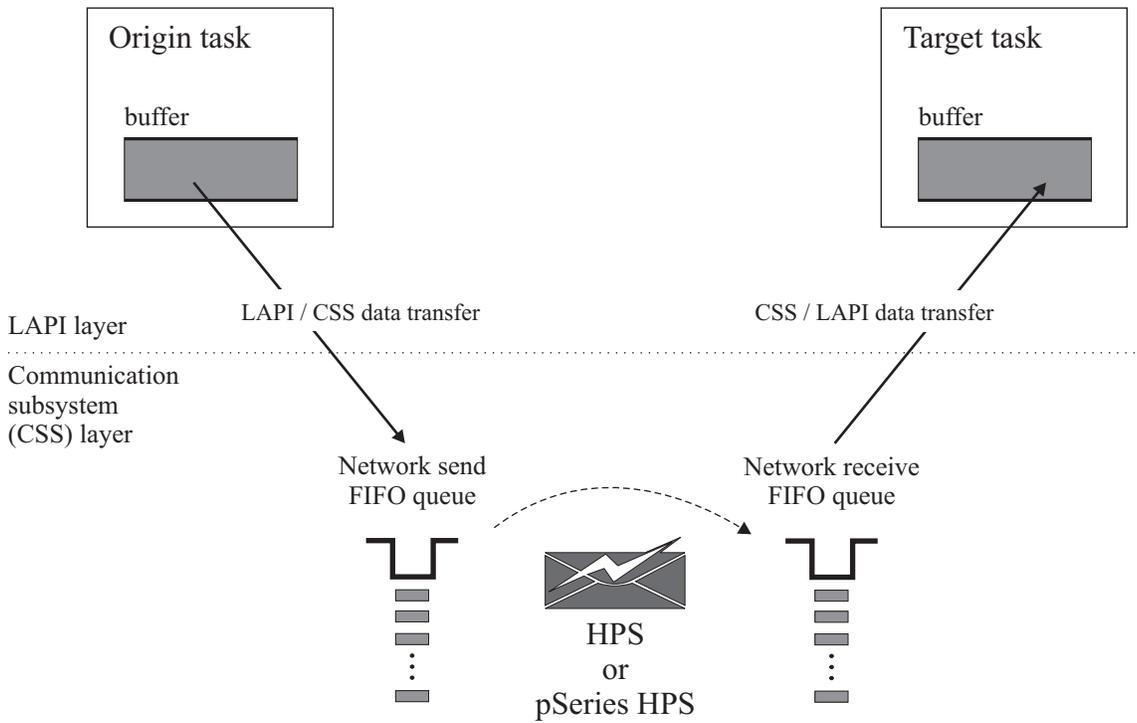


Figure 19. LAPI packet data flow

For larger messages, the AIX communication subsystem (CSS) supports a DMA method on both sides of the communication, resulting on one less data copy on each of the sending and receiving tasks. Figure 20 on page 101 illustrates the flow of data for a LAPI bulk transfer. Using a rendezvous protocol, the origin and target tasks establish DMA connections with the adapter firmware. The target task then pulls the data across the switch. Using the DMA method, the switch adapter transfers the data from the origin task's address space directly into the target task's address space. If any step in the sequence fails, LAPI will use packet mode for message delivery.

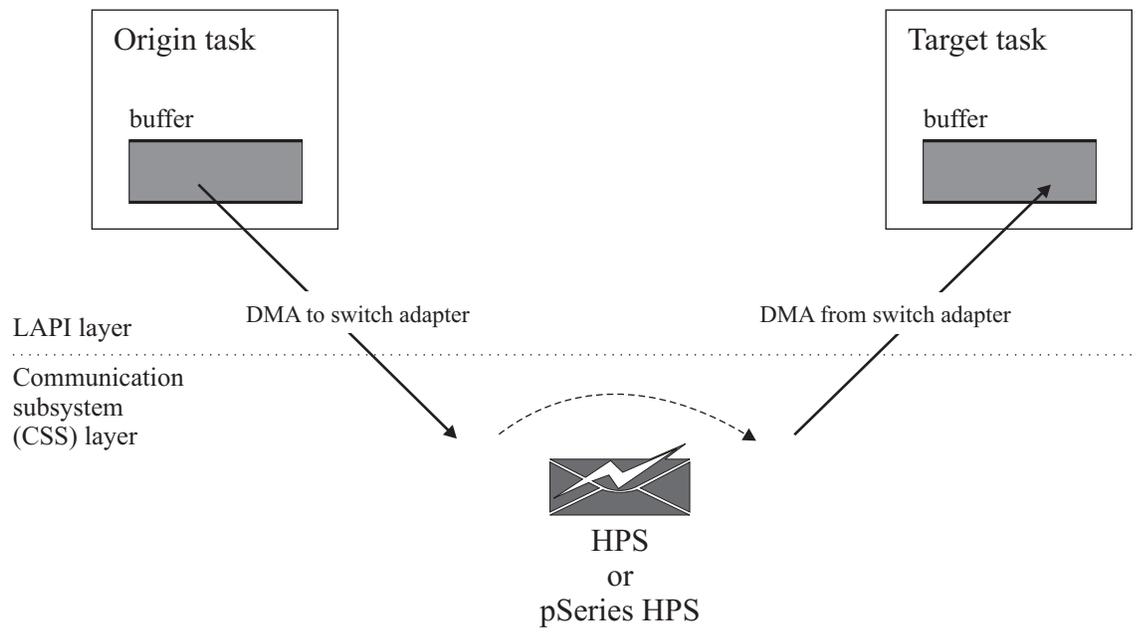


Figure 20. LAPI bulk data flow

Chapter 14. Striping, failover, and recovery

This chapter describes RSCT LAPI's striping, failover, and recovery functions. On systems with multiple HPS or pSeries HPS adapters per node, a job can request the use of multiple LAPI instances for each of the job tasks in order to obtain higher availability and performance in the presence of link/adapter failures. When these multiple instances span multiple physical adapters, LAPI can improve availability of communication when it is used in conjunction with the group services component of RSCT. Using group services, LAPI can quickly determine when an adapter's ability to communicate has ceased, and can then "fail over" all communication to the remaining instances where communication is still possible. Correspondingly, using group services, LAPI is also able to determine when an adapter's ability to communicate has resumed, and can then resume communication using the corresponding instances. During a job run, any failover and recovery operations are, for the most part, transparent to the user.

In this book, adapters that have lost their ability to communicate (as detected by group services), either due to an adapter failure or due to a failure in the network path leading to the adapter, are referred to as "down" adapters. Adapters that still have the ability to communicate are referred to as "up" adapters.

Using failover and recovery

LAPI's failover and recovery function consists of two basic elements:

1. Monitoring and receiving notification about the communication status of HPS or pSeries HPS adapters.

This element depends on the group services component of RSCT and a component of LAPI called the Network Availability Matrix (NAM).

2. The use of multiple HPS or pSeries HPS adapters for redundancy, to enable failover.

This element depends on LoadLeveler, with corresponding POE functions that serve as a wrapper to convey requests to LoadLeveler.

Failover and recovery cannot be provided for a job if either of these elements is absent.

Monitoring adapter status

Adapter status monitoring depends on NAM and group services.

Network Availability Matrix (NAM) overview

The Network Availability Matrix (NAM) is a pseudo-device component that is packaged as a separate LAPI fileset (**rsct.lapi.nam**). To make use of LAPI's failover and recovery function, you must have the NAM pseudo-device "Available" on all of the nodes that are running your job tasks. The pseudo-device **nampd0** is automatically created and configured in the boot process after the **rsct.lapi.nam** fileset is installed. See Chapter 4, "Installing RSCT LAPI," on page 25 for more information.

On any particular node, the NAM pseudo-device serves as a repository for status information about the HPS or pSeries HPS adapters that are within the same RSCT peer domain as that node. The NAM serves as an interface through which the communication status of adapters that are monitored by the group services component can be easily conveyed to LAPI tasks. A change in the communication status of an adapter may reflect that the adapter has gone "down" or has come

back "up". Every time the group services component detects a change in communication status of any of the HPS or pSeries HPS adapters, it conveys the new status of the changed adapters to the NAM pseudo-device on each of the nodes within the corresponding RSCT peer domain. This, in turn, triggers a notification to LAPI tasks running on those nodes that use multiple instances. On receiving this notification, the LAPI protocol for these tasks can fail over communication from "down" adapters to other "up" adapters. The LAPI protocol can also recover the use of adapters that have again become "up" to resume sharing the communication load.

RSCT peer domains and group services

As noted in "Network Availability Matrix (NAM) overview" on page 103, the group services component only updates adapter status in the NAMs of the nodes within a given peer domain. It follows that for LAPI failover and recovery to be possible for a given job, job tasks must all run on nodes that belong to the same peer domain. Preferably, all of the nodes in the system must be configured as part of a single RSCT peer domain. For information about setting up an RSCT peer domain, see *RSCT: Administration Guide*.

If, for some reason, multiple peer domains are required, you must make sure that each peer domain forms a separate LoadLeveler resource pool so that jobs requiring failover and recovery do not span multiple peer domains. The updating of the NAM pseudo-device by group services is transparent to the user.

Requesting the use of multiple adapters

You can use POE environment variables or LoadLeveler job control file (JCF) keywords to request the use of multiple adapters.

Using POE environment variables

In order for there to be sufficient redundancy to handle at least one adapter failure, each task of the job needs to be allocated communication instances across at least two different HPS or pSeries HPS adapters. An *instance* is an entity that is required for communication over an adapter device. In the user space (US) communication mode, which is specified by setting **MP_EUILIB=us**, an instance corresponds to an adapter window. On the other hand, in the IP communication mode, which is specified by setting **MP_EUILIB=ip**, an instance corresponds to the IP address of a given adapter to be used for communication.

Depending on the number of networks in the system and the number of adapters each node has on each of the networks, you can request the allocation of multiple instances for your job tasks by using a combination of the POE environment variables **MP_EUIDEVICE** and **MP_INSTANCES**. The distribution of these requested instances among the various HPS or pSeries HPS adapters on the nodes is done by LoadLeveler. Depending on whether the job is using user space or IP, and on the resources available on each of the adapters, LoadLeveler will try to allocate these instances on different adapters.

To request the use of multiple instances on a system where all nodes have adapters on each of the n networks in the system, you can set **MP_EUIDEVICE** to the value **sn_all**. This setting translates to a request for the default number of instances (**1**) from adapters on each of the networks in the system, and a request for a total of n instances for each of the job tasks. You do not have to set the **MP_INSTANCES** environment variable. If **MP_EUIDEVICE** is set to **sn_all** and you do set the **MP_INSTANCES** variable to a value m (where m is a number from **1** through the value of the case-insensitive string **max**), this translates to a request of m instances from each of the networks in the system for each job task. For user

space, this corresponds to a request for ($m * n$) different windows for each job task. For IP, this corresponds to a request for the same number of HPS or pSeries HPS IP devices.

You must take the following considerations into account while defining the number of instances to use and the value specified for **MP_EUIDEVICE**:

- If m is greater than the number of adapters a node has on one of the networks, multiple windows will be allocated from some of the adapters. For IP, the same adapter device will be allocated multiple times.
- LoadLeveler translates the value **max** as a request to allocate the number of instances (as specified by the *max_protocol_instances* variable) that are defined for this job class in the LoadLeveler **LoadL_admin** file. See *LoadLeveler for AIX 5L and Linux: Using and Administering* for more information. If you request more instances than the value of *max_protocol_instances*, LoadLeveler allocates a number of instances that is equal to the value of *max_protocol_instances*. To have your job use all adapters on the system across all the networks, you can have the administrator set *max_protocol_instances* for your job class to the number of adapters each node has on each network (assuming that each node has the same number of adapters on each network), and then run your job with **MP_EUIDEVICE=sn_all** and **MP_INSTANCES=max**.
- On a system where every node is connected to more than one common network, setting **MP_EUIDEVICE=sn_all** is sufficient to allocate instances from distinct adapters for all job tasks. You do not need to set **MP_INSTANCES**. This is because an adapter is connected to exactly one network, this is a request for instances from each network, and if the request is satisfied, at least two distinct adapters have been allocated for each of the job tasks. In the case of user space, if all windows on the adapters of one or more networks are all used up, the job will not be scheduled until windows are available on adapters of each network.

To request the use of multiple instances on a system where all nodes are connected to a single HPS or pSeries HPS network, or where nodes are connected to multiple networks, but you want your tasks to use adapters that are connected to only one of those networks, you can set **MP_EUIDEVICE=sn_single** and **MP_INSTANCES=m**, where m is a number from 1 through the value of the (case-insensitive) string **max**. This translates to a request for m instances on one network only; not, as in the previous case, on each of the n networks in the system. With such a request, if **MP_EUILIB=us**, it is not guaranteed that LoadLeveler will allocate the multiple windows from distinct adapters if window resources on some of the adapters are all used up by previously-scheduled jobs. In this scenario, LoadLeveler may allocate the multiple windows all from a single adapter and one or more of the job tasks will be without a redundant adapter to fail over to in the case of a communication problem. Thus, the only guaranteed way to get multiple adapters allocated to the job to satisfy the basic requirements for LAPI's failover and recovery function, is to have the nodes in the system connect to multiple HPS or pSeries HPS networks and setting **MP_EUIDEVICE=sn_all**.

POE will post an attention message stating that failover and recovery operations may not be possible for the job if multiple instances are requested, but one or more job tasks are allocated instances that are all from the same adapter. The interaction among the values of **MP_INSTANCES**, **MP_EUIDEVICE**, and **MP_EUILIB**, in terms of the total instances that are allocated to every task of the job, and whether use of the failover and recovery function is possible as a result are shown in Table 12 on page 106:

Table 12. Failover and recovery operations

MP_EUIDEVICE=	Instances allocated per task with MP_EUILIB=us		Instances allocated per task with MP_EUILIB=ip	
	MP_INSTANCES is not set	MP_INSTANCES=m	MP_INSTANCES is not set	MP_INSTANCES=m
sn_single	1 no failover	m failover may not be possible	1 no failover	m failover is possible if num_adapters per network > 1
sn_all	num_networks failover is possible if num_networks > 1	m * num_networks failover is possible if num_networks > 1	num_networks failover is possible if num_networks > 1	m * num_networks failover is possible if num_networks > 1

Using LoadLeveler JCF keywords

The use of the LoadLeveler job class attribute *max_protocol_instances* is described in “Using POE environment variables” on page 104. For more information about this attribute, and for the syntax to specify the request for multiple instances on a single network or on all networks in the system using a LoadLeveler job control file (JCF), see *LoadLeveler for AIX 5L and Linux: Using and Administering*.

Note: Although more than eight instances are allowed using a combination of LoadLeveler’s *max_protocol_instances* setting and the **MP_INSTANCES** environment variable, LAPI ignores all window allocations beyond the first eight, because LAPI supports a maximum of eight adapters per operating system instance and the best performance can be obtained with one window on each of them. Using multiple windows on a given adapter provides no performance advantage.

Failover and recovery restrictions

- When a job with a failed adapter is preempted, LoadLeveler may not be able to continue with the job if it (LoadLeveler) cannot reload the switch table on the failed adapter. Any adapter failure that causes switch tables to be unloaded will not be recovered during the job run.
- In single-network scenarios, LoadLeveler attempts to allocate adapter windows on separate adapters, but does not always succeed. Correspondingly, failover and recovery are not always possible in single-network scenarios. The user will get POE attention messages at job startup time when LoadLeveler fails to get windows on at least two separate adapters.
- Failover and recovery are not supported for non-snX adapters or for standalone (non-POE) LAPI.

Data striping

When running parallel jobs on processors with High Performance Switches or pSeries High Performance Switches, it is possible to stripe data through multiple adapter windows. This is supported for both IP and US protocols.

If the system has more than one switch network, the resource manager allocates adapter windows from multiple adapters. A switch network is the circuit of adapters that connect to the same HPS or pSeries HPS. One window is assigned to an adapter, with one adapter each selected from a different switch network.

If the system has only one switch network, the adapter windows are most likely allocated from different adapters, provided that there are sufficient windows available on each adapter. If there are not enough windows available on one of the adapters, the adapter windows may all be allocated from a single adapter.

LAPI manages communication among multiple adapter windows. Using resources that LoadLeveler allocates, LAPI opens multiple user space windows for communication. Every task of the job opens the same number of user space windows, and a particular window on a task can only communicate with the corresponding window on other tasks. These windows form a set of "virtual networks", in which each "virtual network" consists of a window from each task that can communicate with the corresponding windows from the other tasks. The distribution of data among the various windows on a task is referred to as *striping*, which has the potential to improve communication bandwidth performance for LAPI clients.

To enable striping in user space mode, use environment variable settings that result in the allocation of multiple instances. For a multi-network system, this can be done by setting **MP_EUIDEVICE** to **sn_all**. On a single-network system with multiple adapters per operating system image, this can be done by setting **MP_EUIDEVICE** to **sn_single** and **MP_INSTANCES** to a value that is greater than 1. See "Requesting the use of multiple adapters" on page 104 for more information.

For example, on a node with two adapter links, in a configuration where each link is part of a separate network, the result of setting **MP_EUIDEVICE** to **sn_all** is a window on each of the two networks, which are independent paths from one node to others. For IP communication and for messages that use the user space FIFO mechanism (in which LAPI creates packets and copies them to the user space FIFOs for transmission), striping provides no performance improvement. Therefore, LAPI does not perform striping for short messages, non-contiguous messages, and all communication in which bulk transfer is disabled through environment variable settings.

For large contiguous messages that use bulk transfer, striping provides a vast improvement in communication performance. Bandwidth scaling is nearly linear with the number of adapters (up to a limit of 8) for sufficiently-large messages. This improvement in communication bandwidth stems from: 1) the low overhead that is needed to initiate the remote direct memory access (RDMA) operations used to facilitate the bulk transfer, 2) the major proportion of RDMA work that is being done by the adapters, and 3) high levels of concurrency in the RDMA operations for various parts of the contiguous message that are being transferred by RDMA by each of the adapters. For more information about RDMA, see Chapter 13, "Bulk transfer of messages," on page 99.

To activate striping or failover for an interactive parallel job, you must set the **MP_EUIDEVICE** and **MP_INSTANCES** environment variables as follows:

- For instances from multiple networks:
MP_EUIDEVICE=sn_all — Guarantees that the adapters assigned will be from different networks.
- For instances from a single network:
MP_EUIDEVICE=sn_single and **MP_INSTANCES=*n*** (where *n* is greater than 1 and less than *max_protocol_instances*) — Improved striping performance using RDMA can only be seen if windows are allocated from multiple adapters on the

single network. Such an allocation may not be possible if there is only one adapter on the network or if there are multiple adapters, but there are available resources on only one of the adapters.

To activate striping for a parallel job submitted to the LoadLeveler batch system, the network statement of the LoadLeveler command file must be coded accordingly.

- Use this network statement for a LAPI US job that uses pSeries High Performance Switches on multiple networks:

```
#@ network.lapi = sn_all,shared,us
```

- Use this network statement for an MPI and LAPI US job that uses pSeries High Performance Switches on multiple networks and shares adapter windows:

```
#@ network.mpi_lapi = sn_all,shared,us
```

The value of **MP_INSTANCES** ranges from **1** to the maximum value specified by *max_protocol_instances*, as defined in the LoadLeveler **LoadL_admin** file. The default value of *max_protocol_instances* is **1**. See *LoadLeveler for AIX 5L and Linux: Using and Administering* for more information.

Communication and memory considerations

Depending on the mode of communication, when multiple HPS or pSeries HPS adapters are used for data striping or for failover and recovery, additional memory or address space resources are used for data structures that are associated with each communication instance. In 32-bit applications, these additional requirements have implications that you must consider before deciding whether to use striping or failover and recovery and the extent to which you will use these functions.

IP communication

- When multiple HPS or pSeries HPS instances are used for IP communication, LAPI allocates these data structures from the user heap. Some 32-bit applications may therefore need to be recompiled to use additional data segments for their heap by using the **-bmaxdata** compilation flag and requesting a larger number of segments. The default amount of data that can be allocated for 64-bit programs is practically unlimited, so no changes are needed. Alternatively, you can modify the 32-bit executable using the **ldedit** command or by setting the **LDR_CNTRL** environment variable to **MAXDATA**. Base the increase to **-bmaxdata** on what is needed rather than setting it to the maximum allowed (**0x80000000**). Using more segments than required may make certain shared memory features unusable, which can result in poor performance. Also, applications that require the eight allowed segments for their own user data (thus leaving no space for LAPI to allocate structures) must use a single IP instance only (**MP_EUIDevice=sn_single**).

For more information about **ldedit**, see *AIX 5L Version 5.2 Commands Reference* or *AIX 5L Version 5.3 Commands Reference*. For more information about **LDR_CNTRL**, see *AIX 5L Version 5.2: Performance Management Guide* or *AIX 5L Version 5.3: Performance Management Guide*.

- When multiple adapters from each of one or more networks are used for IP communication, with multiple adapters in each IP subnet, failover can occur only if the AIX IP routing table is updated appropriately to remove the failed adapters. This is because: 1) AIX's IP routing alternates communication to a given target IP address among all the routes to that target in its routing table, 2) with multiple adapters on each IP subnet, multiple routes may be detected and stored in the IP routing table for each remote adapter that is on the same subnet and, 3) the loss of an adapter does not automatically result in an update of the IP routing table. Without such an update to the routing table, the underlying route used to

reach a given target IP address may still go through a failed adapter connection internally, thus preventing proper communication even if LAPI has failed over to a UDP socket that leads to a target adapter that is detected to be still functioning well. This problem *does not* affect the case where the multiple adapters on each node are organized one each on separate IP subnets. See *AIX 5L Version 5.2: System Management Guide, Communications and Networks* or *AIX 5L Version 5.3: System Management Guide, Communications and Networks* for more information.

US communication

When multiple HPS or pSeries HPS instances are used for US communication, you need to consider the following segment usage information when deciding whether to use striping or failover and recovery. The communication subsystem uses segment registers for several different purposes. The AIX memory model for 32-bit applications uses five segment registers. In a 32-bit executable, there are only 16 segment registers available. In a 64-bit executable, the number of segment registers is essentially unbounded. Because segment registers are abundant in 64-bit job runs, this discussion is important only for 32-bit job runs.

By default, the amount of memory that is available for application data structures (the heap) in a 32-bit job run is somewhat less than 256MB. You can use the compilation flag **-bmaxdata:0x80000000** to allocate 2GB of heap, but this requires eight segment registers. Smaller **-bmaxdata** values use fewer segment registers, but these values limit the size of application data structures. If you try to use every available feature of the communication subsystem and allow 2GB for heap, there will not be enough registers, and your application will lose some performance or perhaps not be able to start. The communication subsystem uses segments as follows:

- One US instance (window): 2
- Each additional instance: 1
- Switch clock: 1 (applies *only* to MPI when it is used in conjunction with LAPI on the HPS or the pSeries HPS)
- Shared memory: 1
- Shared memory cross-memory attach: 1

Using MPI and LAPI together with separate windows consumes segments beyond the minimum. Using striping also consumes extra windows. When MPI is used in conjunction with LAPI, access to the switch clock by MPI for the **MPI_WTIME_IS_GLOBAL** attribute requires a dedicated segment register on the HPS or the pSeries HPS. Turning shared memory communication on requires one segment register for basic functions and a second segment register to exploit cross-memory attach, to accelerate large messages between tasks on the same node. If your application requires a large heap, you may need to forgo some communication subsystem options. For most MPI applications, if you are using the HPS or the pSeries HPS, you can set **MP_CLOCK_SOURCE=AIX** and free one register. If MPI and LAPI calls are used in the application, make sure **MP_MSG_API** is set to **MPI_LAPI** rather than **MPI,LAPI**. Because shared memory uses one pair of registers per protocol, using **MPI_LAPI** rather than **MPI,LAPI** is especially important when combining shared memory and user space. If you do not need to use the striping and failover functions, make sure that **MP_EUIDEVICE** is set to **sn_single** and that **MP_INSTANCES** is not set (in which case, it defaults to **1**) or is set to **1** explicitly.

| For 32-bit executables that are compiled to use small pages, the segment registers
| that are reserved by AIX and by **-bmaxdata** are claimed first. The initialization of
| user space comes second. If there are not enough registers left, your job will not
| start. The initialization of shared memory comes last. If there are no registers left,
| the job will still run, but without shared memory. If there is only one register left,
| shared memory will be enabled, but the optimization to speed large messages with
| cross-memory attach will not be used. If there are no registers left, shared memory
| will be bypassed and on-node communication will go through the network.

| For 32-bit executables that use large pages, dynamic segment allocation (DSA) is
| turned on automatically, so any **-bmaxdata** segments requested are not reserved
| first for the user heap, but are instead allocated in the order of usage. Thus, if the
| program allocates memory corresponding to the total size of the requested
| **-bmaxdata** segments before **MPI_Init** or **LAPI_Init** is called, the behavior would be
| similar to the small page behavior that is described in the previous paragraph.
| However, if **MPI_Init** or **LAPI_Init** is called before the memory allocation, segments
| that were intended for use for the program heap may be first obtained and reserved
| for windows and for communication library features such as shared memory. In this
| case, the program will be left with fewer segments to grow the heap than
| **-bmaxdata** had requested. The program is likely to start by claiming all the
| segments required for the initialization of the communication subsystem, but will
| terminate later in the job run on a **malloc** failure as its data structure allocations
| grow to fill the space that the specified **-bmaxdata** value was expected to provide.

| For information about how to use large pages, see *AIX 5L Version 5.2:
| Performance Management Guide* or *AIX 5L Version 5.3: Performance Management
| Guide*. For information about DSA, see *AIX 5L Version 5.2: General Programming
| Concepts, Writing and Debugging Programs* or *AIX 5L Version 5.3: General
| Programming Concepts, Writing and Debugging Programs*.

Chapter 15. Threaded programming

General guidelines

LAPI has no concept of identifying individual threads within a task. No communication can be directed to a specific thread.

In multi-threaded programming, synchronization among threads is the user's responsibility. All LAPI communication is based on the use of handles. If multiple threads share the same handle, their calls to LAPI are serialized in the LAPI library, which makes LAPI safe for multi-threaded programming.

It is recommended that you don't write threaded message-passing programs until you are very familiar with writing and debugging single-threaded multitask programs and multithreaded single-task programs.

Using LAPI_Address_init

Here is an example of using **LAPI_Address_init** in multiple threads. Suppose there are two tasks creating two threads to do the exchange of addresses as follows.

	Task 1	Task2
Thread 1	LAPI_Address_init(hndl, addr1, addr_tab1);	LAPI_Address_init(hndl, addr1, addr_tab1);
Thread 2	LAPI_Address_init(hndl, addr2, addr_tab2);	LAPI_Address_init(hndl, addr2, addr_tab2);

The user expects that all *addr1* will be collected in *addr_tab1* and all *addr2* will be collected in *addr_tab2*. However, if Thread 1 and Thread 2 are running completely in parallel for this code section, the resulting serialization in LAPI can be any one of the following sequences.

Sequence 1	Sequence 2
LAPI_Address_init(hndl, addr1, addr_tab1);	LAPI_Address_init(hndl, addr2, addr_tab2);
LAPI_Address_init(hndl, addr2, addr_tab2);	LAPI_Address_init(hndl, addr1, addr_tab1);

Suppose Task 1 is serialized into Sequence 1 and Task 2 into Sequence 2. The result of address exchange in this scenario will be unexpected because *addr1* of Task 1 will be put into *addr_tab2* of Task 2 and *addr2* of Task 2 will be put into *addr_tab1* of Task 1. To achieve the desired results in *addr_tab1* and *addr_tab2*, you can either put the two **LAPI_Address_init** calls in one thread or use your own thread synchronization mechanism to enforce ordering of the two **LAPI_Address_init** calls. For example, the following code allows threads to exchange counter addresses group by group.

```
volatile int turn = 0;
void *communication_thread(void *param)
{
    int group_id = (int)param;
    lapi_cntr_t tgt_cntr;
    void **tgt_cntr_tab;
    tgt_cntr_tab = malloc(num_tasks * sizeof(void *));
    while (turn != group_id) yield();
    LAPI_Address_init(hndl, &tgt_cntr, tgt_cntr_tab);
    turn++;
    ...
}
```

Making global fence calls

Synchronization of one set of threads across all tasks of the job concurrently with synchronization of another set of threads across all the same tasks cannot be achieved by making one **LAPI_Gfence** call in each thread. This is because, just as with the example of **LAPI_Address_init** above, each task will locally serialize the **LAPI_Gfence** call in undefined order, and it is unpredictable whether the **LAPI_Gfence** call for thread A on task 0 will match the **LAPI_Gfence** call for thread A or thread B on task 1. If you intend to make two global fence calls in two different threads, you should enforce the ordering of the two calls similar to the example of **LAPI_Address_init** ordering above.

Making "wait on counter" calls

Multiple **LAPI_Waitcntr** calls can be issued from different threads without one blocking the others. Note, however, that two threads cannot wait on the same counter. **LAPI_Waitcntr** will return to the user as soon as the counter reaches or exceeds the value that is being waited on, which also implies that an earlier **LAPI_Waitcntr** may return after a later **LAPI_Waitcntr**.

Synchronizing threads across tasks

To synchronize threads in different tasks, you may not want to call **LAPI_Gfence** because it blocks other threads in the same task from any further communication before the global fence operation is complete. A better way to implement such synchronization uses other LAPI functions. The following steps show a possible implementation:

1. Pick a root thread from the group of threads to synchronize.
2. All threads in the group send a barrier message to the root thread task (**LAPI_Amsend**, **LAPI_Put**) with an associated target counter.
3. All threads wait for a response by waiting on a specific counter to be incremented (**LAPI_Waitcntr**).
4. The root thread waits for its counter to reach a value that is equal to the number of threads in this group (**LAPI_Waitcntr**).
5. The root thread then broadcasts a message to all of the threads in the group specifying the counter they are waiting on as the target counter, to release them from the barrier (**LAPI_Amsend**, **LAPI_Put**).

Using handlers

Typically, LAPI invokes the following user-provided handlers to indicate the arrival/completion of a message:

- Header handler: when data first arrives at the receiving side in an active message.
- Send completion handler: when data has been sent at the sending side and the user can modify the data buffer.
- (Receive) Completion handler: when data has been completely received at the receiving side.

LAPI does not guarantee which thread will invoke the above handlers, with the exception of non-inline completion handlers, which are always invoked in the completion handler thread created by LAPI. All other types of handlers can be

invoked in any thread where LAPI communication functions are called and also in the interrupt thread. Refer to the next section for a description of the threads in an executing LAPI program.

LAPI threads

A program running LAPI is inherently multi-threaded, even though the user program may itself be single-threaded. The list of threads in a program running with LAPI is as follows:

- User threads: created by the user.
- Interrupt thread: When interrupt is enabled and there are incoming packets from the adapter, LAPI interrupt handler is called in this thread to process packets. All user-provided handlers except non-inline completion handlers can be invoked in this thread.
- Completion handler threads: LAPI creates these threads to run non-inline completion handlers.
- Shared memory dispatcher thread. If shared memory is on, LAPI creates this thread to handle interrupt in shared memory transport. All user-provided handlers except non-inline completion handlers can be invoked in this thread.

A single-threaded LAPI user program has one user thread, but may have one thread of each of the other types that are transparent to the user.

Chapter 16. Using LAPI on a standalone system

This section includes information that applies only to the use of LAPI on a standalone system. In this book, a *standalone system* refers to a system that is *not* running IBM's Parallel Environment for AIX 5L (PE) licensed program.

Standalone setup

On a standalone system, you need to assign task IDs and initialize jobs on each node. To use LAPI on a standalone system, set the following environment variables:

MP_CHILD Sets the task ID of the current job. **MP_CHILD** needs to be set to a unique value for each task in standalone mode.

MP_LAPI_INET_ADDR
Describes the network setup among LAPI tasks for IP communication. The format is:

```
MP_LAPI_INET_ADDR=@num_instances:IP_addr_in_dotted_decimal,adapter_name[:IP_addr_in_dotted_decimal2,adapter_name2...]
```

where: *num_instances* is the number of network instances, *IP_addr_in_dotted_decimal* is the IP address of the adapter being used, and *adapter_name* is the logical name of the adapter device. The number of *IP_addr_in_dotted_decimal,adapter_name* pairs is equal to the value of *num_instances*. For example, if you want a task to use two separate adapters, **sn0** and **sn1**, for IP communication, you would use something like this:

```
MP_LAPI_INET_ADDR=@2:192.161.0.1,sn0:192.161.1.1,sn1
```

MP_LAPI_NETWORK

Describes the network setup for user space communication among LAPI tasks. The format is:

```
MP_LAPI_NETWORK=@num_instances>window_num,adapter_name[:window_num2,adapter_name2...]
```

where: *num_instances* is the number of network instances and *window_num* and *adapter_name* refer to one or more adapter windows that have been reserved for this task. The number of *window_num,adapter_name* pairs is equal to the value of *num_instances*. For example, if you loaded one adapter instance for this task using window **164** on adapter **sn0**, you would use:

```
MP_LAPI_NETWORK=@1:164,sn0
```

See the **README.LAPI.STANDALONE.US** file in the **lapi/samples/standalone/us** directory of the LAPI samples files for more information. For more information on LAPI sample files, see Chapter 20, "LAPI sample programs," on page 245.

MP_PARTITION

A number that is the same for all tasks in the job. In standalone mode, you need to set this variable to an identical value for each task. In standalone mode for switched communication, the **MP_PARTITION** value must be associated with the network table description file.

MP_PROCS The value of *num_tasks*, which is the total number of program tasks in the job. This number must be the same for all tasks.

See “Variables for standalone systems” on page 274 for more information.

To use LAPI shared memory on a standalone system, set the following environment variables:

LAPI_USE_SHM	Enables or disables the use of shared memory. no -- disables the use of shared memory (the default). yes -- enables the use of shared memory where it is possible. LAPI will communicate using shared memory among all common tasks (tasks that are on the same node) over the selected device (user space over switch, IP over switch, or IP over Ethernet). See MP_EUIDEVICE and MP_EUILIB for tasks on different nodes. Shared memory requires segment registers, which can affect availability to user code in 32-bit applications. only -- communicates only using shared memory. LAPI will fail to initialize if this option is chosen and tasks are assigned to more than one node.
MP_COMMON_TASKS	Is set for shared memory jobs. It is different for each task, and is mapped to the setting of the MP_CHILD environment variable. For each task, MP_COMMON_TASKS contains a string that indicates the number and task IDs of other tasks on the same node (that is, those that can communicate through shared memory).

For **MP_COMMON_TASKS**, the format of the string is:

MP_COMMON_TASKS=*number-of-common-tasks:common-task:other-task-2:...*

For example, for task 1 of a 4-task job running on the same node, the following environment value is set for task 1: **MP_COMMON_TASKS=3:0:2:3**. Notice that task 1 is not in the list. Task numbering starts at 0 to *ntask - 1*, where *ntask* is the total number of tasks.

See “Variables for shared memory” on page 274 for more information.

Note

The descriptions and formats of **MP_COMMON_TASKS**, **MP_LAPI_INET_ADDR**, **MP_LAPI_NETWORK** are provided in this book for informational purposes only. These environment variables are not intended to be used as external programming interfaces. IBM will not guarantee that the formats or values of these variables can continue to be used without change in future releases. Programmers and users who choose to develop applications that depend on these variables do so with the understanding that these variables may be subject to future change. IBM cannot guarantee that such applications can migrate or coexist with future releases without additional changes, nor will IBM ensure that there will be binary compatibility of these variables.

Standalone initialization

See “LAPI_Init” on page 163 for examples of standalone initialization.

Using UDP/IP mode

For standalone UDP/IP initialization, LAPI must have at least one of these two means of transferring UDP information: a user handler or a user list. If both a user handler and a user list are passed in, the user handler is invoked and the user list is ignored.

In UDP/IP mode, LAPI uses a pair of connectionless sockets for each task, one for reading and one for writing. During initialization, the IP address and port information for each task’s read socket must be distributed to all tasks. On a standalone system, you need to distribute this read socket information to all of the tasks. LAPI provides two mechanisms for distributing IP address and port information on standalone systems: user handlers and user lists.

For the user handler mechanism, a user handler is passed to LAPI as a callback pointer to be used during initialization. Before opening the UDP sockets during initialization (in **LAPI_Init**), LAPI calls the handler and expects it to return (by way of a reference parameter) a list of IP address and port information for each task in the job.

For the user list mechanism, you need to pass a pointer to LAPI at initialization time (by way of the **lapi_info_t** structure that is passed in to **LAPI_Init**) that points to user memory that has the required port information.

Using US mode

For standalone initialization in user space (US), no changes are required to C source code that makes LAPI calls, except that the code must be compiled with a non-parallel compiler (**cc_r**, for example). When running a user space program standalone, you need to handle a number of tasks that are normally handled by PE and LoadLeveler. In particular, you need to:

- Determine available adapters and windows, and then load the network tables for the desired network configuration on the chosen adapters and windows.
- Set the **MP_LAPI_NETWORK** environment variable in the following format:

```
MP_LAPI_NETWORK=@num_instances>window_num,snX
```

where *num_instances* is the number of network instances to use, *window_num* is the window on the adapter that has been loaded for this task and *snX* is the name of the adapter that has this window and on which the network table has been loaded. For example, if you loaded one adapter instance for this task using window 164 on adapter sn0, you would set **MP_LAPI_NETWORK** as follows:

```
MP_LAPI_NETWORK=@1:164,sn0
```

Note that for this release of LAPI, *num_instances* must be set to 1.

- Set other environment variables to control various aspects of the LAPI run time. See “Environment variables” on page 269 for more information.
- Invoke each of the tasks separately on the desired set of nodes on which the job is to be run.

For full details on standalone initialization in user space, see the **README.LAPI.STANDALONE.US** file in the **standalone/us** directory of the LAPI sample files.

Compiling LAPI programs on a standalone system

Table 13 shows what commands to enter to compile a LAPI program on a standalone system.

Table 13. Compiling LAPI programs on a standalone system

To compile a C program	cc_r program.c -o program
To compile a C++ program	CC_r program.C -o program
To compile a FORTRAN program	xlf_r program.f -o program

Part 4. LAPI reference

	Chapter 17. LAPI man pages.	121
	lapi_subroutines	122
	Chapter 18. Subroutines for all systems (PE and standalone).	125
	LAPI_Addr_get	126
	LAPI_Addr_set	128
	LAPI_Address	130
	LAPI_Address_init	132
	LAPI_Address_init64	134
	LAPI_Amsend	136
	LAPI_Amsendv	143
	LAPI_Fence	149
	LAPI_Get	151
	LAPI_Getcptr	154
	LAPI_Getv	156
	LAPI_Gfence	161
	LAPI_Init	163
	LAPI_Msg_string	169
	LAPI_Msgpoll	171
	LAPI_Probe	174
	LAPI_Put	176
	LAPI_Putv	179
	LAPI_Qenv	184
	LAPI_Rmw	188
	LAPI_Rmw64	192
	LAPI_Senv	196
	LAPI_Setcptr	198
	LAPI_Term	201
	LAPI_Util	203
	LAPI_Waitcptr	217
	LAPI_Xfer	219
	Chapter 19. Subroutines for standalone systems.	235
	LAPI_Nopoll_wait	236
	LAPI_Purge_totask	238
	LAPI_Resume_totask	240
	LAPI_Setcptr_wstatus	242
	Chapter 20. LAPI sample programs	245
	Sample program directory structure	245
	Using the LAPI sample programs	250
	Summary of constructs and techniques for LAPI programming	250

Chapter 17. LAPI man pages

For each subroutine in Chapter 18, “Subroutines for all systems (PE and standalone),” on page 125 and Chapter 19, “Subroutines for standalone systems,” on page 235, information about some or all of the following topics is included, as appropriate: purpose, library, C syntax, FORTRAN syntax, parameters, description, restrictions, return values, location, C examples, FORTRAN examples, and related information. Review “lapi_subroutines” on page 122 before proceeding to get a better understanding of how the subroutine information in Chapter 18, “Subroutines for all systems (PE and standalone),” on page 125 and Chapter 19, “Subroutines for standalone systems,” on page 235 is structured.

lapi_subroutines

Purpose

Provides overview information about LAPI subroutines, including some sample sections of the man pages for these subroutines.

Library

Availability Library (**liblapi_r.a**)

C syntax

```
#include <lapi.h>

int lapi_subroutines(parm1, parm2...)
type1 parm1;
type2 parm2;
:
```

FORTRAN syntax

```
include 'lapif.h'

LAPI_SUBROUTINES(parm1, parm2..., ierror)
TYPE1 :: parm1;
TYPE2 :: parm2;
:
INTEGER ierror
```

Parameters

Parameter definitions are listed as follows:

INPUT

parm1 Describes *parm1*.

INPUT/OUTPUT

parm2 This section includes all LAPI counters.

OUTPUT

Function calls are nonblocking, so counter behavior is asynchronous with respect to the function call.

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

This man page provides overview information about LAPI subroutines, including some sample sections of the man pages for these subroutines.

Programming with C++

LAPI subroutines provide **extern "C"** declarations for C++ programming.

Profiling

See Chapter 9, "Using LAPI's profiling interface," on page 65 for more information.

Querying runtime values

You can find out the size (or size range) of certain parameters by calling the **LAPI_Qenv** subroutine with the appropriate query type. For example, call **LAPI_Qenv** with the **LOC_ADDRTBL_SZ** query type to find out the size of the address table used by the **LAPI_Addr_set** subroutine:

```
LAPI_Qenv(hdl, LOC_ADDRTBL_SZ, ret_val)
```

Now, suppose you want to register a function address using **LAPI_Addr_set**:

```
LAPI_Addr_set (hdl, addr, addr_hdl)
```

The value of index *addr_hdl* must be in the range:

```
1 <= addr_hdl < LOC_ADDRTBL_SZ
```

When used to show the size of a parameter, a comparison of values, or a range of values, valid values for the *query* parameter of the **LAPI_Qenv** subroutine appear in **SMALL, BOLD** capital letters. For example:

```
NUM_TASKS
```

is a shorthand notation for:

```
LAPI_Qenv(hdl, NUM_TASKS, ret_val)
```

See “LAPI_Qenv” on page 184 for a list of the *query* parameter’s valid values.

Restrictions

Any specific restrictions for the subroutine appear here.

Also, see Appendix G, “LAPI restrictions,” on page 285 for more information.

Return values

```
LAPI_SUCCESS
```

Indicates that the function call completed successfully.

Any other return values for the subroutine appear here.

For a complete list, see “LAPI return values” on page 264.

