4 Program Communication: PathLib

Chapter 1 provided a brief overview of program communication on the iWarp system. This chapter describes the Pathway Library (PathLib). This library contains a set of calls used to implement program-based communication. Appendix B provides thorough descriptions of all PathLib calls. Appendix D provides definitions for PathLib terminology.

NOTE

It is essential to have a clear understanding of the iWarp architecture to effectively design programs using the PathLib communication calls. For a complete description of the iWarp architecture, refer to the *iWarp Macroarchitecture Specification*.

Communication Overview

Each cell's communication agent maintains a hardware structure called the Pathway Control Table (PCT). Together, the communication agent and PCT provide the mechanisms necessary to support logical channels, which are the means by which cells communicate over the array.

The PCT contains a collection of PCT entries. The PCT entry is the primary element of a logical channel. A logical channel is a chain of PCT entries that link a sender cell to a receiver cell. All cells in the path from the sender cell to the receiver cell contribute PCT entries to support a logical channel, as do the sender and receiver cells.

When a logical channel is being created, it competes with other logical channels for cell resources. Once resources are allocated to a logical channel, they carry data sent over this logical channel until the channel is destroyed by the deallocation of its PCT resources.

The communication agent PCT contains twenty entries that provide hardware support for a maximum of twenty simultaneous logical channels per cell. These resources are shared between the runtime system and PathLib. The runtime system uses four entries, and sixteen are available to the PathLib user.

For more information about the iWarp processor architecture, refer to the *iWarp Macroarchitecture Specification*.

PathLib Overview

PathLib supports the following communication operations:

- the creation and termination of logical channels, called connections
- user creation and termination of messages over logical channels, known as messages and marks
- user management of streamgates
- spooled block data movement over logical channels, called spooling
- the creation and termination of aggregate logical channels, called joining and splitting, and known in PathLib as express

Connections

A connection is an instance of a logical channel created by PathLib. A connection is known only at its endpoints: one endpoint on the origin cell of the connection and one on the destination cell. Connections are created and destroyed via the production and consumption of connection headers and trailers. The PathLib connection calls support the creation and termination of connections.

PathLib includes the following connection calls:

pl_send_oc()	Opens a connection with the calling cell as its origin and the route and destination specified via the user-supplied arguments. The connection is created by the sending of a connection header encoded with routing and destination information.
pl_recv_oc()	Accepts a connection header on the calling cell. The use of this call assumes that some cell sent a connection with this cell as its destination. Upon return from this call, the creation of the connection is complete, and all resources to support it are established.
pl_send_cc()	Closes a connection originating from the calling cell. The communication agent resources that comprise the connection are deallocated by the sending of a connection trailer.
pl_recv_cc()	Concludes a connection on the calling cell. The connection trailer is consumed, thus freeing the last resources associated with the connection.

The index of the PCT entry for a connection is returned when a create or accept connection call is called. This value is the user's handle on the connection. The user is expected to supply this value to identify connections to PathLib calls that perform operations on connections.

Messages

A message is a block of data transmitted over a connection. All data must be sent within the context of a message. It is up to the user to establish messages by sending and receiving message headers, and to end messages by sending and receiving message trailers.

PathLib includes the following message calls:

pl_send_om()	Opens a message by sending a message header. The message header will follow the path of the connection until it is stopped. A message header stops when reaching the destination cell of the connection or aggregate logical channel of which the connection is a constituent (refer to the description of express in Appendix B). A message header may be stopped or consumed by any cell along the path of the connection. Cells along the connection may request that generic message headers be stopped, or a message header may be matched by a cell's address match cam, which always causes the header to stop.
pl_recv_om()	Allows the calling cell to receive a message header from a connection. This action can be performed by any cell along the path of the logical channel that stops the header.
pl_send_cm()	Allows the calling cell to end a message by sending a message trailer over a connection. Like a message header, a message trailer may be stopped and consumed by any cell along the path of a logical channel. If not stopped by any intermediate cell, the message trailer will flow to the ultimate destination of a logical channel.
pl_recv_cm()	Closes a message on the calling cell by consuming the message trailer.

The consume calls are also provided in forms that cause the header or trailer to be sampled (copied), but not consumed. Refer to Appendix B for a complete description of these forms.

Message headers and trailers can be recognized by the communication agent (refer to the description of express in Appendix B). The semantics associated with this capability are up to the user. There are some guidelines that should be observed regarding the use of message headers and trailers, and these are discussed in the section "Using Messages," later in this chapter.

