IBM System Blue Gene Solution
BlueGene/Q Code Development and Tools Interface

Compute Node Kernel, runjob, and CIOS tool interfaces

Tool messages

Tool commands

John Attinella
Sam Miller
Gary Lakner

Redpaper

ibm.com/redbooks

International Technical Support Organization

IBM System Blue Gene Solution: Blue Gene/Q Code Development and Tools Interface

July 2012

REDP-4659-00

Note: Before using this information and the product it supports, read the information in “Notices” on page v.

First Edition (July 2012)

This edition applies to Version 1, Release 1, Modification 0 of IBM System Blue Gene/Q Solution (product number 0207-200).

Contents

Notices

Trademarks

Preface

This book is one in a series of IBM® publications written specifically for the IBM System Blue Gene® supercomputer, Blue Gene/Q®, which is the third generation of massively parallel supercomputers from IBM in the Blue Gene series. This IBM Redpaper™ publication helps you develop tools, such as a debugger, that can be used on the IBM Blue Gene/Q platform to control and monitor application threads and processes.

The team who wrote this paper

Gary Lakner is a Staff Software Engineer for IBM Rochester on assignment in the ITSO. He is a member of the IBM Blue Gene Support Team in the IBM Rochester Support Center, where he specializes in both Blue Gene hardware and software and performs customer installations. Prior to joining the Blue Gene team, Gary supported TCP/IP communications on the IBM System i® platform. Gary has been with IBM since 1998.

John Attinella is a Senior Software Engineer for IBM Rochester. He graduated from Lafayette College in 1981 with a B.S. in Electrical Engineering. He is a member of the Blue Gene Kernel Development Team. John has worked at IBM for over 30 years in hardware and software processor development. Prior to joining the Blue Gene team, John designed and developed system software on various platforms including IBM S/390®, IBM System i, and Power hypervisor.

Sam Miller is a Staff Software Engineer for IBM Rochester. He graduated from Iowa State University in 2003 with a B.S. in Computer Engineering and received his M.S. in 2006. He is currently the High Level Control System Team Leader for Blue Gene software. He has worked on all three generations of Blue Gene.

Thanks to the following people for their contributions to this project:

Thomas Budnik
Blue Gene Development, IBM Rochester

Annette Bauerle
International Technical Support Organization

Now you can become a published author, too!

Here’s an opportunity to spotlight your skills, grow your career, and become a published author—all at the same time! Join an ITSO residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction, as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our papers to be as helpful as possible. Send us your comments about this paper or other IBM Redbooks® publications in one of the following ways:

- Use the online **Contact us** review Redbooks form found at:

  ibm.com/redbooks

- Send your comments in an email to:

  redbooks@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

Stay connected to IBM Redbooks

- Find us on Facebook:

  http://www.facebook.com/IBMRedbooks

- Follow us on Twitter:

  http://twitter.com/ibmredbooks

- Look for us on LinkedIn:

  http://www.linkedin.com/groups?home=&gid=2130806

- Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:

  https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm

- Stay current on recent Redbooks publications with RSS Feeds:

  http://www.redbooks.ibm.com/rss.html

1

Code Development and Tools Interface

The Blue Gene/Q Code Development and Tools Interface (CDTI) provides the support for launching and controlling tools on a Blue Gene system. The Blue Gene/Q CDTI is a major update to the interfaces provided on prior Blue Gene systems. The Blue Gene/Q CDTI improves efficiency, adds additional function, and takes into account the hardware differences in a Blue Gene/Q system. A tool written for a prior Blue Gene system must be ported to the new interfaces described in this document.

1.1 Requirements

Tools can be supplied with the Blue Gene system software, provided by other vendors, or developed by Blue Gene customers. The primary example of a tool is a debugger. A tool operates in the context of a Blue Gene job. A job can consist of as little as a single node to a large block on tens or hundreds of racks with thousands of nodes. A tool executes on the I/O nodes of Blue Gene, so it must be in a file system accessible from the I/O node. A user can run any tool that they have authority to. A tool can access and modify the processor state and processor memory.

Tool launching has the following features:

- Multiple tools can be launched for a job (maximum of four).
- Tools can be launched on a subset of the I/O nodes servicing a job.
- Tools can be launched at job start or they can be launched after a job begins (referred to as *tool attach*).
- Tools can be terminated before a job ends and can end at different times on different I/O nodes.

1.2 System overview

Figure 1-1 on page 3 illustrates how an external tool interfaces with the system. The illustration shows the portion of the system that is the focus of this paper: The interaction between the compute nodes and I/O nodes on the torus network, the I/O nodes and the service node, front end node and the file servers on the functional network.

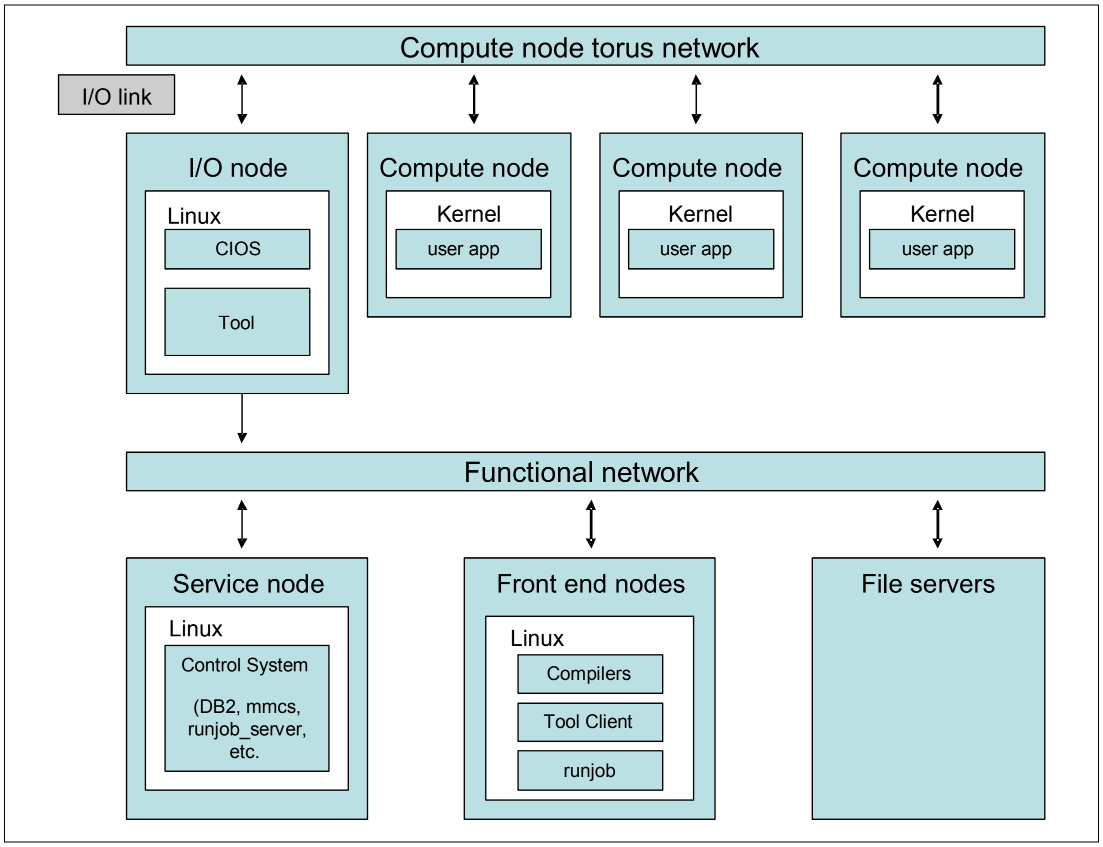

Figure 1-1 Tool interface architecture

Keep in mind that this figure is only a partial representation of the Blue Gene/Q system architecture:

- Multiple compute nodes use the services provided by an I/O node. The exact ratio depends upon the system configuration.
- Compute nodes and I/O nodes communicate over the torus network. A special I/O link connects an I/O node to the compute node torus network.
- Users interact with the system from a front end node (also referred to as *login node*). The front end node has compilers for generating user applications and the **runjob** executable, which is used to launch jobs.
- **runjob** communicates with the Control System running on the service node to submit the job to the system. Job submission is dependent upon what job scheduler is in use. The job scheduler is selected and installed by the customer.