For information about LAPI error messages, see *RSCT: Messages*.

Location

```
/usr/lib/liblapi_r.a
```

C examples

Any C examples of the subroutine appear here.

lapi_subroutines

FORTRAN examples

Any FORTRAN examples of the subroutine appear here.

Related information

Any information that is related to the subroutine (including names of related subroutines) appears here.

Chapter 18. Subroutines for all systems (PE and standalone)

Use the subroutines in this chapter on systems that are running Parallel Environment (PE) and on standalone systems.

LAPI_Addr_get

Purpose

Retrieves a function address that was previously registered using **LAPI_Addr_set**.

Library

Availability Library (**liblapi_r.a**)

C syntax

```
#include <lapi.h>

int LAPI_Addr_get(hndl, addr, addr_hndl)
lapi_handle_t hndl;
void **addr;
int addr_hndl;
```

FORTRAN syntax

```
include 'lapif.h'

LAPI_ADDR_GET(hndl, addr, addr_hndl, ierror)
INTEGER hndl
INTEGER (KIND=LAPI_ADDR_TYPE) :: addr
INTEGER addr_hndl
INTEGER ierror
```

Parameters

INPUT

hndl Specifies the LAPI handle.

addr_hndl Specifies the index of the function address to retrieve. You should have previously registered the address at this index using **LAPI_Addr_set**. The value of this parameter must be in the range $1 \leq \text{addr_hndl} < \underline{\text{LOC_ADDRTBL_SZ}}$.

OUTPUT

addr Returns a function address that the user registered with LAPI.

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: local address manipulation

Use this subroutine to get the pointer that was previously registered with LAPI and is associated with the index *addr_hndl*. The value of *addr_hndl* must be in the range $1 \leq \text{addr_hndl} < \underline{\text{LOC_ADDRTBL_SZ}}$.

Return values

LAPI_SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_ADDR_HNDL_RANGE Indicates that the value of *addr_hndl* is not in the

range

$1 \leq addr_hdl < LOC_ADDRTBL_SZ$.

LAPI_ERR_HNDL_INVALID Indicates that the *hdl* passed in is not valid (not initialized or in terminated state).

LAPI_ERR_RET_PTR_NULL Indicates that the value of the *addr* pointer is NULL (in C) or that the value of *addr* is **LAPI_ADDR_NULL** (in FORTRAN).

Location

`/usr/lib/liblapi_r.a`

C examples

To retrieve a header handler address that was previously registered using **LAPI_Addr_set**:

```
lapi_handle_t  hndl;      /* the LAPI handle          */
void          **addr;     /* the address to retrieve */
int           addr_hndl; /* the index returned from LAPI_Addr_set */
:
:
addr_hndl = 1;
LAPI_Addr_get(hndl, &addr, addr_hndl);

/* addr now contains the address that was previously registered */
/* using LAPI_Addr_set                                         */
```

Related information

Subroutines: **LAPI_Addr_set**, **LAPI_Qenv**

LAPI_Addr_set

Purpose

Registers the address of a function.

Library

Availability Library (**liblapi_r.a**)

C syntax

```
#include <lapi.h>

int LAPI_Addr_set(hndl, addr, addr_hndl)
lapi_handle_t hndl;
void *addr;
int addr_hndl;
```

FORTRAN syntax

```
include 'lapif.h'

LAPI_ADDR_SET(hndl, addr, addr_hndl, ierror)
INTEGER hndl
INTEGER (KIND=LAPI_ADDR_TYPE) :: addr
INTEGER addr_hndl
INTEGER ierror
```

Parameters

INPUT

hndl Specifies the LAPI handle.

addr Specifies the address of the function handler that the user wants to register with LAPI.

addr_hndl Specifies a user function address that can be passed to LAPI calls in place of a header handler address. The value of this parameter must be in the range $1 \leq \text{addr_hndl} < \underline{\text{LOC_ADDRTBL_SZ}}$.

OUTPUT

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: local address manipulation

Use this subroutine to register the address of a function (*addr*). LAPI maintains the function address in an internal table. The function address is indexed at location *addr_hndl*. In subsequent LAPI calls, *addr_hndl* can be used in place of *addr*. The value of *addr_hndl* must be in the range $1 \leq \text{addr_hndl} < \underline{\text{LOC_ADDRTBL_SZ}}$.

For active message communication, you can use *addr_hndl* in place of the corresponding header handler address. LAPI only supports this indexed substitution for remote header handler addresses (but not other remote addresses, such as target counters or base data addresses). For these other types of addresses, the actual address value must be passed to the API call.

Return values

LAPI_SUCCESS	Indicates that the function call completed successfully.
LAPI_ERR_ADDR_HNDL_RANGE	Indicates that the value of <i>addr_hndl</i> is not in the range $1 \leq \text{addr_hndl} < \text{LOC_ADDRTBL_SZ}$.
LAPI_ERR_HNDL_INVALID	Indicates that the <i>hndl</i> passed in is not valid (not initialized or in terminated state).

Location

`/usr/lib/liblapi_r.a`

C examples

To register a header handler address:

```

lapi_handle_t hndl;      /* the LAPI handle           */
void          *addr;     /* the remote header handler address */
int           addr_hndl; /* the index to associate          */

:

addr = my_func;
addr_hndl = 1;
LAPI_Addr_set(hndl, addr, addr_hndl);

/* addr_hndl can now be used in place of addr in LAPI_Amsend, */
/* LAPI_Amsendv, and LAPI_Xfer calls                          */
:

```

Related information

Subroutines: **LAPI_Addr_get**, **LAPI_Amsend**, **LAPI_Amsendv**, **LAPI_Qenv**, **LAPI_Xfer**

LAPI_Address

Purpose

Returns an unsigned long value for a specified user address.

Library

Availability Library (**liblapi_r.a**)

C syntax

```
#include <lapi.h>

int LAPI_Address(my_addr, ret_addr)
void *my_addr;
ulong *ret_addr;
```

Note: This subroutine is meant to be used by FORTRAN programs. The C version of **LAPI_Address** is provided for compatibility purposes only.

FORTRAN syntax

```
include 'lapif.h'

LAPI_ADDRESS(my_addr, ret_addr, ierror)
INTEGER (KIND=any_fortran_type) :: my_addr
INTEGER (KIND=LAPI_ADDR_TYPE) :: ret_addr
INTEGER ierror
```

where:

any_fortran_type

Is any FORTRAN datatype. This type declaration has the same meaning as the type **void *** in C.

Parameters

INPUT

my_addr Specifies the address to convert. The value of this parameter cannot be NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

OUTPUT

ret_addr Returns the address that is stored in *my_addr* as an unsigned long for use in LAPI calls. The value of this parameter cannot be NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: local address manipulation

Use this subroutine in FORTRAN programs when you need to store specified addresses in an array. In FORTRAN, the concept of address (**&**) does not exist as it does in C. **LAPI_Address** provides FORTRAN programmers with this function.

Return values

LAPI_SUCCESS	Indicates that the function call completed successfully.
LAPI_ERR_ORG_ADDR_NULL	Indicates that the value of <i>my_addr</i> is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_TGT_ADDR_NULL	Indicates that the value of <i>ret_addr</i> is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

Location

`/usr/lib/liblapi_r.a`

FORTRAN examples

To retrieve the address of a variable:

```
! Contains the address of the target counter
INTEGER (KIND=LAPI_ADDR_TYPE) :: cntr_addr

! Target counter
TYPE (LAPI_CNTR_T) :: tgt_cntr

! Return code
INTEGER :: ierror

CALL LAPI_ADDRESS(tgt_cntr, cntr_addr, ierror)

! cntr_addr now contains the address of tgt_cntr
```

Related information

Subroutines: **LAPI_Address_init**, **LAPI_Address_init64**

LAPI_Address_init

Purpose

Creates a remote address table.

Library

Availability Library (**liblapi_r.a**)

C syntax

```
#include <lapi.h>

int LAPI_Address_init(hndl, my_addr, add_tab)
lapi_handle_t hndl;
void          *my_addr;
void          *add_tab[ ];
```

FORTRAN syntax

```
include 'lapif.h'

LAPI_ADDRESS_INIT(hndl, my_addr, add_tab, ierror)
INTEGER hndl
INTEGER (KIND=LAPI_ADDR_TYPE) :: my_addr
INTEGER (KIND=LAPI_ADDR_TYPE) :: add_tab(*)
INTEGER ierror
```

Parameters

INPUT

hndl Specifies the LAPI handle.

my_addr Specifies the entry supplied by each task. The value of this parameter can be NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

OUTPUT

add_tab Specifies the address table containing the addresses that are to be supplied by all tasks. *add_tab* is an array of pointers, the size of which is greater than or equal to **NUM_TASKS**. The value of this parameter cannot be NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: collective communication (blocking)

LAPI_Address_init exchanges virtual addresses among tasks of a parallel application. Use this subroutine to create tables of such items as header handlers, target counters, and data buffer addresses.

LAPI_Address_init is a *collective call* over the LAPI handle *hndl*, which fills the table *add_tab* with the virtual address entries that each task supplies. Collective calls must be made in the same order at all participating tasks.

The addresses that are stored in the table *add_tab* are passed in using the *my_addr* parameter. Upon completion of this call, *add_tab[i]* contains the virtual address entry that was provided by task *i*. The array is opaque to the user.

Return values

LAPI_SUCCESS	Indicates that the function call completed successfully.
LAPI_ERR_COLLECTIVE_PSS	Indicates that a collective call was made while in persistent subsystem (PSS) mode.
LAPI_ERR_HNDL_INVALID	Indicates that the <i>hndl</i> passed in is not valid (not initialized or in terminated state).
LAPI_ERR_RET_PTR_NULL	Indicates that the value of the <i>add_tab</i> pointer is NULL (in C) or that the value of <i>add_tab</i> is LAPI_ADDR_NULL (in FORTRAN).

Location

`/usr/lib/liblapi_r.a`

C examples

To collectively transfer target counter addresses for use in a communication API call, in which all nodes are either 32-bit or 64-bit:

```
lapi_handle_t hndl;           /* the LAPI handle */
void          *addr_tbl[NUM_TASKS]; /* the table for all tasks' addresses */
lapi_cntr_t   tgt_cntr;      /* the target counter */
:
:
LAPI_Address_init(hndl, (void *)&tgt_cntr, addr_tbl);

/* for communication with task t, use addr_tbl[t] */
/* as the address of the target counter */
:
:

```

For a combination of 32-bit and 64-bit nodes, use **LAPI_Address_init64**.

Related information

Subroutines: **LAPI_Address**, **LAPI_Address_init64**

LAPI_Address_init64

Purpose

Creates a 64-bit remote address table.

Library

Availability Library (**liblapi_r.a**)

C syntax

```
#include <lapi.h>

int LAPI_Address_init64(hndl, my_addr, add_tab)
lapi_handle_t hndl;
lapi_long_t my_addr;
lapi_long_t *add_tab;
```

FORTRAN syntax

```
include 'lapif.h'

LAPI_ADDRESS_INIT64(hndl, my_addr, add_tab, ierror)
INTEGER hndl
INTEGER (KIND=LAPI_ADDR_TYPE) :: my_addr
INTEGER (KIND=LAPI_LONG_LONG_TYPE) :: add_tab(*)
INTEGER ierror
```

Parameters

INPUT

hndl Specifies the LAPI handle.

my_addr Specifies the address entry that is supplied by each task. The value of this parameter can be NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN). To ensure 32-bit/64-bit interoperability, it is passed as a **lapi_long_t** type in C.

OUTPUT

add_tab Specifies the 64-bit address table that contains the 64-bit values supplied by all tasks. *add_tab* is an array of type **lapi_long_t** (in C) or **LAPI_LONG_LONG_TYPE** (in FORTRAN). The size of *add_tab* is greater than or equal to **NUM_TASKS**. The value of this parameter cannot be NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: collective communication (blocking)

LAPI_Address_init64 exchanges virtual addresses among a mixture of 32-bit and 64-bit tasks of a parallel application. Use this subroutine to create 64-bit tables of such items as header handlers, target counters, and data buffer addresses.

LAPI_Address_init64 is a *collective call* over the LAPI handle *hndl*, which fills the 64-bit table *add_tab* with the virtual address entries that each task supplies. Collective calls must be made in the same order at all participating tasks.

The addresses that are stored in the table *add_tab* are passed in using the *my_addr* parameter. Upon completion of this call, *add_tab[i]* contains the virtual address entry that was provided by task *i*. The array is opaque to the user.

Return values

LAPI_SUCCESS	Indicates that the function call completed successfully.
LAPI_ERR_COLLECTIVE_PSS	Indicates that a collective call was made while in persistent subsystem (PSS) mode.
LAPI_ERR_HNDL_INVALID	Indicates that the <i>hndl</i> passed in is not valid (not initialized or in terminated state).
LAPI_ERR_RET_PTR_NULL	Indicates that the value of the <i>add_tab</i> pointer is NULL (in C) or that the value of <i>add_tab</i> is LAPI_ADDR_NULL (in FORTRAN).

Location

`/usr/lib/liblapi_r.a`

C examples

To collectively transfer target counter addresses for use in a communication API call with a mixed task environment (any combination of 32-bit and 64-bit):

```

lapi_handle_t hndl;           /* the LAPI handle */
lapi_long_t  addr_tbl[NUM_TASKS]; /* the table for all tasks' addresses */
lapi_long_t  tgt_cntr;       /* the target counter */
:
LAPI_Address_init64(hndl, (lapi_long_t)&tgt_cntr, addr_tbl);

/* For communication with task t, use addr_tbl[t] as the address */
/* of the target counter. For mixed (32-bit and 64-bit) jobs, */
/* use the LAPI_Xfer subroutine for communication. */

```

Related information

Subroutines: **LAPI_Address**, **LAPI_Address_init**, **LAPI_Xfer**

LAPI_Amsend

Purpose

Transfers a user message to a remote task, obtaining the target address on the remote task from a user-specified header handler.

Library

Availability Library (**liblapi_r.a**)

C syntax

```
#include <lapi.h>

typedef void (compl_hdlr_t) (hdl, user_info);

lapi_handle_t *hdl;      /* pointer to LAPI context passed in from LAPI_Amsend */
void          *user_info; /* buffer (user_info) pointer passed in                */
                          /* from header handler (void *(hdr_hdlr_t))              */

typedef void *(hdr_hdlr_t)(hdl, uhdr, uhdr_len, msg_len, comp_h, user_info);

lapi_handle_t *hdl;      /* pointer to LAPI context passed in from LAPI_Amsend */
void          *uhdr;     /* uhdr passed in from LAPI_Amsend                    */
uint          *uhdr_len; /* uhdr_len passed in from LAPI_Amsend                */
ulong         *msg_len;  /* udata_len passed in fom LAPI_Amsend                 */
compl_hdlr_t **comp_h;  /* function address of completion handler              */
                          /* (void (compl_hdlr_t)) that needs to be filled       */
                          /* out by this header handler function.                */
void          **user_info; /* pointer to the parameter to be passed              */
                          /* in to the completion handler                        */

int LAPI_Amsend(hdl, tgt, hdr_hdl, uhdr, uhdr_len, udata, udata_len,
                tgt_cntr, org_cntr, cpl_cntr)

lapi_handle_t hdl;
uint          tgt;
void          *hdr_hdl;
void          *uhdr;
uint          uhdr_len;
void          *udata;
ulong         udata_len;
lapi_cntr_t  *tgt_cntr;
lapi_cntr_t  *org_cntr;
lapi_cntr_t  *cpl_cntr;
```

FORTRAN syntax

```
include 'lapif.h'

INTEGER SUBROUTINE COMPL_H (hdl, user_info)
INTEGER hdl
INTEGER user_info

INTEGER FUNCTION HDR_HDL (hdl, uhdr, uhdr_len, msg_len, comp_h, user_info)
INTEGER hdl
INTEGER uhdr
INTEGER uhdr_len
INTEGER (KIND=LAPI_LONG_TYPE) :: msg_len
EXTERNAL INTEGER FUNCTION comp_h
TYPE (LAPI_ADDR_T) :: user_info
```

```

LAPI_AMSEND(hdl, tgt, hdr_hdl, uhdr, uhdr_len, udata, udata_len,
            tgt_cntr, org_cntr, cmpl_cntr, ierror)
INTEGER hdl
INTEGER tgt
EXTERNAL INTEGER FUNCTION hdr_hdl
INTEGER uhdr
INTEGER uhdr_len
TYPE (LAPI_ADDR_T) :: udata
INTEGER (KIND=LAPI_LONG_TYPE) :: udata_len
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_cntr
TYPE (LAPI_CNTR_T) :: org_cntr
TYPE (LAPI_CNTR_T) :: cmpl_cntr
INTEGER ierror

```

Parameters

INPUT

<i>hdl</i>	Specifies the LAPI handle.
<i>tgt</i>	Specifies the task ID of the target task. The value of this parameter must be in the range $0 \leq \textit{tgt} < \text{NUM_TASKS}$.
<i>hdr_hdl</i>	Specifies the pointer to the remote header handler function to be invoked at the target. The value of this parameter can take an address handle that has already been registered using LAPI_Addr_set . The value of this parameter cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
<i>uhdr</i>	Specifies the pointer to the user header data. This data will be passed to the user header handler on the target. If <i>uhdr_len</i> is 0, The value of this parameter can be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
<i>uhdr_len</i>	Specifies the length of the user's header. The value of this parameter must be a multiple of the processor's word size in the range $0 \leq \textit{uhdr_len} \leq \text{MAX_UHDR_SZ}$.
<i>udata</i>	Specifies the pointer to the user data. If <i>udata_len</i> is 0, The value of this parameter can be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
<i>udata_len</i>	Specifies the length of the user data in bytes. The value of this parameter must be in the range $0 \leq \textit{udata_len} \leq$ the value of LAPI constant LAPI_MAX_MSG_SZ .

INPUT/OUTPUT

<i>tgt_cntr</i>	Specifies the target counter address. The target counter is incremented after the completion handler (if specified) completes or after the completion of data transfer. If the value of this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the target counter is not updated.
<i>org_cntr</i>	Specifies the origin counter address (in C) or the origin counter (in FORTRAN). The origin counter is incremented after data is copied out of the origin address (in C) or the origin (in FORTRAN). If the value of this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the origin counter is not updated.
<i>cmpl_cntr</i>	Specifies the counter at the origin that signifies completion of the completion handler. It is updated once the completion handler completes. If no completion handler is specified, the counter is

LAPI_Amsend

incremented at the completion of message delivery. If the value of this parameter is NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN), the completion counter is not updated.

OUTPUT

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: point-to-point communication (non-blocking)

Use this subroutine to transfer data to a target task, where it is desirable to run a handler on the target task before message delivery begins or after delivery completes. **LAPI_Amsend** allows the user to provide a header handler and optional completion handler. The header handler is used to specify the target buffer address for writing the data, eliminating the need to know the address on the origin task when the subroutine is called.

User data (*uhdr* and *udata*) are sent to the target task. Once these buffers are no longer needed on the origin task, the origin counter is incremented, which indicates the availability of origin buffers for modification. Using the **LAPI_Xfer** call with the **LAPI_AM_XFER** type provides the same type of transfer, with the option of using a send completion handler instead of the origin counter to specify buffer availability.

Upon arrival of the first data packet at the target, the user's header handler is invoked. Note that a header handler must be supplied by the user because it returns the base address of the buffer in which LAPI will write the data sent from the origin task (*udata*). See "Receive-side optimization for single-packet messages" on page 79 for more information.

The header handler also provides additional information to LAPI about the message delivery, such as the completion handler. **LAPI_Amsend** and similar calls (such as **LAPI_Amsendv** and corresponding **LAPI_Xfer** transfers) also allow the user to specify their own message header information, which is available to the header handler. The user may also specify a completion handler parameter from within the header handler. LAPI will pass the information to the completion handler at execution.

Note that the header handler is run inline by the thread running the LAPI dispatcher. For this reason, the header handler must be non-blocking because no other progress on messages will be made until it returns. It is also suggested that execution of the header handler be simple and quick. The completion handler, on the other hand, is normally enqueued for execution by a separate thread. It is possible to request that the completion handler be run inline. See "Inline completion handlers" on page 77 for more information.

If a completion handler was not specified (that is, set to **LAPI_ADDR_NULL** in FORTRAN or its pointer set to NULL in C), the arrival of the final packet causes LAPI to increment the target counter on the remote task and send an internal message back to the origin task. The message causes the completion counter (if it is not NULL in C or **LAPI_ADDR_NULL** in FORTRAN) to increment on the origin task.

If a completion handler was specified, the above steps take place after the completion handler returns. To guarantee that the completion handler has executed

on the target, you must wait on the completion counter. See “Flow of active message operations” on page 55 for more information.

User details

As mentioned above, the user must supply the address of a header handler to be executed on the target upon arrival of the first data packet. The signature of the header handler is as follows:

```
void *hdr_hdlr(lapi_handle_t *hdl, void *uhdr, uint *uhdr_len, ulong *msg_len,
              compl_hdlr_t **compl_hdlr, void **user_info);
```

The value returned by the header handler is interpreted by LAPI as an address for writing the user data (*udata*) that was passed to the **LAPI_Amsend** call. The *uhdr* and *uhdr_len* parameters are passed by LAPI into the header handler and contain the information passed by the user to the corresponding parameters of the **LAPI_Amsend** call.

Use of LAPI_Addr_set

Remote addresses are commonly exchanged by issuing a collective **LAPI_Address_init** call within a few steps of initializing LAPI. LAPI also provides the **LAPI_Addr_set** mechanism, whereby users can register one or more header handler addresses in a table, associating an index value with each address. This index can then be passed to **LAPI_Amsend** instead of an actual address. On the target side, LAPI will use the index to get the header handler address. Note that, if all tasks use the same index for their header handler, the initial collective communication can be avoided. Each task simply registers its own header handler address using the well-known index. Then, on any **LAPI_Amsend** calls, the reserved index can be passed to the header handler address parameter.

Role of the header handler

The user optionally returns the address of a completion handler function through the *compl_hdlr* parameter and a completion handler parameter through the *user_info* parameter. The address passed through the *user_info* parameter can refer to memory containing a datatype defined by the user and then cast to the appropriate type from within the completion handler if desired.

The signature for a user completion handler is as follows:

```
typedef void (compl_hdlr_t)(lapi_handle_t *hdl, void *completion_param);
```

The argument returned by reference through the *user_info* member of the user's header handler will be passed to the *completion_param* argument of the user's completion handler. See the **C Examples** for an example of setting the completion handler and parameter in the header handler.

As mentioned above, the value returned by the header handler must be an address for writing the user data sent from the origin task. There is one exception to this rule. In the case of a single-packet message, LAPI passes the address of the packet in the receive FIFO, allowing the entire message to be consumed within the header handler. In this case, the header handler should return NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN) so that LAPI does not copy the message to a target buffer. See “Receive-side optimization for single-packet messages” on page 79 for more information.

Passing additional information through `lapi_return_info_t`

LAPI allows additional information to be passed to and returned from the header handler by passing a pointer to `lapi_return_info_t` through the `msg_len` argument. On return from a header handler that is invoked by a call to **LAPI_Amsend**, the `ret_flags` member of `lapi_return_info_t` can contain one of these values: **LAPI_NORMAL** (the default), **LAPI_SEND_REPLY** (to run the completion handler inline), or **LAPI_LOCAL_STATE** (no reply is sent). The `dgsp_handle` member of `lapi_return_info_t` should not be used in conjunction with **LAPI_Amsend**.

For a complete description of the `lapi_return_info_t` type, see “The enhanced header handler interface” on page 75.

Inline execution of completion handlers

Under normal operation, LAPI uses a separate thread for executing user completion handlers. After the final packet arrives, completion handler pointers are placed in a queue to be handled by this thread. For performance reasons, the user may request that a given completion handler be run inline instead of being placed on this queue behind other completion handlers. This mechanism gives users a greater degree of control in prioritizing completion handler execution for performance-critical messages.

LAPI places no restrictions on completion handlers that are run “normally” (that is, by the completion handler thread). Inline completion handlers should be short and should not block, because no progress can be made while the main thread is executing the handler. The user must use caution with inline completion handlers so that LAPI’s internal queues do not fill up while waiting for the handler to complete. I/O operations must not be performed with an inline completion handler.

Return values

LAPI_SUCCESS	Indicates that the function call completed successfully.
LAPI_ERR_DATA_LEN	Indicates that the value of <code>udata_len</code> is greater than the value of LAPI constant LAPI_MAX_MSG_SZ .
LAPI_ERR_HDR_HNDLR_NULL	Indicates that the value of the <code>hdr_hdl</code> passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_HNDL_INVALID	Indicates that the <code>hdl</code> passed in is not valid (not initialized or in terminated state).
LAPI_ERR_ORG_ADDR_NULL	Indicates that the value of the <code>udata</code> parameter passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but the value of <code>udata_len</code> is greater than 0.
LAPI_ERR_TGT	Indicates that the <code>tgt</code> passed in is outside the range of tasks defined in the job.
LAPI_ERR_TGT_PURGED	Indicates that the subroutine returned early because LAPI_Purge_totask() was called.
LAPI_ERR_UHDR_LEN	Indicates that the <code>uhdr_len</code> value passed in is greater than MAX_UHDR_SZ or is not a multiple of the processor’s doubleword size.

LAPI_ERR_UHDR_NULL Indicates that the *uhdr* passed in is NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN), but *uhdr_len* is not 0.

Location

`/usr/lib/liblapi_r.a`

C examples

To send an active message and then wait on the completion counter:

```

/* header handler routine to execute on target task */
void *hdr_hdlr(lapi_handle_t *hdl, void *uhdr, uint *uhdr_len,
               ulong *msg_len, compl_hdlr_t **cpl_hdlr,
               void **user_info)
{
/* set completion handler pointer and other information */
/* return base address for LAPI to begin its data copy */
}

{
    lapi_handle_t hdl;           /* the LAPI handle */
    int task_id;                 /* the LAPI task ID */
    int num_tasks;               /* the total number of tasks */
    void *hdr_hdlr_list[NUM_TASKS]; /* the table of remote header handlers */
    int buddy;                   /* the communication partner */
    lapi_cntr_t cpl_cntr;        /* the completion counter */
    int data_buffer[DATA_LEN];  /* the data to transfer */

    .
    .
    .
/* retrieve header handler addresses */
LAPI_Address_init(hdl, (void *)&hdr_hdlr, hdr_hdlr_list);

/*
** up to this point, all instructions have executed on all
** tasks. we now begin differentiating tasks.
*/
if ( sender ) {                /* origin task */

    /* initialize data buffer, cpl_cntr, etc. */
    .
    .
    .
/* synchronize before starting data transfer */
LAPI_Gfence(hdl);

    LAPI_Amsend(hdl, buddy, (void *)hdr_hdlr_list[buddy], NULL,
                0,&(data_buffer[0]),DATA_LEN*(sizeof(int)),
                NULL, NULL, cpl_cntr);

/* Wait on completion counter before continuing. Completion */
/* counter will update when message completes at target. */

} else {                        /* receiver */
    .
    .
    .
/* to match the origin's synchronization before data transfer */
LAPI_Gfence(hdl);
}
}

```

LAPI_Amsend

```
    .  
    .  
    .  
}
```

For a complete program listing, see “Using LAPI_Amsend: a complete LAPI program” on page 56. Sample code illustrating the **LAPI_Amsend** call can be found in the LAPI sample files. See Chapter 20, “LAPI sample programs,” on page 245 for more information.

Related information

Subroutines: **LAPI_Addr_get**, **LAPI_Addr_set**, **LAPI_Getcntr**, **LAPI_Msgpoll**, **LAPI_Qenv**, **LAPI_Setcntr**, **LAPI_Waitcntr**, **LAPI_Xfer**

LAPI_Amsendv

Purpose

Transfers a user vector to a remote task, obtaining the target address on the remote task from a user-specified header handler.

Library

Availability Library (**liblapi_r.a**)

C syntax

```
#include <lapi.h>

typedef void (compl_hdlr_t) (hdl, user_info);
lapi_handle_t *hdl; /* the LAPI handle passed in from LAPI_Amsendv */
void *user_info; /* the buffer (user_info) pointer passed in */
/* from vhdr_hdlr (void *(vhdr_hdlr_t)) */

typedef lapi_vec_t *(vhdr_hdlr_t) (hdl, uhdr, uhdr_len, len_vec, comp_h, uinfo);

lapi_handle_t *hdl; /* pointer to the LAPI handle passed in from LAPI_Amsendv */
void *uhdr; /* uhdr passed in from LAPI_Amsendv */
uint uhdr_len; /* uhdr_len passed in from LAPI_Amsendv */
ulong *len_vec[ ]; /* vector of lengths passed in LAPI_Amsendv */
compl_hdlr_t **comp_h; /* function address of completion handler */
/* (void (compl_hdlr_t)) that needs to be */
/* filled out by this header handler function */
void **user_info; /* pointer to the parameter to be passed */
/* in to the completion handler */

int LAPI_Amsendv(hdl, tgt, hdr_hdl, uhdr, uhdr_len, org_vec,
                tgt_cntr, org_cntr, compl_cntr);

lapi_handle_t hdl;
uint tgt;
void *hdr_hdl;
void *uhdr;
uint uhdr_len;
lapi_vec_t *org_vec;
lapi_cntr_t *tgt_cntr;
lapi_cntr_t *org_cntr;
lapi_cntr_t *compl_cntr;
```

FORTRAN syntax

```
include 'lapif.h'

INTEGER SUBROUTINE COMPL_H (hdl, user_info)
INTEGER hdl
INTEGER user_info(*)

INTEGER FUNCTION VHDR_HDL (hdl, uhdr, uhdr_len, len_vec, comp_h, user_info)
INTEGER hdl
INTEGER uhdr
INTEGER uhdr_len
INTEGER (KIND=LAPI_LONG_TYPE) :: len_vec
EXTERNAL INTEGER FUNCTION comp_h
TYPE (LAPI_ADDR_T) :: user_info

LAPI_AMSENDV(hdl, tgt, hdr_hdl, uhdr, uhdr_len, org_vec,
             tgt_cntr, org_cntr, compl_cntr, ierror)
INTEGER hdl
INTEGER tgt
```

LAPI_Amsendv

```
EXTERNAL INTEGER FUNCTION hdr_hdl
INTEGER uhdr
INTEGER uhdr_len
TYPE (LAPI_VEC_T) :: org_vec
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_cntr
TYPE (LAPI_CNTR_T) :: org_cntr
TYPE (LAPI_CNTR_T) :: cmpl_cntr
INTEGER ierror
```

Parameters

<i>hdl</i>	Specifies the LAPI handle.
<i>tgt</i>	Specifies the task ID of the target task. The value of this parameter must be in the range $0 \leq \textit{tgt} < \text{NUM_TASKS}$.
<i>hdr_hdl</i>	Points to the remote header handler function to be invoked at the target. The value of this parameter can take an address handle that had been previously registered using the LAPI_Addr_set/LAPI_Addr_get mechanism. The value of this parameter cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
<i>uhdr</i>	Specifies the pointer to the local header (parameter list) that is passed to the handler function. If <i>uhdr_len</i> is 0, The value of this parameter can be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
<i>uhdr_len</i>	Specifies the length of the user's header. The value of this parameter must be a multiple of the processor's doubleword size in the range $0 \leq \textit{uhdr_len} \leq \text{MAX_UHDR_SZ}$.
<i>org_vec</i>	Points to the origin vector.

INPUT/OUTPUT

<i>tgt_cntr</i>	Specifies the target counter address. The target counter is incremented after the completion handler (if specified) completes or after the completion of data transfer. If the value of this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the target counter is not updated.
<i>org_cntr</i>	Specifies the origin counter address (in C) or the origin counter (in FORTRAN). The origin counter is incremented after data is copied out of the origin address (in C) or the origin (in FORTRAN). If the value of this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the origin counter is not updated.
<i>cmpl_cntr</i>	Specifies the counter at the origin that signifies completion of the completion handler. It is updated once the completion handler completes. If no completion handler is specified, the counter is incremented at the completion of message delivery. If the value of this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the completion counter is not updated.

OUTPUT

<i>ierror</i>	Specifies a FORTRAN return code. This is always the last parameter.
---------------	---

Description

Type of call: point-to-point communication (non-blocking)

LAPI_Amsendv is the vector-based version of the **LAPI_Amsend** call. You can use it to specify multi-dimensional and non-contiguous descriptions of the data to transfer. Whereas regular LAPI calls allow the specification of a single data buffer address and length, the vector versions allow the specification of a vector of address and length combinations. Additional information is allowed in the data description on the origin task and the target task.

Use this subroutine to transfer a vector of data to a target task, when you want a handler to run on the target task before message delivery begins or after message delivery completes.

To use **LAPI_Amsendv**, you must provide a header handler, which returns the address of the target vector description that LAPI uses to write the data that is described by the origin vector. The header handler is used to specify the address of the vector description for writing the data, which eliminates the need to know the description on the origin task when the subroutine is called. The header handler is called upon arrival of the first data packet at the target.

Optionally, you can also provide a completion handler. The header handler provides additional information to LAPI about the message delivery, such as the completion handler. You can also specify a completion handler parameter from within the header handler. LAPI passes the information to the completion handler at execution.

With the exception of the address that is returned by the completion handler, the use of counters, header handlers, and completion handlers in **LAPI_Amsendv** is identical to that of **LAPI_Amsend**. In both cases, the user header handler returns information that LAPI uses for writing at the target. See **LAPI_Amsend** for more information.

This is a non-blocking call. The calling task cannot change the *uhdr* (origin header) and *org_vec* data until completion at the origin is signaled by the *org_cntr* being incremented. The calling task cannot assume that the *org_vec* structure can be changed before the origin counter is incremented. The structure (of type **lapi_vec_t**) that is returned by the header handler cannot be modified before the target counter has been incremented. Also, if a completion handler is specified, it may execute asynchronously, and can only be assumed to have completed after the target counter increments (on the target) or the completion counter increments (at the origin).

The length of the user-specified header (*uhdr_len*) is constrained by the implementation-specified maximum value **MAX_UHDR_SZ**. *uhdr_len* must be a multiple of the processor's doubleword size. To get the best bandwidth, *uhdr_len* should be as small as possible.

If the following requirement is not met, an error condition occurs:

- If a strided vector is being transferred, the size of each block must not be greater than the stride size in bytes.

LAPI does not check for any overlapping regions among vectors either at the origin or the target. If the overlapping regions exist on the target side, the contents of the target buffer are undefined after the operation.

Return values

LAPI_SUCCESS	Indicates that the function call completed successfully.
---------------------	--

LAPI_ERR_HDR_HNDLR_NULL	Indicates that the <i>hdr_hdl</i> passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_HNDL_INVALID	Indicates that the <i>hdl</i> passed in is not valid (not initialized or in terminated state).
LAPI_ERR_ORG_EXTENT	Indicates that the <i>org_vec</i> 's extent (stride * <i>num_vecs</i>) is greater than the value of LAPI constant LAPI_MAX_MSG_SZ .
LAPI_ERR_ORG_STRIDE	Indicates that the <i>org_vec</i> stride is less than block.
LAPI_ERR_ORG_VEC_ADDR	Indicates that the <i>org_vec->info[i]</i> is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but its length (<i>org_vec->len[i]</i>) is not 0.
LAPI_ERR_ORG_VEC_LEN	Indicates that the sum of <i>org_vec->len</i> is greater than the value of LAPI constant LAPI_MAX_MSG_SZ .
LAPI_ERR_ORG_VEC_NULL	Indicates that <i>org_vec</i> is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_ORG_VEC_TYPE	Indicates that the <i>org_vec->vec_type</i> is not valid.
LAPI_ERR_STRIDE_ORG_VEC_ADDR_NULL	Indicates that the strided vector address <i>org_vec->info[0]</i> is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_TGT	Indicates that the <i>tgt</i> passed in is outside the range of tasks defined in the job.
LAPI_ERR_TGT_PURGED	Indicates that the subroutine returned early because LAPI_Purge_totask() was called.
LAPI_ERR_UHDR_LEN	Indicates that the <i>uhdr_len</i> value passed in is greater than MAX_UHDR_SZ or is not a multiple of the processor's doubleword size.
LAPI_ERR_UHDR_NULL	Indicates that the <i>uhdr</i> passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but <i>uhdr_len</i> is not 0.

Location

`/usr/lib/liblapi_r.a`

C examples

1. To send a **LAPI_GEN_IOVECTOR** using active messages:

```

/* header handler routine to execute on target task */
lapi_vec_t *hdr_hdlr(lapi_handle_t *handle, void *uhdr, uint *uhdr_len,
                    ulong *len_vec[ ], compl_hdlr_t **completion_handler,
                    void **user_info)
{
    /* set completion handler pointer and other info */

    /* set up the vector to return to LAPI
    /* for a LAPI_GEN_IOVECTOR: num_vecs, vec_type, and len must all have */
    /* the same values as the origin vector. The info array should */

```

```

/* contain the buffer addresses for LAPI to write the data */
vec->num_vecs = NUM_VECS;
vec->vec_type = LAPI_GEN_IOVECTOR;
vec->len      = (unsigned long *)malloc(NUM_VECS*sizeof(unsigned long));
vec->info     = (void **) malloc(NUM_VECS*sizeof(void *));
for( i=0; i < NUM_VECS; i++ ) {
    vec->info[i] = (void *) &data_buffer[i];
    vec->len[i]  = (unsigned long)(sizeof(int));
}

return vec;
}

{

.
.
.

void      *hdr_hdlr_list[NUM_TASKS]; /* table of remote header handlers */
lapi_vec_t *vec;                    /* data for data transfer */

vec->num_vecs = NUM_VECS;
vec->vec_type = LAPI_GEN_IOVECTOR;
vec->len      = (unsigned long *) malloc(NUM_VECS*sizeof(unsigned long));
vec->info     = (void **) malloc(NUM_VECS*sizeof(void *));

/* each vec->info[i] gets a base address */
/* each vec->len[i] gets the number of bytes to transfer from vec->info[i] */

LAPI_Amsendv(hndl, tgt, (void *) hdr_hdl_list[buddy], NULL, 0, vec,
             tgt_cntr, org_cntr, compl_cntr);

/* data will be copied as follows: */
/* len[0] bytes of data starting from address info[0] */
/* len[1] bytes of data starting from address info[1] */
.
.
.
/* len[NUM_VECS-1] bytes of data starting from address info[NUM_VECS-1] */
}

```

The above example could also illustrate the **LAPI_GEN_GENERIC** type, with the following modifications:

- Both vectors would need **LAPI_GEN_GENERIC** as the `vec_type`.
- There are no restrictions on symmetry of number of vectors and lengths between the origin and target sides.

2. To send a **LAPI_STRIDED_VECTOR** using active messages:

```

/* header handler routine to execute on target task */
lapi_vec_t *hdr_hdlr(lapi_handle_t *handle, void *uhdr, uint *uhdr_len,
                    ulong *len_vec[ ], compl_hdlr_t **completion_handler,
                    void **user_info)
{

    int block_size;          /* block size */
    int data_size;          /* stride */
    .
    .
    .
    vec->num_vecs = NUM_VECS; /* NUM_VECS = number of vectors to transfer */
                             /* must match that of the origin vector */
}

```

LAPI_Amsendv

```
vec->vec_type = LAPI_GEN_STRIDED_XFER;          /* same as origin vector */

/* see comments in origin vector setup for a description of how data
/* will be copied based on these settings.
vec->info[0] = buffer_address; /* starting address for data copy
vec->info[1] = block_size;    /* bytes of data to copy
vec->info[2] = stride;        /* distance from copy block to copy block */
.
.
.
return vec;
}

{
.
.
lapi_vec_t *vec;          /* data for data transfer */

vec->num_vecs = NUM_VECS; /* NUM_VECS = number of vectors to transfer */
/* must match that of the target vector
vec->vec_type = LAPI_GEN_STRIDED_XFER; /* same as target vector

vec->info[0] = buffer_address; /* starting address for data copy
vec->info[1] = block_size;    /* bytes of data to copy
vec->info[2] = stride;        /* distance from copy block to copy block */
/* data will be copied as follows:
/* block_size bytes will be copied from buffer_address
/* block_size bytes will be copied from buffer_address+stride
/* block_size bytes will be copied from buffer_address+(2*stride)
/* block_size bytes will be copied from buffer_address+(3*stride)
.
.
.
/* block_size bytes will be copied from buffer_address+((NUM_VECS-1)*stride)
.
.
.
/* if uhdr isn't used, uhdr should be NULL and uhdr_len should be 0
/* tgt_cntr, org_cntr and cmpl_cntr can all be NULL
LAPI_Amsendv(hndl, tgt, (void *) hdr_hdl_list[buddy], uhdr, uhdr_len,
             vec, tgt_cntr, org_cntr, cmpl_cntr);
.
.
.
}
```

For complete examples, see the sample programs shipped with LAPI.

Related information

“Using vectors” on page 37, for information about vector data transfer

Subroutines: **LAPI_Addr_get**, **LAPI_Addr_set**, **LAPI_Address_init**,
LAPI_Amsend, **LAPI_Getcntr**, **LAPI_Getv**, **LAPI_Putv**, **LAPI_Qenv**,
LAPI_Waitcntr, **LAPI_Xfer**

LAPI_Fence

Purpose

Enforces order on LAPI calls.

Library

Availability Library (**liblapi_r.a**)

C syntax

```
#include <lapi.h>

int LAPI_Fence(hdl)
lapi_handle_t hdl;
```

FORTRAN syntax

```
include 'lapif.h'

LAPI_FENCE(hdl, ierror)
INTEGER hdl
INTEGER ierror
```

Parameters

INPUT

hdl Specifies the LAPI handle.

OUTPUT

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: Local data synchronization (blocking) (may require progress on the remote task)

Use this subroutine to enforce order on LAPI calls. If a task calls **LAPI_Fence**, all the LAPI operations that were initiated by that task, before the fence using the LAPI context *hdl*, are guaranteed to complete at the target tasks. This occurs before any of its communication operations using *hdl*, initiated after the **LAPI_Fence**, start transmission of data. This is a data fence which means that the data movement is complete. This is not an operation fence which would need to include active message completion handlers completing on the target.

LAPI_Fence may require internal protocol processing on the remote side to complete the fence request.

Return values

LAPI_SUCCESS	Indicates that the function call completed successfully.
LAPI_ERR_HNDL_INVALID	Indicates that the <i>hdl</i> passed in is not valid (not initialized or in terminated state).