User Data Movement

PathLib provides streamgate binding operations to support user-programmed send and receives and spooled block data movement. Single-word and double-word streaming are provided by the systolic library, providing calls such as pl_sendi(), pl_sendf(), pl_sendd(), pl_recvi(), pl_recvf(), and pl_recvd(). Each of these calls sends or receives a single piece of data from a streamgate. The data can be integer, floating-point, or double-precision floating-point. In addition, there are sampling and non-blocking routines available.

Streaming

Streaming is the user-programmed word-by-word transfer of data using streamgates. A streamgate is the mechanism that allows the computation agent to access a connection. A streamgate is bound to a connection for input, output, or both. The send and receive operations may be performed to or from the connection by specifying the streamgate.

There are two streamgates. You can use each streamgate for both input and output. The streamgates are PL_GATE0 and PL_GATE1. Streamgates are also used by the connection, message, and mark calls to perform the data production and consumption necessary to implement these operations. The user is expected to supply a streamgate specifier to these calls to identify the gate to be used to perform the operation. For some of these calls, the streamgate binding will be changed as a side-effect of the requested operation.

PathLib includes the following streamgate calls:

pl_bind_input_gate()	Bind an input gate to an inbound connection.
pl_bind_output_gate()	Bind an output gate to an outbound connection.
pl_save_input_binding()	Save the current binding for an input gate in a user-supplied variable.
pl_save_output_binding()	Save the current binding for an output gate in a user supplied variable.
pl_restore_input_binding()	Restore an input gate binding from a user-supplied variable.
pl_restore_output_binding()	Restore an output gate binding from a user-supplied variable.
pl_unbind_input_pct_entry()	Unbind either or both input streamgates from an inbound connection.

Spooling

Spooling allows the iWarp hardware to perform send and receive operations to or from messages and local memory without programmed word-by-word copy operations. Spooling can be thought of as a DMA operation between local memory and logical channels.

PathLib provides a simple interface to the iWarp spooling mechanisms to allow you to use spooling to send and receive blocks of data.

PathLib includes the following spool calls:

pl_bind_spool()	Request the use of a spoolgate for a connection. This call allocates a spoolgate and configures it.
pl_start_spool()	Use a previously bound spoolgate to begin data movement through a connection.
pl_spool_complete()	Determine if a requested spool transfer is complete.
pl_unbind_spool()	Deallocate a spool.

NOTE

Use of spooling can cause programs that contain multiclock operations as the third word of a compute-and-access instruction to be unreliable. For a complete description of this problem and associated programming suggestions, please refer to the `spooling` manual page.

Aggregate Logical Channels (Express Calls)

The user may form aggregate logical channels by plugging connections together to form chains. The connecting cells may selectively participate in communications over the chain. Cells may join and split the connections as necessary to suit the requirements of the application.

When a cell joins connections, data is forwarded from the inbound neighbor to the outbound neighbor. Forwarding is handled by the communication agent, allowing the computation agent to perform other work. Forwarding is stopped by a user-specified sentinel or as a result of the arrival of a matching message header. The various sentinels are message header or trailer, software mark, connection trailer, or when the cell is between two messages on the channel.

PathLib express calls create and destroy aggregate logical channels by providing operations to set the sentinel, forward data, and resume programmed data movement after forwarding has been stopped.

PathLib includes the following express calls:

pl_set_sentinel()	Set the sentinel to stop forwarding.
pl_get_sentinel()	Obtain the sentinel in effect.
pl_express()	Request that data be forwarded.
pl_cancel_requested_express()	Attempt to cancel a previously requested express operation.
pl_stopped()	Determine if forwarding has been stopped.
pl_between_messages()	Determine if the cell is between messages.
pl_synced()	Determine if data production can be resumed over an outbound connection that was joined to an inbound connection using an express operation.

Software Marks

A software mark is a data word that can be detected by the communication agent. The semantics associated with software marks are defined by the user. You can specify an express sentinel that causes express forwarding to stop when a software mark arrives at a cell. The following calls allow the user to send, receive and sample software marks:

pl_send_mark()	Send a software mark.
pl_recv_mark()	Receive a software mark.
pl_sample_mark()	Copy a software mark.

Support Calls

PathLib also provides calls to initialize an instance of the library linked to a user's program and handle and print errors reported by PathLib calls.

PathLib includes the following initialization calls:

pl_init()	Initialize the iWarp hardware for use by PathLib. The user specifies which of the PCT entries and spoolgates available on a cell may be used by the PathLib interface for this instance of the library.
pl_rpe_configure()	Establish which of the PathLib PCT entries in the calling cell's neighbors are to support inbound/express logical channels propagated to them by this cell, and which local PCT entries this cell can use for creating connections.
pl_load_amc()	Load an address matchcam and cam mask on this cell.