If the user wants to launch a tool during their job, they must tell the Control System. The **start_tool** executable tells the Control System to start tools on all of the I/O nodes that are servicing the job to be controlled or monitored. The tools communicate with the Common I/O Services (CIOS) daemons running on the I/O nodes to pass messages back and forth to the compute nodes, where the user code is running. No user code runs on the I/O node; instead, it acts as a proxy/manager for compute nodes.

A tool running on the I/O node communicates using a standard TCP/IP connection on the functional network. For example, a tool server on the I/O node communicates with a tool client on the front end node. In addition, a tool server on one I/O node communicates with a tool server on a different I/O node using TCP/IP.

Tools are free to start additional processes on the I/O node. On Blue Gene/Q systems with a high compute node to I/O node ratio, this approach can be an effective way to divide the work of managing many compute node processes. The interface for sending and receiving messages from compute nodes allows for messages to be in flight to multiple compute nodes simultaneously.

1.2.1 Compute nodes

The Compute Node Kernel (CNK) can support a range of process configurations ranging from one process with 64 hardware threads to 64 processes each with one hardware thread. The 64 hardware threads used by application code are spread across 16 cores, each containing four hardware threads. The CDTI supports jobs that occupy all nodes in the system down to jobs that occupy a single node. The CDTI does not support attachment to application agent processes that might be running on the 17th core. Each hardware thread can support a fixed number of threads (the current number is five). Each pthread is managed as a runnable entity by the kernel. After a pthread is bound to a hardware thread, it does not move from that hardware thread unless acted upon by a set affinity action, such as pthread_setaffinity. The main thread of a process cannot be moved to another hardware thread. It remains on the original hardware thread it was started on.

1.2.2 I/O nodes

CIOS provides services on the I/O node that are used by the Control System and CNK when running jobs. A tool is started on the I/O node by the job control daemon. The I/O node provides a fairly complete Red Hat Enterprise Linux (RHEL) 6 environment with TCP/IP connectivity to the functional network and disk storage through file systems mounted over network file systems.

The I/O node Linux environment can be customized to suit individual Blue Gene installations.

2

The runjob tool interface

The primary end-user interface to the system is the **`runjob`** command. It is expected that tools interface with **`runjob`** as defined by the MPI debug specification:
`http://www.mcs.anl.gov/research/projects/mpi/mpi-debug/`

This interface allows for a tool front end to attach to an MPI job and learn about the job in a standard way.

Note: Most references in this chapter assume that **`runjob`** is a command. It is also possible for this functionality to be exposed through your job scheduler, which can replace **`runjob`** with its own interface. Consult your job scheduler documentation for further details.

2.1 Variables and functions

The MPI specification defines a set of variables that a tool can use to interface with **runjob** primarily for the purposes of debugging. The following sections describe the variables that are implemented in the **runjob** executable.

2.1.1 MPIR_Breakpoint

MPIR_Breakpoint is a routine that **runjob** calls at points of interest. Specifically, the two points are:

- After the MPIR_proctable is initialized and the I/O nodes are told to start the tools, **runjob** sets MPIR_debug_state to MPIR_DEBUG_SPAWNED and calls this function. This process tells the tool front end that it can read the proctable.
- If a signal tries to kill **runjob**, the signal handler in **runjob** sets MPIR_debug_state to MPIR_DEBUG_ABORTING and calls this function. This process allows the tool front end to perform cleanup before **runjob** exits. Some signals cannot be caught, so it is not completely reliable.

Global subroutine definition:

```
extern “C” void MPIR_Breakpoint();
```

2.1.2 MPIR_debug_state

The MPIR_debug_state variable is set to a value before invoking MPIR_Breakpoint to indicate why the call is occurring. The two values are:

- MPIR_DEBUG_SPAWNED: The tools were told to start on the I/O nodes.
- MPIR_DEBUG_ABORTING: The **runjob** program is ending due to a signal being sent to it.

Macro definitions:

```
extern "C" volatile int MPIR_debug_state;
#define MPIR_DEBUG_SPAWNED 1
#define MPIR_DEBUG_ABORTING 2
```

2.1.3 MPIR_proctable

The MPIR_proctable is an array of MPIR_PROCDESC structures. Each entry in the array represents a process within the job. The entries in the array are sorted by the MPI rank of the processes in COMM_World. Depending on the configuration of the job, multiple ranks can exist within a compute node.

Global variable definition:

```
extern "C" MPIR_PROCDESC *MPIR_proctable;
```

The **runjob** program gets information about the compute nodes that form the job. The content in Example 2-1 on page 7 is used to build the proctable.

Example 2-1 MPIR_PROCDESC type definition

```
typedef struct {
  char * host_name; /* IP address of I/O node controlling the
              compute node that process is running on */
  char * executable_name;/* name of executable */
  int    pid;/* A number representing a compute node within the job */
} MPIR_PROCDESC;
```

The pid is a value that corresponds to a symbolic link in the /jobs/toolctl_node/ directory within the I/O node to the data channel of a compute node. By using the index into this array of objects and by using the pid field within the objects, the association between ranks and compute nodes can be determined. For more information, see 3.1.2, “Job information” on page 12.

2.1.4 MPIR_proctable_size

The MPIR_proctable_size variable is set to the number of compute node processes that are part of the MPI job.

Global variable definition:

```
extern "C" int MPIR_proctable_size;
```

2.1.5 MPIR_being_debugged

The MPIR_being_debugged variable is set by the tool client to tell **runjob** that a tool is attached. It is used for the case when **runjob** is started under debug, and when the tool client attaches to an already running **runjob**. This variable must be reset back to 0 when a tool client finishes the debugging session and wants to detach. Resetting the variable allows other tools to attach to the same job in the future.

Global variable definition:

```
extern "C" volatile int MPIR_being_debugged;
```

2.1.6 MPIR_executable_path

The MPIR_executable_path is set to the full path and filename of the tool server executable. It is set by the tool client and consists of a null terminated string. The maximum length is 256 bytes, including the null character for termination.

Global variable definition:

```
extern "C" char MPIR_executable_path[256];
```

This is a Blue Gene unique field.