LAPI_Fence

Location

`/usr/lib/liblapi_r.a`

C examples

To establish a data barrier in a single task:

```
lapi_handle_t hnd1; /* the LAPI handle */
:
/* API communication call 1 */
/* API communication call 2 */
:
/* API communication call n */
LAPI_Fence(hnd1);
/* all data movement from above communication calls has completed by this point */
/* any completion handlers from active message calls could still be running. */
```

Related information

Subroutines: **LAPI_Amsend**, **LAPI_Gfence**

LAPI_Get

Purpose

Copies data from a remote task to a local task.

Library

Availability Library (**liblapi_r.a**)

C syntax

```
#include <lapi.h>

int LAPI_Get(hndl, tgt, len, tgt_addr, org_addr, tgt_cntr, org_cntr)
lapi_handle_t hndl;
uint          tgt;
ulong        len;
void         *tgt_addr;
void         *org_addr;
lapi_cntr_t  *tgt_cntr;
lapi_cntr_t  *org_cntr;
```

FORTRAN syntax

```
include 'lapif.h'

LAPI_GET(hndl, tgt, len, tgt_addr, org_addr, tgt_cntr, org_cntr, ierror)
INTEGER hndl
INTEGER tgt
INTEGER (KIND=LAPI_LONG_TYPE) :: len
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_addr
INTEGER (KIND=LAPI_ADDR_TYPE) :: org_addr
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_cntr
TYPE (LAPI_CNTR_T) :: org_cntr
INTEGER ierror
```

Parameters

INPUT

hndl Specifies the LAPI handle.

tgt Specifies the task ID of the target task that is the source of the data. The value of this parameter must be in the range $0 \leq tgt < \text{NUM_TASKS}$.

len Specifies the number of bytes of data to be copied. This parameter must be in the range $0 \leq len \leq$ the value of LAPI constant **LAPI_MAX_MSG_SZ**.

tgt_addr Specifies the target buffer address of the data source. If *len* is **0**, The value of this parameter can be NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

INPUT/OUTPUT

tgt_cntr Specifies the target counter address. The target counter is incremented once the data buffer on the target can be modified. If the value of this parameter is NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN), the target counter is not updated.

org_cntr Specifies the origin counter address (in C) or the origin counter (in FORTRAN). The origin counter is incremented after data arrives at

LAPI_Get

the origin. If the value of this parameter is NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN), the origin counter is not updated.

OUTPUT

org_addr Specifies the local buffer address into which the received data is copied. If *len* is 0, The value of this parameter can be NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: point-to-point communication (non-blocking)

Use this subroutine to transfer data from a remote (target) task to a local (origin) task. Note that in this case the origin task is actually the *receiver* of the data. This difference in transfer type makes the counter behavior slightly different than in the normal case of origin sending to target.

The origin buffer will still increment on the origin task upon availability of the origin buffer. But in this case, the origin buffer becomes available once the transfer of data is complete. Similarly, the target counter will increment once the target buffer is available. Target buffer availability in this case refers to LAPI no longer needing to access the data in the buffer.

This is a non-blocking call. The caller *cannot* assume that data transfer has completed upon the return of the function. Instead, counters should be used to ensure correct buffer addresses as defined above.

Note that a zero-byte message does not transfer data, but it does have the same semantic with respect to counters as that of any other message.

Return values

LAPI_SUCCESS	Indicates that the function call completed successfully.
LAPI_ERR_DATA_LEN	Indicates that the value of <i>udata_len</i> is greater than the value of LAPI constant LAPI_MAX_MSG_SZ .
LAPI_ERR_HNDL_INVALID	Indicates that the <i>hndl</i> passed in is not valid (not initialized or in terminated state).
LAPI_ERR_ORG_ADDR_NULL	Indicates that the <i>org_addr</i> passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but <i>len</i> is greater than 0.
LAPI_ERR_TGT	Indicates that the <i>tgt</i> passed in is outside the range of tasks defined in the job.
LAPI_ERR_TGT_ADDR_NULL	Indicates that the <i>tgt_addr</i> passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but <i>len</i> is greater than 0.
LAPI_ERR_TGT_PURGED	Indicates that the subroutine returned early because LAPI_Purge_totask() was called.

Location

`/usr/lib/liblapi_r.a`

C examples

```
{
    /* initialize the table buffer for the data addresses */
    /* get remote data buffer addresses */
    LAPI_Address_init(hndl, (void *)data_buffer, data_buffer_list);
    .
    .
    .
    LAPI_Get(hndl, tgt, (ulong) data_len, (void *) (data_buffer_list[tgt]),
            (void *) data_buffer, tgt_cntr, org_cntr);

    /* retrieve data_len bytes from address data_buffer_list[tgt] on task tgt. */
    /* write the data starting at address data_buffer. tgt_cntr and org_cntr */
    /* can be NULL. */
}
```

Related information

Subroutines: **LAPI_Address_init**, **LAPI_Getcntr**, **LAPI_Put**, **LAPI_Qenv**,
LAPI_Waitcntr, **LAPI_Xfer**

LAPI_Getcntr

Purpose

Gets the integer value of a specified LAPI counter.

Library

Availability Library (**liblapi_r.a**)

C syntax

```
#include <lapi.h>

int LAPI_Getcntr(hdl, cntr, val)
lapi_handle_t hdl;
lapi_cntr_t *cntr;
int *val;
```

FORTRAN syntax

```
include 'lapif.h'

LAPI_GETCNTR(hdl, cntr, val, ierror)
INTEGER hdl
TYPE (LAPI_CNTR_T) :: cntr
INTEGER val
INTEGER ierror
```

Parameters

INPUT

hdl Specifies the LAPI handle.

cntr Specifies the address of the counter. The value of this parameter cannot be NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

OUTPUT

val Returns the integer value of the counter *cntr*. The value of this parameter cannot be NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: Local counter manipulation

This subroutine gets the integer value of *cntr*. It is used to check progress on *hdl*.

Return values

LAPI_SUCCESS	Indicates that the function call completed successfully.
LAPI_ERR_CNTR_NULL	Indicates that the <i>cntr</i> pointer is NULL (in C) or that the value of <i>cntr</i> is LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_HNDL_INVALID	Indicates that the <i>hndl</i> passed in is not valid (not initialized or in terminated state).
LAPI_ERR_RET_PTR_NULL	Indicates that the value of the <i>val</i> pointer is NULL (in C) or that the value of <i>val</i> is LAPI_ADDR_NULL (in FORTRAN).

Location

`/usr/lib/liblapi_r.a`

C examples

```

{
    lapi_cntr_t cntr;
    int         val;

    /* cntr is initialized */

    /* processing/communication takes place */

    LAPI_Getcncr(hndl, &cntr, &val)

    /* val now contains the current value of cntr */
}

```

Related information

Subroutines: **LAPI_Amsend**, **LAPI_Amsendv**, **LAPI_Get**, **LAPI_Getv**, **LAPI_Put**, **LAPI_Putv**, **LAPI_Rmw**, **LAPI_Setcncr**, **LAPI_Waitcncr**, **LAPI_Xfer**

LAPI_Getv

Purpose

Copies vectors of data from a remote task to a local task.

Library

Availability Library (**liblapi_r.a**)

C syntax

```
#include <lapi.h>

int LAPI_Getv(hndl, tgt, tgt_vec, org_vec, tgt_cntr, org_cntr)
lapi_handle_t hndl;
uint          tgt;
lapi_vec_t    *tgt_vec;
lapi_vec_t    *org_vec;
lapi_cntr_t   *tgt_cntr;
lapi_cntr_t   *org_cntr;

typedef struct {
    lapi_vectype_t  vec_type; /* operation code      */
    uint            num_vecs; /* number of vectors */
    void            **info;   /* vector of information */
    ulong           *len;     /* vector of lengths   */
} lapi_vec_t;
```

FORTTRAN syntax

```
include 'lapif.h'

LAPI_GETV(hndl, tgt, tgt_vec, org_vec, tgt_cntr, org_cntr, ierror)
INTEGER hndl
INTEGER tgt
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_vec
TYPE (LAPI_VEC_T) :: org_vec
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_cntr
TYPE (LAPI_CNTR_T) :: org_cntr
INTEGER ierror
```

The 32-bit version of the **LAPI_VEC_T** type is defined as:

```
TYPE LAPI_VEC_T
SEQUENCE
    INTEGER(KIND = 4) :: vec_type
    INTEGER(KIND = 4) :: num_vecs
    INTEGER(KIND = 4) :: info
    INTEGER(KIND = 4) :: len
END TYPE LAPI_VEC_T
```

The 64-bit version of the **LAPI_VEC_T** type is defined as:

```
TYPE LAPI_VEC_T
SEQUENCE
    INTEGER(KIND = 4) :: vec_type
    INTEGER(KIND = 4) :: num_vecs
    INTEGER(KIND = 8) :: info
    INTEGER(KIND = 8) :: len
END TYPE LAPI_VEC_T
```

Parameters

INPUT

<i>hndl</i>	Specifies the LAPI handle.
<i>tgt</i>	Specifies the task ID of the target task. The value of this parameter must be in the range $0 \leq tgt < \text{NUM_TASKS}$.
<i>tgt_vec</i>	Points to the target vector description.
<i>org_vec</i>	Points to the origin vector description.

INPUT/OUTPUT

<i>tgt_cntr</i>	Specifies the target counter address. The target counter is incremented once the data buffer on the target can be modified. If the value of this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the target counter is not updated.
<i>org_cntr</i>	Specifies the origin counter address (in C) or the origin counter (in FORTRAN). The origin counter is incremented after data arrives at the origin. If the value of this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the origin counter is not updated.

OUTPUT

<i>ierror</i>	Specifies a FORTRAN return code. This is always the last parameter.
---------------	---

Description

Type of call: point-to-point communication (non-blocking)

This subroutine is the vector version of the **LAPI_Get** call. Use **LAPI_Getv** to transfer vectors of data from the target task to the origin task. Both the origin and target vector descriptions are located in the address space of the origin task. But, the values specified in the *info* array of the target vector must be addresses in the address space of the target task.

The calling program *cannot* assume that the origin buffer can be changed or that the contents of the origin buffers on the origin task are ready for use upon function return. After the origin counter (*org_cntr*) is incremented, the origin buffers can be modified by the origin task. After the target counter (*tgt_cntr*) is incremented, the target buffers can be modified by the target task. If you provide a completion counter (*cmpl_cntr*), it is incremented at the origin after the target counter (*tgt_cntr*) has been incremented at the target. If the values of any of the counters or counter addresses are NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN), the data transfer occurs, but the corresponding counter increments do not occur.

If any of the following requirements are not met, an error condition occurs:

- The vector types *org_vec*->*vec_type* and *tgt_vec*->*vec_type* must be the same.
- If a strided vector is being transferred, the size of each block must not be greater than the stride size in bytes.
- The length of any vector that is pointed to by *tgt_vec* must be equal to the length of the corresponding vector that is pointed to by *org_vec*.

LAPI_Getv

LAPI does not check for any overlapping regions among vectors either at the origin or the target. If the overlapping regions exist on the origin side, the contents of the origin buffer are undefined after the operation.

See **LAPI_Amsendv** for details about communication using different LAPI vector types. (**LAPI_Getv** does not support the **LAPI_GEN_GENERIC** type.)

Return values

LAPI_SUCCESS	Indicates that the function call completed successfully.
LAPI_ERR_HNDL_INVALID	Indicates that the <i>hndl</i> passed in is not valid (not initialized or in terminated state).
LAPI_ERR_ORG_EXTENT	Indicates that the <i>org_vec</i> 's extent (stride * <i>num_vecs</i>) is greater than the value of LAPI constant LAPI_MAX_MSG_SZ .
LAPI_ERR_ORG_STRIDE	Indicates that the <i>org_vec</i> stride is less than block size.
LAPI_ERR_ORG_VEC_ADDR	Indicates that some <i>org_vec->info[i]</i> is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but the corresponding length (<i>org_vec->len[i]</i>) is not 0 .
LAPI_ERR_ORG_VEC_LEN	Indicates that the total sum of all <i>org_vec->len[i]</i> (where [i] is in the range $0 \leq i \leq \text{org_vec} \rightarrow \text{num_vecs}$) is greater than the value of LAPI constant LAPI_MAX_MSG_SZ .
LAPI_ERR_ORG_VEC_NULL	Indicates that the <i>org_vec</i> is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_ORG_VEC_TYPE	Indicates that the <i>org_vec->vec_type</i> is not valid.
LAPI_ERR_STRIDE_ORG_VEC_ADDR_NULL	Indicates that the strided vector base address <i>org_vec->info[0]</i> is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_STRIDE_TGT_VEC_ADDR_NULL	Indicates that the strided vector address <i>tgt_vec->info[0]</i> is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_TGT	Indicates that the <i>tgt</i> passed in is outside the range of tasks defined in the job.
LAPI_ERR_TGT_EXTENT	Indicates that <i>tgt_vec</i> 's extent (stride * <i>num_vecs</i>) is greater than the value of LAPI constant LAPI_MAX_MSG_SZ .
LAPI_ERR_TGT_PURGED	Indicates that the subroutine returned early because LAPI_Purge_totask() was called.
LAPI_ERR_TGT_STRIDE	Indicates that the <i>tgt_vec</i> 's stride is less than its block size.
LAPI_ERR_TGT_VEC_ADDR	Indicates that the <i>tgt_vec->info[i]</i> is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but its length (<i>tgt_vec->len[i]</i>) is not 0 .

LAPI_ERR_TGT_VEC_LEN	Indicates that the sum of <i>tgt_vec->len</i> is greater than the value of LAPI constant LAPI_MAX_MSG_SZ .
LAPI_ERR_TGT_VEC_NULL	Indicates that <i>tgt_vec</i> is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_TGT_VEC_TYPE	Indicates that the <i>tgt_vec->vec_type</i> is not valid.
LAPI_ERR_VEC_LEN_DIFF	Indicates that <i>org_vec</i> and <i>tgt_vec</i> have different lengths (<i>len[]</i>).
LAPI_ERR_VEC_NUM_DIFF	Indicates that <i>org_vec</i> and <i>tgt_vec</i> have different <i>num_vecs</i> .
LAPI_ERR_VEC_TYPE_DIFF	Indicates that <i>org_vec</i> and <i>tgt_vec</i> have different vector types (<i>vec_type</i>).

Location

`/usr/lib/liblapi_r.a`

C examples

To get a LAPI_GEN_IOVECTOR:

```
{
    /* retrieve a remote data buffer address for data to transfer, */
    /* such as through LAPI_Address_init */
    /* task that calls LAPI_Getv sets up both org_vec and tgt_vec */
    org_vec->num_vecs = NUM_VECS;
    org_vec->vec_type = LAPI_GEN_IOVECTOR;
    org_vec->len = (unsigned long *)
    malloc(NUM_VECS*sizeof(unsigned long));
    org_vec->info = (void **) malloc(NUM_VECS*sizeof(void *));

    /* each org_vec->info[i] gets a base address on the origin task */
    /* each org_vec->len[i] gets the number of bytes to write to */
    /* org_vec->info[i] */

    tgt_vec->num_vecs = NUM_VECS;
    tgt_vec->vec_type = LAPI_GEN_IOVECTOR;
    tgt_vec->len = (unsigned long *)
    malloc(NUM_VECS*sizeof(unsigned long));
    tgt_vec->info = (void **) malloc(NUM_VECS*sizeof(void *));

    /* each tgt_vec->info[i] gets a base address on the target task */
    /* each tgt_vec->len[i] gets the number of bytes to transfer */
    /* from vec->info[i] */
    /* For LAPI_GEN_IOVECTOR, num_vecs, vec_type, and len must be */
    /* the same */

    LAPI_Getv(hndl, tgt, tgt_vec, org_vec, tgt_cntr, org_cntr);
    /* tgt_cntr and org_cntr can both be NULL */

    /* data will be retrieved as follows: */
    /* org_vec->len[0] bytes will be retrieved from */
    /* tgt_vec->info[0] and written to org_vec->info[0] */
    /* org_vec->len[1] bytes will be retrieved from */
    /* tgt_vec->info[1] and written to org_vec->info[1] */
    :
}
```

LAPI_Getv

```
        *  
        /* org_vec->len[NUM_VECS-1] bytes will be retrieved */  
        /* from tgt_vec->info[NUM_VECS-1] and written to    */  
        /* org_vec->info[NUM_VECS-1]                        */  
    }
```

For examples of other vector types, see **LAPI_Amsendv**.

Related information

Subroutines: **LAPI_Amsendv**, **LAPI_Getcntr**, **LAPI_Putv**, **LAPI_Qenv**,
LAPI_Waitcntr

LAPI_Gfence

Purpose

Enforces order on LAPI calls across all tasks and provides barrier synchronization among them.

Library

Availability Library (**liblapi_r.a**)

C syntax

```
#include <lapi.h>

int LAPI_Gfence(hndl)
lapi_handle_t hndl;
```

FORTRAN syntax

```
include 'lapif.h'

LAPI_GFENCE(hndl, ierror)
INTEGER hndl
INTEGER ierror
```

Parameters

INPUT

hndl Specifies the LAPI handle.

OUTPUT

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: collective data synchronization (blocking)

Use this subroutine to enforce global order on LAPI calls. This is a *collective call*. Collective calls must be made in the same order at all participating tasks.

On completion of this call, it is assumed that all LAPI communication associated with *hndl* from all tasks has quiesced. Although *hndl* is local, it represents a set of tasks that were associated with it at **LAPI_Init**, all of which must participate in this operation for it to complete. This is a data fence, which means that the data movement is complete. This is not an operation fence, which would need to include active message completion handlers completing on the target.

Return values

LAPI_SUCCESS	Indicates that the function call completed successfully.
LAPI_ERR_HNDL_INVALID	Indicates that the <i>hndl</i> passed in is not valid (not initialized or in terminated state).

LAPI_Gfence

Location

`/usr/lib/liblapi_r.a`

Related information

Subroutines: **LAPI_Fence**

LAPI_Init

Purpose

Initializes a LAPI context.

Library

Availability Library (**liblapi_r.a**)

C syntax

```
#include <lapi.h>

int LAPI_Init(hndl,lapi_info)
lapi_handle_t *hndl;
lapi_info_t *lapi_info;
```

FORTRAN syntax

```
include 'lapif.h'

LAPI_INIT(hndl,lapi_info,ierror)
INTEGER hndl
TYPE (LAPI_INFO_T) :: lapi_info
INTEGER ierror
```

Parameters

INPUT/OUTPUT

lapi_info Specifies a structure that provides the parallel job information with which this LAPI context is associated. The value of this parameter cannot be NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

OUTPUT

hndl Specifies a pointer to the LAPI handle to initialize.

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: Local initialization

Use this subroutine to instantiate and initialize a new LAPI context. A handle to the newly-created LAPI context is returned in *hndl*. All subsequent LAPI calls can use *hndl* to specify the context of the LAPI operation. Except for **LAPI_Address()** and **LAPI_Msg_string()**, the user cannot make any LAPI calls before calling **LAPI_Init()**.

The *lapi_info* structure (**lapi_info_t**) must be "zeroed out" before any fields are filled in. To do this in C, use this statement: **bzero (lapi_info, size of (lapi_info_t))**. In FORTRAN, you need to "zero out" each field manually in the **LAPI_INFO_T** type. Fields with a description of Future support should not be used because the names of those fields might change.

LAPI_Init

The **lapi_info_t** structure is defined as follows:

```
typedef struct {
    lapi_dev_t    protocol;        /* Protocol device returned          */
    lapi_lib_t    lib_vers;        /* LAPI library version -- user-supplied */
    uint          epoch_num;       /* No longer used                    */
    int           num_compl_hdlr_thr; /* Number of completion handler threads */
    uint          instance_no;     /* Instance of LAPI to initialize [1-16] */
    int           info6;           /* Future support                    */
    LAPI_err_hdlr *err_hdlr;       /* User-registered error handler      */
    com_thread_info_t *lapi_thread_attr; /* Support thread att and init function */
    void          *adapter_name;   /* What adapter to initialize, i.e. css0, m10 */
    lapi_extend_t *add_info;       /* Additional structure extension     */
} lapi_info_t;
```

The fields are used as follows:

protocol LAPI sets this field to the protocol that has been initialized.

lib_vers Is used to indicate a library version to LAPI for compatibility purposes. Valid values for this field are:

- L1_LIB** Provides basic functionality (this is the default).
- L2_LIB** Provides the ability to use counters as structures.
- LAST_LIB** Provides the most current level of functionality. For new users of LAPI, *lib_vers* should be set to **LAST_LIB**.

This field must be set to **L2_LIB** or **LAST_LIB** to use **LAPI_Nopoll_wait** and **LAPI_Setcntr_wstatus**.

epoch_num This field is no longer used.

num_compl_hdlr_thr Indicates to LAPI the number of completion handler threads to initialize.

instance_no Specifies the instance of LAPI to initialize (**1** to **16**).

info6 This field is for future use.

err_hdlr Use this field to optionally pass a callback pointer to an error-handler routine.

lapi_thread_attr Supports thread attributes and initialization function.

adapter_name Is used in persistent subsystem (PSS) mode to pass an adapter name.

add_info Is used for additional information in standalone UDP mode.

Return values

LAPI_SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_ALL_HNDL_IN_USE All available LAPI instances are in use.

LAPI_ERR_BOTH_NETSTR_SET Both the **MP_LAPI_NETWORK** and **MP_LAPI_INET** statements are set (only one should be set).

LAPI_ERR_CSS_LOAD_FAILED	LAPI is unable to load the communication utility library.
LAPI_ERR_HNDL_INVALID	The lapi_handle_t * passed to LAPI for initialization is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_INFO_NONZERO_INFO	The future support fields in the lapi_info_t structure that was passed to LAPI are not set to zero (and should be).
LAPI_ERR_INFO_NULL	The lapi_info_t pointer passed to LAPI is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_MEMORY_EXHAUSTED	LAPI is unable to obtain memory from the system.
LAPI_ERR_MSG_API	Indicates that the MP_MSG_API environment variable is not set correctly.
LAPI_ERR_NO_NETSTR_SET	No network statement is set. Note that if running with POE, this will be returned if MP_MSG_API is not set correctly.
LAPI_ERR_NO_UDP_HNDLR	You passed a value of NULL (in C) or LAPI_ADDR_NULL (in FORTRAN) for both the UDP handler and the UDP list. One of these (the UDP handler or the UDP list) must be initialized for standalone UDP initialization. This error is returned in standalone UDP mode only.
LAPI_ERR_PSS_NON_ROOT	You tried to initialize the persistent subsystem (PSS) protocol as a non- root user.
LAPI_ERR_SHM_KE_NOT_LOADED	LAPI's shared memory kernel extension is not loaded.
LAPI_ERR_SHM_SETUP	LAPI is unable to set up shared memory. This error will be returned if LAPI_USE_SHM=only and tasks are assigned to more than one node.
LAPI_ERR_UDP_PKT_SZ	The UDP packet size you indicated is not valid.
LAPI_ERR_UNKNOWN	An internal error has occurred.
LAPI_ERR_USER_UDP_HNDLR_FAIL	The UDP handler you passed has returned a non-zero error code. This error is returned in standalone UDP mode only.

Location

`/usr/lib/liblapi_r.a`

C examples

The following environment variable must be set before LAPI is initialized:

```
MP_MSG_API=[ lapi | [ lapi,mpi | mpi,lapi ] | mpi_lapi ]
```

LAPI_Init

The following environment variables are also commonly used:

```
MP_EUILIB=[ ip | us ] (ip is the default)
```

```
MP_PROCS=number_of_tasks_in_job
```

```
LAPI_USE_SHM=[ yes | no | only ] (no is the default)
```

To initialize LAPI, follow these steps:

1. Set environment variables (as described in “Setting environment variables” on page 29) before the user application is invoked. The remaining steps are done in the user application.
2. Clear **lapi_info_t**, then set any fields.
3. Call **LAPI_Init**.

For systems running PE

Both US and UDP/IP are supported for shared handles as long as they are the same for both handles. Mixed transport protocols such as LAPI IP and shared user space (US) are not supported.

To initialize a LAPI handle:

```
{
    lapi_handle_t hndl;
    lapi_info_t info;

    bzero(&info, sizeof(lapi_info_t)); /* clear lapi_info */

    LAPI_Init(&hndl, &info);
}
```

To initialize a LAPI handle and register an error handler:

```
void my_err_hdlr(lapi_handle_t *hndl, int *error_code, lapi_err_t *err_type,
                int *task_id, int *src )
{
    /* examine passed parameters and delete desired information */

    if ( user wants to terminate ) {
        LAPI_Term(*hndl);          /* will terminate LAPI */
        exit(some_return_code);
    }

    /* any additional processing */

    return; /* signals to LAPI that error is non-fatal; execution should continue */
}

{
    lapi_handle_t hndl;
    lapi_info_t info;

    bzero(&info, sizeof(lapi_info_t)); /* clear lapi_info */

    /* set error handler pointer */
    info.err_hdlr = (LAPI_err_hdlr) my_err_hdlr;

    LAPI_Init(&hndl, &info);
}
```

For standalone systems (not running PE)

To initialize a LAPI handle for UDP/IP communication using a user handler:

```
int my_udp_hdlr(lapi_handle_t *hdl, lapi_udp_t *local_addr, lapi_udp_t *addr_list,
               lapi_udpinfo_t *info)
{
    /* LAPI will allocate and free addr_list pointer when using */
    /* a user handler */

    /* use the AIX inet_addr call to convert an IP address */
    /* from a dotted quad to a long */
    task_0_ip_as_long = inet_addr(task_0_ip_as_string);
    addr_list[0].ip_addr = task_0_ip_as_long;
    addr_list[0].port_no = task_0_port_as_unsigned;

    task_1_ip_as_long = inet_addr(task_1_ip_as_string);
    addr_list[1].ip_addr = task_1_ip_as_long;
    addr_list[1].port_no = task_1_port_as_unsigned;
    .
    .
    .
    task_num_tasks-1_ip_as_long = inet_addr(task_num_tasks-1_ip_as_string);
    addr_list[num_tasks-1].ip_addr = task_num_tasks-1_ip_as_long;
    addr_list[num_tasks-1].port_no = task_num_tasks-1_port_as_unsigned;
}

{
    lapi_handle_t hdl;
    lapi_info_t info;
    lapi_extend_t extend_info;

    bzero(&info, sizeof(lapi_info_t)); /* clear lapi_info */
    bzero(&extend_info, sizeof(lapi_extend_t)); /* clear lapi_extend_info */

    extend_info.udp_hdlr = (udp_init_hdlr *) my_udp_hdlr;
    info.add_info = &extend_info;

    LAPI_Init(&hdl, &info);
}
```

To initialize a LAPI handle for UDP/IP communication using a user list:

```
{
    lapi_handle_t hdl;
    lapi_info_t info;
    lapi_extend_t extend_info;
    lapi_udp_t *addr_list;

    bzero(&info, sizeof(lapi_info_t)); /* clear lapi_info */
    bzero(&extend_info, sizeof(lapi_extend_t)); /* clear lapi_extend_info */

    /* when using a user list, the user is responsible for allocating */
    /* and freeing the list pointer */
    addr_list = malloc(num_tasks);

    /* Note, since we need to know the number of tasks before LAPI is */
    /* initialized, we can't use LAPI_Qenv. getenv("MP_PROCS") will */
    /* do the trick. */

    /* populate addr_list */
}
```

LAPI_Init

```
/* use the AIX inet_addr call to convert an IP address */
/* from a dotted quad to a long */
task_0_ip_as_long = inet_addr(task_0_ip_as_string);
addr_list[0].ip_addr = task_0_ip_as_long;
addr_list[0].port_no = task_0_port_as_unsigned;

task_1_ip_as_long = inet_addr(task_1_ip_as_string);
addr_list[1].ip_addr = task_1_ip_as_long;
addr_list[1].port_no = task_1_port_as_unsigned;
.
.
.
task_num_tasks-1_ip_as_long = inet_addr(task_num_tasks-1_ip_as_string);
addr_list[num_tasks-1].ip_addr = task_num_tasks-1_ip_as_long;
addr_list[num_tasks-1].port_no = task_num_tasks-1_port_as_unsigned;

/* then assign to extend pointer */
extend_info.add_udp_addrs = addr_list;

info.add_info = &extend_info;

LAPI_Init(&hdl, &info);
.
.
.

/* user's responsibility only in the case of user list */
free(addr_list);

}
```

See the LAPI sample programs for complete examples of initialization in standalone mode.

To initialize a LAPI handle for user space (US) communication in standalone mode:

```
export MP_MSG_API=lapi
export MP_EUILIB=us
export MP_PROCS=                /* number of tasks in job */
export MP_PARTITION=           /* unique job key */
export MP_CHILD=                /* unique task ID */
export MP_LAPI_NETWORK=@1:164,sn0 /* LAPI network information */
```

run LAPI jobs as normal

See the **README.LAPI.STANDALONE.US** file in the **standalone/us** directory of the LAPI sample files for complete details.

Related information

- “Initializing LAPI” on page 31
- “Standalone initialization” on page 117
- Chapter 13, “Bulk transfer of messages,” on page 99

Subroutines: **LAPI_Msg_string**, **LAPI_Term**

LAPI_Msg_string

Purpose

Retrieves the message that is associated with a subroutine return code.

Library

Availability Library (**liblapi_r.a**)

C syntax

```
#include <lapi.h>

LAPI_Msg_string(error_code, buf)
int error_code;
void *buf;
```

FORTRAN syntax

```
include 'lapif.h'

LAPI_MSG_STRING(error_code, buf, ierror)
INTEGER error_code
CHARACTER buf(LAPI_MAX_ERR_STRING)
INTEGER ierror
```

Parameters

INPUT

error_code Specifies the return value of a previous LAPI call.

OUTPUT

buf Specifies the buffer to store the message string.

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: local queries

Use this subroutine to retrieve the message string that is associated with a LAPI return code. LAPI tries to find the messages of any return codes that come from the AIX operating system or its communication subsystem.

Return values

LAPI_SUCCESS	Indicates that the function call completed successfully.
LAPI_ERR_CATALOG_FAIL	Indicates that the message catalog cannot be opened. An English-only string is copied into the user's message buffer (<i>buf</i>).
LAPI_ERR_CODE_UNKNOWN	Indicates that <i>error_code</i> is outside of the range known to LAPI.

LAPI_Msg_string

LAPI_ERR_RET_PTR_NULL Indicates that the value of the *buf* pointer is NULL (in C) or that the value of *buf* is **LAPI_ADDR_NULL** (in FORTRAN).

Location

`/usr/lib/liblapi_r.a`

C examples

To get the message string associated with a LAPI return code:

```
{  
  
    char msg_buf[LAPI_MAX_ERR_STRING]; /* constant defined in lapi.h */  
    int rc, errc;  
  
    rc = some_LAPI_call();  
  
    errc = LAPI_Msg_string(rc, msg_buf);  
  
    /* msg_buf now contains the message string for the return code */  
  
}
```

LAPI_Msgpoll

Purpose

Allows the calling thread to check communication progress.

Library

Availability Library (**liblapi_r.a**)

C syntax

```
#include <lapi.h>

int LAPI_Msgpoll(hndl, cnt, info)
lapi_handle_t    hndl;
uint            cnt;
lapi_msg_info_t *info;

typedef struct {
    lapi_msg_state_t status;    /* Message status returned from LAPI_Msgpoll */
    ulong            reserve[10]; /* Reserved */
} lapi_msg_info_t;
```

FORTRAN syntax

```
include 'lapif.h'

LAPI_MSGPOLL(hndl, cnt, info, ierror)
INTEGER hndl
INTEGER cnt
TYPE (LAPI_MSG_STATE_T) :: info
INTEGER ierror
```

Parameters

INPUT

hndl Specifies the LAPI handle.

cnt Specifies the maximum number of times the dispatcher should loop with no progress before returning.

info Specifies a status structure that contains the result of the **LAPI_Msgpoll()** call. LAPI will set flags depending on the status of any send or receive completions.

OUTPUT

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: local progress monitor (blocking)

The **LAPI_Msgpoll** subroutine allows the calling thread to check communication progress. With this subroutine, LAPI provides a means of running the dispatcher several times until either progress is made or a specified maximum number of dispatcher loops have executed. Here, *progress* is defined as the completion of either a message send operation or a message receive operation.

LAPI_Msgpoll

LAPI_Msgpoll is intended to be used when interrupts are turned off. If the user has not explicitly turned interrupts off, LAPI temporarily disables interrupt mode while in this subroutine because the dispatcher is called, which will process any pending receive operations. If the LAPI dispatcher loops for the specified maximum number of times, the call returns. If progress is made before the maximum count, the call will return immediately. In either case, LAPI will report status through a data structure that is passed by reference.

The **lapi_msg_info_t** structure contains a flags field (*status*), which is of type **lapi_msg_state_t**. Flags in the *status* field are set as follows:

LAPI_DISP_CNTR	If the dispatcher has looped <i>cnt</i> times without making progress
LAPI_SEND_COMPLETE	If a message send operation has completed
LAPI_RECV_COMPLETE	If a message receive operation has completed
LAPI_BOTH_COMPLETE	If both a message send operation and a message receive operation have completed
LAPI_POLLING_NET	If another thread is already polling the network or shared memory completion

Return values

LAPI_SUCCESS	Indicates that the function call completed successfully.
LAPI_ERR_HNDL_INVALID	Indicates that the <i>hndl</i> passed in is not valid (not initialized or in terminated state).
LAPI_ERR_MSG_INFO_NULL	Indicates that the <i>info</i> pointer is NULL (in C) or that the value of <i>info</i> is LAPI_ADDR_NULL (in FORTRAN).

Location

`/usr/lib/liblapi_r.a`

C examples

To loop through the dispatcher no more than 1000 times, then check what progress has been made:

```
{
    lapi_msg_info_t msg_info;
    int cnt = 1000;
    .
    .
    .
    LAPI_Msgpoll(hndl, cnt, &msg_info);

    if ( msg_info.status & LAPI_BOTH_COMPLETE ) {
        /* both a message receive and a message send have been completed */
    } else if ( msg_info.status & LAPI_RECV_COMPLETE ) {
        /* just a message receive has been completed */
    } else if ( msg_info.status & LAPI_SEND_COMPLETE ) {
        /* just a message send has been completed */
    } else {
        /* cnt loops and no progress */
    }
}
```

```
}
```

```
}
```

Related information

Subroutines: [LAPI_Getcntr](#), [LAPI_Probe](#), [LAPI_Setcntr](#), [LAPI_Waitcntr](#)

LAPI_Probe

Purpose

Transfers control to the communication subsystem to check for arriving messages and to make progress in polling mode.

Library

Availability Library (**liblapi_r.a**)

C syntax

```
#include <lapi.h>

int LAPI_Probe(hdl)
lapi_handle_t hdl;
```

FORTRAN syntax

```
include 'lapif.h'

int LAPI_PROBE(hdl, ierror)
INTEGER hdl
INTEGER ierror
```

Parameters

INPUT

hdl Specifies the LAPI handle.

OUTPUT

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: local progress monitor (non-blocking)

This subroutine transfers control to the communication subsystem in order to make progress on messages associated with the context *hdl*. A **LAPI_Probe** operation lasts for one round of the communication dispatcher.

Note: There is no guarantee about receipt of messages on the return from this function.

Return values

LAPI_SUCCESS	Indicates that the function call completed successfully.
LAPI_ERR_HNDL_INVALID	Indicates that the <i>hdl</i> passed in is not valid (not initialized or in terminated state).

Location

/usr/lib/liblapi_r.a

Related information

Subroutines: **LAPI_Getcptr**, **LAPI_Msgpoll**, **LAPI_Nopoll_wait**, **LAPI_Waitcptr**

LAPI_Put

Purpose

Transfers data from a local task to a remote task.

Library

Availability Library (**liblapi_r.a**)

C syntax

```
#include <lapi.h>

int LAPI_Put(hndl, tgt, len, tgt_addr, org_addr, tgt_cntr, org_cntr, cmpl_cntr)
lapi_handle_t hndl;
uint      tgt;
ulong     len;
void      *tgt_addr;
void      *org_addr;
lapi_cntr_t *tgt_cntr;
lapi_cntr_t *org_cntr;
lapi_cntr_t *cmpl_cntr;
```

FORTTRAN syntax

```
include 'lapif.h'

int LAPI_PUT(hndl, tgt, len, tgt_addr, org_addr, tgt_cntr, org_cntr, ierror)
INTEGER hndl
INTEGER tgt
INTEGER (KIND=LAPI_LONG_TYPE) :: len
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_addr
INTEGER org_addr
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_cntr
TYPE (LAPI_CNTR_T) :: org_cntr
TYPE (LAPI_CNTR_T) :: cmpl_cntr
INTEGER ierror
```

Parameters

INPUT

<i>hndl</i>	Specifies the LAPI handle.
<i>tgt</i>	Specifies the task ID of the target task. The value of this parameter must be in the range 0 <= <i>tgt</i> < NUM_TASKS .
<i>len</i>	Specifies the number of bytes to be transferred. This parameter must be in the range 0 <= <i>len</i> <= the value of LAPI constant LAPI_MAX_MSG_SZ .
<i>tgt_addr</i>	Specifies the address on the target task where data is to be copied into. If <i>len</i> is 0 , The value of this parameter can be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
<i>org_addr</i>	Specifies the address on the origin task from which data is to be copied. If <i>len</i> is 0 , The value of this parameter can be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

INPUT/OUTPUT

<i>tgt_cntr</i>	Specifies the target counter address. The target counter is
-----------------	---

incremented upon message completion. If this parameter is NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN), the target counter is not updated.

<i>org_cntr</i>	Specifies the origin counter address (in C) or the origin counter (in FORTRAN). The origin counter is incremented at buffer availability. If this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the origin counter is not updated.
<i>cmpl_cntr</i>	Specifies the completion counter address (in C) or the completion counter (in FORTRAN) that is a reflection of <i>tgt_cntr</i> . The completion counter is incremented at the origin after <i>tgt_cntr</i> is incremented. If this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the completion counter is not updated.

OUTPUT

<i>ierror</i>	Specifies a FORTRAN return code. This is always the last parameter.
---------------	---

Description

Type of call: point-to-point communication (non-blocking)

Use this subroutine to transfer data from a local (origin) task to a remote (target) task. The origin counter will increment on the origin task upon origin buffer availability. The target counter will increment on the target and the completion counter will increment at the origin task upon message completion. Because there is no completion handler, message completion and target buffer availability are the same in this case.

This is a non-blocking call. The caller *cannot* assume that the data transfer has completed upon the return of the function. Instead, counters should be used to ensure correct buffer accesses as defined above.

Note that a zero-byte message does not transfer data, but it does have the same semantic with respect to counters as that of any other message.

Return values

LAPI_SUCCESS	Indicates that the function call completed successfully.
LAPI_ERR_DATA_LEN	Indicates that the value of <i>len</i> is greater than the value of LAPI constant LAPI_MAX_MSG_SZ .
LAPI_ERR_HNDL_INVALID	Indicates that the <i>hndl</i> passed in is not valid (not initialized or in terminated state).
LAPI_ERR_ORG_ADDR_NULL	Indicates that the <i>org_addr</i> parameter passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but <i>len</i> is greater than 0.
LAPI_ERR_TGT	Indicates that the <i>tgt</i> passed in is outside the range of tasks defined in the job.

LAPI_Put

LAPI_ERR_TGT_ADDR_NULL

Indicates that the *tgt_addr* parameter passed in is NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN), but *len* is greater than 0.

LAPI_ERR_TGT_PURGED

Indicates that the subroutine returned early because **LAPI_Purge_totask()** was called.

Location

`/usr/lib/liblapi_r.a`

C examples

```
{  
  
    /* initialize the table buffer for the data addresses          */  
  
    /* get remote data buffer addresses                          */  
    LAPI_Address_init(hndl, (void *)data_buffer, data_buffer_list);  
    .  
    .  
    .  
    LAPI_Put(hndl, tgt, (ulong) data_len, (void *) (data_buffer_list[tgt]),  
            (void *) data_buffer, tgt_cntr, org_cntr, compl_cntr);  
  
    /* transfer data_len bytes from local address data_buffer.   */  
    /* write the data starting at address data_buffer_list[tgt] on */  
    /* task tgt. tgt_cntr, org_cntr, and compl_cntr can be NULL. */  
  
}
```

Related information

Subroutines: **LAPI_Get**, **LAPI_Getcptr**, **LAPI_Qenv**, **LAPI_Setcptr**,
LAPI_Waitcptr, **LAPI_Xfer**

LAPI_Putv

Purpose

Transfers vectors of data from a local task to a remote task.

Library

Availability Library (**liblapi_r.a**)

C syntax

```
#include <lapi.h>

int LAPI_Putv(hndl, tgt, tgt_vec, org_vec, tgt_cntr, org_cntr, cmpl_cntr)

lapi_handle_t hndl;
uint          tgt;
lapi_vec_t    *tgt_vec;
lapi_vec_t    *org_vec;
lapi_cntr_t   *tgt_cntr;
lapi_cntr_t   *org_cntr;
lapi_cntr_t   *cmpl_cntr;

typedef struct {
    lapi_vectype_t vec_type; /* operation code */
    uint          num_vecs; /* number of vectors */
    void          **info;   /* vector of information */
    ulong         *len;     /* vector of lengths */
} lapi_vec_t;
```

FORTRAN syntax

```
include 'lapif.h'

LAPI_PUTV(hndl, tgt, tgt_vec, org_vec, tgt_cntr, org_cntr, cmpl_cntr, ierror)
INTEGER hndl
INTEGER tgt
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_vec
TYPE (LAPI_VEC_T) :: org_vec
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_cntr
TYPE (LAPI_CNTR_T) :: org_cntr
TYPE (LAPI_CNTR_T) :: cmpl_cntr
INTEGER ierror
```

The 32-bit version of the **LAPI_VEC_T** type is defined as:

```
TYPE LAPI_VEC_T
  SEQUENCE
    INTEGER(KIND = 4) :: vec_type
    INTEGER(KIND = 4) :: num_vecs
    INTEGER(KIND = 4) :: info
    INTEGER(KIND = 4) :: len
END TYPE LAPI_VEC_T
```

The 64-bit version of the **LAPI_VEC_T** type is defined as:

```
TYPE LAPI_VEC_T
  SEQUENCE
    INTEGER(KIND = 4) :: vec_type
    INTEGER(KIND = 4) :: num_vecs
    INTEGER(KIND = 8) :: info
    INTEGER(KIND = 8) :: len
END TYPE LAPI_VEC_T
```

Parameters

INPUT

<i>hndl</i>	Specifies the LAPI handle.
<i>tgt</i>	Specifies the task ID of the target task. The value of this parameter must be in the range $0 \leq tgt < \text{NUM_TASKS}$.
<i>tgt_vec</i>	Points to the target vector description.
<i>org_vec</i>	Points to the origin vector description.