For a description of the error handling operations provided by PathLib, refer to the description of the plerror calls in Appendix B.

Guidelines for Using PathLib

Because of the flexibility required to use PathLib for program communication on the iWarp array, there are few strictly-enforced policies. While this provides the needed flexibility, it also creates hazards for the programmer.

The following guidelines will help avoid some basic pitfalls involved in designing programs for the iWarp array using PathLib communication.

Cell Naming

Cell naming and addressing are based upon the hardware address-matchcam. Each cell has four matchcams that together control what connection and message headers the cell recognizes.

The user has access to a portion of one cell matchcam. The PathLib call pl_load_amc() provides the user with control over the low-order 18 bits of the PathLib matchcam id and mask. Thus, the user has exclusive access to one quarter of the available match address space. The other three quarters is reserved for iW/RTS and PathLib internal use.

No inter-cell synchronization is provided by PathLib, such as that required to ensure that a receiver's address-matchcam is loaded before a connection reaches the cell. This synchronization is the user's responsibility.

PathLib doesn't address any aspect of how cells are named. This is up to the runtime system and the user. The current iW/RTS gives every cell a canonical name.

Using Connections

To reach its destination, a connection must be granted resources along the route of the logical channel. The propagation of the connection header will stall if it encounters a resource shortage at any cell along its route. If resources become available, the header will then proceed. Once a connection is established, traffic over the connection is guaranteed a share of the available physical bus capacity in all processors along the path of the connection.

The routing of connections is a result of information encoded in the connection header and configuration information installed on the cells. The user has responsibility for correct management of both of these details.

Before you can create connections, proper recipient PCT entry (RPE) configuration must be in place within the array. If not properly configured, cells may forward connection headers in unpredictable directions.

You can create connections that traverse the array in any way you wish. Some important aspects of routing to consider are:

- ensuring that routes are deadlock free
- minimizing the cost of routes in terms of logical channel resources
- maximizing the use of available physical pathway bus capacity of the array
- enhancing the re-usability of cell and array programs by characterizing, consolidating, and standardizing the entry and exit points of inter-processor communications

In particular, the issue of deadlock must be addressed when setting up connections among cells. Due to a lack of synchronization and pathway cleanup primitives, you should only use static configurations of connections.

Assuming static connection allocations, observing the following guidelines will help avoid deadlock situations when setting up connections:

- All cells must have names before connections are created. iW/RTS installs a canonical name for every cell before any user codes may be loaded and run on the array. An alias may be established for a cell through the use of pl_load_amc(). For a description of the cell naming conventions, refer to the iWarp `numbering` man page.

NOTE

Aliases should not be used for connection setup. They should only be used for messages, and the call to install the alias should be made before any connection setup code is run. There is no inherent synchronization available to ensure that cells have their names in place before communication requiring these names occurs.

- All cells must have established RPE configurations before they begin connection setup. These configurations are established by calling pl_rpe_configure(). The configuration established must provide sufficient resources on all cells to allow all connection setup work to complete.

NOTE

Once RPE configurations are established, cells may not change them without restarting the system.

- Connection setup and data movement over connections must be handled in two distinct and separate steps. First, connection setup should be performed in its entirety. Second, data movement can occur. Observing this method avoids deadlock that might occur in cases where cells create connections to cells from which they also accept connections.

  For example, if two cells create connections to each other, then start their respective send operations before accepting their partner's connection, they will deadlock if there isn't sufficient PCT entry queue space between each other to hold the transmissions.

  A simple way to avoid problems of this kind is to program a cell to create all the connections for which it will be the origin, then accept all the connections for which it will be the destination. After both of these steps are complete, data movement can occur.

- When using pl_recv_oc() and more than one connection header is pending at the cell, the header accepted is arbitrary. The user must program any protocol semantics associated with connections. For example, this might involve accepting the connection and processing a user-defined protocol message that follows the header.

- Connections must be terminated using the same cells that created or accepted them.

Using Messages

The communication agent does not require that data transmitted over logical channels be encapsulated by messages. Message headers and trailers may occur arbitrarily, but a more restrictive usage model should be used to allow modular development of array programs, so independently-developed programs and systems can later be used together.

Observe the following guidelines when using messages:

- Messages should not be nested between any two cells on a logical channel.
- Data words and software marks must not be found outside of a message by any cell participating in a logical channel.
- Cells may produce, consume, and forward (express) message headers and trailers arbitrarily, but between any two cells on a logical channel, data should always be within the context of a message.