2.1.7 MPIR_server_arguments

The MPIR_server_arguments variable specifies the arguments to pass to the tool server at launch on the I/O node. It is set by the tool client and consists of an array of null terminated strings with the last element of the array set to NULL to indicate the end of the array. The maximum length is 1024 bytes. The format of the array is described in 3.1.2, “Job information” on page 12.

Global variable definition:

```
extern "C" char MPIR_server_arguments[1024];
```

This is a Blue Gene unique field.

2.1.8 MPIR_subset_attach

The MPIR_subset_attach fields specify the subset of MPI ranks on which to launch a tool server. The contents of the string specify the ranks that must have tool servers launched for them. A subset specification is a space-separated list, where each element in the list can have one of three forms:

rank	Means that rank only.
rank1-rank2	Means all ranks between rank1 and rank2, inclusive.
rank1-rank2:stride	Means every strideth rank between rank1 and rank2.

A rank specification can be either a number, "$max", or "max". Both variables represent the last rank in the MPI job, for example, the following syntax attaches to ranks 1, 2, 4, 5, 6, 7, 9, 11, 13. … and so on. If this field is left empty by the tool, the default behavior is to launch the servers on all I/O nodes.

```
1 2 4-6 7-$max:2
```

Global variable definition:

```
extern "C" const char* MPIR_subset_attach;
```

This is a Blue Gene unique field.

2.2 The runjob program flow

Tools can be started from within the job or attached to an already running job. The following sections describe the program flow for both cases.

2.2.1 Starting a job under tool control

A tool can be started by the system software before the job gains control, which can be useful for debugging applications before the main() function. This section outlines the steps required to start a job under the control of a tool:

1. The user starts the tool front end, which starts `runjob`.
2. The tool front end sets an early breakpoint at MPIR_Breakpoint.
3. When the breakpoint is hit:
   a. The tool front end sets the MPIR_executable_path and MPIR_server_arguments.
   b. The tool front end sets the MPIR_being_debugged flag.
   c. The program execution continues.
4. The `runjob` program continues execution and checks the MPIR_being_debugged flag. Because the flag is set, `runjob` proceeds as follows:
   a. Instructs the I/O nodes to load the job.
   b. Fills in the MPIR_proctable and MPIR_proctable_size.
   c. Launches the tool daemons on the I/O nodes.
   d. Sets MPIR_debug_state.
   e. Calls MPIR_Breakpoint.

5. The tool front end gets control again because of the call to MPIR_Breakpoint. The tool front end can continue setting up:
   a. The tool front end can read the MPIR_proctable structure.
   b. The tool front end establishes communications with the tools running on the I/O nodes.
   c. The tool front end can attach to compute nodes.
6. When **runjob** is allowed to start executing, it tells the job to start executing:
   a. Compute nodes that the tool front end attached to halt after they execute the first instruction.
   b. Compute nodes that the tool front end did not attach to run normally.

2.2.2 Attaching to a running job

This section outlines the steps required to attach a tool to a job that is already running:

1. The user starts **runjob**.
2. The tool front end attaches to the **runjob** executable and specifies the PID of **runjob**:
   a. A breakpoint at MPIR_Breakpoint is set.
   b. The tool front end sets the MPIR_executable_path and MPIR_server_arguments.
   c. The tool front end sets the MPIR_being_debugged flag.
   d. Program execution continues.
3. The **runjob** program eventually checks the MPIR_being_debugged flag and performs the following steps:
   a. Fills in MPIR_proctable and MPIR_proctable_size.
   b. Starts the tool daemons on the I/O nodes.
   c. Sets MPIR_debug_state.
   d. Calls MPIR_Breakpoint.
4. At this point, the tool front end can read the proctable and establish connections with the tools that are running on the I/O nodes.

2.3 Starting and stopping multiple tools

The **runjob** tool interface presented in 2.2, “The runjob program flow” on page 8 describes how a single tool can interface with a job and start tool daemons on I/O nodes. In many instances, it might be beneficial to start and stop additional tools while a job executes. To accomplish this, the commands **start_tool** and **end_tool** are provided. Sample usage is shown in the following sections.

2.3.1 Starting a tool

A tool can be started after a job is running using the **start_tool** command. In addition to the interactive use, this command exposes the same MPIR_proctable interface described in 2.1.3, “MPIR_proctable” on page 6. The –pid argument can be used instead of the –id. Using the –pid argument places the added restriction that **start_tool** must be invoked from the same front end node that **runjob** is invoked from.

This command is synchronous. It returns control after the tool daemon has started on all I/O nodes requested.

```
$~> start_tool --id 123 --tool /path/to/my_great_tool –args "–one hello –two
world"
tool /path/to/my_great_tool has been started with ID 1.
$~>
```

2.3.2 Stopping a tool

By default, ending a tool daemon delivers SIGTERM, which can be changed with the –signal argument. This command is asynchronous. It returns control after the signal is delivered but before the tool ends.

```
$~> end_tool --id 123 --tool 1
tool 1 has been stopped
$~>
```

3

CIOS tool interface

When a tool is started by the **runjob** interface, a tool process is started on each I/O node. The tool programs then use the Common I/O Services (CIOS) interfaces described in this chapter to monitor and control the compute nodes.

3.1 Job control daemon

The job control daemon (jobctld) running on the I/O node provides the interface that `runjob` uses to manage running of jobs on the I/O and compute nodes.

3.1.1 Starting a tool

The job control daemon starts a tool on the I/O node when it gets a message from the Control System. The message from the Control System includes the full path to the tool executable and a set of arguments to pass to the tool. Multiple tools can be running simultaneously for a job. A tool is started by jobctld using fork() and exec(). A tool is given two environment variables:

- BG_JOBID is set to the job identifier for the job. The value is a string representing an unsigned 64-bit decimal number.
- BG_TOOLID is set to the tool identifier assigned by `runjob`. The value is a string representing an unsigned 32-bit decimal number.

A tool needs both values when sending messages to the Compute Node Kernel (CNK), as described in 4.1, "Tool protocol message format" on page 16.

3.1.2 Job information

Information about the job is placed in the /jobs directory on the I/O node. When a job starts, jobctld creates a directory in /jobs using the job identifier. When a job ends, jobctld removes the directory. The job information directory contains the following objects:

exe	A symbolic link to the executable. The value comes from the `runjob` –exe parameter.
wdir	A symbolic link to the initial working directory. The value comes from the `runjob –cwd` parameter.
cmdline	A file containing the argument strings. Each string is terminated by a null byte. The end of the strings is denoted by two null bytes. The value comes from the `runjob –args` parameters.
environ	A file containing the environment variable strings. Each string is terminated by a null byte. The end of the strings is denoted by two null bytes. The value comes from the `runjob –envs` parameters.
loginuid	A file containing the login user ID value as a string.
logingids	A file containing the login group ID values as strings. Each string is terminated by a null byte. The end of the strings is denoted by two null bytes.
toolctl_rank	A directory with symbolic links to the tool control service data-channel local sockets to enable attaching to a specific rank in the job.
toolctl_node	A directory with symbolic links to the tool control service data-channel local sockets to enable attaching to all ranks in the job. The naming of the symbolic links matches the PID field within the MPIR_PROCDESC structure as discussed in 2.1.3, "MPIR_proctable" on page 6. The MPIR_PROCDESC entries are ordered by rank, and each MPIR_PROCDESC structure contains an I/O node address and a PID value that represents the local socket to the compute node. Therefore, the relationships between processes within a compute node, compute

	nodes serviced by an I/O node, and ranks associated with I/O nodes and processes can be determined.
tools	A directory with symbolic links to the tools that are running. When a tool is started, a symbolic link is added that is named with the tool ID and points to the tool executable path. When a tool ends, its symbolic link is removed.
tools/protocol	A file containing the Code Development and Tools Interface (CDTI) version number associated with the Blue Gene system software as a string.
tools/status	A directory with status files for tools that are running. When a tool is started, a zero-length file is added that is named with the tool ID. A tool updates the modification time of its file to tell jobctld that it is still active. When a tool ends, its status file is removed.