INPUT/OUTPUT

<i>tgt_cntr</i>	Specifies the target counter address. The target counter is incremented upon message completion. If this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the target counter is not updated.
<i>org_cntr</i>	Specifies the origin counter address (in C) or the origin counter (in FORTRAN). The origin counter is incremented at buffer availability. If this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the origin counter is not updated.
<i>cmpl_cntr</i>	Specifies the completion counter address (in C) or the completion counter (in FORTRAN) that is a reflection of <i>tgt_cntr</i> . The completion counter is incremented at the origin after <i>tgt_cntr</i> is incremented. If this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the completion counter is not updated.

OUTPUT

<i>ierror</i>	Specifies a FORTRAN return code. This is always the last parameter.
---------------	---

Description

Type of call: point-to-point communication (non-blocking)

LAPI_Putv is the vector version of the **LAPI_Put** call. Use this subroutine to transfer vectors of data from the origin task to the target task. The origin vector descriptions and the target vector descriptions are located in the address space of the *origin* task. However, the values specified in the *info* array of the target vector must be addresses in the address space of the *target* task.

The calling program *cannot* assume that the origin buffer can be changed or that the contents of the target buffers on the target task are ready for use upon function return. After the origin counter (*org_cntr*) is incremented, the origin buffers can be modified by the origin task. After the target counter (*tgt_cntr*) is incremented, the target buffers can be modified by the target task. If you provide a completion counter (*cmpl_cntr*), it is incremented at the origin after the target counter (*tgt_cntr*) has been incremented at the target. If the values of any of the counters or counter addresses are NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN), the data transfer occurs, but the corresponding counter increments do not occur.

If a strided vector is being transferred, the size of each block must not be greater than the stride size in bytes.

The length of any vector pointed to by *org_vec* must be equal to the length of the corresponding vector pointed to by *tgt_vec*.

LAPI does not check for any overlapping regions among vectors either at the origin or the target. If the overlapping regions exist on the target side, the contents of the target buffer are undefined after the operation.

See **LAPI_Amsendv** for more information about using the various vector types. (**LAPI_Putv** does not support the **LAPI_GEN_GENERIC** type.)

Return values

LAPI_SUCCESS	Indicates that the function call completed successfully.
LAPI_ERR_HNDL_INVALID	Indicates that the <i>hndl</i> passed in is not valid (not initialized or in terminated state).
LAPI_ERR_ORG_EXTENT	Indicates that the <i>org_vec</i> 's extent (stride * <i>num_vecs</i>) is greater than the value of LAPI constant LAPI_MAX_MSG_SZ .
LAPI_ERR_ORG_STRIDE	Indicates that the <i>org_vec</i> stride is less than block.
LAPI_ERR_ORG_VEC_ADDR	Indicates that the <i>org_vec->info[i]</i> is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but its length (<i>org_vec->len[i]</i>) is not 0.
LAPI_ERR_ORG_VEC_LEN	Indicates that the sum of <i>org_vec->len</i> is greater than the value of LAPI constant LAPI_MAX_MSG_SZ .
LAPI_ERR_ORG_VEC_NULL	Indicates that the <i>org_vec</i> is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_ORG_VEC_TYPE	Indicates that the <i>org_vec->vec_type</i> is not valid.
LAPI_ERR_STRIDE_ORG_VEC_ADDR_NULL	Indicates that the strided vector address <i>org_vec->info[0]</i> is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_STRIDE_TGT_VEC_ADDR_NULL	Indicates that the strided vector address <i>tgt_vec->info[0]</i> is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_TGT	Indicates that the <i>tgt</i> passed in is outside the range of tasks defined in the job.
LAPI_ERR_TGT_EXTENT	Indicates that <i>tgt_vec</i> 's extent (stride * <i>num_vecs</i>) is greater than the value of LAPI constant LAPI_MAX_MSG_SZ .
LAPI_ERR_TGT_PURGED	Indicates that the subroutine returned early because LAPI_Purge_totask() was called.
LAPI_ERR_TGT_STRIDE	Indicates that the <i>tgt_vec</i> stride is less than block.
LAPI_ERR_TGT_VEC_ADDR	Indicates that the <i>tgt_vec->info[i]</i> is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but its length (<i>tgt_vec->len[i]</i>) is not 0.
LAPI_ERR_TGT_VEC_LEN	Indicates that the sum of <i>tgt_vec->len</i> is greater than the value of LAPI constant LAPI_MAX_MSG_SZ .

LAPI_Putv

LAPI_ERR_TGT_VEC_NULL	Indicates that <i>tgt_vec</i> is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_TGT_VEC_TYPE	Indicates that the <i>tgt_vec->vec_type</i> is not valid.
LAPI_ERR_VEC_LEN_DIFF	Indicates that <i>org_vec</i> and <i>tgt_vec</i> have different lengths (<i>len[]</i>).
LAPI_ERR_VEC_NUM_DIFF	Indicates that <i>org_vec</i> and <i>tgt_vec</i> have different <i>num_vecs</i> .
LAPI_ERR_VEC_TYPE_DIFF	Indicates that <i>org_vec</i> and <i>tgt_vec</i> have different vector types (<i>vec_type</i>).

Location

`/usr/lib/liblapi_r.a`

C examples

To put a LAPI_GEN_IOVECTOR:

```
{
    /* retrieve a remote data buffer address for data to transfer, */
    /* such as through LAPI_Address_init */

    /* task that calls LAPI_Putv sets up both org_vec and tgt_vec */
    org_vec->num_vecs = NUM_VECS;
    org_vec->vec_type = LAPI_GEN_IOVECTOR;
    org_vec->len      = (unsigned long *)
    malloc(NUM_VECS*sizeof(unsigned long));
    org_vec->info     = (void **) malloc(NUM_VECS*sizeof(void *));

    /* each org_vec->info[i] gets a base address on the origin task */
    /* each org_vec->len[i] gets the number of bytes to transfer */
    /* from org_vec->info[i] */

    tgt_vec->num_vecs = NUM_VECS;
    tgt_vec->vec_type = LAPI_GEN_IOVECTOR;
    tgt_vec->len      = (unsigned long *)
    malloc(NUM_VECS*sizeof(unsigned long));
    tgt_vec->info     = (void **) malloc(NUM_VECS*sizeof(void *));

    /* each tgt_vec->info[i] gets a base address on the target task */
    /* each tgt_vec->len[i] gets the number of bytes to write to vec->info[i] */
    /* For LAPI_GEN_IOVECTOR, num_vecs, vec_type, and len must be the same */

    LAPI_Putv(hndl, tgt, tgt_vec, org_vec, tgt_cntr, org_cntr, compl_cntr);
    /* tgt_cntr, org_cntr and compl_cntr can all be NULL */

    /* data will be transferred as follows: */
    /* org_vec->len[0] bytes will be retrieved from */
    /* org_vec->info[0] and written to tgt_vec->info[0] */
    /* org_vec->len[1] bytes will be retrieved from */
    /* org_vec->info[1] and written to tgt_vec->info[1] */
    .
    .
    .
    /* org_vec->len[NUM_VECS-1] bytes will be retrieved */
    /* from org_vec->info[NUM_VECS-1] and written to */
    /* tgt_vec->info[NUM_VECS-1] */
}
}
```

See the example in **LAPI_Amsendv** for information on other vector types.

Related information

Subroutines: **LAPI_Amsendv**, **LAPI_Getcntr**, **LAPI_Getv**, **LAPI_Qenv**,
LAPI_Setcntr, **LAPI_Waitcntr**, **LAPI_Xfer**

LAPI_Qenv

Purpose

Used to query LAPI for runtime task information.

Library

Availability Library (**liblapi_r.a**)

C syntax

```
#include <lapif.h>

int LAPI_Qenv(hndl, query, ret_val)
lapi_handle_t hndl;
lapi_query_t query;
int *ret_val; /* ret_val's type varies (see Additional query types) */
```

FORTRAN syntax

```
include 'lapif.h'

LAPI_QENV(hndl, query, ret_val, ierror)
INTEGER hndl
INTEGER query
INTEGER ret_val /* ret_val's type varies (see Additional query types) */
INTEGER ierror
```

Parameters

INPUT

hndl Specifies the LAPI handle.

query Specifies the type of query you want to request. In C, the values for *query* are defined by the **lapi_query_t** enumeration in **lapi.h**. In FORTRAN, these values are defined explicitly in the 32-bit version and the 64-bit version of **lapif.h**.

OUTPUT

ret_val Specifies the reference parameter for LAPI to store as the result of the query. The value of this parameter cannot be NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: local queries

Use this subroutine to query runtime settings and statistics from LAPI. LAPI defines a set of query types as an enumeration in **lapi.h** for C and explicitly in the 32-bit and 64-bit versions of **lapif.h** for FORTRAN.

For example, you can query the size of the table that LAPI uses for the **LAPI_Addr_set** subroutine using a *query* value of **LOC_ADDRTBL_SZ**:

```
LAPI_Qenv(hndl, LOC_ADDRTBL_SZ, &ret_val);
```

ret_val will contain the upper bound on the table index. A subsequent call to **LAPI_Addr_set** (*hndl, addr, addr_hndl*); could then ensure that the value of *addr_hndl* is between 0 and *ret_val*.

When used to show the size of a parameter, a comparison of values, or a range of values, valid values for the *query* parameter of the **LAPI_Genv** subroutine appear in **SMALL, BOLD** capital letters. For example:

NUM_TASKS

is a shorthand notation for:

LAPI_Genv(*hndl, NUM_TASKS, ret_val*)

In C, **lapi_query_t** defines the valid types of LAPI queries:

```
typedef enum {
    TASK_ID=0,      /* Query the task ID of the current task in the job      */
    NUM_TASKS,     /* Query the number of tasks in the job                  */
    MAX_UHDR_SZ,   /* Query the maximum user header size for active messaging */
    MAX_DATA_SZ,   /* Query the maximum data length that can be sent        */
    ERROR_CHK,     /* Query and set parameter checking on (1) or off (0)    */
    TIMEOUT,       /* Query and set the current communication timeout setting */
                  /* in seconds                                             */
    MIN_TIMEOUT,  /* Query the minimum communication timeout setting in seconds */
    MAX_TIMEOUT,  /* Query the maximum communication timeout setting in seconds */
    INTERRUPT_SET, /* Query and set interrupt mode on (1) or off (0)        */
    MAX_PORTS,    /* Query the maximum number of available communication ports */
    MAX_PKT_SZ,   /* This is the payload size of 1 packet                  */
    NUM_REX_BUFS, /* Number of retransmission buffers                     */
    REX_BUF_SZ,   /* Size of each retransmission buffer in bytes           */
    LOC_ADDRTBL_SZ, /* Size of address store table used by LAPI_Addr_set     */
    EPOCH_NUM,    /* No longer used by LAPI (supports legacy code)        */
    USE_THRESH,   /* No longer used by LAPI (supports legacy code)        */
    RCV_FIFO_SIZE, /* No longer used by LAPI (supports legacy code)        */
    MAX_ATOM_SIZE, /* Query the maximum atom size for a DGSP accumulate transfer */
    BUF_CP_SIZE, /* Query the size of the message buffer to save (default 128b) */
    MAX_PKTS_OUT, /* Query the maximum number of messages outstanding /    */
                  /* destination                                           */
    ACK_THRESHOLD, /* Query and set the threshold of acknowledgments going  */
                  /* back to the source                                    */
    QUERY_SHM_ENABLED, /* Query to see if shared memory is enabled             */
    QUERY_SHM_NUM_TASKS, /* Query to get the number of tasks that use shared    */
                  /* memory                                                */
    QUERY_SHM_TASKS, /* Query to get the list of task IDs that make up shared */
                  /* memory; pass in an array of size QUERY_SHM_NUM_TASKS */
    QUERY_STATISTICS, /* Query to get packet statistics from LAPI, as         */
                  /* defined by the lapi_statistics_t structure. For     */
                  /* this query, pass in 'lapi_statistics_t *' rather   */
                  /* than 'int *ret_val'; otherwise, the data will     */
                  /* overflow the buffer.                               */
    PRINT_STATISTICS, /* Query debug print function to print out statistics  */
    QUERY_SHM_STATISTICS, /* Similar query as QUERY_STATISTICS for shared      */
                  /* memory path.                                       */
    QUERY_LOCAL_SEND_STATISTICS, /* Similar query as QUERY_STATISTICS */
                  /* for local copy path.                             */
    BULK_XFER, /* Query to see if bulk transfer is enabled (1) or disabled (0) */
    BULK_MIN_MSG_SIZE, /* Query the current bulk transfer minimum message size */
    LAST_QUERY
}
```

LAPI_Qenv

```
} lapi_query_t;

typedef struct {
    lapi_long_t Tot_dup_pkt_cnt;      /* Total duplicate packet count */
    lapi_long_t Tot_retrans_pkt_cnt; /* Total retransmit packet count */
    lapi_long_t Tot_gho_pkt_cnt;     /* Total Ghost packet count */
    lapi_long_t Tot_pkt_sent_cnt;    /* Total packet sent count */
    lapi_long_t Tot_pkt_rcv_cnt;     /* Total packet receive count */
    lapi_long_t Tot_data_sent;       /* Total data sent */
    lapi_long_t Tot_data_rcv;        /* Total data receive */
} lapi_statistics_t;
```

In FORTRAN, the valid types of LAPI queries are defined in **lapif.h** as follows:

```
integer TASK_ID,NUM_TASKS,MAX_UHDR_SZ,MAX_DATA_SZ,ERROR_CHK
integer TIMEOUT,MIN_TIMEOUT,MAX_TIMEOUT
integer INTERRUPT_SET,MAX_PORTS,MAX_PKT_SZ,NUM_REX_BUFS
integer REX_BUF_SZ,LOC_ADDRTBL_SZ,EPOCH_NUM,USE_THRESH
integer RCV_FIFO_SIZE,MAX_ATOM_SIZE,BUF_CP_SIZE
integer MAX_PKTS_OUT,ACK_THRESHOLD,QUERY_SHM_ENABLED
integer QUERY_SHM_NUM_TASKS,QUERY_SHM_TASKS
integer QUERY_STATISTICS,PRINT_STATISTICS
integer QUERY_SHM_STATISTICS,QUERY_LOCAL_SEND_STATISTICS
integer BULK_XFER,BULK_MIN_MSG_SIZE,
integer LAST_QUERY
parameter (TASK_ID=0,NUM_TASKS=1,MAX_UHDR_SZ=2,MAX_DATA_SZ=3)
parameter (ERROR_CHK=4,TIMEOUT=5,MIN_TIMEOUT=6)
parameter (MAX_TIMEOUT=7,INTERRUPT_SET=8,MAX_PORTS=9)
parameter (MAX_PKT_SZ=10,NUM_REX_BUFS=11,REX_BUF_SZ=12)
parameter (LOC_ADDRTBL_SZ=13,EPOCH_NUM=14,USE_THRESH=15)
parameter (RCV_FIFO_SIZE=16,MAX_ATOM_SIZE=17,BUF_CP_SIZE=18)
parameter (MAX_PKTS_OUT=19,ACK_THRESHOLD=20)
parameter (QUERY_SHM_ENABLED=21,QUERY_SHM_NUM_TASKS=22)
parameter (QUERY_SHM_TASKS=23,QUERY_STATISTICS=24)
parameter (PRINT_STATISTICS=25)
parameter (QUERY_SHM_STATISTICS=26,QUERY_LOCAL_SEND_STATISTICS=27)
parameter (BULK_XFER=28,BULK_MIN_MSG_SIZE=29)
parameter (LAST_QUERY=30)
```

Additional query types

LAPI provides additional query types for which the behavior of **LAPI_Qenv** is slightly different:

PRINT_STATISTICS

When passed this query type, LAPI sends data transfer statistics to standard output. In this case, *ret_val* is unaffected. However, LAPI's error checking requires that the value of *ret_val* is not NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN) for all **LAPI_Qenv** types (including **PRINT_STATISTICS**).

QUERY_LOCAL_SEND_STATISTICS

When passed this query type, **LAPI_Qenv** interprets *ret_val* as a pointer to type **lapi_statistics_t**. Upon function return, the fields of the structure contain LAPI's data transfer statistics for data transferred through intra-task local copy. The packet count will be 0.

QUERY_SHM_STATISTICS

When passed this query type, **LAPI_Qenv** interprets *ret_val* as a pointer to type **lapi_statistics_t**. Upon function return, the fields of

|
|

the structure contain LAPI's data transfer statistics for data transferred through shared memory.

QUERY_SHM_TASKS

When passed this query type, **LAPI_Qenv** returns a list of task IDs with which this task can communicate using shared memory. *ret_val* must be an **int *** with enough space to hold **NUM_TASKS** integers. For each task *i*, if it is possible to use shared memory, *ret_val[i]* will contain the shared memory task ID. If it is not possible to use shared memory, *ret_val[i]* will contain **-1**.

QUERY_STATISTICS

When passed this query type, **LAPI_Qenv** interprets *ret_val* as a pointer to type **lapi_statistics_t**. Upon function return, the fields of the structure contain LAPI's data transfer statistics for data transferred using the user space (US) protocol or UDP/IP.

|

Return values**LAPI_SUCCESS**

Indicates that the function call completed successfully.

LAPI_ERR_HNDL_INVALID

Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

LAPI_ERR_QUERY_TYPE

Indicates that the query passed in is not valid.

LAPI_ERR_RET_PTR_NULL

Indicates that the value of the *ret_val* pointer is NULL (in C) or that the value of *ret_val* is **LAPI_ADDR_NULL** (in FORTRAN).

Location

/usr/lib/liblapi_r.a

C examples

To query runtime values from LAPI:

```
{
    int          task_id;
    lapi_statistics_t stats;
    .
    .
    .
    LAPI_Qenv(hndl, TASK_ID, &task_id);
    /* task_id now contains the task ID */
    .
    .
    .
    LAPI_Qenv(hndl, QUERY_STATISTICS, (int *)&stats);
    /* the fields of the stats structure are now
       filled in with runtime values */
    .
    .
    .
}
```

Related information

Subroutines: **LAPI_Amsend**, **LAPI_Get**, **LAPI_Put**, **LAPI_Senv**, **LAPI_Xfer**

LAPI_Rmw

Purpose

Provides data synchronization primitives.

Library

Availability Library (**liblapi_r.a**)

C syntax

```
#include <lapi.h>

int LAPI_Rmw(hndl, op, tgt, tgt_var, in_val, prev_tgt_val, org_cntr)

lapi_handle_t hndl;
RMW_ops_t op;
uint tgt;
int *tgt_var;
int *in_val;
int *prev_tgt_val;
lapi_cntr_t *org_cntr;
```

FORTRAN syntax

```
include 'lapif.h'

LAPI_RMW(hndl, op, tgt, tgt_var, in_val, prev_tgt_val, org_cntr, ierror)
INTEGER hndl
INTEGER op
INTEGER tgt
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_var
INTEGER in_val
INTEGER prev_tgt_val
TYPE (LAPI_CNTR_T) :: org_cntr
INTEGER ierror
```

Parameters

INPUT

<i>hndl</i>	Specifies the LAPI handle.
<i>op</i>	Specifies the operation to be performed. The valid operations are: <ul style="list-style-type: none"> • COMPARE_AND_SWAP • FETCH_AND_ADD • FETCH_AND_OR • SWAP
<i>tgt</i>	Specifies the task ID of the target task where the read-modify-write (Rmw) variable resides. The value of this parameter must be in the range $0 \leq tgt < \text{NUM_TASKS}$.
<i>tgt_var</i>	Specifies the target read-modify-write (Rmw) variable (in FORTRAN) or its address (in C). The value of this parameter cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
<i>in_val</i>	Specifies the value that is passed in to the operation (<i>op</i>). This value cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

INPUT/OUTPUT

prev_tgt_val Specifies the location at the origin in which the previous *tgt_var* on the target task is stored before the operation (*op*) is executed. The value of this parameter can be NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

org_cntr Specifies the origin counter address (in C) or the origin counter (in FORTRAN). If *prev_tgt_val* is set, the origin counter (*org_cntr*) is incremented when *prev_tgt_val* is returned to the origin side. If *prev_tgt_val* is not set, the origin counter (*org_cntr*) is updated after the operation (*op*) is completed at the target side.

OUTPUT

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: point-to-point communication (non-blocking)

Use this subroutine to synchronize two independent pieces of data, such as two tasks sharing a common data structure. The operation is performed at the target task (*tgt*) and is atomic. The operation takes an input value (*in_val*) from the origin and performs one of four operations (*op*) on a variable (*tgt_var*) at the target (*tgt*), and then replaces the target variable (*tgt_var*) with the results of the operation (*op*). The original value (*prev_tgt_val*) of the target variable (*tgt_var*) is returned to the origin.

The operations (*op*) are performed over the context referred to by *hndl*. The outcome of the execution of these calls is as if the following code was executed atomically:

```
*prev_tgt_val = *tgt_var;
*tgt_var      = f(*tgt_var, *in_val);
```

where:

$f(a,b) = a + b$ for **FETCH_AND_ADD**

$f(a,b) = a | b$ for **FETCH_AND_OR** (bitwise or)

$f(a,b) = b$ for **SWAP**

For **COMPARE_AND_SWAP**, *in_val* is treated as a pointer to an array of two integers, and the *op* is the following atomic operation:

```
if(*tgt_var == in_val[0]) {
    *prev_tgt_val = TRUE;
    *tgt_var      = in_val[1];
} else {
    *prev_tgt_val = FALSE;
}
```

All **LAPI_Rmw** calls are non-blocking. To test for completion, use the **LAPI_Getcntr** and **LAPI_Waitcntr** subroutines. **LAPI_Rmw** does not include a target counter (*tgt_cntr*), so **LAPI_Rmw** calls do not provide any indication of completion on the target task (*tgt*).

Return values

LAPI_SUCCESS	Indicates that the function call completed successfully.
LAPI_ERR_HNDL_INVALID	Indicates that the <i>hndl</i> passed in is not valid (not initialized or in terminated state).
LAPI_ERR_IN_VAL_NULL	Indicates that the <i>in_val</i> pointer is NULL (in C) or that the value of <i>in_val</i> is LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_RMW_OP	Indicates that <i>op</i> is not valid.
LAPI_ERR_TGT	Indicates that the <i>tgt</i> passed in is outside the range of tasks defined in the job.
LAPI_ERR_TGT_PURGED	Indicates that the subroutine returned early because LAPI_Purge_totask() was called.
LAPI_ERR_TGT_VAR_NULL	Indicates that the tgt_var address is NULL (in C) or that the value of tgt_var is LAPI_ADDR_NULL (in FORTRAN).

Location

`/usr/lib/liblapi_r.a`

C examples

1. To synchronize a data value between two tasks (with **FETCH_AND_ADD**):

```
{
    int local_var;
    int *addr_list;

    /* both tasks initialize local_var to a value */

    /* local_var addresses are exchanged and stored */
    /* in addr_list (using LAPI_Address_init). */
    /* addr_list[tgt] now contains the address of */
    /* local_var on tgt */
    .
    .
    /* add value to local_var on some task */

    /* use LAPI to add value to local_var on remote task */
    LAPI_Rmw(hndl, FETCH_AND_ADD, tgt, addr_list[tgt],
             value, prev_tgt_val, &org_cntr);

    /* local_var on the remote task has been increased */
    /* by value. prev_tgt_val now contains the value */
    /* of local_var on remote task before the addition */
}
```

2. To synchronize a data value between two tasks (with **SWAP**):

```
{
    int local_var;
    int *addr_list;

    /* local_var addresses are exchanged and stored */
```

```

/* in addr_list (using LAPI_Address_init). */
/* addr_list[tgt] now contains the address of */
/* local_var on tgt. */
.
.
.
/* local_var is assigned some value */

/* assign local_var to local_var on remote task */
LAPI_Rmw(hndl, SWAP, tgt, addr_list[tgt],
         local_var, prev_tgt_val, &org_cntr);

/* local_var on the remote task is now equal to */
/* local_var on the local task. prev_tgt_val now */
/* contains the value of local_var on the remote */
/* task before the swap. */
}

```

3. To conditionally swap a data value (with **COMPARE_AND_SWAP**):

```

{

    int local_var;
    int *addr_list;
    int in_val[2];

    /* local_var addresses are exchanged and stored */
    /* in addr_list (using LAPI_Address_init). */
    /* addr_list[tgt] now contains the address of */
    /* local_var on tgt. */
    .
    .
    .
    /* if local_var on remote_task is equal to comparator, */
    /* assign value to local_var on remote task */

    in_val[0] = comparator;
    in_val[1] = value;

    LAPI_Rmw(hndl, COMPARE_AND_SWAP, tgt, addr_list[tgt],
            in_val, prev_tgt_val, &org_cntr);

    /* local_var on the remote task is now in_val[1] if it */
    /* had previously been equal to in_val[0]. If the swap */
    /* was performed, prev_tgt_val now contains TRUE; */
    /* otherwise, it contains FALSE. */
}

```

Related information

Subroutines: **LAPI_Address_init**, **LAPI_Getcntr**, **LAPI_Genv**, **LAPI_Rmw64**,
LAPI_Setcntr, **LAPI_Waitcntr**, **LAPI_Xfer**

LAPI_Rmw64

Purpose

Provides data synchronization primitives for 64-bit applications.

Library

Availability Library (**liblapi_r.a**)

C syntax

```
#include <lapi.h>

int LAPI_Rmw64(hndl, op, tgt, tgt_var, in_val, prev_tgt_val, org_cntr)

lapi_handle_t hndl;
Rmw_ops_t op;
uint tgt;
long long *tgt_var;
long long *in_val;
long long *prev_tgt_val;
lapi_cntr_t *org_cntr;
```

FORTRAN syntax

```
include 'lapif.h'

LAPI_RMW64(hndl, op, tgt, tgt_var, in_val, prev_tgt_val, org_cntr, ierror)

INTEGER hndl
INTEGER op
INTEGER tgt
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_var
INTEGER (KIND=LAPI_LONG_LONG_TYPE) :: in_val, prev_tgt_val
TYPE (LAPI_CNTR_T) :: org_cntr
INTEGER ierror
```

Parameters

INPUT

<i>hndl</i>	Specifies the LAPI handle.
<i>op</i>	Specifies the operation to be performed. The valid operations are: <ul style="list-style-type: none"> • COMPARE_AND_SWAP • FETCH_AND_ADD • FETCH_AND_OR • SWAP
<i>tgt</i>	Specifies the task ID of the target task where the read-modify-write (Rmw64) variable resides. The value of this parameter must be in the range $0 \leq tgt < \text{NUM_TASKS}$.
<i>tgt_var</i>	Specifies the target read-modify-write (Rmw64) variable (in FORTRAN) or its address (in C). The value of this parameter cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
<i>in_val</i>	Specifies the value that is passed in to the operation (<i>op</i>). This value cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

INPUT/OUTPUT

prev_tgt_val Specifies the location at the origin in which the previous *tgt_var* on the target task is stored before the operation (*op*) is executed. The value of this parameter can be NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

org_cntr Specifies the origin counter address (in C) or the origin counter (in FORTRAN). If *prev_tgt_val* is set, the origin counter (*org_cntr*) is incremented when *prev_tgt_val* is returned to the origin side. If *prev_tgt_val* is not set, the origin counter (*org_cntr*) is updated after the operation (*op*) is completed at the target side.

OUTPUT

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: point-to-point communication (non-blocking)

This subroutine is the 64-bit version of **LAPI_Rmw**. It is used to synchronize two independent pieces of 64-bit data, such as two tasks sharing a common data structure. The operation is performed at the target task (*tgt*) and is atomic. The operation takes an input value (*in_val*) from the origin and performs one of four operations (*op*) on a variable (*tgt_var*) at the target (*tgt*), and then replaces the target variable (*tgt_var*) with the results of the operation (*op*). The original value (*prev_tgt_val*) of the target variable (*tgt_var*) is returned to the origin.

The operations (*op*) are performed over the context referred to by *hdl*. The outcome of the execution of these calls is as if the following code was executed atomically:

```
*prev_tgt_val = *tgt_var;
*tgt_var      = f(*tgt_var, *in_val);
```

where:

$f(a,b) = a + b$ for **FETCH_AND_ADD**

$f(a,b) = a | b$ for **FETCH_AND_OR** (bitwise or)

$f(a,b) = b$ for **SWAP**

For **COMPARE_AND_SWAP**, *in_val* is treated as a pointer to an array of two integers, and the *op* is the following atomic operation:

```
if(*tgt_var == in_val[0]) {
    *prev_tgt_val = TRUE;
    *tgt_var      = in_val[1];
} else {
    *prev_tgt_val = FALSE;
}
```

This subroutine can also be used on a 32-bit processor.

All **LAPI_Rmw64** calls are non-blocking. To test for completion, use the **LAPI_Getcntr** and **LAPI_Waitcntr** subroutines. **LAPI_Rmw64** does not include a target counter (*tgt_cntr*), so **LAPI_Rmw64** calls do not provide any indication of completion on the target task (*tgt*).

Return values

LAPI_SUCCESS	Indicates that the function call completed successfully.
LAPI_ERR_HNDL_INVALID	Indicates that the <i>hndl</i> passed in is not valid (not initialized or in terminated state).
LAPI_ERR_IN_VAL_NULL	Indicates that the <i>in_val</i> pointer is NULL (in C) or that the value of <i>in_val</i> is LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_RMW_OP	Indicates that <i>op</i> is not valid.
LAPI_ERR_TGT	Indicates that the <i>tgt</i> passed in is outside the range of tasks defined in the job.
LAPI_ERR_TGT_PURGED	Indicates that the subroutine returned early because LAPI_Purge_totask() was called.
LAPI_ERR_TGT_VAR_NULL	Indicates that the tgt_var address is NULL (in C) or that the value of tgt_var is LAPI_ADDR_NULL (in FORTRAN).

Location

`/usr/lib/liblapi_r.a`

C examples

1. To synchronize a data value between two tasks (with **FETCH_AND_ADD**):

```
{
    long long local_var;
    long long *addr_list;

    /* both tasks initialize local_var to a value */
    /* local_var addresses are exchanged and stored */
    /* in addr_list (using LAPI_Address_init64) */
    /* addr_list[tgt] now contains address of */
    /* local_var on tgt */
    .
    .
    /* add value to local_var on some task */
    /* use LAPI to add value to local_var on remote task */
    LAPI_Rmw64(hndl, FETCH_AND_ADD, tgt, addr_list[tgt],
               value, prev_tgt_val, &org_cntr);
    /* local_var on remote task has been increased */
    /* by value. prev_tgt_val now contains value of */
    /* local_var on remote task before the addition */
}
```

2. To synchronize a data value between two tasks (with **SWAP**):

```
{
    long long local_var;
    long long *addr_list;

    /* local_var addresses are exchanged and stored */
    /* local_var on remote task has been increased */
    /* by value. prev_tgt_val now contains value of */
    /* local_var on remote task before the addition */
}
```

```

/* in addr_list (using LAPI_Address_init64).          */
/* addr_list[tgt] now contains the address of        */
/* local_var on tgt.                                  */
.
.
.
/* local_var is assigned some value                  */

/* assign local_var to local_var on the remote task */
LAPI_Rmw64(hndl, SWAP, tgt, addr_list[tgt],
           local_var, prev_tgt_val, &org_cntr);

/* local_var on the remote task is now equal to local_var */
/* on the local task. prev_tgt_val now contains the value */
/* of local_var on the remote task before the swap.      */
}

```

3. To conditionally swap a data value (with **COMPARE_AND_SWAP**):

```

{
    long long local_var;
    long long *addr_list;
    long long in_val[2];

    /* local_var addresses are exchanged and stored */
    /* in addr_list (using LAPI_Address_init64).    */
    /* addr_list[tgt] now contains the address of    */
    /* local_var on tgt.                              */
    .
    .
    .
    /* if local_var on remote_task is equal to comparator, */
    /* assign value to local_var on the remote task          */

    in_val[0] = comparator;
    in_val[1] = value;

    LAPI_Rmw64(hndl, COMPARE_AND_SWAP, tgt, addr_list[tgt],
              in_val, prev_tgt_val, &org_cntr);

    /* local_var on remote task is now in_val[1] if it */
    /* had previously been equal to in_val[0]. If the  */
    /* swap was performed, prev_tgt_val now contains   */
    /* TRUE; otherwise, it contains FALSE.            */
}

```

Related information

Subroutines: **LAPI_Address_init64**, **LAPI_Getcntr**, **LAPI_Qenv**, **LAPI_Rmw**, **LAPI_Setcntr**, **LAPI_Waitcntr**, **LAPI_Xfer**

LAPI_Senv

Purpose

Used to set a runtime variable.

Library

Availability Library (**liblapi_r.a**)

C syntax

```
#include <lapif.h>

int LAPI_Senv(hndl, query, set_val)
lapi_handle_t hndl;
lapi_query_t query;
int set_val;
```

FORTRAN syntax

```
include 'lapif.h'

LAPI_SENV(hndl, query, set_val, ierror)
INTEGER hndl
INTEGER query
INTEGER set_val
INTEGER ierror
```

Parameters

INPUT

hndl Specifies the LAPI handle.

query Specifies the type of query that you want to set. In C, the values for *query* are defined by the **lapi_query_t** enumeration in **lapi.h**. In FORTRAN, these values are defined explicitly in the 32-bit version and the 64-bit version of **lapif.h**.

set_val Specifies the integer value of the query that you want to set.

OUTPUT

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: local queries

Use this subroutine to set runtime attributes for a specific LAPI instance. In C, the **lapi_query_t** enumeration defines the attributes that can be set at runtime. These attributes are defined explicitly in FORTRAN. See **LAPI_Qenv** for more information.

You can use **LAPI_Senv** to set these runtime attributes: **ACK_THRESHOLD**, **ERROR_CHK**, **INTERRUPT_SET**, and **TIMEOUT**.

Return values

LAPI_SUCCESS	Indicates that the function call completed successfully.
---------------------	--

LAPI_ERR_HNDL_INVALID	Indicates that the <i>hndl</i> passed in is not valid (not initialized or in terminated state).
LAPI_ERR_QUERY_TYPE	Indicates the query passed in is not valid.
LAPI_ERR_SET_VAL	Indicates the <i>set_val</i> pointer is not in valid range.

Location

`/usr/lib/liblapi_r.a`

C examples

The following values can be set using **LAPI_Senv**:

```

ACK_THRESHOLD:
int value;
LAPI_Senv(hndl, ACK_THRESHOLD, value);
/* LAPI sends packet acknowledgements (acks) in groups, waiting until */
/* ACK_THRESHOLD packets have arrived before returning a group of acks */
/* The valid range for ACK_THRESHOLD is (1 <= value <= 30) */
/* The default is 30. */

ERROR_CHK:
boolean toggle;
LAPI_Senv(hndl, ERROR_CHK, toggle);
/* Indicates whether LAPI should perform error checking. If set, LAPI */
/* calls will perform bounds-checking on parameters. Error checking */
/* is disabled by default. */

INTERRUPT_SET:
boolean toggle;
LAPI_Senv(hndl, INTERRUPT_SET, toggle);
/* Determines whether LAPI will respond to interrupts. If interrupts */
/* are disabled, LAPI will poll for message completion. */
/* toggle==True will enable interrupts, False will disable. */
/* Interrupts are enabled by default. */

TIMEOUT:
int value;
LAPI_Senv(hndl, TIMEOUT, value);
/* LAPI will time out on a communication if no response is received */
/* within timeout seconds. Valid range is (10 <= timeout <= 86400). */
/* 86400 seconds = 24 hours. Default value is 900 (15 minutes). */

```

Related information

Subroutines: **LAPI_Qenv**

LAPI_Setcncr

Purpose

Used to set a counter to a specified value.

Library

Availability Library (**liblapi_r.a**)

C syntax

```
#include <lapi.h>

int LAPI_Setcncr(hdl, cntr, val)
lapi_handle_t hdl;
lapi_cntr_t *cntr;
int val;
```

FORTRAN syntax

```
include 'lapif.h'

LAPI_SETCNTR(hdl, cntr, val, ierror)
INTEGER hdl
TYPE (LAPI_CNTR_T) :: cntr
INTEGER val
INTEGER ierror
```

Parameters

INPUT

hdl Specifies the LAPI handle.

val Specifies the value to which the counter needs to be set.

INPUT/OUTPUT

cntr Specifies the address of the counter to be set (in C) or the counter structure (in FORTRAN). The value of this parameter cannot be NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

OUTPUT

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: Local counter manipulation

This subroutine sets *cntr* to the value specified by *val*. Because the **LAPI_Getcncr/LAPI_Setcncr** sequence cannot be made atomic, you should only use **LAPI_Setcncr** when you know there will not be any competing operations.

Return values

LAPI_SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_CNTR_NULL Indicates that the *cntr* value passed in is NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

LAPI_ERR_HNDL_INVALID Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

Location

`/usr/lib/liblapi_r.a`

C examples

To initialize a counter for use in a communication API call:

```
{
    lapi_cntr_t  my_tgt_cntr, *tgt_cntr_array;
    int          initial_value, expected_value, current_value;
    lapi_handle_t hndl;
    .
    .
    .
    /*
     * Note: the code below is executed on all tasks
     */

    /* initialize, allocate and create structures */
    initial_value = 0;
    expected_value = 1;

    /* set the cntr to zero */
    LAPI_Setcctr(hndl, &my_tgt_cntr, initial_value);
    /* set other counters */
    .
    .
    .
    /* exchange counter addresses, LAPI_Address_init synchronizes */
    LAPI_Address_init(hndl, &my_tgt_cntr, tgt_cntr_array);
    /* more address exchanges */
    .
    .
    .
    /* Communication calls using my_tgt_cntr */
    LAPI_Put(....., tgt_cntr_array[tgt], ....);
    .
    .
    .
    /* Wait for counter to reach value */
    for (;;) {
        LAPI_Getcctr(hndl, &my_tgt_cntr, &current_value);
        if (current_value >= expected_value) {
            break; /* out of infinite loop */
        } else {
            LAPI_Probe(hndl);
        }
    }
    .
    .
    .
    /* Quiesce/synchronize to ensure communication using our counter is done */
    LAPI_Gfence(hndl);
    /* Reset the counter */
    LAPI_Setcctr(hndl, &my_tgt_cntr, initial_value);
    /*
     * Synchronize again so that no other communication using the counter can
     * begin from any other task until we're all finished resetting the counter.
     */
    LAPI_Gfence(hndl);

    /* More communication calls */
}
```

LAPI_Setcntr

```
    .  
    .  
    .  
}
```

Related information

Subroutines: [LAPI_Getcntr](#), [LAPI_Waitcntr](#)

LAPI_Term

Purpose

Terminates and cleans up a LAPI context.

Library

Availability Library (**liblapi_r.a**)

C syntax

```
#include <lapi.h>

int LAPI_Term(hdl)
lapi_handle_t hdl;
```

FORTRAN syntax

```
include 'lapif.h'

LAPI_TERM(hdl, ierror)
INTEGER hdl
INTEGER ierror
```

Parameters

INPUT

hdl Specifies the LAPI handle.

OUTPUT

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: local termination

Use this subroutine to terminate the LAPI context that is specified by *hdl*. Any LAPI notification threads that are associated with this context are terminated. An error occurs when any LAPI calls are made using *hdl* after **LAPI_Term** is called.

A DGSP that is registered under that LAPI handle remains valid even after **LAPI_Term** is called on *hdl*.

Return values

LAPI_SUCCESS	Indicates that the function call completed successfully.
LAPI_ERR_HNDL_INVALID	Indicates that the <i>hdl</i> passed in is not valid (not initialized or in terminated state).

Location

/usr/lib/liblapi_r.a

LAPI_Term

C examples

To terminate a LAPI context (represented by *hdl*):

```
LAPI_Term(hdl);
```

Related information

Subroutines: **LAPI_Init**, **LAPI_Purge_totask**, **LAPI_Resume_totask**

LAPI_Util

Purpose

Serves as a wrapper function for such data gather/scatter operations as registration and reservation, for updating UDP port information, and for obtaining pointers to locking and signaling functions that are associated with a shared LAPI lock.

Library

Availability Library (**liblapi_r.a**)

C syntax

```
#include <lapi.h>

int LAPI_Util(hdl, util_cmd)
lapi_handle_t hdl;
lapi_util_t *util_cmd;
```

FORTRAN syntax

```
include 'lapif.h'

LAPI_UTIL(hdl, util_cmd, ierror)
INTEGER hdl
TYPE (LAPI_UTIL_T) :: util_cmd
INTEGER ierror
```

Parameters

INPUT

hdl Specifies the LAPI handle.

INPUT/OUTPUT

util_cmd Specifies the command type of the utility function.

OUTPUT

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: Data gather/scatter program (DGSP), UDP port information, and lock sharing utilities

This subroutine is used for several different operations, which are indicated by the command type value in the beginning of the command structure. The **lapi_util_t** structure is defined as:

```
typedef union {
    lapi_util_type_t    Util_type;
    lapi_reg_dgsp_t     RegDgsp;
    lapi_dref_dgsp_t    DrefDgsp;
    lapi_resv_dgsp_t    ResvDgsp;
    lapi_reg_ddm_t      DdmFunc;
    lapi_add_udp_port_t  Udp;
    lapi_pack_dgsp_t    PackDgsp;
    lapi_unpack_dgsp_t  UnpackDgsp;
    lapi_thread_func_t  ThreadFunc;
} lapi_util_t;
```

The enumerated type **lapi_util_type_t** has these values:

Table 14. *lapi_util_type_t* types

Value of <i>Util_type</i>	Union member as interpreted by LAPI_Util
LAPI_REGISTER_DGSP	lapi_reg_dgsp_t
LAPI_UNRESERVE_DGSP	lapi_dref_dgsp_t
LAPI_RESERVE_DGSP	lapi_resv_dgsp_t
LAPI_REG_DDM_FUNC	lapi_reg_ddm_t
LAPI_ADD_UDP_DEST_PORT	lapi_add_udp_port_t
LAPI_DGSP_PACK	lapi_pack_dgsp_t
LAPI_DGSP_UNPACK	lapi_unpack_dgsp_t
LAPI_GET_THREAD_FUNC	lapi_thread_func_t

hndl is not checked for command type **LAPI_REGISTER_DGSP**, **LAPI_RESERVE_DGSP**, or **LAPI_UNRESERVE_DGSP**.