  For example, a cell other than the cell that accepted a message header may stop and consume the message trailer for a message. If this is done, the message is complete through the cell that consumes the trailer. If the cell that accepted the header is farther along the path of a logical channel than the cell that consumes the message trailer, the message is closed between the origin of the trailer and the cell that consumed it. However, the message remains open from the cell that intercepted the trailer to the cell that consumed the header. Similarly, if the origin of the trailer is beyond the origin of the header, a message is still open from the cell that was the origin of the header to the cell that is origin of the trailer.

Express

Observe the following guidelines when using express:

- A connection trailer should not be allowed to be forwarded through the cell that is the ultimate destination of the connection.
- A single connection may not be joined to more than one partner at a time.
- Joined connections must be split before streaming or spooling may be resumed over the connections. Downstream neighbor synchronization must be performed before streaming or spooling is resumed on the outbound partner after a split is detected.
- The job of terminating connections that have been members of aggregate logical channels is an array-level activity, not a cell-level activity. Connections are terminated using individual cells. The termination of connections can have array-wide consequences, particularly in the case of terminating connections that have been used to form aggregate logical channels. A basic guideline to observe is that aggregate logical channels must be terminated by splitting them into their constituent connections and terminating each of these using the PathLib connections interface.

PathLib Example

This section contains a simple PathLib example in which a single logical channel is produced and consumed by a cell. The channel is sent out in the Xright direction from the cell, and returns to the cell from the Xleft direction, after traveling around the path of cells that comprise the row of the array in which the cell resides.

The example assumes the following:

- all cells in the array have names that match release 3.0 iW/RTS names

NOTE

For a description of the cell naming conventions, refer to the iWarp `numbering` man page.

- the systems software has configured cell logical channel resources and has initialized the PL module.

```
/* The channel is sent out to the right and is returned to this cell
 * on the left after traveling around the entire path of cells that
 * make up this cell's row in the iWarp array. */

#include <stdio.h>
#include <pathlib/pl.h>
#include <iwsys/getcfg.h>

main()
{
  pl_pcti   i_chan;      /* logical channel descriptor   */
  pl_pcti   o_chan;      /* logical channel descriptor   */
  unsigned  dest_id[1];  /* name of the destination cell */
  unsigned  header;      /* storage for input header     */
  int i;
  int tmp;
  struct iwcfg  cfg;

  /* Find out which cell this is. */
  getcfg(&cfg);

  /* Initialize the PL module. Use two PCTEs (one in, one out). */
  pl_init(0x3);

  /* Reserve one pcte in the cell to the
        right and one in your own cell */
  pl_rpe_configure(0, 0x1, 0, 0, 0x2);

  /* Only cell 0 is going to do anything. */
  if (cfg.cellid != 0)  {
    pl_exit();
    exit(0);
  }

  /* Create a logical channel from this cell to itself
      in the X-right direction. */
  dest_id[0] = cfg.cellid|PL_DESTINATION;
  o_chan = pl_send_oc(PL_GATE0, PL_DIR_XR,
                          dest_id, 1);
  pl_send_om(PL_GATE0, PL_NOMATCH);

  /* Accept the logical channel just created. */
  i_chan = pl_recv_oc(PL_GATE1, &header);
  if (pl_recv_om(PL_GATE1, &header) != PL_SUCCESS)  {
    fprintf(stderr, "Open message not received.\n");
    exit(1);
  }

  /* Send and receive 1000 words over the channel. */
  printf("Cellid %d: start\n", cfg.cellid);
  for (i=0; i<1000; i++) {
    pl_sendi(PL_GATE0, i);
    tmp = pl_recvi(PL_GATE1);
    if (tmp != i)  {
```

```
        fprintf(stderr, "Error: sent %d, recd %d\n",
               i, tmp);
        exit(1);
      }
    }
    printf("cellid %d: finish\n", cfg.cellid);
    /* Terminate the output channel */
    pl_send_cm(PL_GATE0);
    pl_send_cc(o_chan, PL_GATE0);

    /* Terminate the input channel */
    if (pl_recv_cm(PL_GATE1) != PL_SUCCESS)  {
      fprintf(stderr, "Close message not received.\n");
      exit(1);
    }
    if (pl_recv_cc(i_chan, PL_GATE1) != PL_SUCCESS)  {
      fprintf(stderr, "Close connection not received.\n");
      exit(1);
    }
    /* Give the pathways time to empty out. */
    pl_exit();
    exit(0);
  }
```

This example provides a simple illustration of programming on the iWarp system using PathLib communication calls. Chapter 5 provides further examples of developing and executing programs on the iWarp system.