3.1.3 Stopping a tool

Jobctld stops a tool on the I/O node when it gets a message from the Control System or when a job ends. A tool is stopped by sending a SIGTERM signal to the tool process. When a tool receives the signal, it must clean up the state and detach from all compute node processes that it is attached to, cleanup any resources it allocated on the I/O node, and then end.

When a job is ended by a RAS event with an END_JOB control action, `runjob` waits for tools to complete any cleanup to recover from the failure. As long as a tool is updating its status file in the /jobs directory every 60 seconds, `runjob` does not force the job to terminate.

3.2 Tool control service

The CNK communicates with the I/O node over the torus network. On the I/O node, tool control messages are sent and received using an OpenFabrics Enterprise Distribution (OFED) InfiniBand verbs interface. The tool control service manages the torus network interface to a compute node and multiplexes messages from multiple tools. There is one tool control service daemon (toolctld) for each physical compute node. A tool connects to the data-channel local socket of toolctld to exchange tool control messages with the CNK. A local socket is a socket of the AF_LOCAL (or AF_UNIX) address family that supports full duplex message exchanges.

There are two ways to connect to toolctld:

- When a tool is attaching to a specific rank in a job. The toolctl_rank directory in the job information directory contains the path to the data channel local socket for each rank. For example: to connect to rank 3 in job 5072, a tool connects to the local socket /jobs/5072/toolctl_rank/3.
- When a tool is attaching to all ranks in a job. The toolctl_node directory in the job information directory contains the path to the data-channel local socket for each compute node in the job. A tool connects to all of the local sockets contained in the directory.

The tool control service does no processing of a message. It simply transfers messages between the CNK and a tool. After connecting to the data-channel local socket of toolctld (either through toolctl_rank directory or toolctl_node directory), a tool does further operations by reading and writing messages on the same local socket descriptor.

4

Compute Node Kernel interface

A tool communicates to the compute node indirectly through the tool control daemons. The tool control daemon sends and receives messages between the tool and the compute node. The daemon passes these messages unaltered between the tools and the compute nodes. This chapter defines the tool interfaces and interactions with the compute node.

4.1 Tool protocol message format

There are multiple message protocols for communicating with the Compute Node Kernel (CNK). The protocols are differentiated by the service type of that message. All messages have a standard header that includes the service, protocol version, and a type that further differentiates the message within that service. Tools use the tool control service. The header files that contain the C++ declarations of the fields presented in this section are in the MessageHeader.h and ToolctlMessages.h files in /bgsys/drivers/ppcfloor/ramdisk/include/services.

The message header fields include the following information:

- Service (for example, tool control, job control)
- Protocol version number associated with the message service
- Type of message within this service
- Rank target for this message
- Sequence ID to correlate requests and acknowledgments
- Return code providing result of request
- Error detail which is typically an error value
- Amount of data in message (including this header)
- Job identifier target for this message

The protocol version number that is associated with the message service is used to determine the CDTI interface level that a component is built with. The compute node provides backward compatibility. However, message sizes and content sent from the compute node are dictated by the protocol version that the compute node was built with. For a tool that was compiled to an older version of the CDTI interface to operate with a kernel built with a newer version of the interface, the tool must accept messages from the CNK that contain a larger size than defined in the tool's version of the interface. Conversely, if a tool was compiled to a newer version of the interface than the CNK, the tool can send messages to the CNK that are larger than the definitions in the CNK CDTI interface. The CNK accepts the larger messages, but operates only on the fields in the message that are defined in its version of the interface. The sizes of the various messages are not reduced when moving to a newer version of the protocol. The semantics of the interface maintain compatibility across protocol versions.

The tool control service message defines the following message types. Each of the following message types has a corresponding acknowledgment message.

- Attach
- Detach
- Control
- Query
- Update
- Notify

The Query and Update message types include one or more message primitives. These message primitives are referred to as commands. There are commands for specific actions that a tool can perform. More complex operations can be accomplished by combining commands in a single message.

The Query and Update messages contain the following fields to allow specification of the commands:

- Number of commands in the message
- List of command descriptors

Each command within the message is represented by the following command descriptor. The tool must set up this descriptor list with the commands it wants to execute along with any additional data associated with each command. This additional data is placed in the message data area located after the list of command descriptors, with the offset field in the descriptor entry pointing to it. The return code field in the descriptor is not applicable when sending the message. This return code field is set in the descriptor entry within the corresponding acknowledgment message by the CNK. The tool must then test this return code field to determine if the command executed successfully.

Command descriptor fields:

- Command type
- Offset to command in the message
- Length of the command data
- Command return code

Each message, including the message header, cannot exceed 65,536 bytes. Transfers between the compute node and I/O node use remote direct memory access (RDMA). The number of bytes provided in message acknowledgments are also limited to 65,536 bytes.

4.2 Processing the command list

When a message that contains a list of commands is processed by the CNK, all of the commands in the list that target a specific hardware thread are grouped together. The CNK interrupts all of the necessary hardware threads, sending to them the list of pertinent commands. Each hardware thread then processes its list of commands in parallel with the other hardware threads on the node. The commands within a hardware thread are processed together within the same interrupt instance. Therefore, the information returned by all of the commands in a message is presented as a consistent set of data across pthreads that are running in that hardware thread. This happens regardless of whether the target threads were actively running when the request was processed.

4.3 Messages and commands sent from a tool to CNK

Table 4-1 shows the messages and the corresponding commands that a tool can send to the CNK. Each message and command also contains a corresponding acknowledgment message. If a command or message is defined to return data to the caller, this data is contained in the acknowledgment message. The returned data for each command is accessed using the same command descriptor data structures previously described.