LAPI_REGISTER_DGSP

You can use this operation to register a LAPI DGSP that you have created. To register a LAPI DGSP, **lapi_dgsp_descr_t** *idgsp* must be passed in. LAPI returns a handle (**lapi_dg_handle_t** *dgsp_handle*) to use for all future LAPI calls. The *dgsp_handle* that is returned by a register operation is identified as a **lapi_dg_handle_t** type, which is the appropriate type for **LAPI_Xfer** and **LAPI_Util** calls that take a DGSP. This returned *dgsp_handle* is also defined to be castable to a pointer to a **lapi_dgsp_descr_t** for those situations where the LAPI user requires read-only access to information that is contained in the cached DGSP. The register operation delivers a DGSP to LAPI for use in future message send, receive, pack, and unpack operations. LAPI creates its own copy of the DGSP and protects it by reference count. All internal LAPI operations that depend on a DGSP cached in LAPI ensure the preservation of the DGSP by incrementing the reference count when they begin a dependency on the DGSP and decrementing the count when that dependency ends. A DGSP, once registered, can be used from any LAPI instance. **LAPI_Term** does not discard any DGSPs.

You can register a DGSP, start one or more LAPI operations using the DGSP, and then unreserve it with no concern about when the LAPI operations that depend on the DGSP will be done using it. See **LAPI_RESERVE_DGSP** and **LAPI_UNRESERVE_DGSP** for more information.

In general, the DGSP you create and pass in to the **LAPI_REGISTER_DGSP** call using the *dgsp* parameter is discarded after LAPI makes and caches its own copy. Because DGSP creation is complex, user errors may occur, but extensive error checking at data transfer time would hurt performance. When developing code that creates DGSPs, you can invoke extra validation at the point of registration by setting the **LAPI_VERIFY_DGSP** environment variable. **LAPI_Util** will return any detected errors. Any errors that exist and are not detected at registration time will cause problems during data transfer. Any errors detected during data transfer will be reported by an asynchronous error handler. A segmentation fault is one common symptom of a faulty DGSP. If multiple DGSPs are in use, the asynchronous error handler will not be able to identify which DGSP caused the error. For more information about asynchronous error handling, see **LAPI_Init**.

LAPI_REGISTER_DGSP uses the **lapi_reg_dgsp_t** command structure.

Table 15. The *lapi_reg_dgsp_t* fields

lapi_reg_dgsp_t field	lapi_reg_dgsp_t field type	lapi_reg_dgsp_t usage
<i>Util_type</i>	lapi_util_type_t	LAPI_REGISTER_DGSP
<i>idgsp</i>	lapi_dgsp_descr_t	IN - pointer to DGSP program
<i>dgsp_handle</i>	lapi_dg_handle_t	OUT - handle for a registered DGSP program
<i>in_usr_func</i>	lapi_usr_fcall_t	For debugging only
<i>status</i>	lapi_status_t	OUT - future support

LAPI_RESERVE_DGSP

You can use this operation to reserve a DGSP. This operation is provided because a LAPI client might cache a LAPI DGSP handle for later use. The client needs to ensure the DGSP will not be discarded before the cached handle is used. A DGSP handle, which is defined to be a pointer to a DGSP description that is already cached inside LAPI, is passed to this operation. The DGSP handle is also defined to be a structure pointer, so that client programs can get direct access to information in the DGSP. Unless the client can be certain that the DGSP will not be "unreserved" by another thread while it is being accessed, the client should bracket the access window with its own reserve/unreserve operation. The client is not to modify the cached DGSP, but LAPI has no way to enforce this. The reserve operation increments the user reference count, thus protecting the DGSP until an unreserve operation occurs. This is needed because the thread that placed the reservation will expect to be able to use or examine the cached DGSP until it makes an unreserve call (which decrements the user reference count), even if the unreserve operation that matches the original register operation occurs within this window on some other thread.

LAPI_RESERVE_DGSP uses the **lapi_resv_dgsp_t** command structure.

Table 16. The *lapi_resv_dgsp_t* fields

lapi_resv_dgsp_t field	lapi_resv_dgsp_t field type	lapi_resv_dgsp_t usage
<i>Util_type</i>	lapi_util_type_t	LAPI_RESERVE_DGSP
<i>dgsp_handle</i>	lapi_dg_handle_t	OUT - handle for a registered DGSP program
<i>in_usr_func</i>	lapi_usr_fcall_t	For debugging only
<i>status</i>	lapi_status_t	OUT - future support

LAPI_UNRESERVE_DGSP

You can use this operation to unregister or unreserve a DGSP. This operation decrements the user reference count. If external and internal reference counts are zero, this operation lets LAPI free the DGSP. All operations that decrement a reference count cause LAPI to check to see if the counts have both become 0 and if they have, dispose of the DGSP. Several internal LAPI activities increment and decrement a second reference count. The cached DGSP is disposable only when all activities (internal and external) that depend on it and use reference counting to preserve it have discharged their reference. The DGSP handle is passed to LAPI as a value parameter and LAPI does not nullify the caller's handle. It is your

responsibility to not use this handle again because in doing an unreserve operation, you have indicated that you no longer count on the handle remaining valid.

LAPI_UNRESERVE_DGSP uses the **lapi_dref_dgsp_t** command structure.

Table 17. The *lapi_dref_dgsp_t* fields

lapi_dref_dgsp_t field	lapi_dref_dgsp_t field type	lapi_dref_dgsp_t usage
<i>Util_type</i>	lapi_util_type_t	LAPI_UNRESERVE_DGSP
<i>dgsp_handle</i>	lapi_dg_handle_t	OUT - handle for a registered DGSP program
<i>in_usr_func</i>	lapi_usr_fcall_t	For debugging only
<i>status</i>	lapi_status_t	OUT - future support

LAPI_REG_DDM_FUNC

You can use this operation to register data distribution manager (DDM) functions. It works in conjunction with the DGSM CONTROL instruction. Primarily, it is used for **MPI_Accumulate**, but LAPI clients can provide any DDM function. It is also used to establish a callback function for processing data that is being scattered into a user buffer on the destination side.

The native LAPI user can install a callback without affecting the one MPI has registered for **MPI_Accumulate**. The function prototype for the callback function is:

```
typedef long ddm_func_t (      /* return number of bytes processed */
    void *in,                /* pointer to inbound data */
    void *inout,             /* pointer to destination space */
    long bytes,              /* number of bytes inbound */
    int operand,             /* CONTROL operand value */
    int operation            /* CONTROL operation value */
);
```

A DDM function acts between the arrival of message data and the target buffer. The most common usage is to combine inbound data with data already in the target buffer. For example, if the target buffer is an array of integers and the incoming message consists of integers, the DDM function can be written to add each incoming integer to the value that is already in the buffer. The *operand* and *operation* fields of the DDM function allow one DDM function to support a range of operations with the CONTROL instruction by providing the appropriate values for these fields.

See “Using data gather/scatter programs (DGSPs)” on page 43 for more information.

LAPI_REG_DDM_FUNC uses the **lapi_reg_ddm_t** command structure. Each call replaces the previous function pointer, if there was one.

Table 18. The *lapi_reg_ddm_t* fields

lapi_reg_ddm_t field	lapi_reg_ddm_t field type	lapi_reg_ddm_t usage
<i>Util_type</i>	lapi_util_type_t	LAPI_REG_DDM_FUNC
<i>ddm_func</i>	ddm_func_t *	IN - DDM function pointer
<i>in_usr_func</i>	lapi_usr_fcall_t	For debugging only
<i>status</i>	lapi_status_t	OUT - future support

LAPI_DGSP_PACK

You can use this operation to gather data to a pack buffer from a user buffer under control of a DGSP. A single buffer may be packed by a series of calls. The caller provides a *position* value that is initialized to the starting offset within the buffer. Each pack operation adjusts *position*, so the next pack operation can begin where the previous pack operation ended. In general, a series of pack operations begins with *position* initialized to **0**, but any offset is valid. There is no state carried from one pack operation to the next. Each pack operation starts at the beginning of the DGSP it is passed.

LAPI_DGSP_PACK uses the **lapi_pack_dgsp_t** command structure.

Table 19. The *lapi_pack_dgsp_t* fields

lapi_pack_dgsp_t field	lapi_pack_dgsp_t field type	lapi_pack_dgsp_t usage
<i>Util_type</i>	lapi_util_type_t	LAPI_DGSP_PACK
<i>dgsp_handle</i>	lapi_dg_handle_t	OUT - handle for a registered DGSP program
<i>in_buf</i>	void *	IN - source buffer to pack
<i>bytes</i>	ulong	IN - number of bytes to pack
<i>out_buf</i>	void *	OUT - output buffer for pack
<i>out_size</i>	ulong	IN - output buffer size in bytes
<i>position</i>	ulong	IN/OUT - current buffer offset
<i>in_usr_func</i>	lapi_usr_fcall_t	For debugging only
<i>status</i>	lapi_status_t	OUT - future support

LAPI_DGSP_UNPACK

You can use this operation to scatter data from a packed buffer to a user buffer under control of a DGSP. A single buffer may be unpacked by a series of calls. The caller provides a *position* value that is initialized to the starting offset within the packed buffer. Each unpack operation adjusts *position*, so the next unpack operation can begin where the previous unpack operation ended. In general, a series of unpack operations begins with *position* initialized to **0**, but any offset is valid. There is no state carried from one unpack operation to the next. Each unpack operation starts at the beginning of the DGSP it is passed.

LAPI_DGSP_UNPACK uses the **lapi_unpack_dgsp_t** command structure.

Table 20. The *lapi_unpack_dgsp_t* fields

lapi_unpack_dgsp_t field	lapi_unpack_dgsp_t field type	lapi_unpack_dgsp_t usage
<i>Util_type</i>	lapi_util_type_t	LAPI_DGSP_UNPACK
<i>dgsp_handle</i>	lapi_dg_handle_t	OUT - handle for a registered DGSP program
<i>buf</i>	void *	IN - source buffer for unpack
<i>in_size</i>	ulong	IN - source buffer size in bytes
<i>out_buf</i>	void *	OUT - output buffer for unpack
<i>bytes</i>	ulong	IN - number of bytes to unpack
<i>out_size</i>	ulong	IN - output buffer size in bytes

Table 20. The *lapi_unpack_dgsp_t* fields (continued)

lapi_unpack_dgsp_t field	lapi_unpack_dgsp_t field type	lapi_unpack_dgsp_t usage
<i>position</i>	ulong	IN/OUT - current buffer offset
<i>in_usr_func</i>	lapi_usr_fcall_t	For debugging only
<i>status</i>	lapi_status_t	OUT - future support

LAPI_ADD_UDP_DEST_PORT

You can use this operation to update UDP port information about the destination task. This operation can be used when you have written your own UDP handler (*udp_hndlr*) and you need to support recovery of failed tasks. You cannot use this operation under the POE runtime environment.

LAPI_ADD_UDP_DEST_PORT uses the **lapi_add_udp_port_t** command structure.

Table 21. The *lapi_add_udp_port_t* fields

lapi_add_udp_port_t field	lapi_add_udp_port_t field type	lapi_add_udp_port_t usage
<i>Util_type</i>	lapi_util_type_t	LAPI_ADD_UDP_DEST_PORT
<i>tgt</i>	uint	IN - destination task ID
<i>udp_port</i>	lapi_udp_t *	IN - UDP port information for the target
<i>instance_no</i>	uint	IN - Instance number of UDP
<i>in_usr_func</i>	lapi_usr_fcall_t	For debugging only
<i>status</i>	lapi_status_t	OUT - future support

LAPI_GET_THREAD_FUNC

You can use this operation to retrieve various shared locking and signalling functions. Retrieval of these functions is valid only after LAPI is initialized and before LAPI is terminated. You should not call any of these functions after LAPI is terminated.

LAPI_GET_THREAD_FUNC uses the **lapi_thread_func_t** command structure.

Table 22. The *lapi_thread_func_t* fields

lapi_thread_func_t field	lapi_thread_func_t field type	lapi_thread_func_t usage
<i>Util_type</i>	lapi_util_type_t	LAPI_GET_THREAD_FUNC
<i>mutex_lock</i>	lapi_mutex_lock_t	OUT - mutex lock function pointer
<i>mutex_unlock</i>	lapi_mutex_unlock_t	OUT - mutex unlock function pointer
<i>mutex_trylock</i>	lapi_mutex_trylock_t	OUT - mutex try lock function pointer
<i>mutex_getowner</i>	lapi_mutex_getowner_t	OUT - mutex get owner function pointer
<i>cond_wait</i>	lapi_cond_wait_t	OUT - condition wait function pointer
<i>cond_timedwait</i>	lapi_cond_timedwait_t	OUT - condition timed wait function pointer
<i>cond_signal</i>	lapi_cond_signal_t	OUT - condition signal function pointer

Table 22. The *lapi_thread_func_t* fields (continued)

lapi_thread_func_t field	lapi_thread_func_t field type	lapi_thread_func_t usage
<i>cond_init</i>	lapi_cond_init_t	OUT - initialize condition function pointer
<i>cond_destroy</i>	lapi_cond_destroy_t	OUT - destroy condition function pointer

LAPI uses the pthread library for thread ID management. You can therefore use **pthread_self()** to get the running thread ID and **lapi_mutex_getowner_t** to get the thread ID that owns the shared lock. Then, you can use **pthread_equal()** to see if the two are the same.

Mutex thread functions: **LAPI_GET_THREAD_FUNC** includes the following mutex thread functions: mutex lock, mutex unlock, mutex try lock, and mutex get owner.

Mutex lock function pointer

```
int (*lapi_mutex_lock_t)(lapi_handle_t hndl);
```

This function acquires the lock that is associated with the specified LAPI handle. The call blocks if the lock is already held by another thread. Deadlock can occur if the calling thread is already holding the lock. You are responsible for preventing and detecting deadlocks.

Parameters

INPUT

hndl Specifies the LAPI handle.

Return values

0 Indicates that the lock was acquired successfully.

EINVAL Is returned if the lock is not valid because of an incorrect *hndl* value.

Mutex unlock function pointer

```
int (*lapi_mutex_unlock_t)(lapi_handle_t hndl);
```

This function releases the lock that is associated with the specified LAPI handle. A thread should only unlock its own locks.

Parameters

INPUT

hndl Specifies the LAPI handle.

Return values

0 Indicates that the lock was released successfully.

EINVAL Is returned if the lock is not valid because of an incorrect *hndl* value.

Mutex try lock function pointer

```
int (*lapi_mutex_trylock_t)(lapi_handle_t hndl);
```

This function tries to acquire the lock that is associated with the specified LAPI handle, but returns immediately if the lock is already held.

Parameters

INPUT

hndl Specifies the LAPI handle.

Return values

0 Indicates that the lock was acquired successfully.

EBUSY Indicates that the lock is being held.

EINVAL Is returned if the lock is not valid because of an incorrect *hndl* value.

Mutex get owner function pointer

```
int (*lapi_mutex_getowner_t)(lapi_handle_t hndl, pthread_t *tid);
```

This function gets the pthread ID of the thread that is currently holding the lock associated with the specified LAPI handle. **LAPI_NULL_THREAD_ID** indicates that the lock is not held at the time the function is called.

Parameters

INPUT

hndl Specifies the LAPI handle.

OUTPUT

tid Is a pointer to hold the pthread ID to be retrieved.

Return values

0 Indicates that the lock owner was retrieved successfully.

EINVAL Is returned if the lock is not valid because of an incorrect *hndl* value.

Condition functions: **LAPI_GET_THREAD_FUNC** includes the following condition functions: condition wait, condition timed wait, condition signal, initialize condition, and destroy condition.

Condition wait function pointer

```
int (*lapi_cond_wait_t)(lapi_handle_t hndl, lapi_cond_t *cond);
```

This function waits on a condition variable (*cond*). The user must hold the lock associated with the LAPI handle (*hndl*) before making the call. Upon the return of the call, LAPI guarantees that the lock is still being held. The same LAPI handle must be supplied to concurrent **lapi_cond_wait_t** operations on the same condition variable.

Parameters

INPUT

hdl Specifies the LAPI handle.
cond Is a pointer to the condition variable to be waited on.

Return values

0 Indicates that the condition variable has been signaled.
EINVAL Indicates that the value specified by *hdl* or *cond* is not valid.

Condition timed wait function pointer

```
int (*lapi_cond_timedwait_t)(lapi_handle_t hndl,
                           lapi_cond_t *cond,
                           struct timespec *timeout);
```

This function waits on a condition variable (*cond*). The user must hold the lock associated with the LAPI handle (*hdl*) before making the call. Upon the return of the call, LAPI guarantees that the lock is still being held. The same LAPI handle must be supplied to concurrent **lapi_cond_timedwait_t** operations on the same condition variable.

Parameters**INPUT**

hdl Specifies the LAPI handle.
cond Is a pointer to the condition variable to be waited on.
timeout Is a pointer to the absolute time structure specifying the timeout.

Return values

0 Indicates that the condition variable has been signaled.
ETIMEDOUT Indicates that time specified by *timeout* has passed.
EINVAL Indicates that the value specified by *hdl*, *cond*, or *timeout* is not valid.

Condition signal function pointer

```
int (*lapi_cond_wait_t)(lapi_handle_t hndl, lapi_cond_t *cond);
typedef int (*lapi_cond_signal_t)(lapi_handle_t hndl, lapi_cond_t *cond);
```

This function signals a condition variable (*cond*) to wake up a thread that is blocked on the condition. If there are multiple threads waiting on the condition variable, which thread to wake up is decided randomly.

Parameters**INPUT**

hdl Specifies the LAPI handle.
cond Is a pointer to the condition variable to be signaled.

Return values

0 Indicates that the condition variable has been signaled.

EINVAL Indicates that the value specified by *hdl* or *cond* is not valid.

Initialize condition function pointer

```
int (*lapi_cond_init_t)(lapi_handle_t hndl, lapi_cond_t *cond);
```

This function initializes a condition variable.

Parameters

INPUT

hdl Specifies the LAPI handle.

cond Is a pointer to the condition variable to be initialized.

Return values

0 Indicates that the condition variable was initialized successfully.

EAGAIN Indicates that the system lacked the necessary resources (other than memory) to initialize another condition variable.

ENOMEM Indicates that there is not enough memory to initialize the condition variable.

EINVAL Is returned if the *hdl* value is not valid.

Destroy condition function pointer

```
int (*lapi_cond_destroy_t)(lapi_handle_t hndl, lapi_cond_t *cond);
```

This function destroys a condition variable.

Parameters

INPUT

hdl Specifies the LAPI handle.

cond Is a pointer to the condition variable to be destroyed.

Return values

0 Indicates that the condition variable was destroyed successfully.

EBUSY Indicates that the implementation has detected an attempt to destroy the object referenced by *cond* while it is referenced (while being used in a **`lapi_cond_wait_t`** or **`lapi_cond_timedwait_t`** by another thread, for example).

EINVAL Indicates that the value specified by *hdl* or *cond* is not valid.

Return values

LAPI_SUCCESS Indicates that the function call completed successfully.

LAPI_ERR_DGSP Indicates that the DGSP that was passed in is

	NULL (in C) or LAPI_ADDR_NULL (in FORTRAN) or is not a registered DGSP.
LAPI_ERR_DGSP_ATOM	Indicates that the DGSP has an <i>atom_size</i> that is less than 0 or greater than <u>MAX_ATOM_SIZE</u> .
LAPI_ERR_DGSP_BRANCH	Indicates that the DGSP attempted a branch that fell outside of the code array. This is returned only in validation mode.
LAPI_ERR_DGSP_COPY_SZ	Is returned with DGSP validation turned on when MCOPY block < 0 or COPY instruction with bytes < 0 . This is returned only in validation mode.
LAPI_ERR_DGSP_FREE	Indicates that LAPI tried to free a DGSP that is not valid or is no longer registered. There should be one LAPI_UNRESERVE_DGSP operation to close the LAPI_REGISTER_DGSP operation and one LAPI_UNRESERVE_DGSP operation for each LAPI_RESERVE_DGSP operation.
LAPI_ERR_DGSP_OPC	Indicates that the DGSP <i>opcode</i> is not valid. This is returned only in validation mode.
LAPI_ERR_DGSP_STACK	Indicates that the DGSP has a greater GOSUB depth than the allocated stack supports. Stack allocation is specified by the <i>dgsp->depth</i> member. This is returned only in validation mode.
LAPI_ERR_HNDL_INVALID	Indicates that the <i>hndl</i> passed in is not valid (not initialized or in terminated state).
LAPI_ERR_MEMORY_EXHAUSTED	Indicates that LAPI is unable to obtain memory from the system.
LAPI_ERR_UDP_PORT_INFO	Indicates that the <i>udp_port</i> information pointer is NULL (in C) or that the value of <i>udp_port</i> is LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_UTIL_CMD	Indicates that the command type is not valid.

Location

`/usr/lib/liblapi_r.a`

C examples

1. To create and register a DGSP:

```
{
    /*
    ** DGSP code array. DGSP instructions are stored
    ** as ints (with constants defined in lapi.h for
    ** the number of integers needed to store each
    ** instruction). We will have one COPY and one ITERATE
    ** instruction in our DGSP. We use LAPI's constants
    ** to allocate the appropriate storage.
    */
    int code[LAPI_DGSM_COPY_SIZE+LAPI_DGSM_ITERATE_SIZE];

    /* DGSP description */
    lapi_dgsp_descr_t dgsp_d;
```

```

/*
** Data structure for the xfer call.
*/
lapi_xfer_t  xfer_struct;

/* DGSP data structures */
lapi_dgsm_copy_t  *copy_p;  /* copy instruction */
lapi_dgsm_iterate_t  *iter_p;  /* iterate instruction */
int  *code_ptr; /* code pointer */

/* constant for holding code array info */
int  code_less_iterate_size;

/* used for DGSP registration */
lapi_reg_dgsp_t  reg_util;

/*
** Set up dgsp description
*/

/* set pointer to code array */
dgsp_d.code = &code[0];

/* set size of code array */
dgsp_d.code_size = LAPI_DGSM_COPY_SIZE + LAPI_DGSM_ITERATE_SIZE;

/* not using DGSP gosub instruction */
dgsp_d.depth = 1;

/*
** set density to show internal gaps in the
** DGSP data layout
*/
dgsp_d.density = LAPI_DGSM_SPARSE;

/* transfer 4 bytes at a time */
dgsp_d.size = 4;

/* advance the template by 8 for each iteration */
dgsp_d.extent = 8;

/*
** ext specifies the memory 'footprint' of
** data to be transferred. The ltext specifies
** the offset from the base address to begin
** viewing the data. The rtext specifies the
** length from the base address to use.
*/
dgsp_d.ltext = 0;
dgsp_d.rtext = 4;
/* atom size of 0 lets LAPI choose the packet size */
dgsp_d.atom_size = 0;

/*
** set up the copy instruction
*/
copy_p = (lapi_dgsm_copy_t *) (dgsp_d.code);
copy_p->opcode = LAPI_DGSM_COPY;

/* copy 4 bytes at a time */
copy_p->bytes = (long) 4;

/* start at offset 0 */
copy_p->offset = (long) 0;

```

```

/* set code pointer to address of iterate instruction */
code_less_iterate_size = dgsp_d.code_size - LAPI_DGSM_ITERATE_SIZE;
code_ptr = ((int *) (code)) + code_less_iterate_size;

/*
** Set up iterate instruction
*/
iter_p = (lapi_dgsm_iterate_t *) code_ptr;
iter_p->opcode = LAPI_DGSM_ITERATE;
iter_p->iter_loc = (-code_less_iterate_size);

/* Set up and do DGSP registration */
reg_util.Util_type = LAPI_REGISTER_DGSP;
reg_util.idgsp = &dgsp_d;
LAPI_Util(hndl, (lapi_util_t *) &reg_util);

/*
** LAPI returns a usable DGSP handle in
** reg_util.dgsp_handle
** Use this handle for subsequent reserve/unreserve
** and Xfer calls. On the receive side, this
** handle can be returned by the header handler using the
** return_info_t mechanism. The DGSP will then be used for
** scattering data.
*/
}

```

2. To reserve a DGSP handle:

```

{
    reg_util.dgsp_handle = dgsp_handle;

    /*
    ** dgsp_handle has already been created and
    ** registered as in the above example
    */

    reg_util.Util_type = LAPI_RESERVE_DGSP;
    LAPI_Util(hndl, (lapi_util_t *) &reg_util);

    /*
    ** LAPI's internal reference count to dgsp_handle
    ** will be incremented. DGSP will
    ** remain available until an unreserve is
    ** done for each reserve, plus one more for
    ** the original registration.
    */
}

```

3. To unreserve a DGSP handle:

```

{
    reg_util.dgsp_handle = dgsp_handle;

    /*
    ** dgsp_handle has already created and
    ** registered as in the above example, and
    ** this thread no longer needs it.
    */

    reg_util.Util_type = LAPI_UNRESERVE_DGSP;
}

```

LAPI_Util

```
LAPI_Util(hndl, (lapi_util_t *)&reg_util);  
  
/*  
** An unreserve is required for each reserve,  
** plus one more for the original registration.  
*/  
  
}
```

Related information

Subroutines: **LAPI_Init**, **LAPI_Xfer**

LAPI_Waitcncntr

Purpose

Waits until a specified counter reaches the value specified.

Library

Availability Library (**liblapi_r.a**)

C syntax

```
#include <lapi.h>

int LAPI_Waitcncntr(hndl, cncntr, val, cur_cncntr_val)
lapi_handle_t hndl;
lapi_cncntr_t *cncntr;
int val;
int *cur_cncntr_val;
```

FORTRAN syntax

```
include 'lapif.h'

LAPI_WAITCNCNTR(hndl, cncntr, val, cur_cncntr_val, ierror)
INTEGER hndl
TYPE (LAPI_CNCNTR_T) :: cncntr
INTEGER val
INTEGER cur_cncntr_val
INTEGER ierror
```

Parameters

INPUT

hndl Specifies the LAPI handle.

val Specifies the value the counter needs to reach.

INPUT/OUTPUT

cncntr Specifies the counter structure (in FORTRAN) to be waited on or its address (in C). The value of this parameter cannot be NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

OUTPUT

cur_cncntr_val Specifies the integer value of the current counter. This value can be NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: local progress monitor (blocking)

This subroutine waits until *cncntr* reaches or exceeds the specified *val*. Once *cncntr* reaches *val*, *cncntr* is decremented by the value of *val*. In this case, "decremented" is used (as opposed to "set to zero") because *cncntr* could have contained a value that was greater than the specified *val* when the call was made. This call may or may not check for message arrivals over the LAPI context *hndl*. The *cur_cncntr_val* variable is set to the current counter value.

LAPI_Waitcncntr

Return values

LAPI_SUCCESS	Indicates that the function call completed successfully.
LAPI_ERR_CNTR_NULL	Indicates that the <i>cntr</i> pointer is NULL (in C) or that the value of <i>cntr</i> is LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_HNDL_INVALID	Indicates that the <i>hndl</i> passed in is not valid (not initialized or in terminated state).

Location

`/usr/lib/liblapi_r.a`

C examples

To wait on a counter to reach a specified value:

```
{  
  
    int          val;  
    int          cur_cntr_val;  
    lapi_cntr_t  some_cntr;  
    .  
    .  
    .  
    LAPI_Waitcncntr(hndl, &some_cntr, val, &cur_cntr_val);  
    /* Upon return, some_cntr has reached val */  
  
}
```

Related information

Subroutines: **LAPI_Amsend**, **LAPI_Amsendv**, **LAPI_Get**, **LAPI_Getcncntr**, **LAPI_Getv**, **LAPI_Put**, **LAPI_Putv**, **LAPI_Rmw**, **LAPI_Rmw64**, **LAPI_Setcncntr**, **LAPI_Xfer**

LAPI_Xfer

Purpose

Serves as a wrapper function for LAPI data transfer functions.

Library

Availability Library (**liblapi_r.a**)

C syntax

```
#include <lapi.h>

int LAPI_Xfer(hndl, xfer_cmd)
lapi_handle_t hndl;
lapi_xfer_t *xfer_cmd;

typedef struct {
    uint          src;          /* Target task ID */
    uint          reason;      /* LAPI return codes */
    ulong        reserve[6];   /* Reserved */
} lapi_sh_info_t;

typedef void (scompl_hdlr_t)(lapi_handle_t *hndl, void *completion_param,
                             lapi_sh_info_t *info);
```

FORTRAN syntax

```
include 'lapif.h'

LAPI_XFER(hndl, xfer_cmd, ierror)
INTEGER hndl
TYPE (fortran_xfer_type) :: xfer_cmd
INTEGER ierror
```

Parameters

INPUT

hndl Specifies the LAPI handle.

xfer_cmd Specifies the name and parameters of the data transfer function.

OUTPUT

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: point-to-point communication (non-blocking)

The **LAPI_Xfer** subroutine provides a superset of the functionality of these subroutines: **LAPI_Amsend**, **LAPI_Amsendv**, **LAPI_Put**, **LAPI_Putv**, **LAPI_Get**, **LAPI_Getv**, and **LAPI_Rmw**. In addition, **LAPI_Xfer** provides data gather/scatter program (DGSP) messages transfer.

In C, the **LAPI_Xfer** command is passed a pointer to a union. It examines the first member of the union, **Xfer_type**, to determine the transfer type, and to determine which union member was passed. **LAPI_Xfer** expects every field of the identified

LAPI_Xfer

union member to be set. It does not examine or modify any memory outside of the identified union member. **LAPI_Xfer** treats all union members (except **status**) as read-only data.

This subroutine provides the following functions:

- The remote address fields are expanded to be of type **lapi_long_t**, which is long enough for a 64-bit address. This allows a 32-bit task to send data to 64-bit addresses, which may be important in client/server programs.
- **LAPI_Xfer** allows the origin counter to be replaced with a send completion callback.
- **LAPI_Xfer** is used to transfer data using LAPI's data gather/scatter program (DGSP) interface.

The **lapi_xfer_t** structure is defined as:

```
typedef union {
    lapi_xfer_type_t  Xfer_type;
    lapi_get_t       Get;
    lapi_am_t        Am;
    lapi_rmw_t       Rmw;
    lapi_put_t       Put;
    lapi_getv_t      Getv;
    lapi_putv_t      Putv;
    lapi_amv_t       Amv;
    lapi_amdgspt_t   Dgsp;
} lapi_xfer_t;
```

Though the **lapi_xfer_t** structure applies only to the C version of **LAPI_Xfer**, the following tables include the FORTRAN equivalents of the C datatypes.

Table 23 list the values of the **lapi_xfer_type_t** structure for C and the explicit *Xfer_type* values for FORTRAN.

Table 23. LAPI_Xfer structure types

Value of <i>Xfer_type</i> (C or FORTRAN)	Union member as interpreted by LAPI_Xfer (C)	Value of <i>fortran_xfer_type</i> (FORTRAN)
LAPI_AM_XFER	lapi_am_t	LAPI_AM_T
LAPI_AMV_XFER	lapi_amv_t	LAPI_AMV_T
LAPI_DGSP_XFER	lapi_amdgspt_t	LAPI_AMDGSP_T
LAPI_GET_XFER	lapi_get_t	LAPI_GET_T
LAPI_GETV_XFER	lapi_getv_t	LAPI_GETV_T
LAPI_PUT_XFER	lapi_put_t	LAPI_PUT_T
LAPI_PUTV_XFER	lapi_putv_t	LAPI_PUTV_T
LAPI_RM_W_XFER	lapi_rmw_t	LAPI_RM_W_T

lapi_am_t details

Table 24 on page 221 shows the correspondence among the parameters of the **LAPI_Amsend** subroutine, the fields of the C **lapi_am_t** structure and their datatypes, and the equivalent FORTRAN datatypes. The **lapi_am_t** fields are listed in Table 24 on page 221 in the order that they occur in the **lapi_xfer_t** structure.

Table 24. LAPI_Amsend and lapi_am_t equivalents

lapi_am_t field name (C)	lapi_am_t field type (C)	Equivalent FORTRAN datatype	Equivalent LAPI_Amsend parameter
<i>Xfer_type</i>	lapi_xfer_type_t	INTEGER(KIND = 4)	implicit in C LAPI_Xfer value in FORTRAN: LAPI_AM_XFER
<i>flags</i>	int	INTEGER(KIND = 4)	none LAPI_Xfer parameter in FORTRAN: <i>flags</i>
<i>tgt</i>	uint	INTEGER(KIND = 4)	<i>tgt</i>
none	none	INTEGER(KIND = 4)	LAPI_Xfer parameter in FORTRAN: <i>pad</i>
<i>hdr_hdl</i>	lapi_long_t	INTEGER(KIND = 8)	<i>hdr_hdl</i>
<i>uhdr_len</i>	uint	INTEGER(KIND = 4)	<i>uhdr_len</i>
none	none	INTEGER(KIND = 4)	LAPI_Xfer parameter in FORTRAN (64-bit): <i>pad2</i>
<i>uhdr</i>	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	<i>uhdr</i>
<i>udata</i>	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	<i>udata</i>
<i>udata_len</i>	ulong	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	<i>udata_len</i>
<i>shdlr</i>	scompl_hndlr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	none LAPI_Xfer parameter in FORTRAN: <i>shdlr</i>
<i>sinfo</i>	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	none LAPI_Xfer parameter in FORTRAN: <i>sinfo</i>
<i>tgt_cntr</i>	lapi_long_t	INTEGER(KIND = 8)	<i>tgt_cntr</i>
<i>org_cntr</i>	lapi_cntr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	<i>org_cntr</i>
<i>cmpl_cntr</i>	lapi_cntr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	<i>cmpl_cntr</i>

When the origin data buffer is free to be used, the pointer to the send completion handler (*shdlr*) is called with the send completion data (*sinfo*) if *shdlr* is not a NULL pointer (in C) or **LAPI_ADDR_NULL** (in FORTRAN). Otherwise, the behavior is identical to that of **LAPI_Amsend**.

lapi_amv_t details

Table 25 on page 222 shows the correspondence among the parameters of the **LAPI_Amsendv** subroutine, the fields of the C **lapi_amv_t** structure and their datatypes, and the equivalent FORTRAN datatypes. The **lapi_amv_t** fields are listed in Table 25 on page 222 in the order that they occur in the **lapi_xfer_t** structure.

LAPI_Xfer

Table 25. LAPI_Amsendv and lapi_amv_t equivalents

lapi_amv_t field name (C)	lapi_amv_t field type (C)	Equivalent FORTRAN datatype	Equivalent LAPI_Amsendv parameter
<i>Xfer_type</i>	lapi_xfer_type_t	INTEGER(KIND = 4)	implicit in C LAPI_Xfer value in FORTRAN: LAPI_AMV_XFER
<i>flags</i>	int	INTEGER(KIND = 4)	none LAPI_Xfer parameter in FORTRAN: <i>flags</i>
<i>tgt</i>	uint	INTEGER(KIND = 4)	<i>tgt</i>
none	none	INTEGER(KIND = 4)	LAPI_Xfer parameter in FORTRAN: <i>pad</i>
<i>hdr_hdl</i>	lapi_long_t	INTEGER(KIND = 8)	<i>hdr_hdl</i>
<i>uhdr_len</i>	uint	INTEGER(KIND = 4)	<i>uhdr_len</i>
none	none	INTEGER(KIND = 4)	LAPI_Xfer parameter in FORTRAN (64-bit): <i>pad2</i>
<i>uhdr</i>	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	<i>uhdr</i>
<i>shdlr</i>	scompl_hndlr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	none LAPI_Xfer parameter in FORTRAN: <i>shdlr</i>
<i>sinfo</i>	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	none LAPI_Xfer parameter in FORTRAN: <i>sinfo</i>
<i>org_vec</i>	lapi_vec_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	<i>org_vec</i>
none	none	INTEGER(KIND = 4)	LAPI_Xfer parameter in FORTRAN (32-bit): <i>pad2</i>
<i>tgt_cntr</i>	lapi_long_t	INTEGER(KIND = 8)	<i>tgt_cntr</i>
<i>org_cntr</i>	lapi_cntr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	<i>org_cntr</i>
<i>cmpl_cntr</i>	lapi_cntr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	<i>cmpl_cntr</i>

lapi_amdgsp_t details

Table 26 on page 223 shows the correspondence among the fields of the C **lapi_amdgsp_t** structure and their datatypes, how they are used in **LAPI_Xfer**, and the equivalent FORTRAN datatypes. The **lapi_amdgsp_t** fields are listed in Table 26 on page 223 in the order that they occur in the **lapi_xfer_t** structure.

Table 26. The `lapi_amdgsp_t` fields

lapi_amdgsp_t field name (C)	lapi_amdgsp_t field type (C)	Equivalent FORTRAN datatype	LAPI_Xfer usage
<i>Xfer_type</i>	<code>lapi_xfer_type_t</code>	INTEGER(KIND = 4)	LAPI_DGSP_XFER
<i>flags</i>	<code>int</code>	INTEGER(KIND = 4)	This field allows users to specify directives or hints to LAPI. If you do not want to use any directives or hints, you must set this field to 0 . See “The <code>lapi_amdgsp_t</code> flags field” for more information.
<i>tgt</i>	<code>uint</code>	INTEGER(KIND = 4)	target task
none	none	INTEGER(KIND = 4)	<i>pad</i> (padding alignment for FORTRAN only)
<i>hdr_hdl</i>	<code>lapi_long_t</code>	INTEGER(KIND = 8)	header handler to invoke at target
<i>uhdr_len</i>	<code>uint</code>	INTEGER(KIND = 4)	user header length (multiple of processor’s doubleword size)
none	none	INTEGER(KIND = 4)	<i>pad2</i> (padding alignment for 64-bit FORTRAN only)
<i>uhdr</i>	<code>void *</code>	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	pointer to user header
<i>udata</i>	<code>void *</code>	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	pointer to user data
<i>udata_len</i>	<code>ulong</code>	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	user data length
<i>shdlr</i>	<code>scompl_hdlr_t *</code>	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	send completion handler (optional)
<i>sinfo</i>	<code>void *</code>	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	data pointer to pass to send completion handler (optional)
<i>tgt_cntr</i>	<code>lapi_long_t</code>	INTEGER(KIND = 8)	target counter (optional)
<i>org_cntr</i>	<code>lapi_cntr_t *</code>	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	origin counter (optional)
<i>cmpl_cntr</i>	<code>lapi_cntr_t *</code>	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	completion counter (optional)
<i>dgsp</i>	<code>lapi_dg_handle_t</code>	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	Handle of a registered DGSP
<i>status</i>	<code>lapi_status_t</code>	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	Status to return (future use)
none	none	INTEGER(KIND = 4)	<i>pad3</i> (padding alignment for 64-bit FORTRAN only)

When the origin data buffer is free to be modified, the send completion handler (*shdlr*) is called with the send completion data (*sinfo*), if *shdlr* is not a NULL pointer (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

See “Using `lapi_am_dgsp_t` for scatter-side DGSP processing” on page 224 for more information.

The `lapi_amdgsp_t flags` field: One or more flags can be set using the `|` (bitwise or) operator. User directives are always followed and could result in incorrect results

if used improperly. Appropriate hints might improve performance, but they may be ignored by LAPI. Inappropriate hints might degrade performance, but they will not cause incorrect results.

The following directive flags are defined:

USE_TGT_VEC_TYPE Instructs LAPI to use the vector type of the target vector (*tgt_vec*). In other words, *tgt_vec* is to be interpreted as type **lapi_vec_t**; otherwise, it is interpreted as type **lapi_lvec_t**. The **lapi_lvec_t** type uses **lapi_long_t**. The **lapi_vec_t** type uses **void *** or **long**. Incorrect results will occur if one type is used in place of the other.

BUFFER_BOTH_CONTIGUOUS

Instructs LAPI to treat all data to be transferred as contiguous, which can improve performance. If this flag is set when non-contiguous data is sent, data will likely be corrupted.

The following hint flags are defined:

LAPI_NOT_USE_BULK_XFER

Instructs LAPI not to use bulk transfer, independent of the current setting for the task.

LAPI_USE_BULK_XFER

Instructs LAPI to use bulk transfer, independent of the current setting for the task.

If neither of these hint flags is set, LAPI will use the behavior defined for the task. If both of these hint flags are set, **LAPI_NOT_USE_BULK_XFER** will take precedence.

These hints may or may not be honored by the communication library.

Using *lapi_am_dgsp_t* for scatter-side DGSP processing: Beginning with AIX 5.2, LAPI allows additional information to be returned from the header handler through the use of the **lapi_return_info_t** datatype. See “The enhanced header handler interface” on page 75 for full details. In the case of transfer type **lapi_am_dgsp_t**, this mechanism can be used to instruct LAPI to run a user DGSP to scatter data on the receive side.

To use this mechanism, pass a **lapi_return_info_t *** pointer back to LAPI through the *msg_len* member of the user header handler. The *dgsp_handle* member of the passed structure must point to a DGSP description that has been registered on the receive side. See **LAPI_Util** and “Using data gather/scatter programs (DGSPs)” on page 43 for details on building and registering DGSPs.

lapi_get_t details

Table 27 on page 225 shows the correspondence among the parameters of the **LAPI_Get** subroutine, the fields of the C **lapi_get_t** structure and their datatypes, and the equivalent FORTRAN datatypes. The **lapi_get_t** fields are listed in Table 27 on page 225 in the order that they occur in the **lapi_xfer_t** structure.

Table 27. LAPI_Get and lapi_get_t equivalents

lapi_get_t field name (C)	lapi_get_t field type (C)	Equivalent FORTRAN datatype	Equivalent LAPI_Get parameter
<i>Xfer_type</i>	lapi_xfer_type_t	INTEGER(KIND = 4)	implicit in C LAPI_Xfer value in FORTRAN: LAPI_GET_XFER
<i>flags</i>	int	INTEGER(KIND = 4)	none LAPI_Xfer parameter in FORTRAN: <i>flags</i>
<i>tgt</i>	uint	INTEGER(KIND = 4)	<i>tgt</i>
none	none	INTEGER(KIND = 4)	LAPI_Xfer parameter in FORTRAN: <i>pad</i>
<i>tgt_addr</i>	lapi_long_t	INTEGER(KIND = 8)	<i>tgt_addr</i>
<i>org_addr</i>	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	<i>org_addr</i>
<i>len</i>	ulong	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	<i>len</i>
<i>tgt_cntr</i>	lapi_long_t	INTEGER(KIND = 8)	<i>tgt_cntr</i>
<i>org_cntr</i>	lapi_cntr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	<i>org_cntr</i>
<i>chndlr</i>	compl_hndlr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	none LAPI_Xfer parameter in FORTRAN: <i>chndlr</i>
<i>cinfo</i>	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	none LAPI_Xfer parameter in FORTRAN: <i>cinfo</i>

When the origin data buffer has completely arrived, the pointer to the completion handler (*chndlr*) is called with the completion data (*cinfo*), if *chndlr* is not a NULL pointer (in C) or **LAPI_ADDR_NULL** (in FORTRAN). Otherwise, the behavior is identical to that of **LAPI_Get**.

lapi_getv_t details

Table 28 shows the correspondence among the parameters of the **LAPI_Getv** subroutine, the fields of the C **lapi_getv_t** structure and their datatypes, and the equivalent FORTRAN datatypes. The **lapi_getv_t** fields are listed in Table 27 in the order that they occur in the **lapi_xfer_t** structure.

Table 28. LAPI_Getv and lapi_getv_t equivalents

lapi_getv_t field name (C)	lapi_getv_t field type (C)	Equivalent FORTRAN datatype	Equivalent LAPI_Getv parameter
<i>Xfer_type</i>	lapi_xfer_type_t	INTEGER(KIND = 4)	implicit in C LAPI_Xfer value in FORTRAN: LAPI_GETV_XFER
<i>flags</i>	int	INTEGER(KIND = 4)	none LAPI_Xfer parameter in FORTRAN: <i>flags</i>
<i>tgt</i>	uint	INTEGER(KIND = 4)	<i>tgt</i>

LAPI_Xfer

Table 28. LAPI_Getv and lapi_getv_t equivalents (continued)

lapi_getv_t field name (C)	lapi_getv_t field type (C)	Equivalent FORTRAN datatype	Equivalent LAPI_Getv parameter
none	none	INTEGER(KIND = 4)	LAPI_Xfer parameter in FORTRAN (64-bit): <i>pad</i>
<i>org_vec</i>	lapi_vec_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	<i>org_vec</i>
<i>tgt_vec</i>	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	<i>tgt_vec</i>
none	none	INTEGER(KIND = 4)	LAPI_Xfer parameter in FORTRAN (32-bit): <i>pad</i>
<i>tgt_cntr</i>	lapi_long_t	INTEGER(KIND = 8)	<i>tgt_cntr</i>
<i>org_cntr</i>	lapi_cntr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	<i>org_cntr</i>
<i>chndlr</i>	compl_hndlr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	none LAPI_Xfer parameter in FORTRAN: <i>chndlr</i>
<i>cinfo</i>	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	none LAPI_Xfer parameter in FORTRAN: <i>cinfo</i>
none	none	INTEGER(KIND = 4)	LAPI_Xfer parameter in FORTRAN (32-bit): <i>pad2</i>

For this release, the **flags** field accepts **USE_TGT_VEC_TYPE** (see “The lapi_amdgrp_t flags field” on page 223) to indicate that **tgt_vec** is to be interpreted as type **lapi_vec_t**; otherwise, it is interpreted as type **lapi_lvec_t**. Note the corresponding field is **lapi_vec_t** in the **LAPI_Getv** call.

When the origin data buffer has completely arrived, the pointer to the completion handler (**chndlr**) is called with the completion data (**cinfo**) if **chndlr** is not a NULL pointer (in C) or **LAPI_ADDR_NULL** (in FORTRAN). Otherwise, the behavior is identical to that of **LAPI_Getv**.

lapi_put_t details

Table 29 shows the correspondence among the parameters of the **LAPI_Put** subroutine, the fields of the C **lapi_put_t** structure and their datatypes, and the equivalent FORTRAN datatypes. The **lapi_put_t** fields are listed in Table 29 in the order that they occur in the **lapi_xfer_t** structure.

Table 29. LAPI_Put and lapi_put_t equivalents

lapi_put_t field name (C)	lapi_put_t field type (C)	Equivalent FORTRAN datatype	Equivalent LAPI_Put parameter
<i>Xfer_type</i>	lapi_xfer_type_t	INTEGER(KIND = 4)	implicit in C LAPI_Xfer value in FORTRAN: LAPI_PUT_XFER
<i>flags</i>	int	INTEGER(KIND = 4)	none LAPI_Xfer parameter in FORTRAN: <i>flags</i>
<i>tgt</i>	uint	INTEGER(KIND = 4)	<i>tgt</i>

Table 29. LAPI_Put and lapi_put_t equivalents (continued)

lapi_put_t field name (C)	lapi_put_t field type (C)	Equivalent FORTRAN datatype	Equivalent LAPI_Put parameter
none	none	INTEGER(KIND = 4)	LAPI_Xfer parameter in FORTRAN: <i>pad</i>
<i>tgt_addr</i>	lapi_long_t	INTEGER(KIND = 8)	<i>tgt_addr</i>
<i>org_addr</i>	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	<i>org_addr</i>
<i>len</i>	ulong	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	<i>len</i>
<i>shdlr</i>	scompl_hndlr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	none LAPI_Xfer parameter in FORTRAN: <i>shdlr</i>
<i>sinfo</i>	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	none LAPI_Xfer parameter in FORTRAN: <i>sinfo</i>
<i>tgt_cntr</i>	lapi_long_t	INTEGER(KIND = 8)	<i>tgt_cntr</i>
<i>org_cntr</i>	lapi_cntr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	<i>org_cntr</i>
<i>cmpl_cntr</i>	lapi_cntr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	<i>cmpl_cntr</i>

When the origin data buffer is free to be used, the pointer to the send completion handler (*shdlr*) is called with the send completion data (*sinfo*), if *shdlr* is not a NULL pointer (in C) or **LAPI_ADDR_NULL** (in FORTRAN). Otherwise, the behavior is identical to that of **LAPI_Put**.

lapi_putv_t details

Table 30 shows the correspondence among the parameters of the **LAPI_Putv** subroutine, the fields of the C **lapi_putv_t** structure and their datatypes, and the equivalent FORTRAN datatypes. The **lapi_putv_t** fields are listed in Table 29 on page 226 in the order that they occur in the **lapi_xfer_t** structure.

Table 30. LAPI_Putv and lapi_putv_t equivalents

lapi_putv_t field name (C)	lapi_putv_t field type (C)	Equivalent FORTRAN datatype	Equivalent LAPI_Putv parameter
<i>Xfer_type</i>	lapi_xfer_type_t	INTEGER(KIND = 4)	implicit in C LAPI_Xfer value in FORTRAN: LAPI_PUT_XFER
<i>flags</i>	int	INTEGER(KIND = 4)	none LAPI_Xfer parameter in FORTRAN: <i>flags</i>
<i>tgt</i>	uint	INTEGER(KIND = 4)	<i>tgt</i>
none	none	INTEGER(KIND = 4)	LAPI_Xfer parameter in FORTRAN (64-bit): <i>pad</i>
<i>shdlr</i>	scompl_hndlr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	none LAPI_Xfer parameter in FORTRAN: <i>shdlr</i>

LAPI_Xfer

Table 30. LAPI_Putv and lapi_putv_t equivalents (continued)

lapi_putv_t field name (C)	lapi_putv_t field type (C)	Equivalent FORTRAN datatype	Equivalent LAPI_Putv parameter
<i>sinfo</i>	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	none LAPI_Xfer parameter in FORTRAN: <i>sinfo</i>
<i>org_vec</i>	lapi_vec_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	<i>org_vec</i>
<i>tgt_vec</i>	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	<i>tgt_vec</i>
none	none	INTEGER(KIND = 4)	LAPI_Xfer parameter in FORTRAN (32-bit): <i>pad</i>
<i>tgt_cntr</i>	lapi_long_t	INTEGER(KIND = 8)	<i>tgt_cntr</i>
<i>org_cntr</i>	lapi_cntr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	<i>org_cntr</i>
<i>cmpl_cntr</i>	lapi_cntr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	<i>cmpl_cntr</i>

For this release, the **flags** field accepts **USE_TGT_VEC_TYPE** (see “The lapi_amdgrp_t flags field” on page 223) to indicate that **tgt_vec** is to be interpreted as **lapi_vec_t**; otherwise, it is interpreted as a **lapi_lvec_t**. Note that the corresponding field is **lapi_vec_t** in the **LAPI_Putv** call.

When the origin data buffer is free to be modified, the pointer to the send completion handler (**shdlr**) is called with the send completion data (**sinfo**), if **shdlr** is not a NULL pointer (in C) or **LAPI_ADDR_NULL** (in FORTRAN). Otherwise, the behavior is identical to that of **LAPI_Putv**.

lapi_rmw_t details

Table 31 shows the correspondence among the parameters of the **LAPI_Rmw** subroutine, the fields of the C **lapi_rmw_t** structure and their datatypes, and the equivalent FORTRAN datatypes. The **lapi_rmw_t** fields are listed in Table 29 on page 226 in the order that they occur in the **lapi_xfer_t** structure.

Table 31. LAPI_Rmw and lapi_rmw_t equivalents

lapi_rmw_t field name (C)	lapi_rmw_t field type (C)	Equivalent FORTRAN datatype	Equivalent LAPI_Rmw parameter
<i>Xfer_type</i>	lapi_xfer_type_t	INTEGER(KIND = 4)	implicit in C LAPI_Xfer value in FORTRAN: LAPI_RMW_XFER
<i>op</i>	Rmw_ops_t	INTEGER(KIND = 4)	<i>op</i>
<i>tgt</i>	uint	INTEGER(KIND = 4)	<i>tgt</i>
<i>size</i>	uint	INTEGER(KIND = 4)	implicit in C LAPI_Xfer parameter in FORTRAN: <i>size</i> (must be 32 or 64)
<i>tgt_var</i>	lapi_long_t	INTEGER(KIND = 8)	<i>tgt_var</i>
<i>in_val</i>	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	<i>in_val</i>

Table 31. LAPI_Rmw and lapi_rmw_t equivalents (continued)

lapi_rmw_t field name (C)	lapi_rmw_t field type (C)	Equivalent FORTRAN datatype	Equivalent LAPI_Rmw parameter
<i>prev_tgt_val</i>	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	<i>prev_tgt_val</i>
<i>org_cntr</i>	lapi_cntr t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	<i>org_cntr</i>
<i>shdlr</i>	scompl_hndlr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	none LAPI_Xfer parameter in FORTRAN: <i>shdlr</i>
<i>sinfo</i>	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	none LAPI_Xfer parameter in FORTRAN: <i>shdlr</i>
none	none	INTEGER(KIND = 4)	LAPI_Xfer parameter in FORTRAN (32-bit): <i>pad</i>

When the origin data buffer is free to be used, the pointer to the send completion handler (*shdlr*) is called with the send completion data (*sinfo*), if *shdlr* is not a NULL pointer (in C) or **LAPI_ADDR_NULL** (in FORTRAN). The *size* value must be either **32** or **64**, indicating whether you want the *in_val* and *prev_tgt_val* fields to point to a 32-bit or 64-bit quantity, respectively. Otherwise, the behavior is identical to that of **LAPI_Rmw**.

Return values

LAPI_SUCCESS	Indicates that the function call completed successfully.
LAPI_ERR_DATA_LEN	Indicates that the value of <i>udata_len</i> or <i>len</i> is greater than the value of LAPI constant LAPI_MAX_MSG_SZ .
LAPI_ERR_DGSP	Indicates that the DGSP that was passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN) or is not a registered DGSP.
LAPI_ERR_DGSP_ATOM	Indicates that the DGSP has an <i>atom_size</i> that is less than 0 or greater than MAX_ATOM_SIZE .
LAPI_ERR_DGSP_BRANCH	Indicates that the DGSP attempted a branch that fell outside the code array.
LAPI_ERR_DGSP_CTL	Indicates that a DGSP control instruction was encountered in a non-valid context (such as a gather-side control or scatter-side control with an atom size of 0 at gather, for example).
LAPI_ERR_DGSP_OPC	Indicates that the DGSP op-code is not valid.
LAPI_ERR_DGSP_STACK	Indicates that the DGSP has greater GOSUB depth than the allocated stack supports. Stack allocation is specified by the <i>dgsp->depth</i> member.
LAPI_ERR_HDR_HNDLR_NULL	Indicates that the <i>hdr_hdl</i> passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_HNDL_INVALID	Indicates that the <i>hndl</i> passed in is not valid (not initialized or in terminated state).
LAPI_ERR_IN_VAL_NULL	Indicates that the <i>in_val</i> pointer is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_MEMORY_EXHAUSTED	LAPI is unable to obtain memory from the system.
LAPI_ERR_OP_SZ	Indicates that the lapi_rmw_t <i>size</i> field is not set to 32 or 64 .
LAPI_ERR_ORG_ADDR_NULL	Indicates either that the <i>udata</i> parameter passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN) and <i>udata_len</i> is greater than 0 , or that the <i>org_addr</i> passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN) and <i>len</i> is greater than 0 . Note: if <i>Xfer_type</i> = LAPI_DGSP_XFER , the case in which <i>udata</i> is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN) and <i>udata_len</i> is greater than 0 is valid, so an error is not returned.
LAPI_ERR_ORG_EXTENT	Indicates that the <i>org_vec</i> 's extent (stride * <i>num_vecs</i>) is greater than the value of LAPI constant LAPI_MAX_MSG_SZ .
LAPI_ERR_ORG_STRIDE	Indicates that the <i>org_vec</i> stride is less than block.
LAPI_ERR_ORG_VEC_ADDR	Indicates that the <i>org_vec->info[i]</i> is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but its length (<i>org_vec->len[i]</i>) is not 0 .
LAPI_ERR_ORG_VEC_LEN	Indicates that the sum of <i>org_vec->len</i> is greater than the value of LAPI constant LAPI_MAX_MSG_SZ .
LAPI_ERR_ORG_VEC_NULL	Indicates that the <i>org_vec</i> value is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_ORG_VEC_TYPE	Indicates that the <i>org_vec->vec_type</i> is not valid.
LAPI_ERR_RMW_OP	Indicates the op is not valid.
LAPI_ERR_STRIDE_ORG_VEC_ADDR_NULL	Indicates that the strided vector address <i>org_vec->info[0]</i> is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_STRIDE_TGT_VEC_ADDR_NULL	Indicates that the strided vector address <i>tgt_vec->info[0]</i> is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_TGT	Indicates that the <i>tgt</i> passed in is outside the range of tasks defined in the job.
LAPI_ERR_TGT_ADDR_NULL	Indicates that <i>ret_addr</i> is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_TGT_EXTENT	Indicates that <i>tgt_vec</i> 's extent (stride * <i>num_vecs</i>) is greater than the value of LAPI constant LAPI_MAX_MSG_SZ .
LAPI_ERR_TGT_PURGED	Indicates that the subroutine returned early because LAPI_Purge_totask() was called.
LAPI_ERR_TGT_STRIDE	Indicates that the <i>tgt_vec</i> stride is less than block.
LAPI_ERR_TGT_VAR_NULL	Indicates that the <i>tgt_var</i> address is NULL (in C) or that the value of <i>tgt_var</i> is LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_TGT_VEC_ADDR	Indicates that the <i>tgt_vec->info[i]</i> is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but its length (<i>tgt_vec->len[i]</i>) is not 0.
LAPI_ERR_TGT_VEC_LEN	Indicates that the sum of <i>tgt_vec->len</i> is greater than the value of LAPI constant LAPI_MAX_MSG_SZ .
LAPI_ERR_TGT_VEC_NULL	Indicates that <i>tgt_vec</i> is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_TGT_VEC_TYPE	Indicates that the <i>tgt_vec->vec_type</i> is not valid.
LAPI_ERR_UHDR_LEN	Indicates that the <i>uhdr_len</i> value passed in is greater than MAX_UHDR_SZ or is not a multiple of the processor's doubleword size.
LAPI_ERR_UHDR_NULL	Indicates that the <i>uhdr</i> passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but <i>uhdr_len</i> is not 0.
LAPI_ERR_VEC_LEN_DIFF	Indicates that <i>org_vec</i> and <i>tgt_vec</i> have different lengths (<i>len[]</i>).
LAPI_ERR_VEC_NUM_DIFF	Indicates that <i>org_vec</i> and <i>tgt_vec</i> have different <i>num_vecs</i> .
LAPI_ERR_VEC_TYPE_DIFF	Indicates that <i>org_vec</i> and <i>tgt_vec</i> have different vector types (<i>vec_type</i>).
LAPI_ERR_XFER_CMD	Indicates that the <i>Xfer_cmd</i> is not valid.

Location

`/usr/lib/liblapi_r.a`

C examples

1. To run the sample code shown in **LAPI_Get** using the **LAPI_Xfer** interface:

```
{
    lapi_xfer_t xfer_struct;

    /* initialize the table buffer for the data addresses */

    /* get remote data buffer addresses */
    LAPI_Address_init(hndl,(void *)data_buffer,data_buffer_list);
    .
    .
    .
    /* retrieve data_len bytes from address data_buffer_list[tgt] on */
}
```

LAPI_Xfer

```
/* task tgt. write the data starting at address data_buffer. */
/* tgt_cntr and org_cntr can be NULL. */

xfer_struct.Get.Xfer_type = LAPI_GET_XFER;
xfer_struct.Get.flags = 0;
xfer_struct.Get.tgt = tgt;
xfer_struct.Get.tgt_addr = data_buffer_list[tgt];
xfer_struct.Get.org_addr = data_buffer;
xfer_struct.Get.len = data_len;
xfer_struct.Get.tgt_cntr = tgt_cntr;
xfer_struct.Get.org_cntr = org_cntr;

LAPI_Xfer(hndl, &xfer_struct);
```

```
}
```

2. To implement the **LAPI_STRIDED_VECTOR** example from **LAPI_Amsendv** using the **LAPI_Xfer** interface:

```
{
    lapi_xfer_t  xfer_struct;           /* info for LAPI_Xfer call */
    lapi_vec_t   vec;                 /* data for data transfer */
    .
    .
    .
    vec->num_vecs = NUM_VECS;          /* NUM_VECS = number of vectors to transfer */
                                       /* must match that of the target vector */
    vec->vec_type = LAPI_GEN_STRIDED_XFER; /* same as target vector */

    vec->info[0] = buffer_address;      /* starting address for data copy */
    vec->info[1] = block_size;          /* bytes of data to copy */
    vec->info[2] = stride;              /* distance from copy block to copy block */
    /* data will be copied as follows: */
    /* block_size bytes will be copied from buffer_address */
    /* block_size bytes will be copied from buffer_address+stride */
    /* block_size bytes will be copied from buffer_address+(2*stride) */
    /* block_size bytes will be copied from buffer_address+(3*stride) */
    .
    .
    .
    /* block_size bytes will be copied from buffer_address+((NUM_VECS-1)*stride) */
    .
    .
    .
    xfer_struct.Amv.Xfer_type = LAPI_AMV_XFER;
    xfer_struct.Amv.flags     = 0;
    xfer_struct.Amv.tgt       = tgt;
    xfer_struct.Amv.hdr_hdl   = hdr_hdl_list[tgt];
    xfer_struct.Amv.uhdr_len  = uhdr_len; /* user header length */
    xfer_struct.Amv.uhdr      = uhdr;

    /* LAPI_AMV_XFER allows the use of a send completion handler */
    /* If non-null, the shdlr function is invoked at the point */
    /* the origin counter would increment. Note that both the */
    /* org_cntr and shdlr can be used. */
    /* The user's shdlr must be of type scompl_hdlr_t *. */
    /* scompl_hdlr_t is defined in /usr/include/lapi.h */
    xfer_struct.shdlr = shdlr;

    /* Use sinfo to pass user-defined data into the send */
    /* completion handler, if desired. */
    xfer_struct.sinfo = sinfo; /* send completion data */

    xfer_struct.org_vec = vec;
    xfer_struct.tgt_cntr = tgt_cntr;
```

```
xfer_struct.org_cntr = org_cntr;  
xfer_struct.cmpl_cntr = cmpl_cntr;  
  
LAPI_Xfer(hndl, &xfer_struct);  
.  
.  
.  
}
```

See **LAPI_Amsendv** for more information about the header handler definition.

Related information

Chapter 13, “Bulk transfer of messages,” on page 99

Subroutines: **LAPI_Amsend**, **LAPI_Amsendv**, **LAPI_Get**, **LAPI_Getv**, **LAPI_Put**,
LAPI_Putv, **LAPI_Rmw**

Chapter 19. Subroutines for standalone systems

Use the subroutines in this chapter on standalone systems.

LAPI_Nopoll_wait

Purpose

Waits for a counter update without polling.

Library

Availability Library (**liblapi_r.a**)

C syntax

```
#include <lapi.h>

void LAPI_Nopoll_wait(hndl, cntr_ptr, val, cur_cntr_val)
lapi_handle_t hndl;
lapi_cntr_t *cntr_ptr;
int val;
int *cur_cntr_val;
```

FORTRAN syntax

```
include 'lapif.h'

int LAPI_NOPOLL_WAIT(hndl, cntr, val, cur_cntr_val, ierror)
INTEGER hndl
TYPE (LAPI_CNTR_T) :: cntr
INTEGER val
INTEGER cur_cntr_val
INTEGER ierror
```

Parameters

INPUT

hndl Specifies the LAPI handle.

val Specifies the relative counter value (starting from 1) that the counter needs to reach before returning.

cur_cntr_val Specifies the integer value of the current counter. The value of The value of this parameter can be NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

INPUT/OUTPUT

cntr_ptr Points to the **lapi_cntr_t** structure in C.

cntr Is the **lapi_cntr_t** structure in FORTRAN.

OUTPUT

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: recovery (blocking)

This subroutine waits for a counter update without polling (that is, without explicitly invoking LAPI's internal communication dispatcher). This call may or may not check for message arrivals over the LAPI context *hndl*. The *cur_cntr_val* variable is set to the current counter value. Although it has higher latency than **LAPI_Waitcntr**, **LAPI_Nopoll_wait** frees up the processor for other uses.

Note: To use this subroutine, the *lib_vers* field in the **lapi_info_t** structure must be set to **L2_LIB** or **LAST_LIB**.

Restrictions

Use of this subroutine is *not* recommended on a system that is running Parallel Environment (PE).

Return values

LAPI_SUCCESS	Indicates that the function call completed successfully.
LAPI_ERR_CNTR_NULL	Indicates that the <i>cntr_ptr</i> pointer is NULL (in C) or that the value of <i>cntr</i> is LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_CNTR_VAL	Indicates that the <i>val</i> passed in is less than or equal to 0 .
LAPI_ERR_HNDL_INVALID	Indicates that the <i>hndl</i> passed in is not valid (not initialized or in terminated state).
LAPI_ERR_MULTIPLE_WAITERS	Indicates that more than one thread is waiting for the counter.
LAPI_ERR_TGT_PURGED	Indicates that the subroutine returned early because LAPI_Purge_totask() was called.

Location

`/usr/lib/liblapi_r.a`

Related information

Subroutines: **LAPI_Init**, **LAPI_Purge_totask**, **LAPI_Resume_totask**, **LAPI_Setcntr_wstatus**

LAPI_Purge_totask

Purpose

Allows a task to cancel messages to a given destination.

Library

Availability Library (**liblapi_r.a**)

C syntax

```
#include <lapi.h>

int LAPI_Purge_totask(hndl, dest)
lapi_handle_t hndl;
uint dest;
```

FORTRAN syntax

```
include 'lapif.h'

int LAPI_PURGE_TOTASK(hndl, dest, ierror)
INTEGER hndl
INTEGER dest
INTEGER ierror
```

Parameters

INPUT

hndl Specifies the LAPI handle.

dest Specifies the destination instance ID to which pending messages need to be cancelled.

OUTPUT

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: recovery

This subroutine cancels messages and resets the state corresponding to messages in flight or submitted to be sent to a particular target task. This is an entirely local operation. For correct behavior a similar invocation is expected on the destination (if it exists). This function cleans up all the state associated with pending messages to the indicated target task. It is assumed that before the indicated task starts communicating with this task again, it also purges this instance (or that it was terminated and initialized again). It will also wake up all threads that are in **LAPI_Nopoll_wait** depending on how the arguments are passed to the **LAPI_Nopoll_wait** function. The behavior of **LAPI_Purge_totask** is undefined if LAPI collective functions are used.

Note: This subroutine should not be used when the parallel application is running in a PE/LoadLeveler environment.

LAPI_Purge_totask is normally used after connectivity has been lost between two tasks. If connectivity is restored, the tasks can be restored for LAPI communication by calling **LAPI_Resume_totask**.

Restrictions

Use of this subroutine is *not* recommended on a system that is running Parallel Environment (PE).

Return values

LAPI_SUCCESS	Indicates that the function call completed successfully.
LAPI_ERR_HNDL_INVALID	Indicates that the <i>hndl</i> passed in is not valid (not initialized or in terminated state).
LAPI_ERR_TGT	Indicates that <i>dest</i> is outside the range of tasks defined in the job.

Location

`/usr/lib/liblapi_r.a`

Related information

Subroutines: **LAPI_Init**, **LAPI_Nopoll_wait**, **LAPI_Resume_totask**, **LAPI_Term**

LAPI_Resume_totask

Purpose

Re-enables the sending of messages to the task.

Library

Availability Library (**liblapi_r.a**)

C syntax

```
#include <lapi.h>

int LAPI_Resume_totask(hndl, dest)
lapi_handle_t hndl;
uint dest;
```

FORTRAN syntax

```
include 'lapif.h'

int LAPI_RESUME_TOTASK(hndl, dest, ierror)
INTEGER hndl
INTEGER dest
INTEGER ierror
```

Parameters

INPUT

hndl Specifies the LAPI handle.

dest Specifies the destination instance ID with which to resume communication.

OUTPUT

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: recovery

This subroutine is used in conjunction with **LAPI_Purge_totask**. It enables LAPI communication to be reestablished for a task that had previously been purged. The purged task must either restart LAPI or execute a **LAPI_Purge_totask/LAPI_Resume_totask** sequence for this task.

Restrictions

Use of this subroutine is *not* recommended on a system that is running Parallel Environment (PE).

Return values

LAPI_SUCCESS	Indicates that the function call completed successfully.
LAPI_ERR_HNDL_INVALID	Indicates that the <i>hndl</i> passed in is not valid (not initialized or in terminated state).

LAPI_ERR_TGT

Indicates that the *tgt* passed in is outside the range of tasks defined in the job.

Location

/usr/lib/liblapi_r.a

Related information

Subroutines: LAPI_Init, LAPI_Nopoll_wait, LAPI_Purge_totask, LAPI_Term

LAPI_Setcntr_wstatus

Purpose

Used to set a counter to a specified value and to set the associated destination list array and destination status array to the counter.

Library

Availability Library (**liblapi_r.a**)

C syntax

```
#include <lapi.h>

int LAPI_Setcntr_wstatus(hdl, cntr, num_dest, dest_list, dest_status)

lapi_handle_t hdl;
lapi_cntr_t *cntr;
int num_dest;
uint *dest_list;
int *dest_status;
```

FORTRAN syntax

```
include 'lapif.h'

LAPI_SETCNTR_WSTATUS(hdl, cntr, num_dest, dest_list, dest_status, ierror)
INTEGER hdl
TYPE (LAPI_CNTR_T) :: cntr
INTEGER num_dest
INTEGER dest_list(*)
INTEGER dest_status
INTEGER ierror
```

Parameters

INPUT

hdl Specifies the LAPI handle.

num_dest Specifies the number of tasks in the destination list.

dest_list Specifies an array of destinations waiting for this counter update. If the value of this parameter is NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN), no status is returned to the user.

INPUT/OUTPUT

cntr Specifies the address of the counter to be set (in C) or the counter structure (in FORTRAN). The value of this parameter cannot be NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

OUTPUT

dest_status Specifies an array of status that corresponds to *dest_list*. The value of this parameter can be NULL (in C) or **LAPI_ADDR_NULL** (in FORTRAN).

ierror Specifies a FORTRAN return code. This is always the last parameter.

Description

Type of call: recovery

This subroutine sets *cntr* to 0. Use **LAPI_Setcntr_wstatus** to set the associated destination list array (*dest_list*) and destination status array (*dest_status*) to the counter. Use a corresponding **LAPI_Nopoll_wait** call to access these arrays. These arrays record the status of a task from where the thread calling **LAPI_Nopoll_wait()** is waiting for a response.

The return values for *dest_status* are:

LAPI_MSG_INITIAL	The task is purged or is not received.
LAPI_MSG_RECVD	The task is received.
LAPI_MSG_PURGED	The task is purged, but not received.
LAPI_MSG_PURGED_RCVD	The task is received and then purged.
LAPI_MSG_INVALID	Not valid; the task is already purged.

Note: To use this subroutine, the *lib_vers* field in the **lapi_info_t** structure must be set to **L2_LIB** or **LAST_LIB**.

Restrictions

Use of this subroutine is *not* recommended on a system that is running Parallel Environment (PE).

Return values

LAPI_SUCCESS	Indicates that the function call completed successfully.
LAPI_ERR_CNTR_NULL	Indicates that the <i>cntr</i> value passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).
LAPI_ERR_HNDL_INVALID	Indicates that the <i>hndl</i> passed in is not valid (not initialized or in terminated state).
LAPI_ERR_RET_PTR_NULL	Indicates that the value of <i>dest_status</i> is NULL in C (or LAPI_ADDR_NULL in FORTRAN), but the value of <i>dest_list</i> is not NULL in C (or LAPI_ADDR_NULL in FORTRAN).

Location

/usr/lib/liblapi_r.a

Related information

Subroutines: **LAPI_Getcntr**, **LAPI_Nopoll_wait**, **LAPI_Purge_totask**, **LAPI_Setcntr**

LAPI_Setcntr_wstatus

Chapter 20. LAPI sample programs

This chapter describes LAPI's sample files. The files are structured to provide an exhaustive look at basic LAPI operations. The intention is to show users the basic building blocks of the LAPI API, facilitating their use for solving more complex problems.

The sample source code resides in several subdirectories. Each subdirectory illustrates some aspect of the LAPI API, forming an "example group". Each example group is a self-contained unit, with a separate README file and Makefile. Since the samples install into a common system directory, users should copy the sample tree into personal file space for use. That way, the source files can be changed as needed and then built.

There are two types of example groups:

- Normal LAPI operations

Showing Initialization and setup (init group), LAPI API communication calls (lapi_api), an example of two-way LAPI communication (basic), illustration of LAPI's DGSP interface (dgsp), illustration of LAPI's vector interface (vector) and illustration of LAPI's xfer interface (xfer)

- Special cases

Showing operation of LAPI without IBM's Parallel Environment for AIX (PE) and LoadLeveler licensed programs (standalone) and LAPI support for mixed 32/64 bit jobs (interop).

The structure of each normal example group is the same. Each directory contains a README, Makefile, and one or more source files. Complete directions for building and executing the examples can be found in the README file in each directory. In each case, building simply requires execution of the make command. Several source files are created.

Sample program directory structure

The sample files install into system directory `/opt/rsct/lapi/samples`. It is assumed that a non-root user wishing to work with the samples has copied them into their own file space. Throughout the remainder of this discussion, references to samples directories are assumed to be relative to the top directory (the directory from which the user is working). So for example, if the user has copied the contents of `/opt/rsct/lapi/samples` into directory `/u/fred/samples`, the use of "init/init.c" in the discussion will refer to `init.c` that resides in the `init` subdirectory of `/u/fred/samples`.

Details of the subdirectories are given here.

init

Contains examples of initialization, setup and termination functions for LAPI.

- The `Init.c` sample illustrates basic initialization and termination. It also demonstrates support for multiple initialization and termination of LAPI handles. At the first initialization, a user error handler is registered.
- `Addr.c` demonstrates LAPI's address manipulation functions, including `LAPI_Address_Init`, `LAPI_Address_init64`, `LAPI_Addr_set` and `LAPI_Addr_get`. Each task declares two local variables then does a collective exchange of their addresses as a `void *` (using `LAPI_Address_init`) and as a `lapi_long_t` (using `LAPI_Address_init64`).

Each task then does a LAPI_Addr_set followed by a LAPI_Addr_get to demonstrate the use of indexed address tables.

- Qenv_senv.c illustrates the use of LAPI's runtime query facility. All parameters that can be queried from a LAPI instance are shown in the source code, as well as those that can be set. For each parameter, the default value is printed. For settable parameters, the value is changed and the new value printed to verify the change.

lapi_api

Contains an example demonstrating each LAPI communication API call. Both a C file and a FORTRAN file are included for each API call.

In each example, a buddy system is used for pairing tasks. Each buddy pair has one task that drives communication with the other task in the pair (his buddy). A single communication is made between the driving task and his buddy.

In all cases, a basic setup is done, followed by the API call. Finally, normal LAPI cleanup operations are done. Synchronization fences are used throughout the code.

Each source code file in the lapi_api directory is named for the demonstrated API call. The following source files are provided:

- Am.c, Amf.F - LAPI_Amsend
- Amv.c, Amvf.F - LAPI_Amsendv
- Get.c, Getf.F - LAPI_Get
- Getv.c, Getvf.F - LAPI_Getv
- Put.c, Putf.F - LAPI_Put
- Putv.c, Putvf.F - LAPI_Putv
- Rmw.c, Rmwf.F - LAPI_Rmw
- Rmw64.c, Rmw64f.F - LAPI_Rmw64

For purposes of deeper illustration, the source code of the Am.c is included below with line numbers and complete annotation. The Am samples demonstrate user-defined header and completion handlers. See the description of the basic/accumulate_and_return.Am sample as well as the complete annotated description below for more details on execution sequence of header and completion handlers.

basic

Shows examples of different approaches to the same communication using different LAPI API calls. The examples illustrate two-way communication. The driver sends an array of ints to its buddy. Its buddy accumulates the data with some local data, then sends the data back to the original sender. Different approaches to synchronization are illustrated, depending on the nature of the API call being used. Each source file is named accumulate_and_return.API.c, where API is one of Am, Put or Xfer.

There are three examples of this communication, one using each of the API calls implied by the name:

accumulate_and_return.Am.c

accumulate_and_return.Am.c does a LAPI_Amsend call for the original data transfer. This example also shows how to use user header data as part of the transfer, as well as the use of a completion handler parameter to pass data between the header and completion handlers. Note that user header data will be available in the first packet, and is thus available to the header handler. Using the combined facility of a user header and a completion

handler parameter ensures the delivery of data from the message sender all the way through to the completion handler on the receiver.

This example defines a data type to use for the completion handler parameter. The parameter is passed as a void *. Then it is cast to the newly defined type for use in the completion handler.

Execution is as follows:

1. Task 0 initializes a data buffer and sends it to Task 1 with a LAPI_Amsend, then waits on its own flag (not a counter managed by LAPI).
2. The arrival of the first packet on Task 1 causes LAPI to invoke Task 1's header handler (defined in the sample source code). The header handler sets up the data structure to pass to the completion handler, sets the completion handler pointer and completion handler parameter pointer, then returns an address in Task 1's address space.
LAPI uses this address as the base address for writing the transferred data. Note that a header handler definition is required, since it is the means by which LAPI gets the base address for writing. If a completion handler is to be used, the header handler is also where the completion handler pointer is set. Note that a completion handler is optional. If one is not used, the completion handler pointer should be set to NULL to ensure that LAPI does not interpret an uninitialized pointer as the address of a function.
3. Once all data has been transferred, LAPI invokes the completion handler (also defined in the sample source code). The completion handler in this case performs the data computation, using the values passed through the completion handler parameter, and then completes a LAPI_Amsend call back to the original sender. This call invokes the same sequence of steps as above, but on the opposite task.
4. Task 0's header handler is invoked upon the arrival of the first data packet. The completion handler pointer and parameter pointer are set and a buffer address in Task 0's address space is returned.
5. Upon completion of the data transfer, Task 0's completion handler is run. The final step of the completion handler is to increment the flag on which Task 0's main execution path has been waiting since immediately after the original LAPI_Amsend call. This frees Task 0 to drop into the final fence and then cleanup and terminate.

Figure 21 on page 248 illustrates the sequence of execution in program sample accumulate_and_return.Am.

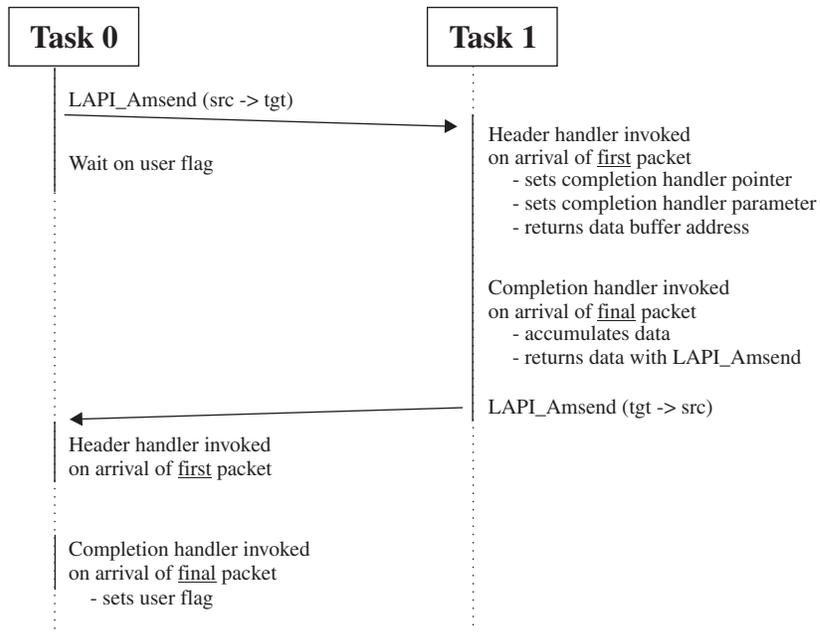


Figure 21. Execution sequence of the `accumulate_and_return.Am` sample

accumulate_and_return.Put.c

`accumulate_and_return.Put.c` uses `LAPI_Put` to do both transfers. Target counters are used for synchronization on both sides. The execution sequence is represented in Figure 21. Task 0 (or any driver task) begins by issuing a `LAPI_Put` to its buddy (Task 1 in the diagram) then immediately goes into a wait on Target Counter 0. Task 1, meanwhile has initiated a wait on Target Counter 1. Once the `LAPI_Put` from Task 0 completes on Task 1, Target Counter 1 will increment, freeing Task 1 from its wait.

Task 1 then does the accumulate and returns the data to Task 0 with another `LAPI_Put`. The completion of this `LAPI_Put` causes Target Counter 0 to increment, releasing Task 0 from its wait. Both tasks then sync in a final fence before implementing cleanup operations and terminating.

Figure 22 illustrates the sequence of execution in program sample `accumulate_and_return.Put`.

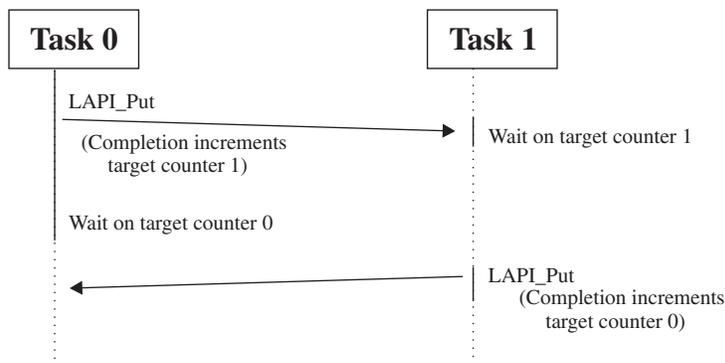


Figure 22. Execution sequence of the `accumulate_and_return.Put` sample

accumulate_and_return.Xfer.c

`accumulate_and_return.Xfer.c` executes the same sequence as

accumulate_and_return.Put except that the LAPI_Xfer interface is used for the first put call. For additional examples of using the LAPI_Xfer interface, see the sample programs in the xfer directory.

vector

Contains illustrations of LAPI's vector interface. accumulate_and_return.Amv.c uses LAPI_Amsendv to perform the same set of tasks as the samples in the basic directory. matrix.c demonstrates a two-dimensional data transfer using LAPI vectors, and strided.c illustrates a strided vector transfer.

dgsp

Provides samples illustrating LAPI's new DGSP interface. Dgsp_simple.c builds a DGSP then executes a data transfer based on it. The data is unpacked sequentially on the receive side.

xfer

Illustrates LAPI's Xfer interface. Am_xfer.c illustrates the use of LAPI_Xfer to do an equivalent communication to a LAPI_Amsend call. Put_Xfer.c illustrates the equivalent of a LAPI_Put call.

The remaining directories focus on special cases of LAPI operation:

interop

Demonstrates the use of the LAPI_Xfer interface for interoperability between 32-bit and 64-bit LAPI applications. Using LAPI_Address_init64 and LAPI_Xfer calls allows remote addresses to be exchanged as 64-bit values in either case. The build instructions are more complicated in this example. A 32-bit and 64-bit executable must be built and run together to truly demonstrate this interoperability. Scripts are provided for convenience in building. See the interop/README.LAPI.INTEROP file for details.

standalone

Illustrates methods for running LAPI in standalone mode, that is, without the use of IBM's Parallel Environment for AIX (PE) or LoadLeveler (LL) products. Setup for standalone mode is slightly different for User Space (US) mode than it is for UDP mode. So a separate example "subgroup" is created for each case.

- Standalone UDP initialization involves providing a means for distributing task address and port information that would normally be distributed by PE. LAPI supports two methods for distributing this information, the use of a user handler, and the use of a user list. Building and execution of files is different than in the usual LAPI environment. Binaries must be created using a non-parallel compiler. Also, certain environment variables that are normally set by PE must be set by hand for each task before executing. Finally, each task must be executed by hand. See the standalone/udp/README.LAPI.STANDALONE.UDP file for details on building and execution for standalone UDP operation.
- Standalone US initialization involves the setting of certain environment variables as well as the execution of each task by hand. The user must also reserve an adapter window for each task and load the network tables on each node. The user must then use the adapter and window information to set the MP_LAPI_NETWORK environment variable before execution of each task. A set of helper applications for loading network tables and grabbing network information are provided in the standalone/us/ntbl directory. Note that these applications require a system administrator for building and installation. Full details on building and executing for standalone user space applications can be found in the standalone/us/README.LAPI.STANDALONE.US file.

Using the LAPI sample programs

The LAPI sample programs are arranged to provide a threaded tutorial for new users, as well as a reference for both experienced and new LAPI users. Users already familiar with LAPI can dive right into the samples anywhere they wish, exploring directories such as `dgsp`, `vector` or `xfer` to learn about specific LAPI constructs or techniques. The sample programs in the `init` subdirectory may be a useful reference for experienced users, especially the `Qenv_senv` sample, which illustrates all runtime parameters that can be queried using `LAPI_Qenv` and set using `LAPI_Senv`. The sample also prints the default values of each of the parameters. The `lapi_api` directory is also a handy reference tool, because it provides a sample for each LAPI communication API call.

IBM suggests that new users:

1. Start with the `init` subdirectory to get a feel for the basics of writing, building and executing LAPI applications, as well as the parameters and API calls that come into play during setup. The execution model is single task for these examples, making them easier to understand for starters.
2. Go through some of the samples in the `lapi_api` directory to understand LAPI's communication APIs.
3. View the remaining directories in any order to demonstrate specific aspects of the LAPI API.