Table 4-1 Available messages and commands

Message	Commands per message	Commands	Target	Scope
Attach	0	n/a	One rank or all ranks	Process or node
Detach	0	n/a	One rank or all ranks	Process or node
Control	0	n/a	rank	Process

Message	Commands per message	Commands	Target	Scope
Query	16	Get special-purpose registers	Thread	Thread
		Get general-purpose registers	Thread	Thread
		Get floating-point registers	Thread	Thread
		Get debug registers	Thread	Thread
		Get memory	Thread	Process (speculative mem:thread)
		Get thread list	Thread	Process
		Get aux vectors	Thread	Process
		Get process data	Thread	Process
		Get thread data	Thread	Thread
		Get file names	Thread	Node
		Get stat data	Thread	Node
		Get file contents	Thread	Node

Message	Commands per message	Commands	Target	Scope
Update	16 (Only one action command allowed per message, and it must be the last command in the list.)	Set general-purpose register	Thread	Thread
		Set general-purpose registers	Thread	Thread
		Set floating-point register	Thread	Thread
		Set floating-point registers	Thread	Thread
		Set debug register	Thread	Thread
		Set debug registers	Thread	Thread
		Set special-purpose register	Thread	Thread
		Set special-purpose registers	Thread	Thread
		Set memory	Thread	Process (speculative mem:thread)
		Hold thread	Thread	Thread
		Release thread	Thread	Thread
		Install trap handler	Thread	Process
		Remove trap handler	Thread	Process
		Allocate memory	Thread	Process
		Free memory	Thread	Process
		Send signal	Thread	Thread
		Set continuations signal	Thread	Process
		Set preferences	Thread	Thread
		Set breakpoint	Thread	Process
		Reset breakpoint	Thread	Process
		Set watchpoint	Thread	Thread
		Reset watchpoint	Thread	Thread
		Step thread (action command)	Thread	Process
		Continue process (action command)	Thread	Thread
		Release control (action command)	Thread	Process

4.4 Messages and commands sent from the CNK to a tool

The CNK sends acknowledgments to all of the messages and commands listed in Table 4-1 on page 17. The individual command acknowledgments are contained within the higher-level message acknowledgment that corresponds to the message that contained the commands. In addition to these messages, the Notify message, is sent from CNK. For more information about this message, see 4.6.6, “Notify” on page 24.

4.5 Multiple tool management

The CNK supports multiple tools attached to processes within the same job. The maximum number of tools that can be attached to any process is four.

4.5.1 Priorities

When a tool attaches, it provides a priority number ranging from 0 to 99. A priority of 99 is the most favored priority value. Each tool must provide a unique priority.

4.5.2 Control authority

When a tool attaches to a compute node, that tool automatically has Query access to the compute node. If a tool wants to modify the state of the compute node using Update commands, it must first obtain Control authority, which is accomplished by calling the Control message. If no other tool currently has Control authority, the message is completed successfully and the tool can now issue Update commands.

4.5.3 Conflicts

If a tool requests Control authority but another tool already has Control authority, the controlling tool’s identification information and priority is returned within the Control message acknowledgment. Also, a Notify message identifying the requesting tool is sent to the tool currently in control. The Notify message contains the requesting tool’s identification and priority.

When a controlling tool releases control, the highest priority tool that was denied Control authority is sent a Notify message indicating that Control authority has been relinquished by the controlling tool.

4.5.4 Example multi-tool usage

Consider a tool that occasionally snapshots the stacks of the threads within the process. This tool might want to set breakpoints to monitor various code paths. This tool attaches at priority 10 and requests to obtain Control authority. Now consider a debugger tool.

This tool connects at priority 90 because it expects to obtain control after it attaches. The following steps show the sequence of messages that allows the debugger tool to obtain control of a process:

1. The Snapshot tool sends an Attach message with a priority 10.
2. The Snapshot tool sends a Control message to request Control authority, and the request succeeds.
3. The Snapshot tool sends Update messages to set breakpoints.
4. The Debugger tool sends Attach message with priority 90.

5. The Debugger tool sends a Control message to request Control authority. Because the Snapshot tool currently has Control authority, the request is denied. The CNK sends a ControlAck message with information about the conflicting Snapshot tool.
6. The CNK sends a Notify message to the Snapshot tool indicating that there is a Control authority conflict. The message contains the tool identification and priority of the Debugger tool.
7. The Snapshot tool sees that the Debugger tool is of higher priority than itself. The Snapshot tool, being a team player, sends an Update messages to clean up any pending alternations/ or breakpoints, and the last Update message contains the Release Control command. The Snapshot tool must be told when the Debugger tool completes. Therefore, it sets an indicator within the Release Control command that causes a Notify message to be sent to it when the Debugger tool releases control.
8. The CNK sends a Notify message to the Debugger tool indicating that Control authority is available.
9. The Debugger tool sends a Control message to request Control authority. This time, the request succeeds and the Debugger tool can proceed.
10. The Snapshot tool remains attached without Control authority. It can choose to operate in this degraded mode.
11. The Debugger tool finishes and sends an Update message with the Release Control command.
12. The CNK sends a Notify message to the Snapshot tool indicating that Control authority is now available.
13. The Snapshot tool sends a Control message and regains Control authority.

4.5.5 Multiple controlling tool considerations

When multiple tools are allowed to update the state of the compute node, coordination across the compute nodes by the tools is necessary. Before a tool releases Control authority, it must perform any necessary cleanup required. For example, if breakpoints are set, they must be removed unless there is some higher-level coordination between the tools that is purposely passing this information or state from one tool to another tool. When a tool attaches, it passes in an 8-byte string that can be used to identify itself. This tag is provided in the Available and Conflict Notify messages to assist in the implementation of higher-level tool coordination.

4.6 Functional description of messages

The following section describes each message and command.

4.6.1 Attach

Attach tells a compute node process and the I/O node tool control daemons that a tool wants to be associated with a compute node. The CNK replies with an Attach acknowledgment after the request is completed. Attach can be used at any time after the job is loaded into the compute node and before the targeted process exits. Thus, a tool can connect to a running program.

The following information is provided by the tool within the Attach message:

- Target rank number or request to attach to all ranks

- Priority level for this tool: 0 - 99
- Character field to be used for tool identification (specific values managed by the tools)

The following information is returned by the compute node:

- Number of processes that were attached on this node
- Rank number for each of the processes that were attached on this node

The following error conditions are detected and reported:

- Tool priority conflict with a tool already attached at the requested priority.
- An invalid Tool ID value was specified.
- A Tool ID conflict was found with a tool that is already attached.
- The maximum number of tools has already been attached to this process.
- The specified job identifier is incorrect.

4.6.2 Detach

The Detach message tells a compute node and the I/O node tool daemons that a tool no longer wants to maintain a connection to the compute node. The CNK replies with a Detach acknowledgment after the request is completed. The Detach message requires that a tool that has Control authority for a process first releases this Control authority using the Release Control command.

The following information is provided by the tool within the Detach message:

- Target rank number or request to detach to all ranks

The following is returned to the tool within the Detach acknowledgment message:

- Number of processes that were detached on this node
- Rank number for each of the processes that were detached on this node

Before detaching, a tool must:

- Remove any break-points and watchpoints that it does not plan to pass to another tool
- Free compute-node process memory allocations
- Release any threads that it was holding that are not being passed to another tool
- Release Control authority

If these actions are not completed, the process might be left in an unusable state. The following error conditions are detected and reported:

- The specified job identifier is incorrect.
- The tool has not released Control authority on at least one process in the node.

If the target tool identifier for the tool to be detached is not found, no error is surfaced.

4.6.3 Control

If an attached tool must modify the state of the process, it must first obtain Control authority. Only one tool can have Control authority to a process at any time.

The following information is provided by the tool for the Control message:

- Signal, if any, to be sent to the process leader when control is acquired:
  - If a signal is requested, all threads in the process are suspended.

- Set of signals that results in a Notify message to the tool when encountered:
  - SIGKILL does not result in a Notify message even if specified in the signal mask.
- Watchpoint trap mode:
  - Deliver watchpoint traps after the instruction that caused the match (default).
  - Deliver watchpoint traps on the instruction that caused the match.
- Job startup notification location for a dynamic application when Control authority is granted before the process starts:
  - First user-state instruction in the dynamic loader, ld.so (default).
  - First instruction within the application_start function.