The `interop` and `standalone` directories are special cases for users who have 32-bit and 64-bit LAPI applications that need to communicate (see the `interop` directory) or for users that are running without IBM's Parallel Environment for AIX (PE) or LoadLeveler (LL) products (see the `standalone` directory).

Summary of constructs and techniques for LAPI programming

Table 32 provides a quick reference of what LAPI constructs and techniques that you can learn from the sample programs.

Table 32. Constructs and techniques for LAPI programming

For examples of this LAPI programming construct or technique:	See this sample:
Origin counter	Vector: <code>strided.c</code>
Target counter	Lapi_api: <code>Put.c</code> , <code>Putf.F</code> , <code>Putv.c</code> , <code>Putvf.F</code> Basic: <code>accumulate_and_return.Put.c</code> , <code>accumulate_and_return.Xfer.c</code> Xfer: <code>Put_xfer.c</code> <code>Put_xferf.F</code>
Completion counter	Lapi_api: <code>Am.c</code> , <code>Amv.c</code> , <code>Putv.c</code> , <code>Putvf.F</code> Dgsp: <code>Dgsp_simple.c</code> Vector: <code>accumulate_and_return.Amv.c</code> , <code>matrix.c</code> , <code>strided.c</code> Xfer: <code>Am_xfer.c</code> , <code>Put_xfer.c</code>
Default values of LAPI runtime parameters	Init: <code>Qenv_senv.c</code>
Setting of LAPI runtime parameters	Init: <code>Qenv_senv.c</code>
User Error handler	Init: <code>Init.c</code>
Initialization of a LAPI handle after termination	Init: <code>Init.c</code>
Address Manipulation in LAPI	Init: <code>Addr.c</code>

Table 32. Constructs and techniques for LAPI programming (continued)

For examples of this LAPI programming construct or technique:	See this sample:
User header data	Basic: accumulate_and_return.Am.c vector: accumulate_and_return.Amv.c
Completion handle	Lapi_api: Am.c, Amf.F, Amv.c Basic: accumulate_and_return.Am.c Dgsp: Dgsp_simple.c Vector: accumulate_and_return.Amv.c, matrix.c, strided.c Xfer: Am_xfer.c
Completion Handler parameter	Basic: accumulate_and_return.Am.c Vector: accumulate_and_return.Amv.c
LAPI communication API calls from within a completion handler	Basic: accumulate_and_return.Am.c Vector: accumulate_and_return.Amv.c

Appendix A. Product-related information

The low-level application programming interface (LAPI) is a part of:

- the Reliable Scalable Cluster Technology (RSCT) component of AIX 5L Versions 5.2 and 5.3.

For more information about RSCT, see the *RSCT: Administration Guide*.

- the IBM Parallel System Support Programs for AIX (PSSP) licensed program.

For more information about PSSP LAPI, see the *PSSP Read This First* document.

For AIX Versions 5.2 and 5.3, the current version of LAPI is shipped with the Reliable Scalable Cluster Technology (RSCT) component of the AIX operating system and with PSSP 3.5. For more information about AIX, go to:

<http://www.ibm.com/servers/aix/library>.

You can use LAPI "standalone" or with the IBM Parallel Environment for AIX (PE) licensed program. Specifically, you can use the parallel operating environment (POE) component of PE to compile and run LAPI parallel programs. POE also provides support for parallel programs to use the Message Passing Interface (MPI) component of PE with or without LAPI. Though the use of PE is optional and requires additional steps, it is recommended that you use PE with LAPI. Unless otherwise noted, this book discusses the use of LAPI in conjunction with PE. For information about using LAPI without PE, see Chapter 16, "Using LAPI on a standalone system," on page 115.

See *Parallel Environment for AIX 5L: Installation* for information about installing PE and POE, *Parallel Environment for AIX 5L: Operation and Use, Volume 1* for information about using POE, and *Parallel Environment for AIX 5L: MPI Programming Guide* for information about using MPI.

RSCT version

This edition applies to:

- For AIX 5.3 —
 - RSCT version 2.4.1.0 (or later) for LAPI (**rsct.lapi.rte** fileset) and NAM (**rsct.lapi.nam** fileset)
 - RSCT version 2.4.0.0 (or later) for group services (part of the **rsct.basic.rte** fileset)
- For AIX 5.2 —
 - RSCT version 2.3.3.0 (or later) for LAPI (**rsct.lapi.rte** fileset) and NAM (**rsct.lapi.nam** fileset)
 - RSCT version 2.3.3.2 (or later) for group services (part of the **rsct.basic.rte** fileset)

As an example, to find out which version of RSCT is running on a particular AIX node, enter:

```
lslpp -L rsct.basic.rte
```

ISO 9000

ISO 9000 registered quality systems were used in the development and manufacturing of this product.

Product-related feedback

To contact the IBM cluster development organization, send your comments by e-mail to:

cluster@us.ibm.com

Appendix B. LAPI execution models

LAPI provides two different execution models:

1. the Internet Protocol / user space (IP/US) execution model
2. the shared memory execution model

The IP/US execution model

LAPI provides a thread-safe environment and supports an execution model that allows for execution concurrency between LAPI instances and LAPI client applications.

Using the setup function (**LAPI_Init**), a user process establishes a LAPI context. Within a LAPI context, the LAPI library is thread-safe and multiple threads can make LAPI calls within the same context. The different calls can run concurrently with each other and with the user threads. In reality, however, execution concurrence among these calls is limited by the locking that is required with LAPI to maintain integrity of its internal data structures and the need to share a single underlying communication channel.

As with any multi-threaded application, coherence of user data is the responsibility of the user. Specifically, if two or more LAPI calls from different threads can run concurrently and if they specify overlapping user buffer areas, the result is undefined. It is the responsibility of the user to coordinate the required synchronization between threads that operate on overlapping buffers.

The user application thread, as well as the completion handlers, cannot hold mutual exclusion resources before making LAPI calls; if they do, it is possible to run into deadlock situations.

Because user-defined handlers can be called concurrently from multiple threads, it is the user's responsibility to make them thread-safe.

Figure 23 on page 256 shows the interaction among an application thread, an interrupt/timer thread, and a completion thread.

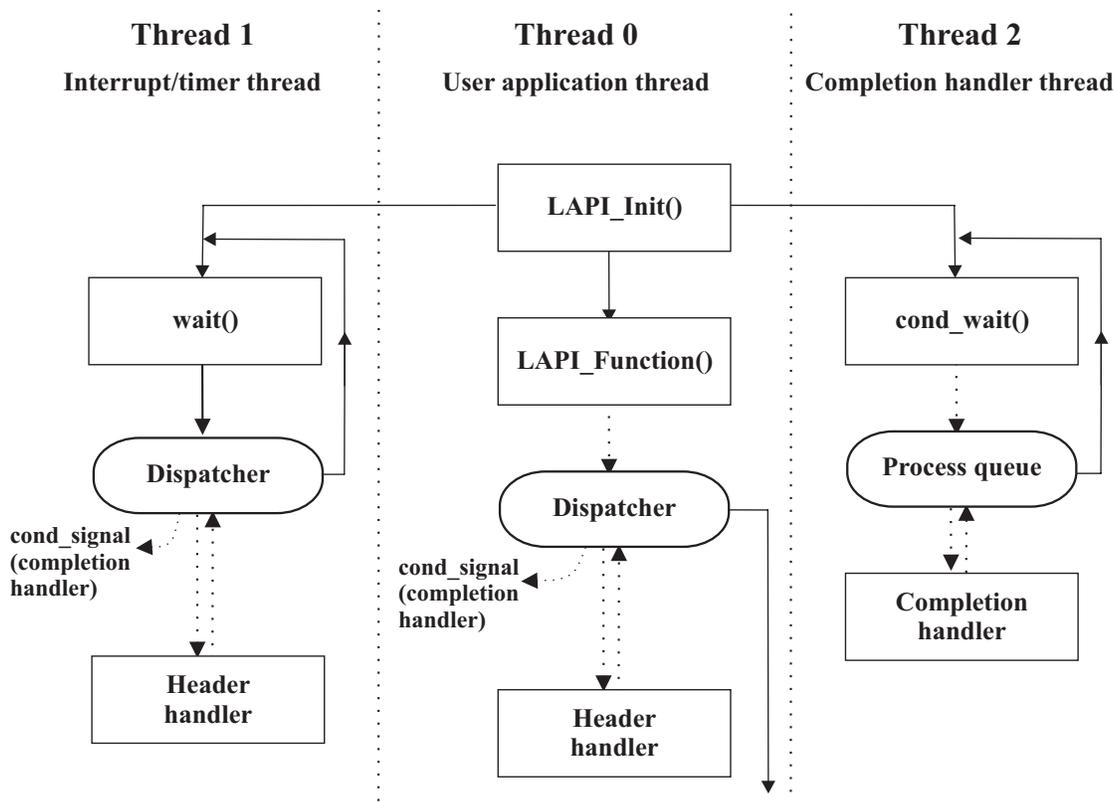


Figure 23. A LAPI thread model

Whenever possible, thread 0 (the application thread) and thread 1 (the interrupt/timer thread) try to call the LAPI dispatcher. This way, progress on incoming and outgoing messages can be made while minimizing additional overhead. Most LAPI calls that are made by the application thread also result in the LAPI dispatcher being run automatically. The interrupt/timer thread waits in the kernel for the occurrence of a notification event. When an event occurs, the kernel “wakes up” the waiting thread. As shown in Figure 23, after the interrupt/timer thread returns from waiting in the kernel, it calls the LAPI dispatcher.

The LAPI dispatcher is the central control point that orchestrates the invocation of the functions and threads needed to process outstanding incoming and outgoing LAPI messages. The LAPI dispatcher can run from the application thread, the interrupt/timer thread, or the completion thread. To maintain integrity, locking is used to ensure that only one instance of the dispatcher runs at a time. On incoming messages, the LAPI dispatcher manages the reassembly of data from different packets — which might arrive out-of-order — into the specified buffer, and then calls the completion handler if necessary.

LAPI_Init creates thread 2 to run completion handlers that are associated with active messages. User-written completion handlers can make LAPI function calls that in turn call the LAPI dispatcher. The completion handler thread processes work from the completion handler queue. When the queue is empty, the thread waits using a **pthread_cond_wait()**. If an active message (**LAPI_Amsend**) includes a completion handler, the dispatcher queues a request on the completion queue after the whole message has arrived and has been reassembled in the specified buffer. The dispatcher then sends a **pthread_cond_signal** to the completion handler thread. If this thread was in a wait state, it will begin processing the completion handler queue; otherwise, if it was not waiting, the thread signal is ignored.

LAPI handlers are not guaranteed to run one at a time. LAPI calls can run concurrently within the origin, the target, or both. The restriction about not holding on to mutual exclusion resources when making LAPI calls still applies.

This discussion of a thread-safe environment and execution concurrence within the LAPI library applies to both polling mode and interrupt mode. In polling mode, any calls to the communication library try to make progress on the context specified in the call. LAPI includes the **LAPI_Probe** subroutine, which lets applications explicitly check for and handle incoming messages.

The execution model of the handlers consists of the following events:

Event	Action
<i>Message arrival</i>	Copies the message from the network into the appropriate data access memory space.
<i>Interrupt or poll</i>	Causes an interrupt if required, based on the mode.
<i>Dispatcher start</i>	Calls LAPI's internal communication dispatcher.
<i>New message packet</i>	Checks the LAPI header and determines (by checking the receive state message reassembly table) whether the packet is part of a pending message or whether it is a new message. For the first packet of a new message, calls the header handler function.
<i>Return from header handler</i>	If the message is contained in more than one packet, the LAPI dispatcher logs that there is a pending message, saves the completion handler address, and saves the user's buffer address to be used during the message reassembly of pending message packets.
<i>Pending message packet</i>	Copies the message to the appropriate portion of the user buffer specified through the header handler. If the packet completes the message, the dispatcher queues the completion handler; otherwise, the dispatcher returns to check for message arrivals.
<i>Return from completion handler</i>	When the completion handler is run, it updates the appropriate target counter before continuing.

The shared memory execution model

When tasks are on the same node, it is more efficient for communication protocol clients to use a shared memory protocol, as opposed to using the switch adapter or "double copy" mode (communication through a shared segment). Using a shared memory protocol reduces switch congestion and optimizes performance. This is where LAPI's shared memory execution model comes in handy.

Figure 24 on page 258 and Figure 25 on page 258 show the different approaches for running **LAPI_Put** when task 0 and task 1 are on the same node. The user interface is identical in the LAPI communication paths for shared memory and for nonshared memory.

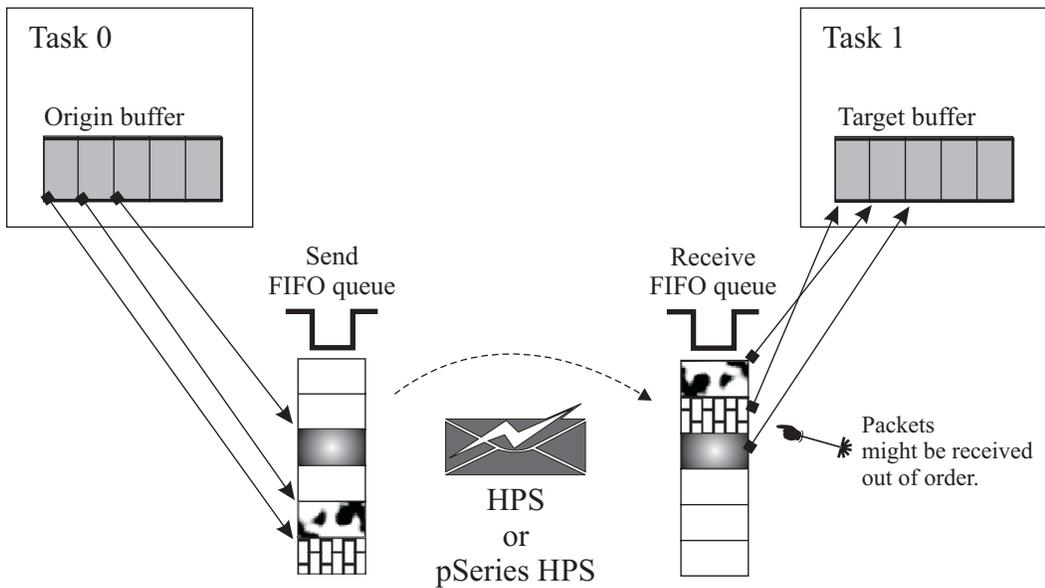


Figure 24. LAPI_Put without shared memory

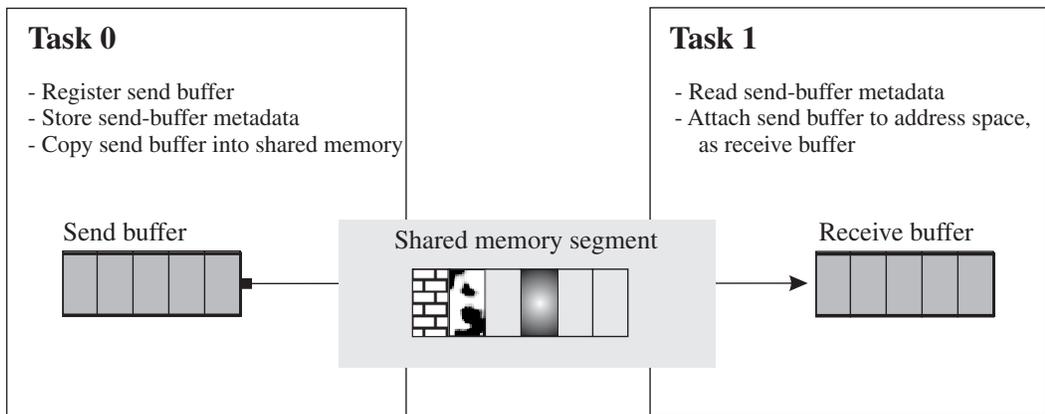


Figure 25. LAPI_Put with shared memory

Cross memory kernel extension

During AIX initialization, a kernel extension that supports LAPI's shared memory execution model is loaded. This kernel extension allows one task to export a portion of its address space to another task of the parallel application. The exported portion of the address space is attached to the address space of the companion task and the data transfer is done by a simple copy. A separate shared-memory region is required for exchanging metadata that describes the exported regions and allows for handshaking between the communicating tasks of the parallel application. This shared-memory region can also be used for transferring small messages or certain types of noncontiguous messages.

As part of **rsct.lapi** installation, the configuration method for the kernel extension is added to the **Config_Rules** object data manager (ODM) database, to be run in phase 3 of the system initialization. This ensures that the kernel extension is available for any **rsct.lapi** installation, whether a switch adapter is present or not. See Chapter 4, "Installing RSCT LAPI," on page 25 for more information.

LAPI shared memory: functional flow

Each task that communicates with other tasks using shared memory has a message queue of command structures, called *slots*. Shared memory task *n* processes commands off of the head of the task *n* message queue. Tasks other than task *n* update the tail of the task *n* message queue. Each slot contains a fixed-size data area.

There are two ways to transfer data using shared memory:

1. For small messages, the message is copied by the message origin into one or more slots at the tail of the target task's message queue. The message is subsequently copied out of these slots by the target task and copied into the user buffers at the target task. This message transfer mode is called *slot mode* and this flow is referred to as the *slot flow*.
2. For large messages, the virtual memory region that contains the message data at the source task is exported using a kernel extension. The kernel extension returns identifiers to the source task. These identifiers can be used to attach to the source task's virtual memory region. The source task transfers these identifiers to the target task using a slot at the tail of the target task's message queue. When this slot is processed on the message target, the identifiers are used to attach the source message region to the target task's address space, and to thereafter copy the data directly from the source task message buffers into user buffers at the target task. This message transfer mode is called *attach mode* and this flow is referred to as the *attach flow*.

LAPI shared memory: requirements and restrictions

Requirements and restrictions for using LAPI shared memory follow:

- To use shared memory, the **LAPI_USE_SHM** environment variable must be set to **yes** or **only**. It is not case-sensitive.
If **LAPI_USE_SHM** is set to **only**, LAPI only uses the shared memory mechanism. If all tasks are not on the same node or if shared memory setup is not successful, LAPI returns an error message.
If **LAPI_USE_SHM** is set to **yes**, the shared memory path is used on tasks within a node, unless initialization of shared memory fails. In that case, the switch path is used. For tasks on different nodes, the network path is used.
- The kernel extension must be loaded.
- The maximum number of shared memory tasks per operating system image is 128.

Appendix C. LAPI messages, return codes, and return values

This appendix lists LAPI's attention messages, error codes, return codes, and return values. For LAPI error messages, see *RSCT: Messages*.

LAPI attention messages

Table 33. LAPI attention messages

Attention message	Description
485	Failover setup failed because an internal error occurred.
486	Failover setup failed because NAM is not installed.
487	Failover setup failed because group services is not installed.
488	Failover setup failed because an old version of POE is installed.
489	Failover setup failed because POE is not installed.
490	Failover setup failed because a non-snX device is being used by the current job.
491	Failover function halted due to an internal LAPI error.
499	Bulk transfer is enabled.
500	Timeout between multiple tasks.
501	LAPI version string.
502	Shared memory initialization failed.
503	Shared memory initialization failed at checkpoint restart.
504	Shared memory was not used because only one task is running.
505	The task was not able to create shared memory.
506	The task was not able to get shared memory.
507	The task was not able to attach shared memory.
508	The task was not able to reserve a segment.
509	Initial communication over port.

LAPI return codes

Table 34. LAPI return codes

Return code	Return value	Description
0	LAPI_SUCCESS	The function call completed successfully.

LAPI error codes

Table 35 lists all of the LAPI error codes and their associated return values in numerical order (by error code).

Table 35. LAPI error codes

Error code	Return value	Description
400	LAPI_ERR_UNKNOWN	An asynchronous, internal communication error occurred.

Table 35. LAPI error codes (continued)

Error code	Return value	Description
401	LAPI_ERR_ALL_HNDL_IN_USE	All available LAPI instances are in use.
402	LAPI_ERR_BOTH_NETSTR_SET	Both network statements are set for a single LAPI instance.
404	LAPI_ERR_CSS_LOAD_FAILED	Unable to load the communication utility library.
405	LAPI_ERR_INFO_NULL	The <i>lapi_info</i> pointer is NULL.
406	LAPI_ERR_MSG_API	The MP_MSG_API environment setting has an error in it.
407	LAPI_ERR_NO_NETSTR_SET	No network statement is set or MP_MSG_API is not set correctly.
408	LAPI_ERR_NO_UDP_HNDLR	Told LAPI to use a user-defined <i>udp_hdlr</i> , but <i>udp_hdlr</i> is set to NULL.
409	LAPI_ERR_HDR_HNDLR_NULL	The header handler is NULL. This error is returned in an asynchronous error handler.
410	LAPI_ERR_PSS_NON_ROOT	Tried to initialize the persistent subsystem (PSS) as non- root .
412	LAPI_ERR_SHM_KE_NOT_LOADED	The shared memory kernel extension is not loaded.
413	LAPI_ERR_TIMEOUT	A communication timeout has occurred. This error is returned in an asynchronous error handler.
414	LAPI_ERR_REG_TIMER	An error occurred while re-registering the timer.
415	LAPI_ERR_UDP_PKT_SZ	The UDP packet size is not valid.
416	LAPI_ERR_USER_UDP_HNDLR_FAIL	The user-defined <i>udp_hdlr</i> failed.
417	LAPI_ERR_HNDL_INVALID	A non-valid handle was passed in to LAPI.
418	LAPI_ERR_RET_PTR_NULL	The output data pointer is NULL.
419	LAPI_ERR_ADDR_HNDL_RANGE	The address handle range is not valid.
420	LAPI_ERR_ADDR_TBL_NULL	The output address table is NULL.
421	LAPI_ERR_TGT_PURGED	The destination task is purged.
422	LAPI_ERR_MULTIPLE_WAITERS	Multiple threads are waiting for the same counter.
423	LAPI_ERR_MEMORY_EXHAUSTED	LAPI is unable to allocate storage.
424	LAPI_ERR_INFO_NONZERO	Unused fields in the lapi_info_t structure need to be zeroed out.
425	LAPI_ERR_ORG_ADDR_NULL	The source address pointer is NULL.
426	LAPI_ERR_TGT_ADDR_NULL	The target address pointer is NULL.
427	LAPI_ERR_DATA_LEN	The length passed in is too big.
428	LAPI_ERR_TGT	The target is not valid.
429	LAPI_ERR_UHDR_NULL	<i>uhdr</i> is NULL, but <i>uhdr_len</i> is greater than 0.
430	LAPI_ERR_UHDR_LEN	<i>uhdr_len</i> is too big.
431	LAPI_ERR_HDR_LEN	<i>uhdr_len</i> is not doubleword-aligned.
432	LAPI_ERR_ORG_EXTENT	The source vector's extent is too big.
433	LAPI_ERR_ORG_STRIDE	The source vector's stride is less than its block.
434	LAPI_ERR_NO_CONNECTIVITY	No connectivity to task.

Table 35. LAPI error codes (continued)

Error code	Return value	Description
435	LAPI_ERR_ADAPTERS_DOWN	All adapters are down.
436	LAPI_ERR_RECV_INCOMP	The "message receive" operation did not complete.
437	LAPI_ERR_SEND_INCOMP	The "message send" operation did not complete.
438	LAPI_ERR_SEND_TIMEOUT	The "message send" operation timed out.
439	LAPI_ERR_SHM_SETUP	The shared memory setup failed.
440	LAPI_ERR_ORG_VEC_ADDR	The source vector address is NULL, but its len is greater than 0.
441	LAPI_ERR_ORG_VEC_LEN	The source vector's length is too big.
442	LAPI_ERR_ORG_VEC_NULL	The source vector pointer is NULL.
443	LAPI_ERR_ORG_VEC_TYPE	The source vector type is not valid.
444	LAPI_ERR_STRIDE_ORG_VEC_ADDR_NULL	The source stride vector address is NULL.
445	LAPI_ERR_STRIDE_TGT_VEC_ADDR_NULL	The target stride vector address is NULL.
446	LAPI_ERR_TGT_EXTENT	The target vector's extent is too big.
447	LAPI_ERR_TGT_STRIDE	The target vector's stride is less than its block.
448	LAPI_ERR_TGT_VEC_ADDR	The target vector address is NULL, but its len is greater than 0.
449	LAPI_ERR_TGT_VEC_LEN	The target vector's length is too big.
451	LAPI_ERR_TGT_VEC_NULL	The target vector pointer is NULL.
452	LAPI_ERR_TGT_VEC_TYPE	The target vector type is not valid.
453	LAPI_ERR_VEC_NUM_DIFF	The source and target vectors have different <i>num_vecs</i> values.
454	LAPI_ERR_VEC_TYPE_DIFF	The source and target vectors have different <i>vec_type</i> values.
455	LAPI_ERR_VEC_LEN_DIFF	The source and target vectors have different <i>len[]</i> values.
456	LAPI_ERR_MSG_INFO_NULL	LAPI_Msgpoll's <i>info</i> pointer is NULL.
458	LAPI_ERR_CNTR_NULL	The counter pointer is NULL.
459	LAPI_ERR_CNTR_VAL	The counter value passed in is less than 0 for the LAPI_Nopoll_wait call.
460	LAPI_ERR_QUERY_TYPE	The query is not a valid query type.
461	LAPI_ERR_IN_VAL_NULL	LAPI_Rmw's <i>in_val</i> pointer is NULL.
462	LAPI_ERR_RMW_OP	The RMW operator is not valid.
463	LAPI_ERR_TGT_VAR_NULL	LAPI_Rmw's <i>tgt_var</i> address is NULL.
464	LAPI_ERR_SET_VAL	LAPI_Senv's <i>set_val</i> value is not valid.
465	LAPI_ERR_DGSP	The DGSP is NULL or is not registered.
466	LAPI_ERR_DGSP_ATOM	The DGSP <i>atom_size</i> is not valid.
467	LAPI_ERR_DGSP_BRANCH	The DGSP processed an incorrect branch.
468	LAPI_ERR_DGSP_CTL	The DGSP CONTROL instruction has errors.
469	LAPI_ERR_DGSP_COPY_SZ	The DGSP has a non-valid copy length.
470	LAPI_ERR_DGSP_FREE	A non-valid attempt was made to free a DGSP.
471	LAPI_ERR_DGSP_OPC	The DGSP <i>opcode</i> is not valid.

Table 35. LAPI error codes (continued)

Error code	Return value	Description
472	LAPI_ERR_DGSP_REPS	The DGSP has a non-valid <i>reps</i> field (its value is less than 0).
473	LAPI_ERR_DGSP_STACK	An insufficient stack depth was allocated for the DGSP stack.
474	LAPI_ERR_OP_SZ	The <i>lapi_rmw_t</i> size is not set to 32 or 64.
475	LAPI_ERR_UDP_PORT_INFO	The <i>udp_port</i> information pointer is NULL.
476	LAPI_ERR_XFER_CMD	The LAPI_Xfer command type is not valid.
477	LAPI_ERR_UTIL_CMD	The LAPI_Util command type is not valid.
478	LAPI_ERR_CATALOG_FAIL	LAPI cannot open the message catalog.
479	LAPI_ERR_PACK_SZ	The pack buffer is too small.
480	LAPI_ERR_DGSP_OTHER	A DGSP error occurred (<i>code_size</i> is equal to 0, for example).
481	LAPI_ERR_UDP_SOCKET	An error occurred during a UDP socket operation.
482	LAPI_ERR_COLLECTIVE_PSS	The persistent subsystem (PSS) attempted a collective call.
492	LAPI_ERR_TGT_CONTEXT	Non-valid target context.
493	LAPI_ERR_SRC_CONTEXT	Non-valid source context.
494	LAPI_ERR_TGT_BUFHNDL	Non-valid target buffer handle.
495	LAPI_ERR_SRC_BUFHNDL	Non-valid source buffer handle.
496	LAPI_ERR_TGT_OFFSET	Non-valid target offset.
497	LAPI_ERR_SRC_OFFSET	Non-valid source offset.
498	LAPI_ERR_NO_RDMA_RESOURCE	No RDMA resources.
499	LAPI_ERR_NO_RDMA_RESOURCE	No RDMA resources.
510	LAPI_ERR_NO_ENV_VAR	A required environment variable is not set.
511	LAPI_ERR_CODE_UNKNOWN	The error code is unknown to LAPI.

LAPI return values

Table 36 lists all of the LAPI return values and their associated return codes in alphabetical order (by return value).

Table 36. LAPI return values

Return value	Return code	Description
LAPI_ERR_ADAPTERS_DOWN	435	All adapters are down.
LAPI_ERR_ADDR_HNDL_RANGE	419	The address handle range is not valid.
LAPI_ERR_ADDR_TBL_NULL	420	The output address table is NULL.
LAPI_ERR_ALL_HNDL_IN_USE	401	All available LAPI instances are in use.
LAPI_ERR_BOTH_NETSTR_SET	402	Both network statements are set for a single LAPI instance.
LAPI_ERR_CATALOG_FAIL	478	LAPI cannot open the message catalog.
LAPI_ERR_CNTR_NULL	458	The counter pointer is NULL.

Table 36. LAPI return values (continued)

Return value	Return code	Description
LAPI_ERR_CNTR_VAL	459	The counter value passed in is less than 0 for the LAPI_Nopoll_wait call.
LAPI_ERR_CODE_UNKNOWN	485	The error code is unknown to LAPI.
LAPI_ERR_COLLECTIVE_PSS	482	The persistent subsystem (PSS) attempted a collective call.
LAPI_ERR_CSS_LOAD_FAILED	404	Unable to load the communication utility library.
LAPI_ERR_DATA_LEN	427	The length passed in is too big.
LAPI_ERR_DGSP	465	The DGSP is NULL or is not registered.
LAPI_ERR_DGSP_ATOM	466	The DGSP <i>atom_size</i> is not valid.
LAPI_ERR_DGSP_BRANCH	467	The DGSP processed an incorrect branch.
LAPI_ERR_DGSP_COPY_SZ	469	The DGSP has a non-valid copy length.
LAPI_ERR_DGSP_CTL	468	The DGSP CONTROL instruction has errors.
LAPI_ERR_DGSP_FREE	470	A non-valid attempt was made to free a DGSP.
LAPI_ERR_DGSP_OPC	471	The DGSP <i>opcode</i> is not valid.
LAPI_ERR_DGSP_OTHER	480	A DGSP error occurred (<i>code_size</i> is equal to 0, for example).
LAPI_ERR_DGSP_REPS	472	The DGSP has a non-valid <i>reps</i> field (its value is less than 0).
LAPI_ERR_DGSP_STACK	473	An insufficient stack depth was allocated for the DGSP stack.
LAPI_ERR_HDR_HNDLR_NULL	409	The header handler is NULL. This error is returned in an asynchronous error handler.
LAPI_ERR_HDR_LEN	431	<i>uhdr_len</i> is not doubleword-aligned.
LAPI_ERR_HNDL_INVALID	417	A non-valid handle was passed in to LAPI.
LAPI_ERR_INFO_NONZERO	424	Unused fields in the lapi_info_t structure need to be zeroed out.
LAPI_ERR_INFO_NULL	405	The <i>lapi_info</i> pointer is NULL.
LAPI_ERR_IN_VAL_NULL	461	LAPI_Rmw's <i>in_val</i> pointer is NULL.
LAPI_ERR_MEMORY_EXHAUSTED	423	LAPI is unable to allocate storage.
LAPI_ERR_MSG_API	406	The MP_MSG_API environment setting has an error in it.
LAPI_ERR_MSG_INFO_NULL	456	LAPI_Msgpoll's <i>info</i> pointer is NULL.
LAPI_ERR_MULTIPLE_WAITERS	422	Multiple threads are waiting for the same counter.
LAPI_ERR_NO_CONNECTIVITY	434	No connectivity to task.
LAPI_ERR_NO_ENV_VAR	510	A required environment variable is not set.
LAPI_ERR_NO_NETSTR_SET	407	No network statement is set or MP_MSG_API is not set correctly.
LAPI_ERR_NO_UDP_HNDLR	408	Told LAPI to use a user-defined <i>udp_hdlr</i> , but <i>udp_hdlr</i> is set to NULL.
LAPI_ERR_OP_SZ	474	The lapi_rmw_t size is not set to 32 or 64.
LAPI_ERR_ORG_ADDR_NULL	425	The source address pointer is NULL.
LAPI_ERR_ORG_EXTENT	432	The source vector's extent is too big.

Table 36. LAPI return values (continued)

Return value	Return code	Description
LAPI_ERR_ORG_STRIDE	433	The source vector's stride is less than its block.
LAPI_ERR_ORG_VEC_ADDR	440	The source vector address is NULL, but its len is greater than 0.
LAPI_ERR_ORG_VEC_LEN	441	The source vector's length is too big.
LAPI_ERR_ORG_VEC_NULL	442	The source vector pointer is NULL.
LAPI_ERR_ORG_VEC_TYPE	443	The source vector type is not valid.
LAPI_ERR_PACK_SZ	479	The pack buffer is too small.
LAPI_ERR_PSS_NON_ROOT	410	Tried to initialize the persistent subsystem (PSS) as non- root .
LAPI_ERR_QUERY_TYPE	460	The query is not a valid query type.
LAPI_ERR_RECV_INCOMP	436	The "message receive" operation did not complete.
LAPI_ERR_REG_TIMER	414	An error occurred while re-registering the timer.
LAPI_ERR_RET_PTR_NULL	418	The output data pointer is NULL.
LAPI_ERR_RMW_OP	462	The RMW operator is not valid.
LAPI_ERR_SEND_INCOMP	437	The "message send" operation did not complete.
LAPI_ERR_SEND_TIMEOUT	438	The "message send" operation timed out.
LAPI_ERR_SET_VAL	464	LAPI_Senv's <i>set_val</i> value is not valid.
LAPI_ERR_SHM_KE_NOT_LOADED	412	The shared memory kernel extension is not loaded.
LAPI_ERR_SHM_SETUP	439	The shared memory setup failed.
LAPI_ERR_SRC_BUFHNDL	495	Non-valid source buffer handle.
LAPI_ERR_SRC_CONTEXT	493	Non-valid source context.
LAPI_ERR_SRC_OFFSET	497	Non-valid source offset.
LAPI_ERR_STRIDE_ORG_VEC_ADDR_NULL	444	The source stride vector address is NULL.
LAPI_ERR_STRIDE_TGT_VEC_ADDR_NULL	445	The target stride vector address is NULL.
LAPI_ERR_TGT	428	The target is not valid.
LAPI_ERR_TGT_ADDR_NULL	426	The target address pointer is NULL.
LAPI_ERR_TGT_BUFHNDL	494	Non-valid target buffer handle.
LAPI_ERR_TGT_CONTEXT	492	Non-valid target context.
LAPI_ERR_TGT_EXTENT	446	The target vector's extent is too big.
LAPI_ERR_TGT_OFFSET	496	Non-valid target offset.
LAPI_ERR_TGT_PURGED	421	The destination task is purged.
LAPI_ERR_TGT_STRIDE	447	The target vector's stride is less than its block.
LAPI_ERR_TGT_VAR_NULL	463	LAPI_Rmw's <i>tgt_var</i> address is NULL.
LAPI_ERR_TGT_VEC_ADDR	448	The target vector address is NULL, but its len is greater than 0.
LAPI_ERR_TGT_VEC_LEN	449	The target vector's length is too big.
LAPI_ERR_TGT_VEC_NULL	451	The target vector pointer is NULL.
LAPI_ERR_TGT_VEC_TYPE	452	The target vector type is not valid.

Table 36. LAPI return values (continued)

Return value	Return code	Description
LAPI_ERR_TIMEOUT	413	A communication timeout has occurred. This error is returned in an asynchronous error handler.
LAPI_ERR_UDP_PKT_SZ	415	The UDP packet size is not valid.
LAPI_ERR_UDP_PORT_INFO	475	The <i>udp_port</i> information pointer is NULL.
LAPI_ERR_UDP_SOCKET	481	An error occurred during a UDP socket operation.
LAPI_ERR_UHDR_LEN	430	<i>uhdr_len</i> is too big.
LAPI_ERR_UHDR_NULL	429	<i>uhdr</i> is NULL, but <i>uhdr_len</i> is greater than 0.
LAPI_ERR_UNKNOWN	400	An asynchronous, internal communication error occurred.
LAPI_ERR_USER_UDP_HNDLR_FAIL	416	The user-defined <i>udp_hdlr</i> failed.
LAPI_ERR_UTIL_CMD	477	The LAPI_Util command type is not valid.
LAPI_ERR_VEC_LEN_DIFF	455	The source and target vectors have different <i>len[]</i> values.
LAPI_ERR_VEC_NUM_DIFF	453	The source and target vectors have different <i>num_vecs</i> values.
LAPI_ERR_VEC_TYPE_DIFF	454	The source and target vectors have different <i>vec_type</i> values.
LAPI_ERR_XFER_CMD	476	The LAPI_Xfer command type is not valid.
LAPI_SUCCESS	0	The function call completed successfully.

Appendix D. LAPI environment variables and runtime attributes

This appendix summarizes the LAPI environment variables and runtime attributes.

For information about MPI and POE environment variables, see the *Parallel Environment for AIX 5L: MPI Programming Guide* or *Parallel Environment for AIX 5L: Operation and Use, Volume 1*.

Environment variables

Variables for communication

Table 37 includes LAPI environment variables for communication:

Table 37. Environment variables for communication

Environment variable	Set:	Possible values	Default value
MP_MSG_API	To determine how adapter windows will be allocated for communication. LAPI will not initialize if this variable is not set. See "Setting environment variables" on page 29 for more information.	lapi lapi,mpi mpi,lapi mpi_lapi mpi	mpi (for users running PE)

Variables for data transfer

Table 38 includes LAPI environment variables for data transfer:

Table 38. Environment variables for data transfer

Environment variable	Set:	Possible values	Default value
LAPI_VERIFY_DGSP	To verify every DGSP at registration time. By default, this variable is set to no because it degrades performance. If it is set to yes , LAPI performs limited correctness checking of users' DGSPs at registration. It is recommended that applications in which users build DGSPs be tested with this variable set to yes , then run with it set to no during performance-critical operation. Note that many DGSP errors are only detectable during data transfer.	yes no	no
MP_BULK_MIN_MSG_SIZE	To change the minimum message size (in bytes) for which LAPI will attempt to make bulk transfers. If you specify a value that is less than 4K , MP_BULK_MIN_MSG_SIZE is set to 4K . This environment variable is a hint that may or may not be honored by the communication library. See Chapter 13, "Bulk transfer of messages," on page 99 for more information.	Any value greater than 4K	150K

Table 38. Environment variables for data transfer (continued)

Environment variable	Set:	Possible values	Default value
MP_USE_BULK_XFER	To enable or disable bulk message transfer using the remote direct memory access (RDMA) protocol. This environment variable is a hint that may or may not be honored by the communication library. See Chapter 13, "Bulk transfer of messages," on page 99 for more information.	yes no	no

Variables for diagnostics

Table 39 includes LAPI environment variables for diagnostics.

Table 39. Environment variables for diagnostics

Environment variable	Set:	Possible values	Default value
MP_DEBUG_NOTIMEOUT	To attach to one or more tasks without the concern that some other task may reach the LAPI timeout. Such a timeout would normally occur if one of the job tasks was continuing to run and tried to communicate with a task to which the programmer has attached using a debugger. With this variable set, LAPI never times out and continues retransmitting message packets forever. The default setting (no) lets LAPI time out.	Any non-null string	no
MP_LAPI_TRACE_LEVEL	The level of tracing to use for debugging (if AIX tracing is enabled). This environment variable is enabled for the libtrace library only.	0 (no LAPI trace output) 1, 2, 3, 4, 5 (increasing levels of LAPI trace output)	0

Variables for performance tuning

Table 40 includes LAPI environment variables that are considered user-tunable for performance. See "Tunable environment variables" on page 80 for more information.

Table 40. Environment variables for performance tuning

Environment variable	Set:	Possible values	Default value
MP_ACK_THRESH	The number of packets that are received before LAPI returns a batch of acknowledgments to the sending task.	A positive integer from 1 to 31	Depends on which communication adapter is used
MP_POLLING_INTERVAL	To control the interval for LAPI timer pops (in microseconds).	Any value greater than 10000	400000 (400 milliseconds)

Table 40. Environment variables for performance tuning (continued)

Environment variable	Set:	Possible values	Default value
MP_RETRANSMIT_INTERVAL	To control how often the communication subsystem library checks to see if it should retransmit packets that have not been acknowledged. The value specified is the number of polling loops between checks.	1000 to INT_MAX	1000000
MP_REXMIT_BUF_CNT	Specifies the number of buffers that LAPI must allocate. The size of each buffer is defined by MP_REXMIT_BUF_SIZE . This count indicates the number of in-flight messages smaller than MP_REXMIT_BUF_SIZE that LAPI can store in its local buffers in order to free up the user's message buffers more quickly.	Any integer greater than 0	128
MP_REXMIT_BUF_SIZE	The maximum message size, in bytes, that LAPI will store in its local buffers in order to more quickly free up the user buffer containing message data. This size indicates the size of the local buffers LAPI will allocate to store such messages, and will impact memory usage, while potentially improving performance. LAPI will use the buffer to store the user header and the user data.	Any integer greater than 0	16384
MP_UDP_PACKET_SIZE	To control the size of LAPI packets for UDP data transfer. LAPI initialization will fail if this variable is set to a value outside the valid range.	1024 to 65536	8192 for non-switch devices 65536 for switch devices

Variables for POE

Table 41 on page 272 includes LAPI environment variables for POE. The variables described in this section are set by the user and interpreted by POE. Although LAPI ignores these variables (except **MP_INFOLEVEL**), they are included in this document because they have an impact on LAPI jobs.

Table 41. Environment variables for POE

Environment variable	Set:	Possible values	Default value
MP_EUIDEVICE	The adapter set to use for message passing: Ethernet, Fiber Distributed Data Interface (FDDI), IP multi-link device, HPS, pSeries HPS, SP Switch2, or token ring.	<p>csss SP Switch2 (multi-plane, for PSSP LAPI)</p> <p>en0 Ethernet</p> <p>fi0 FDDI</p> <p>mI0 IP multi-link device</p> <p>sn_all HPS or pSeries HPS (for RSCT LAPI)</p> <p>sn_single HPS or pSeries HPS (for RSCT LAPI)</p> <p>tr0 token ring</p>	<p>The adapter set that is used as the external network address.</p> <p>IP: en0</p> <p>US: csss (for PSSP LAPI)</p> <p>sn_single (for RSCT LAPI)</p>
MP_EUILIB	The communication subsystem library implementation to use for communication: either the Internet Protocol (IP) communication subsystem or the user space (US) communication subsystem.	<p>These values are case-sensitive:</p> <p>ip</p> <p>us</p>	ip

Table 41. Environment variables for POE (continued)

Environment variable	Set:	Possible values	Default value
MP_INFOLEVEL	<p>The level of message reporting.</p> <p>Any value that is greater or equal to 2 causes LAPI to print out such library information as the LAPI version number and the build timestamp.</p>	<p>One of the following integers:</p> <ul style="list-style-type: none"> 0 Error. 1 Warning and error. 2 Informational, warning, and error. 3 Informational, warning, and error. Also reports high-level diagnostic messages for use by the IBM Support Center. 4 Informational, warning, and error. Also reports high- and low-level diagnostic messages for use by the IBM Support Center. 5 Informational, warning, and error. Also reports high- and low-level diagnostic messages for use by the IBM Support Center. 6 Informational, warning, and error. Also reports high- and low-level diagnostic messages for use by the IBM Support Center. 	0
MP_INSTANCES	<p>To control the number of instances requested for jobs, without requiring direct use of a LoadLeveler job control file (JCF). POE uses the value specified by this environment variable to add an instances=value field to the network statements of the JCF it creates. When multiple protocols (for example: mpi, lapi) are specified using MP_MSG_API, the same number of instances is used for both protocols. (Setting instances for each protocol to different values requires the use of a JCF.) The value in this field is ignored if you supply a JCF to POE.</p>	<p>1 to <i>max</i>, where <i>max</i> (a case-insensitive string), is the maximum number of windows that are usable by a job of the specified job class, as configured by the administrator. POE does not perform error checking. LoadLeveler interprets values that are greater than <i>max</i> as <i>max</i>.</p>	<p>none</p> <p>If POE creates a JCF without the instances field, LoadLeveler interprets this as a request for one window when MP_EUIDEVICE is set to sn_single or for one window on every network when MP_EUIDEVICE is set to sn_all.</p>

Variables for shared memory

Table 42 includes LAPI environment variables for shared memory jobs. Set these variables if you are using LAPI on a standalone system.

Table 42. Environment variables for shared memory

Environment variable	Set:	Possible values	Default value
LAPI_USE_SHM	<p>To enable or disable the use of shared memory.</p> <p>no -- disables the use of shared memory (the default).</p> <p>yes -- enables the use of shared memory where it is possible. LAPI will communicate using shared memory among all common tasks (tasks that are on the same node) over the selected device (user space over switch, IP over switch, or IP over Ethernet). See MP_EUIDEVICE and MP_EUILIB for tasks on different nodes. Shared memory requires segment registers, which can affect availability to user code in 32-bit applications.</p> <p>only -- communicates only using shared memory. LAPI will fail to initialize if this option is chosen and tasks are assigned to more than one node.</p>	<p>yes no only</p>	no

Variables for standalone systems

When LAPI is running in a PE environment, POE sets the variables that are described in Table 43. When LAPI is running in standalone mode, you need to set these variables explicitly. See “Standalone setup” on page 115 for more information.

Table 43. Environment variables for standalone systems

Environment variable	Set:	Possible values	Default value
MP_CHILD	The task ID of the current job. MP_CHILD needs to be set to a unique value for each task in standalone mode.	Any value that is greater than or equal to 0 and less than the value of MP_PROCS	you need to set
MP_COMMON_TASKS	For shared memory jobs.	See “Standalone setup” on page 115.	you need to set
MP_LAPI_INET_ADDR	The network setup among LAPI tasks for IP communication.	See “Standalone setup” on page 115.	you need to set
MP_LAPI_NETWORK	LAPI network information.	See “Standalone setup” on page 115.	you need to set

Table 43. Environment variables for standalone systems (continued)

Environment variable	Set:	Possible values	Default value
MP_PARTITION	A number that is the same for all tasks in the job. In standalone mode, you need to set this variable to an identical value for each task. In standalone mode for switched communication, the MP_PARTITION value must be associated with the network table description file.	any value	you need to set
MP_PROCS	The value of <i>num_tasks</i> , which is the total number of program tasks in the job. This number must be the same for all tasks.	A positive integer from 1 to the maximum number of tasks that is supported by the configuration.	you need to set

The descriptions and formats of **MP_COMMON_TASKS**, **MP_LAPI_INET_ADDR**, **MP_LAPI_NETWORK** are provided in this book for informational purposes only. These environment variables are not intended to be used as external programming interfaces. IBM will not guarantee that the formats or values of these variables can continue to be used without change in future releases. Programmers and users who choose to develop applications that depend on these variables do so with the understanding that these variables may be subject to future change. IBM cannot guarantee that such applications can migrate or coexist with future releases without additional changes, nor will IBM ensure that there will be binary compatibility of these variables.

Runtime attributes

This section includes attributes that you can query during runtime using the **LAPI_Qenv** interface. See “LAPI_Qenv” on page 184 for more information about using **LAPI_Qenv**.

You can set some of these attributes using **LAPI_Senv**. See “Attributes you can query or set.” For more information about using **LAPI_Senv**, see “LAPI_Senv” on page 196.

Attributes you can query or set

Table 44 includes runtime attributes you can set using **LAPI_Senv** or query using **LAPI_Qenv**.

Table 44. Runtime attributes you can query or set

Runtime attribute	Description
ACK_THRESHOLD	This value represents the number of packets received before LAPI sends acknowledgements. It can also be set using LAPI_Senv or with the MP_ACK_THRESH environment variable (see “Variables for performance tuning” on page 270).
ERROR_CHK	This attribute is a user-settable toggle that indicates whether LAPI should perform error checking. If set, LAPI calls will perform bounds- checking on parameters. Due to the potential performance degradation, error checking is disabled by default.
INTERRUPT_SET	This a user-settable toggle value that controls whether LAPI runs with interrupts turned on or off. With interrupts on, a timer-driven interrupt will drive packet acknowledgements and retransmits.

Table 44. Runtime attributes you can query or set (continued)

Runtime attribute	Description
TIMEOUT	This value corresponds to number of seconds that LAPI should wait on receiving packet acknowledgements before considering a remote task as unreachable. It can be set to a value in the range MIN_TIMEOUT < TIMEOUT < MAX_TIMEOUT. The default is 900 seconds (15 minutes).

Attributes you can query

This section includes attribute values that you can query during runtime using the **LAPI_Qenv** interface. **LAPI_Qenv** returns values through a reference parameter.

Attributes that return integers

Table 45 includes attributes that return integers. The actual parameter is expected to be of type **&int**.

Table 45. Attributes that return integers

Runtime attribute	Description
BUF_CP_SIZE	This represents the value of LAPI's send-side copy buffer. It can be set at job startup using the MP_REXMIT_BUF_SIZE environment variable (see "Variables for performance tuning" on page 270).
BULK_MIN_MSG_SIZE	This represents the current minimum message size that will be used for bulk transfer.
BULK_XFER	This value indicates whether bulk transfer is enabled (1) or disabled (0).
LOC_ADDRTBL_SZ	This value represents the upper bound on LAPI's internal table size. For example, the size of the address table used in LAPI_Addr_set is bounded by this value.
MAX_ATOM_SIZE	This represents the maximum atom size for user DGSPs. See "Using data gather/scatter programs (DGSPs)" on page 43 for more information.
MAX_DATA_SZ	This query is deprecated. For 32-bit applications, it will return the value of the macro LAPI_MAX_MSG_SZ defined in lapi.h . For 64-bit applications, it will return the largest unsigned integer. Rather than use this query, it is recommended that users use the macro LAPI_MAX_MSG_SZ directly, which is valid for both 32-bit and 64-bit applications.
MAX_PKT_SZ	This value represents the maximum storage for user data (header + data) in each LAPI packet.
MAX_PKTS_OUT	This value represents the maximum number of packets that can be "in flight" between any two tasks.
MAX_PORTS	This value represents the maximum number of LAPI instances that are available for use.
MAX_TIMEOUT	This is the maximum number of seconds that a user can set for TIMEOUT. If an attempt is made to set TIMEOUT to a value outside the valid range, LAPI_Senv will return an error and TIMEOUT will be unchanged.
MAX_UHDR_SZ	The maximum size in bytes that can be used for user header data. See "LAPI_Amsend" on page 136 for more information on using a user header.
MIN_TIMEOUT	This is the minimum number of seconds that a user can set for TIMEOUT. If an attempt is made to set TIMEOUT to a value outside the valid range, LAPI_Senv will return an error and TIMEOUT will be unchanged.

Table 45. Attributes that return integers (continued)

Runtime attribute	Description
NUM_REX_BUFS	This value represents the number of retransmission buffers that LAPI uses.
NUM_TASKS	The total number of tasks in the job. This corresponds to the value set in MP_PROCS .
QUERY_SHM_ENABLED	This is a boolean value that indicates whether LAPI is communicating using shared memory.
QUERY_SHM_NUM_TASKS	This value represents the number of tasks with which the current task can communicate using shared memory.
REX_BUF_SZ	This value represents the size of LAPI's retransmission buffers.
TASK_ID	The ID that LAPI has for the given task. For standalone jobs, it corresponds to the value set in MP_CHILD .

Attributes that return multiple values

Table 46 includes runtime attributes that return multiple values. These attributes expect a pointer to a different datatype and require that the actual parameter represents sufficient space for the query type. LAPI casts the passed pointer to this type.

Table 46. Attributes that return multiple values

Runtime attribute	Description
PRINT_STATISTICS	Calling LAPI_Qenv with this query value will cause LAPI to dump the values that would be returned in a lapi_statistics_t structure to standard output.
QUERY_LOCAL_SEND_STATISTICS	Returns a number of statistics about the running environment, for the local copy path. Expects a pointer to type lapi_statistics_t . For example: <pre> { lapi_statistics_t stats; LAPI_Qenv(handle, QUERY_LOCAL_SEND_STATISTICS, (int *)&stats)); } </pre> <p>Note that the address of <i>stats</i> is cast to int *. This is required to match the signature of LAPI_Qenv.</p>
QUERY_SHM_STATISTICS	Returns a number of statistics about the running environment, for the shared memory path. Expects a pointer to type lapi_statistics_t . For example: <pre> { lapi_statistics_t stats; LAPI_Qenv(handle, QUERY_SHM_STATISTICS, (int *)&stats)); } </pre> <p>Note that the address of <i>stats</i> is cast to int *. This is required to match the signature of LAPI_Qenv.</p>

Table 46. Attributes that return multiple values (continued)

Runtime attribute	Description
<p>QUERY_SHM_TASKS</p>	<p>This query returns a list of shared memory task IDs for each task with which this task can communicate using shared memory. LAPI expects an array large enough to hold an integer index for each task in the job. For example:</p> <pre data-bbox="592 352 1404 772"> { int *shm_task_list; int i; LAPI_Qenv(hndl, NUM_TASKS, &num_tasks); shm_task_list = (int *) (malloc(sizeof(int)*num_tasks)); LAPI_Qenv(hndl, QUERY_SHM_TASKS, shm_task_list); for(i = 0; i < num_tasks; i++) { printf("task[%d] has shm_task_id %d, (num_tasks:%d)\n", i, shm_task_list[i], num_tasks); } free(shm_task_list); } </pre>
<p>QUERY_STATISTICS</p>	<p>Returns a number of statistics about the running environment. Expects a pointer to type lapi_statistics_t. For example:</p> <pre data-bbox="592 856 1226 1014"> { lapi_statistics_t stats; LAPI_Qenv(handle, QUERY_STATISTICS, (int *)&stats)); } </pre> <p>Note that the address of <i>stats</i> is cast to int *. This is required to match the signature of LAPI_Qenv.</p>

Attributes for legacy code

LAPI no longer uses the following runtime attributes. They are included for support of legacy code.

- **EPOCH_NUM**
- **RCV_FIFO_SIZE**
- **USE_THRESH**

Appendix E. LAPI datatypes

This appendix lists datatypes that you can use in your LAPI programs.

Table 47. LAPI datatypes

C datatype	FORTRAN datatype	Description
<code>com_thread_info_t</code>	<code>COM_THREAD_INFO_T</code>	For thread attribute and initialization functions.
<code>compl_hndlr_t</code>	<code>COMPL_HNDLR_T</code>	The receive completion handler.
<code>ddm_func_t</code>	<code>DDM_FUNC_T</code>	For data distribution manager (DDM) functions.
<code>hdr_hndlr_t</code>	<code>HDR_HNDLR_T</code>	The header handler for a contiguous DGSP message.
<code>in_addr_t</code>	<code>IN_ADDR_T</code>	For local IP addresses.
<code>in_port_t</code>	<code>IN_PORT_T</code>	For local port addresses.
<code>lapi_add_udp_port_t</code>	<code>LAPI_ADD_UDP_PORT_T</code>	The LAPI_Util command structure (in C) or the datatype (in FORTRAN) for updating the UDP port information of the destination task.
(C equivalent: <code>void *</code>)	<code>LAPI_ADDR_TYPE</code>	For address functions.
<code>lapi_am_t</code>	<code>LAPI_AM_T</code>	The LAPI_Xfer command structure (in C) or the datatype (in FORTRAN) for one contiguous active message.
<code>lapi_amdgsp_t</code>	<code>LAPI_AMDGSP_T</code>	The LAPI_Xfer command structure (in C) or the datatype (in FORTRAN) for one DGSP active message.
<code>lapi_amv_t</code>	<code>LAPI_AMV_T</code>	The LAPI_Xfer command structure (in C) or the datatype (in FORTRAN) for one vector active message.
<code>lapi_cntr_t</code>	<code>LAPI_CNTR_T</code>	Defines a LAPI counter.
<code>lapi_ctl_flags_t</code>		Part of the <code>lapi_return_info_t</code> structure. Instructs LAPI on what it should do with a message after the header handler is called.
<code>lapi_dev_t</code>	<code>LAPI_DEV_T</code>	For protocol devices.
<code>lapi_dg_handle_t</code>	<code>LAPI_DG_HANDLE_T</code>	Defines a DGSP handle.
<code>lapi_dgsm_block_t</code>	<code>LAPI_DGSM_BLOCK_T</code>	Defines a DGSP BLOCK instruction.
<code>lapi_dgsm_control_t</code>	<code>LAPI_DGSM_CONTROL_T</code>	Defines a DGSP CONTROL instruction.
<code>lapi_dgsm_copy_t</code>	<code>LAPI_DGSM_COPY_T</code>	Defines a DGSP COPY instruction.
<code>lapi_dgsm_gosub_t</code>	<code>LAPI_DGSM_GOSUB_T</code>	Defines a DGSP GOSUB instruction.
<code>lapi_dgsm_iterate_t</code>	<code>LAPI_DGSM_ITERATE_T</code>	Defines a DGSP ITERATE instruction.
<code>lapi_dgsm_mcopy_t</code>	<code>LAPI_DGSM_MCOPY_T</code>	Defines a DGSP multiple copy (MCOPY) instruction.
<code>lapi_dgsp_density_t</code>	<code>LAPI_DGSM_SPARSE</code> <code>LAPI_DGSM_CONTIG</code> <code>LAPI_DGSM_UNIT</code>	In C: an enumeration for values in a LAPI DGSP descriptor. In FORTRAN: one of three DGSP data layout types.
<code>lapi_dgsp_descr_t</code>	<code>LAPI_DGSP_DESCR_T</code>	A DGSP descriptor structure.
<code>lapi_dgsp_handle_t</code>	<code>LAPI_DGSP_HANDLE_T</code>	A handle for a registered DGSP.

Table 47. LAPI datatypes (continued)

C datatype	FORTRAN datatype	Description
<code>lapi_dref_dgsp_t</code>	<code>LAPI_DREF_DGSP_T</code>	The <code>LAPI_Util</code> command structure (in C) or the datatype (in FORTRAN) for a DGSP un-reserve operation.
<code>lapi_err_t</code>	<code>LAPI_ERR_T</code>	Error type.
<code>lapi_extend_t</code>	<code>LAPI_EXTEND_T</code>	Additional structure extension.
<code>lapi_get_t</code>	<code>LAPI_GET_T</code>	The <code>LAPI_Xfer</code> command structure (in C) or the datatype (in FORTRAN) for one "get" message.
<code>lapi_getv_t</code>	<code>LAPI_GETV_T</code>	The <code>LAPI_Xfer</code> command structure (in C) or the datatype (in FORTRAN) for one vector "get" message.
<code>lapi_handle_t</code>	<code>LAPI_HANDLE_T</code>	An opaque handle for identifying the LAPI context.
<code>lapi_info_t</code>	<code>LAPI_INFO_T</code>	Command structure for <code>LAPI_Init</code> .
<code>lapi_lib_t</code>	<code>LAPI_LIB_T</code>	For the LAPI library version.
<code>lapi_long_t</code>	<code>LAPI_LONG_TYPE</code>	LAPI long type.
<code>long long</code>	<code>LAPI_LONG_LONG_TYPE</code>	LAPI long long type.
<code>lapi_lvec_t</code>	<code>LAPI_LVEC_T</code>	LAPI long vector type.
<code>lapi_msg_info_t</code>	<code>LAPI_MSG_INFO_T</code>	Information about a <code>LAPI_Msgpoll</code> call.
<code>lapi_msg_state_t</code>	<code>LAPI_MSG_STATE_T</code>	In <code>lapi_msg_info_t</code> , indicates whether there is any completion of send/receive after <code>LAPI_Msgpoll</code> returns.
<code>lapi_msglen_t</code>	<code>LAPI_MSGLEN_T</code>	LAPI message length.
<code>lapi_pack_dgsp_t</code>	<code>LAPI_PACK_DGSP_T</code>	The <code>LAPI_Util</code> command structure (in C) or the datatype (in FORTRAN) for packing data from a memory layout defined by a DGSP to a contiguous buffer.
<code>lapi_put_t</code>	<code>LAPI_PUT_T</code>	The <code>LAPI_Xfer</code> command structure (in C) or the datatype (in FORTRAN) for one "put" message.
<code>lapi_putv_t</code>	<code>LAPI_PUTV_T</code>	The <code>LAPI_Xfer</code> command structure (in C) or the datatype (in FORTRAN) for one vector "put" message.
<code>lapi_query_t</code>	No equivalent type (query types are defined explicitly in the 32-bit and 64-bit versions of <code>lapif.h</code>).	In C: an enumeration that defines all queries supported by <code>LAPI_Qenv</code> and <code>LAPI_Senv</code> .
<code>lapi_reg_ddm_t</code>	<code>LAPI_REG_DDM_T</code>	The <code>LAPI_Util</code> command structure (in C) or the datatype (in FORTRAN) for data distribution manager (DDM) functions.
<code>lapi_reg_dgsp_t</code>	<code>LAPI_REG_DGSP_T</code>	The <code>LAPI_Util</code> command structure (in C) or the datatype (in FORTRAN) for registering a DGSP.
<code>lapi_resv_dgsp_t</code>	<code>LAPI_RESV_DGSP_T</code>	The <code>LAPI_Util</code> command structure (in C) or the datatype (in FORTRAN) for reserving a DGSP.
<code>lapi_ret_flags_t</code>		Part of the <code>lapi_return_info_t</code> structure. Indicates to LAPI whether to run the completion handler inline.
<code>lapi_return_info_t</code>		A structure that extends the header handler interface to pass information between LAPI and a user program.

Table 47. LAPI datatypes (continued)

C datatype	FORTRAN datatype	Description
<code>lapi_rmw_t</code>	<code>LAPI_RMW_T</code>	The LAPI_Xfer command structure (in C) or the datatype (in FORTRAN) for one read-modify-write operation.
<code>lapi_sh_info_t</code>	<code>LAPI_SH_INFO_T</code>	Send completion handler information.
<code>lapi_statistics_t</code>	<code>LAPI_STATISTICS_T</code>	LAPI statistics.
<code>lapi_udpinfo_t</code>	<code>LAPI_UDPINFO_T</code>	UDP information
<code>lapi_unpack_dgsp_t</code>	<code>LAPI_UNPACK_DGSP_T</code>	The LAPI_Util command structure (in C) or the datatype (in FORTRAN) for unpacking data from a contiguous buffer to a memory layout that is defined by a DGSP.
<code>lapi_usr_fcall_t</code>	<code>LAPI_USR_FCALL_T</code>	For debugging only.
<code>lapi_util_t</code>	<code>LAPI_ADD_UDP_DEST_PORT</code> <code>LAPI_DGSP_PACK</code> <code>LAPI_DGSP_UNPACK</code> <code>LAPI_REG_DDM_FUNC</code> <code>LAPI_REGISTER_DGSP</code> <code>LAPI_RESERVE_DGSP</code> <code>LAPI_UNRESERVE_DGSP</code>	In C: the union of all possible command structures for LAPI_Util . In FORTRAN: one of seven types for LAPI_Util .
<code>lapi_util_type_t</code>	<code>LAPI_UTIL_TYPE_T</code>	Specifies the type of utility in the command structures for LAPI_Util .
<code>lapi_vec_t</code>	<code>LAPI_VEC_T</code>	Defines LAPI vector data layout.
<code>lapi_vectype_t</code>	<code>LAPI_VECTYPE_T</code>	The type of a vector.
<code>lapi_xfer_t</code>	<code>LAPI_AM_XFER</code> <code>LAPI_AMV_XFER</code> <code>LAPI_DGSP_XFER</code> <code>LAPI_GET_XFER</code> <code>LAPI_GETV_XFER</code> <code>LAPI_PUT_XFER</code> <code>LAPI_PUTV_XFER</code> <code>LAPI_RMW_XFER</code>	In C: the union of all possible command structures for LAPI_Xfer . In FORTRAN: one of eight transfer types for LAPI_Xfer .
<code>lapi_xfer_type_t</code>	<code>LAPI_XFER_TYPE_T</code>	In C, the type of a LAPI_Xfer command structure.
<code>RMW_ops_t</code>	<code>RMW_OPS_T</code>	The type of read-modify-write operation.
<code>scompl_hndlr_t</code>	<code>SCOMPL_HNDLR_T</code>	The send completion handler.
<code>vhdr_hndlr_t</code>	<code>VHDR_HNDLR_T</code>	The header handler for a vector message.

Appendix F. LAPI constants and size limits

This appendix includes information about LAPI constants and size limits.

The upper bound on the message size that a user can transfer
(**LAPI_MAX_MSG_SZ**):

- For 32-bit applications: **0x7fffffff**
- For 64-bit applications: **0x7fffffffffffffffLL**

The maximum error string length (**LAPI_MAX_ERR_STRING**): 160 characters

The maximum number of nodes (**MAX_NODES**): 2048

The maximum number of tasks allowed within a task (**MAX_TASKS**): 16384

The maximum number of shared memory tasks (**MAX_SHM_TASKS**): 128

The maximum number of slots per task (**MAX_SLOTS_PER_TASK**): 128

Appendix G. LAPI restrictions

General restrictions

Nodes within the same LAPI job must be at the same version of LAPI. Interoperability with previous versions of LAPI is not supported.

You *cannot* make LAPI calls from within the header handler. For contiguous data, you can copy the data to the buffer you specified. For non-contiguous data, you must pass the DGSP handle and a buffer address to LAPI. LAPI will unpack the data to the specified buffer address.

I/O operations and blocking calls, including blocking LAPI calls, should *not* be performed within an inline completion handler. Inline completion handlers should be short, because no progress can be made while the main thread is executing the handler. You must use caution with inline completion handlers so that LAPI's internal queues do not fill up while waiting for the handler to complete. Note that LAPI places no restrictions on completion handlers that are run "normally" (that is, by the completion handler thread).

User application threads and completion handler threads cannot hold mutual exclusion resources before making LAPI calls. If they do, it is possible to run into deadlock situations.

Static linking is *not* supported.

For systems running PE, both US and IP are supported for shared handles as long as they are the same for both handles. Mixed transport protocols such as LAPI IP and LAPI user space (US) are *not* supported.

Use of segment registers (32-bit applications only)

The user space (US) LAPI library uses two segment registers of the 10 that are unassigned in the user's AIX process space. Thus, the user can allocate a maximum of 8 segments (**-bmaxdata=0x80000000**) to extended heap for large data structures. However, programs compiled with 8 segments may experience reduced shared memory performance in a 32-bit environment because there is no extra segment for LAPI to do address space attach, which avoids one extra copy for large messages. See "The shared memory execution model" on page 257 for more information.

Other restrictions

For restrictions related to:

- Lock sharing, see "Implications and restrictions" on page 93
- Striping, failover, and recovery, see "Failover and recovery restrictions" on page 106 and "Communication and memory considerations" on page 108
- Shared memory, see "LAPI shared memory: requirements and restrictions" on page 259

Glossary

access control. The process of limiting access to system objects and resources to authorized principals.

access control list. A list of principals and the type of access allowed to each.

ACL. See *access control list*.

action. The part of the event response resource that contains a command and other information about the command.

attribute. Attributes are either persistent or dynamic. A resource class is defined by a set of persistent and dynamic attributes. A resource is also defined by a set of persistent and dynamic attributes. Persistent attributes define the configuration of the resource class and resource. Dynamic attributes define a state or a performance-related aspect of the resource class and resource. In the same resource class or resource, a given attribute name can be specified as either persistent or dynamic, but not both.

AIX. Advanced Interactive Executive. See *AIX operating system*.

AIX operating system. IBM's implementation of the UNIX operating system.

authentication. The process of validating the identity of an entity, generally based on user name and password. However, it does not address the access rights of that entity. Thus, it simply makes sure a user is who he or she claims to be.

authorization. The process of granting or denying access to an entity to system objects or resources, based on the entity's identity.

checksum. A count of the number of bits in a transmission unit that is included with the unit so that the receiver can check to see whether the same number of bits arrived. If the counts match, it's assumed that the complete transmission was received. TCP and UDP communication layers provide a checksum count and verification as one of their services.

client. Client applications are the ordinary user interface programs that are invoked by users or routines provided by trusted services for other components to use. The client has no network identity of its own: it assumes the identity of the invoking user or of the process where it is called, who must have previously obtained network credentials.

cluster. A group of servers and other resources that act like a single system and enable high availability and, in some cases, load balancing and parallel processing.

clustering. The use of multiple computers (such as UNIX workstations, for example), multiple storage devices, and redundant interconnections to form what appears to users as a single highly-available system. Clustering can be used for load balancing, for high availability, and as a relatively low-cost form of parallel processing for scientific and other applications that lend themselves to parallel operations.

cluster security services. A component of RSCT that is used by RSCT applications and other RSCT components to perform authentication within both management domains and peer domains.

condition. A state of a resource as defined by the event response resource manager (ERRM) that is of interest to a client. It is defined by means of a logical expression called an event expression. Conditions apply to resource classes unless a specific resource is designated.

condition/response association. A link between a condition and a response.

CSM. Clusters Systems Management.

datagram. Synonymous with *UDP packet*.

domain. (1) A set of network resources (such as applications and printers, for example) for a group of users. A user logs in to the domain to gain access to the resources, which could be located on a number of different servers in the network. (2) A group of server and client machines that exist in the same security structure. (3) A group of computers and devices on a network that are administered as a unit with common rules and procedures. Within the Internet, a domain is defined by its Internet Protocol (IP) address. All devices that share a common part of the IP address are said to be in the same domain.

event. Occurs when the event expression of a condition evaluates to True. An evaluation occurs each time an instance of a dynamic attribute is observed.

event expression. A definition of the specific state when an event is true.

event response. One or more actions as defined by the event response resource manager (ERRM) that take place in response to an event or a rearm event.

failover. A backup operation that automatically switches to another adapter if one adapter fails. Failover is an important fault-tolerance function of mission-critical systems that rely on constant accessibility. Automatically and transparently to the user, failover redirects requests from the failed adapter to another adapter that mimics the operations of the failed adapter.

FFDC. See *first failure data capture*.

first failure data capture. Provides a way to track problems back to their origin even though the source problem may have occurred in other layers or subsystems than the layer or subsystem with which the end user is interacting. FFDC provides a correlator called an **ffdc_id** for any error that it writes to the AIX error log. This correlator can be used to link related events together to form a chain.

FIFO. First in first out, usually referring to buffers.

High Performance Switch. The switch that works in conjunction with IBM @server p5 servers (575, 595).

HPS. See *High Performance Switch*.

Internet Protocol. The method by which data is sent from one computer to another on the Internet.

IP. See *Internet Protocol*.

IP address. A 32-bit (in IP Version 4) or 128-bit (in IP Version 6) number identifying each sender or receiver of information that is sent in packets across the Internet.

LAPI. See *low-level application programming interface*.

Linux. A freeware clone of UNIX for 386-based personal computers (PCs). Linux consists of the **linux** kernel (core operating system), originally written by Linus Torvalds, along with utility programs developed by the Free Software Foundation and by others.

LoadLeveler. The IBM LoadLeveler licensed program is a job management system that works with POE to let users run jobs and match processing needs with system resources, in order to make better use of the system.

low-level application programming interface. A low-overhead message-passing protocol that uses a one-sided communication model and active message paradigm to transfer data among tasks. See also *RSCT LAPI*. Contrast with *PSSP LAPI*.

logical unit number. A unique identifier used on a SCSI bus that enables it to differentiate between up to eight separate devices (each of which is a logical unit). Each LUN is a unique number that identifies a specific logical unit, which may be an end user, a file, or an application program.

LUN. See *logical unit number*.

management domain. A set of nodes configured for manageability by the Clusters Systems Management (CSM) licensed program. Such a domain has a management server that is used to administer a number of managed nodes. Only management servers have knowledge of the whole domain. Managed nodes only know about the servers managing them; they know nothing of each other. Contrast with *peer domain*.

Message Passing Interface. A standardized API for implementing the message-passing model.

MPI. See *Message Passing Interface*.

mutex. See *mutual exclusion object*.

mutual exclusion object. A program object that allows multiple program threads to share the same resource, such as file access, but not simultaneously. When a program is started, a mutual exclusion object is created with a unique name. After this stage, any thread that needs the resource must lock the mutual exclusion object from other threads while it is using the resource. The mutual exclusion object is set to unlock when the data is no longer needed or the routine is finished.

network credentials. These represent the data specific to each underlying security mechanism.

OSI. Operating system image.

PAC. See *privileged attribute certificate*.

packet. The unit of data that is routed between an origin and a destination on the Internet or any other packet-switched network.

Parallel Environment. The IBM Parallel Environment for AIX 5L licensed program is an execution and development environment for parallel C, C++, and FORTRAN programs. It also includes tools for debugging, profiling, and tuning parallel programs.

parallel operating environment. An execution environment that smooths the differences between serial and parallel execution. It lets you submit and manage parallel jobs.

Parallel System Support Programs. The IBM Parallel System Support Programs for AIX 5L licensed program is system administration software for the IBM RS/6000® SP system.

PE. See *Parallel Environment*.

peer domain. A set of nodes configured for high availability by the configuration resource manager. Such a domain has no distinguished or master node. All nodes are aware of all other nodes, and administrative commands can be issued from any node in the domain. All nodes also have a consistent view of the domain membership. Contrast with *management domain*.

POE. See *parallel operating environment*.

port. A "logical connection place". Using TCP/IP, the way a client program specifies a particular server program on a computer in a network.

principal. A user, an instance of the server, or an instance of a trusted client whose identity is to be authenticated.

privileged attribute certificate. Contains such information as the client's name and the groups to which it belongs. Its format is dependent on the underlying security mechanism.

protocol. The set of rules that endpoints in a telecommunication connection use when they communicate.

pSeries High Performance Switch. The switch that works in conjunction with IBM @server pSeries servers (655, 690).

pSeries HPS. See *pSeries High Performance Switch*.

PSSP. See *Parallel System Support Programs*.

PSSP LAPI. The version of LAPI that supports the SP Switch2.

rearm event. Occurs when the rearm expression for a condition evaluates to True.

rearm expression. An expression that generates an event which alternates with an original event in the following way: the event expression is used until it is true; then, the rearm expression is used until it is true; then, the event expression is used. The rearm expression is commonly the inverse of the event expression. It can also be used with the event expression to define an upper and lower boundary for a condition of interest.

Reliable Scalable Cluster Technology. A set of software components that together provide a comprehensive clustering environment for AIX and Linux. RSCT is the infrastructure used by a variety of IBM products to provide clusters with improved system availability, scalability, and ease of use.

resource. An entity in the system that provides a set of services. Examples of hardware entities are processors, disk drives, memory, and adapters. Examples of software entities are database applications, processes, and file systems. Each resource in the system has one or more attributes that define the state of the resource.

resource class. A broad category of system resource, for example: node, file system, adapter. Each resource class has a container that holds the functions, information, dynamic attributes, and conditions that apply to that resource class. For example, the **/tmp space used** condition applies to a file system resource class.

resource manager. A process that maps resource and resource-class abstractions into calls and commands for one or more specific types of resources. A resource manager can be a standalone daemon, or it can be integrated into an application or a subsystem directly.

RSCT. See *Reliable Scalable Cluster Technology*.

RSCT LAPI. The version of LAPI that supports the IBM @server pSeries High Performance Switch (pSeries HPS) or an IBM @server High Performance Switch (HPS) for p5 servers. See also *low-level application programming interface*.

RSCT peer domain. See *peer domain*.

SCSI. See *Small System Computer Interface*.

Small System Computer Interface. A parallel interface that can have up to eight devices all attached through a single cable; the cable and the host (computer) adapter make up the SCSI bus. The bus allows the interchange of information between devices independently of the host. In the SCSI program, each device is assigned a unique number, which is either a number between 0 and 7 for an 8-bit (narrow) bus, or between 8 and 16 for a 16-bit (wide) bus. The devices that request input/output (I/O) operations are initiators and the devices that perform these operations are targets. Each target has the capacity to connect up to eight additional devices through its own controller; these devices are the logical units, each of which is assigned a unique number for identification to the SCSI controller for command processing.

SD. Structured data.

security context token. A pointer to an opaque data structure called the context token descriptor. The context token is associated with a connection between a client and the server.

security services token. A pointer to an opaque descriptor called the security token descriptor. It keeps track of the mechanism-independent information and state.

servers. Server programs are usually daemons or other applications running in the background without a user's inherited credentials. A server must acquire its own network identity to get to access other trusted services.

SP Switch2. The switch that works in conjunction with IBM RS/6000 SP systems.

standalone system. A system on which you are using LAPI that is not running IBM's Parallel Environment for AIX licensed program.

striping. The distribution of message data across multiple communication adapters in order to increase bandwidth.

TCP. See *Transmission Control Protocol*.

Transmission Control Protocol. One of the core Internet protocols. TCP ports are 16-bit entities, so a maximum of 65535 different endpoints are possible within a single IP address.

UDP. See *User Datagram Protocol*.

User Datagram Protocol. One of the core Internet protocols. UDP is a layer 4 protocol (Transport layer of the OSI model) within the Internet protocol suite. It provides a mechanism to identify different endpoints on a single host by using ports. UDP deals with single-packet delivery that is provided by the underlying IP. As a stateless protocol, it is often used in applications where data must arrive quickly. This smaller feature set provides quicker data transmittal and lower total overhead. UDP packets (or *datagrams*) contain, in addition to the lower-level headers, a UDP header, which consists of the packet length, source and destination ports, and a checksum. UDP ports are 16-bit entities, so a maximum of 65535 different endpoints are possible within a single IP address.

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.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

For AIX:
IBM Corporation
Department LRAS, Building 003
11400 Burnet Road
Austin, Texas 78758-3498
U.S.A.

For Linux:
IBM Corporation
Department LJEB, MS P905
2455 South Road
Poughkeepsie, New York 12601-5400
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements, or other publicly-available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility, or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

AIX
AIX 5L
@server
eServer
IBM
IBM(logo)
IBMLink
LoadLeveler
POWER
pSeries
RS/6000
SP

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be the trademarks or service marks of others.

Index

A

- active message 55
- active messages 12
- address-related functions 10
- advantages of LAPI 7
- atomicity 5
- audience of this book xiii

B

- bibliography xiv
- blocking calls 4
- books
 - RSCT xiv
- buffer
 - data 3

C

- C bindings 66
- calls
 - blocking 4
 - non-blocking 4
- coexistence 28
- communication behaviors
 - of LAPI 4
- communication operation
 - completion of 4
- completion
 - of communication operation 4
- completion detection 15
- constants 283
- context, LAPI 3
- conventions
 - terminology xiv
 - typographic xiii
- counters 4

D

- data buffer 3
- data transfer 12, 14, 35
 - non-contiguous 37
 - strided 41
 - vector 37
- datatypes 279
 - LAPI_GEN_GENERIC 39
 - LAPI_GEN_IOVECTOR 40
- definitions 287
- destination 3

E

- environment
 - querying 10
 - setting up 10

- error handling 5
- event notification 15
- examples
 - subroutine 121
- execution model
 - IP/US 255

F

- failover 6, 103
- feedback
 - product-related 254
- FORTTRAN bindings 66
- functional overview 9
- functions
 - address-related 10
 - get 11
 - pull 11
 - push 11
 - put 11
- functions of LAPI 9

G

- get functions 11
- glossary 287

H

- handle, LAPI 3
- handlers 4

I

- initialization 10
- instance, LAPI 3
- interrupt mode 6
- introduction 3
- IP/US
 - execution model 255
- ISO 9000 253

L

- LAPI
 - characteristics 3
 - communication behaviors 4
 - segment registers 285
- LAPI constants 283
- LAPI context 3
- LAPI datatypes 279
- LAPI handle 3
- LAPI instance 3
- LAPI size limits 283
- LAPI_Addr_get subroutine 126
- LAPI_Addr_set subroutine 128
- LAPI_Address subroutine 130

- LAPI_Address_init subroutine 132
- LAPI_Address_init64 134
- LAPI_Amsend subroutine 136
- LAPI_Amsendv subroutine 143
- LAPI_Fence subroutine 149
- LAPI_GEN_GENERIC 39
- LAPI_GEN_IOVECTOR 40
- LAPI_Get subroutine 151
- LAPI_Getcptr subroutine 154
- LAPI_Getv subroutine 156
- LAPI_Gfence subroutine 161
- LAPI_Init subroutine 163
- LAPI_Msg_string subroutine 169
- LAPI_Msgpoll subroutine 171
- LAPI_Nopoll_wait subroutine 236
- LAPI_Probe subroutine 174
- LAPI_Purge_totask subroutine 238
- LAPI_Put
 - with shared memory 257
 - without shared memory 258
- LAPI_Put subroutine 176
- LAPI_Putv subroutine 179
- LAPI_Qenv subroutine 184
- LAPI_Resume_totask subroutine 240
- LAPI_Rmw subroutine 188
- LAPI_Rmw64 subroutine 192
- LAPI_Senv subroutine 196
- LAPI_Setcptr subroutine 198
- LAPI_Setcptr_wstatus subroutine 242
- LAPI_Term subroutine 201
- LAPI_Util subroutine 203
- LAPI_Waitcptr subroutine 217
- LAPI_Xfer structure types 220
- LAPI_Xfer subroutine 219
- lapi_xfer_type_t 220
- lock sharing 6, 83
- LookAt xv

M

- message ordering 5
- messages 5
- migration 28
- mpxlf_r 67

N

- non-blocking calls 4
- non-contiguous data transfer 37

O

- operations
 - get 11
 - pull 4, 11
 - push 4, 11
 - put 11
 - remote read-modify-write 13
- origin 3
- overview of LAPI 9

P

- polling mode 6
- prerequisite information xiv
- prerequisite knowledge for this book xiii
- product-related feedback 254
- profiling 6
 - export file 67
 - LAPI name-shift 66
 - library 67
 - shared library 67
- profiling library 66
- programs
 - compiling 71
 - running 71
- progress 5, 15
- publications
 - RSCT xiv
- pull operations 4, 11
- push operations 4, 11
- put functions 11

R

- recovery 6, 103
- related information xiv
- remote read-modify-write operations 13
- RSCT
 - books xiv
 - feedback 254
 - publications xiv
 - version 253
- runtime environment
 - querying 10
 - setting up 10

S

- sample subroutine 121
- sharing locks 6, 83
- size limits 283
- source 3
- standalone operation 6
- statistics 6
- strided data transfer 41
- striping 6, 103
- subroutine sample 121
- subroutines 125, 235
 - LAPI_Addr_get 126
 - LAPI_Addr_set 128
 - LAPI_Address 130
 - LAPI_Address_init 132
 - LAPI_Address_init64 134
 - LAPI_Amsend 136
 - LAPI_Amsendv 143
 - LAPI_Fence 149
 - LAPI_Get 151
 - LAPI_Getcptr 154
 - LAPI_Getv 156
 - LAPI_Gfence 161
 - LAPI_Init 163

subroutines (*continued*)

- LAPI_Msg_string 169
- LAPI_Msgpoll 171
- LAPI_Nopoll_wait 236
- LAPI_Probe 174
- LAPI_Purge_totask 238
- LAPI_Put 176
- LAPI_Putv 179
- LAPI_Qenv 184
- LAPI_Resume_totask 240
- LAPI_Rmw 188
- LAPI_Rmw64 192
- LAPI_Senv 196
- LAPI_Setcptr 198
- LAPI_Setcptr_wstatus 242
- LAPI_Term 201
- LAPI_Util 203
- LAPI_Waitcptr 217
- LAPI_Xfer 219

T

- target 3
- termination 10
- terminology 287
- terminology conventions xiv
- trademarks 292
- types
 - LAPI_GEN_GENERIC 39
 - LAPI_GEN_IOVECTOR 40
- typographic conventions xiii

V

- vector data transfer 37
 - strided 41
- version
 - of RSCT 253

Readers' comments – We'd like to hear from you

IBM Reliable Scalable Cluster Technology for AIX 5L
LAPI Programming Guide

Publication No. SA22-7936-02

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	<input type="checkbox"/>				

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	<input type="checkbox"/>				
Complete	<input type="checkbox"/>				
Easy to find	<input type="checkbox"/>				
Easy to understand	<input type="checkbox"/>				
Well organized	<input type="checkbox"/>				
Applicable to your tasks	<input type="checkbox"/>				

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? Yes No

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

Name

Address

Company or Organization

Phone No.



Fold and Tape

Please do not staple

Fold and Tape



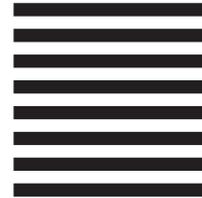
NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Department 55JA, Mail Station P384
2455 South Road
Poughkeepsie NY 12601-5400



Fold and Tape

Please do not staple

Fold and Tape



Program Number: 5765-E62, 5765-G03

SA22-7936-02