The following error conditions are detected and reported:

- Invalid job identifier was specified.
- An invalid rank number was specified.
- Control authority conflict; another tool currently has Control authority:
  - Tool ID and tool tag field of the tool that currently has Control authority (will be self if succeeded).
  - Priority of the tool that currently has Control authority.

4.6.4 Update

The Update message is used to modify the state of the process. The Update message can be executed only by a tool that has Control authority.

The following information is provided by the tool for the Update message:

- Number of commands in message
- List of command descriptors indicating which commands are to be processed
- Command data

To determine if the commands executed successfully, the return code contained within each commands descriptor within the Update acknowledgment message must be checked. The return code in the Update acknowledgment message itself is an indication of whether the commands were able to be processed, not an indication of whether the commands executed successfully. The ordering of the command descriptors in the Update acknowledgment message matches the original ordering of the command descriptors within the Update message. However, the ordering of the data associated with the commands does not necessarily match the ordering of the command descriptors. The data offset field within the command descriptor must be used to locate the data associated with a command.

The following information is returned to the tool within the Update acknowledgment message:

- The number of command acknowledgments contained in the acknowledgment message
- A list of command descriptors containing the command acknowledgments and associated return codes for all of the commands in the Update message
- Command data located in the Update acknowledgment message data area for commands that return data

The following error conditions are detected and reported in the Update acknowledgment message return code:

- An invalid job identifier was specified.
- This tool does not have the necessary priority to perform the requested action.

- An invalid rank number was specified.
- The specified number of commands within the message exceeded the maximum size of the message.
- The process is exiting.

4.6.5 Query

The Query message is used to access information about a process. The Query message can be executed by any attached tool.

The following information is provided by the tool for the Query message:

- The number of commands in the message
- A list of command descriptors indicating which commands are to be processed
- Command data

To determine if the commands executed successfully, the return code contained within each command descriptor entry must be checked. The return code in the Query acknowledgment message itself is an indication of whether the commands were able to be processed, not an indication of whether the commands executed successfully. The ordering of the command descriptors in the Query acknowledgment message matches the original ordering of the command descriptors within the Query message. However, the ordering of the data associated with the commands does not necessarily match the ordering of the command descriptors. The data offset field within the command descriptor must be used to locate the data associated with a command.

The following information is returned to the tool within the Query acknowledgment message:

- The number of command acknowledgments contained in the acknowledgment message
- A list of command descriptors containing the acknowledgments and associated return codes for all of the commands in the Query message
- Command data located in the Query acknowledgment message data area for each command

The following error conditions are detected and reported in the Query acknowledgment message return code:

- An invalid job identifier was specified.
- An invalid rank number was specified.
- The specified number of commands within the message exceeded the maximum size of the message.
- The process is exiting.

4.6.6 Notify

The Notify message is the only unsolicited message that is sent from the CNK to the tool. The Notify message is sent for the following conditions:

- Signal
- Process exit
- Control authority available
- Control authority conflict

Signal

The notify message is sent when a signal is encountered, and a controlling tool is configured to be notified when this signal occurs. The kernel serializes Notify message delivery to the tool. The Continue Process, Release Control, and Step Thread commands satisfy a Notify message. When a thread that generated a Notify message next runs, it is continued with no signal by default. If the tool needs to continue a thread with a non-zero signal, it must use the Set Continuation Signal command at some point after the Notify message and before a command that satisfies the Notify message. The kernel also provides a synchronous suspension of all threads in the process when a signal occurs and resumes all threads when the last pending Notify message is continued. For more information, see the following sections: “Continue Process” on page 33, “Step Thread” on page 34, and “Set Continuation Signal” on page 34.

The following information (in addition to the message header data) is provided within the message:

- Signal number
- Instruction address where the signal occurred
- Address of the data access causing the signal if such a signal occurs (for example, SIGTRAP due to a data address compare [DAC] interrupt)
- Reason for signal notification:
  - Generic signal
  - Breakpoint trap
  - Read watchpoint
  - Write watchpoint
  - Step completion
- Speculation information:
  - Speculation active indicator
  - Instruction address where interrupt condition occurred
  - General Purpose Register 1
  - General Purpose Register 2
  - General Purpose Register 3

Process exit

A process exited (all attached tools are sent this Notify message).

The following information (in addition to the message header data) is provided within the message:

- Process exit state

Control authority available

A tool released Control authority, and the highest-priority conflicting tool requested to be notified when this occurs.

The following information is provided within the message:

- Tool ID and tool tag identification of the tool that released Control authority
- Priority of the tool that released its Control authority

Control authority conflict

Another tool attempted to gain Control authority, but this tool currently has Control authority.

The following information is provided within the message:

- Tool ID and tool tag identification of the tool that requested Control authority
- Priority of the tool that requested Control authority

4.7 Functional description of commands

This section provides a detailed description of each command. The result of executing each command is provided within the return code field of the command descriptor entries in the acknowledgment message. The return codes are defined in ToolctlMessage.h and MessageHeader.h in /bgsys/drivers/ppcfloor/ramdisk/include/services.

The possible command return codes are:

- The command executed successfully.
- The Thread ID specified no longer corresponds to an active thread in this process.
- The command is not recognized.
- Timeout waiting for the target thread to respond.
- Could not allocate necessary memory in the outbound message area.
- An invalid parameter was supplied with the command.
- A failure occurred while attempting to set or clear a breakpoint.
- Invalid value provided for the address parameter.
- Invalid value provided for the length parameter.
- Hardware resource conflict.
- The requested memory allocation cannot be satisfied.
- The requested file name was not found.
- The command conflicts with other commands in the command list.
- The Release Control command cannot complete because of a pending signal notify that is not satisfied.
- A previous command failure with the same message prevented this command from being executed.
- A process is exiting. The command cannot be completed.
- A request to set a watchpoint overlaps with an existing watchpoint.

4.7.1 Thread target

Each command can accept a thread identifier (ID) within the command descriptor entry to be used as the target thread. This thread identifier is the same identifier generated by the kernel when the thread was created using the clone system call. This is also the same thread ID that the kernel expects for other system calls that require a thread target. The Linux equivalent to this field is the lwpid. This is not the pthread_t thread ID value that the GLIBC thread library provides for use by the thread library APIs. If a zero value is supplied by the sender, the CNK chooses a thread identifier. The CNK first attempts to use the thread identifier associated with the most recent Notify Signal message (that is, the thread that generated the most recent signal). If no Notify Signal message occurred, or if that thread no longer exists, the thread identifier of the process leader thread is used. The process is identified from the rank information provided in the message header.

4.7.2 Query commands

This section describes the available query commands that can be specified within a query message. Up to 16 commands can be specified within one message.

Get Special Purpose Registers

Command data provided by the sender:

- Thread identifier

This command returns the special purpose registers for the target thread. The registers that are included in this list are:

- Instruction Address Register
- Link Register
- Machine State Register
- Condition Register
- Count Register
- Integer Exception Register
- Floating Point Status and Control Register
- Data Exception Address Register
- Exception Syndrome Register

Get General Purpose Registers

Command data provided by the sender:

- Thread identifier

This command returns the 32 general-purpose registers.

Get Floating Point Registers

Command data provided by the sender:

- Thread identifier

This command returns the 32 floating-point register sets. Each floating-point register set contains four 64-bit registers representing slots 0, 1, 2, 3, as defined in the quad-processing extensions (QPX) floating-point unit.

Get Debug Registers

Command data provided by the sender:

- Thread identifier

This command returns the debug registers for the target thread. The registers that are included in this list are:

- Debug Control Register 0
- Debug Control Register 1
- Debug Control Register 2
- Debug Control Register 3
- Data Address Compare Register 1
- Data Address Compare Register 2
- Data Address Compare Register 3
- Data Address Compare Register 4
- Instruction Address Compare Register 1
- Instruction Address Compare Register 2
- Instruction Address Compare Register 3

- Instruction Address Compare Register 4
- Debug Status Register

Get Memory

Command data provided by the sender:

- Thread identifier
- Address of the data to be returned
- Length of the data to be read (maximum 65,024 bytes)
- Force nonspeculative read flag

This command returns the following information:

- Address of the data
- Length of the data
- Data

Shared text segment

When executing a statically linked executable, the text segment of the executable is shared across the configured processes within the node. When reading data from the text segment using this Get Memory interface, any memory modifications performed against the text segment by other processes using the Set Memory command are seen by this process. If these memory modifications were performed using the Set Breakpoint command, the kernel tracks these changes. When a Get Memory operation is performed on storage locations modified by another process executing the Set Breakpoint command, this process does not see that modification. For more information, see “Set Breakpoint” on page 35 and “Set Memory” on page 32.

Speculative memory

In general, the specification of a thread target within the Get Memory command is arbitrary. However, if the target thread is currently running speculatively, the data returned by the Get Memory command is that thread’s view of memory. That thread’s view might be different from another thread’s view of the same memory within the same process if the speculative region of that thread’s memory is being read. To force a nonspeculative read of data, regardless of the state of the target thread, the sender can set the *force non-speculative access* indicator in the command data. If the thread is stopped at an interrupt condition that caused a notification signal, it might have been running speculatively when the condition occurred; however, it is no longer in a speculation mode. To read the speculative memory for these conditions, the *attempt speculative access* indicator in the command data must be set.

Get Thread List

Command data provided by the sender:

- Thread identifier

This command returns a list of thread identifiers for threads currently active within the targeted process. For more information about the thread identifier, see 4.7.1, “Thread target” on page 26. In addition to the thread identifier, an indicator is also returned to identify which of the listed thread identifiers is associated with communication threads created by the Blue Gene messaging infrastructure.

Get Aux Vectors

Command data provided by the sender:

- Thread identifier

This command returns the auxiliary vectors associated with the targeted process:

- Length of the data in double words
- Aux vector data consists of 64-bit double word pairs

Get Process Data

Command data provided by the sender:

- Thread identifier

This command returns the following process information:

- Rank
- Thread group ID (equivalent to the PID and TID of the main thread)
- A, B, C, D, E, and T coordinates
- Shared memory region start and end address
- Persistent memory region start and end address
- Heap start and end address
- Heap break address
- Memory map start and end address
- Time job has been running

Get Thread Data

Command data provided by the sender:

- Thread identifier

This command returns the following thread data:

- Processor core identifier on which this thread exists
- Processor thread identifier on which this thread exists
- State of the thread: Run, Sleep, Futex wait, Idle
- Tool state of the thread: Active, Hold, Suspended, Hold and Suspend
- Speculative state: Non-speculative, transactional memory active/invalid, speculative execution active/invalid
- Stack guard page start and end address
- Stack start address
- Current stack address
- Number of stack frame entries in the trace-back
- Stack trace-back data containing frame addresses and saved link registers

Get File Names

Command data provided by the sender:

- None

This command returns the list of file names that exist in the compute node's persistent memory file system.

Get File Stat Data

Command data provided by the sender:

- File path name

This command returns the following file information:

- stat64 structure as defined in the stat system call

Get File Contents

Command data provided by the sender:

- File path name
- File offset
- Number of bytes to read (maximum 65,024 bytes)

This command returns the requested number of bytes from the file starting at the specified offset.

4.7.3 Update commands

This section describes the available update commands that can be specified within an update message. Up to 16 commands can be specified within one message.

Set General Purpose Register

Command data provided by the sender:

- Thread identifier
- Register selection
- Register value to be set into the selected register

This command does not return any data.

Set General Purpose Registers

Command data provided by the sender:

- Thread identifier
- Register values to be set into the registers

This command does not return any data.

Set Floating Point Register

Command data provided by the sender:

- Thread identifier
- Register selection
- Values to be set into the selected QPX register set

This command does not return any data.

Set Floating Point Registers

Command data provided by the sender:

- Thread identifier
- Register values to be set into the registers

This command does not return any data.

Set Debug Register

Command data provided by the sender:

- Thread identifier

- Register selection
- Register value to be set into the selected register

The register selection list includes the following debug registers:

- Debug Control Register 0
- Debug Control Register 1
- Debug Control Register 2
- Debug Control Register 3
- Data Address Compare Register 1
- Data Address Compare Register 2
- Data Address Compare Register 3
- Data Address Compare Register 4
- Instruction Address Compare Register 1
- Instruction Address Compare Register 2
- Instruction Address Compare Register 3
- Instruction Address Compare Register 4
- Debug Status Register

This command does not return any data.

Care must be taken when using this command. Other commands, such as Set Watchpoint, Set Breakpoint, and Step, implicitly use the debug registers. If a tool writes to the debug registers using this command and is also using these other commands, unpredictable behavior might result.

Set Debug Registers

Command data provided by the sender:

- Thread identifier
- Register values to be set into the registers

This command does not return any data.

Set Special Purpose Register

Command data provided by the sender:

- Thread identifier
- Register selection
- Register value to be set into the selected register

The register selection list includes the following debug registers:

- Instruction Address Register
- Link Register
- Machine Status Register
- Condition Register
- Count Register
- Integer Exception Register
- Floating Point Control Register
- Data Effective Address Register
- Exception Status Register

Set Special Purpose Registers

Command data provided by the sender:

- Thread identifier
- Register values to be set into the registers

The list of registers includes:

- Instruction Address Register
- Link Register
- Machine Status Register
- Condition Register
- Count Register
- Integer Exception Register
- Floating Point Control Register
- Data Effective Address Register
- Exception Status Register

This command does not return any data.

Set Memory

Command data provided by the sender:

- Thread identifier
- Address where data is to be written
- Number of bytes to be written
- Data to be written (maximum 65,024 bytes)
- Force nonspeculative write flag
- Allow write to shared process memory flag

This command returns the following data:

- Address where the data was written
- Number of bytes that were written

Shared text segment

When executing a statically linked executable, the text segment of the executable is shared across the configured processes within the node. When writing data into the text segment using this Set Memory interface, the default behavior is to return an error. If a tool needs to write to a shared process memory, the *allow write to shared process memory* indicator can be set in the command data. This process allows the write operation to shared memory to proceed.

However, special processing occurs when a 4-byte write operation is performed to data in the shared text segment. If the data being written is a trap instruction, the operation is allowed to continue regardless of the *allow write to shared process memory indicator*. This write operation is tracked within the CNK breakpoint table allowing the trap condition to not be surfaced outside of the CNK for processes that do not expect to see the trap instruction. If the 4-byte data write operation is not a trap instruction, a test is first made to see if an entry exists in the breakpoint table. If it does, the entry is removed from the table and the command is considered successful. If the address does not exist in the breakpoint table, normal handling of a 4-byte write operation proceeds, including the testing of the *allow write to shared process memory* indicator. For more information about the breakpoint table, see “Step Thread” on page 34 and “Reset Breakpoint” on page 36.

Speculative memory

In general, the specification of a thread target within the Set Memory command is arbitrary. However, if the target thread is currently running speculatively, the data stored by the Set Memory command will be that thread’s version of memory. That thread’s version might be different from another thread’s version of the same virtual address within the same process. If the tool needs to write the data non speculatively regardless of the targeted thread, the *force nonspeculative access* flag can be set in the command data. If a tool wants to write the data speculatively, the *attempt speculative access* flag can be set. If a speculative region exists for the thread, it is written.

Hold Thread

The state of an active thread can be either running or blocked. There are various reasons why a thread might be blocked. Multiple block conditions can exist together. Two special block reasons have been defined for use by tools: Suspend and Hold. The Hold Thread command sets the hold block state within the target thread. The removing of this blocking condition is accomplished by the Release Thread command. When a signal occurs and a corresponding Notify message is sent to a tool, the kernel sets the Suspend blocking state in all threads of the process. The Hold Thread command provides a mechanism for a tool to explicitly place a thread into the hold state. Multiple calls to the Hold Thread command are not stacked or reference counted. For example, two Hold Thread operations followed by one Release Thread operation release the thread.

Command data provided by the sender:

- Thread identifier

This command does not return any data.

Release Thread

The blocking reason of Hold is removed from the target thread. If no other blocking reasons exist, and if the hardware thread is not currently running another thread, this thread is dispatched and begins executing on its hardware thread. For more information about the hold state, see “Hold Thread” on page 33.

Command data provided by the sender:

- Thread identifier

This command does not return any data.

Continue Process

The Continue Process command is used to resume a process that was interrupted by a signal. The target of the command must be the last thread that delivered a Notify Signal message to the tool. If no thread identifier target is supplied in the command, the default thread identifier is the last thread that delivered a Notify Signal message. If there are additional threads currently attempting to send a Notify Signal message when this command is executed, the threads all remain suspended and the next Notify Signal message is sent to the tool. If there are no additional threads currently attempting to send a Notify Signal message when this command is executed, this command removes the Suspend blocking condition from all threads in the process. It continues the targeted thread from the interrupted signal delivery flow. If the Set Continuation Signal command was not previously called, the thread is resumed with no signal. If a continuation signal was set by the Set Continuation Signal command, the signal is delivered before the thread resumes program execution. For more information, see “Set Continuation Signal” on page 34. If a Hold condition is active on any threads in the process, those threads remain in the kernel scheduler and do not return to the application. For more information, see 4.6.6, “Notify” on page 24 describing the Notify Signal message. If any previous command targeted to the same thread failed, this command is not completed. For example, if a Set Continuation Signal command is packaged within the same Update Message and the Set Continuation Signal command fails, this Continue Process command is not attempted and returns an error code.

Command data provided by the sender:

- Thread identifier

This command does not return any data.

Step Thread

The Step Thread command advances the currently executing program to its next instruction. If the targeted thread is a thread that is currently suspended in signal delivery, and if the Set Continuation Signal command was not previously called, the thread is stepped with no signal. If a continuation signal was set by the Set Continuation Signal command and a user signal handler exists, the user signal handler is called before the thread resumes program execution. This results in the step operation stopping at the beginning of the signal handler. Other threads within the process remain suspended while the Step Thread operation is performed. If an attempt is made to perform a step operation when there are other pending Notify Signal messages, the step is not completed and one of the pending notification messages is sent to the tool. If an attempt is made to step a thread that is not in a state it can run before its suspension, for example a futex wait, an error is returned. For more information, see 4.6.6, “Notify” on page 24. If any previous command targeted to the same thread failed, this command is not completed. For example, if a Set Continuation Signal command is packaged within the same Update Message and the Set Continuation Signal command fails, this Step Thread command is not attempted and returns an error code.

Command data provided by the sender:

- Thread identifier

This command does not return any data.

Set Continuation Signal

The Set Continuation Signal command is used to change the signal number of a suspended thread that delivered a Notify message to the tool. When this thread is later resumed as a result of a Continue Process, Step Thread, or Release Control. The specified signal is raised. The signal number can be a zero, which results in no signal being raised. If multiple Set Continuation Signal commands are sent to the same thread, the signal number in the last command is used. For more information, see 4.6.6, “Notify” on page 24.

Command data provided by the sender:

- Thread identifier
- Signal number

Send Signal

This command sends a signal to the targeted thread. If the signal cannot be delivered when the signal is sent, it is made pending. If this is a signal that the tool is interested in, the occurrence of this signal results in a notification message. This command is not to be used to continue a thread that was suspended during a signal delivery by a Notify message. For that condition, use the Set Continuation Signal command. If the target thread is currently suspended in signal delivery, this new signal is added to the pending signals for the target thread. If the target thread is currently held by a Hold Thread command, the signal is added to the pending signals for the target thread.

Command data provided by the sender:

- Thread identifier
- Signal number

This command does not return any data.

Set Breakpoint

This command is used to set a trap instruction into the text segment or heap segment in the case of dynamically loaded code. When setting a trap instruction into a text segment that is shared across multiple processes within a node, the CNK keeps track of all the breakpoints set by the various processes. It manages the reporting of trap exceptions so that only the processes that set traps surface SIGTRAP notifications to the tool. It is recommended that tools use this interface when setting breakpoints. However, a 4-byte Set Memory operation also respects the breakpoint table (see “Set Memory” on page 32). The maximum number of unique breakpoint addresses that can be tracked per node within the shared text segment is 16,384. Requests to set more than this number of unique breakpoint addresses in the shared text segment result in an unsuccessful return code indicating that there was a failure setting a breakpoint. There is no limit on the number of breakpoints that can be set in dynamically loaded libraries or in job configurations that contain only one process per node (that is, the `runjob` option –ranks-per-node=1).

Table 4-2 illustrates the breakpoint capacities for various environments:

Table 4-2 Breakpoint capacities

	One process per node	**More than one process per node**						
		Maximum unique addresses	**Breakpoints in the node for various processes per node configurations (not exceeding the listed maximum unique addresses)**					
			2	**4**	**8**	**16**	**32**	**64**
Statically linked executable	Unlimited	16,384	32,768	65,536	131,072	262,144	524,288	1,048,576
Dynamic executable main text segment	Unlimited	16,384	32,768	65,536	131,072	262,144	524,288	1,048,576
Dynamic executable libraries	Unlimited	Unlimited	Unlimited	Unlimited	Unlimited	Unlimited	Unlimited	Unlimited

Normal and fast trap breakpoints

When the Set Breakpoint instruction is called, the trap instruction is provided by the caller. Trap Word instructions, conforming to the X-form as documented in the *Power Instruction Set Architecture* (Power ISA), are treated as normal breakpoints. Trap Word Immediate instructions, conforming to the D-form as documented in the Power ISA, are treated as fast-trap breakpoints. This distinction is important when the CNK is deciding if a tool is to be notified of a breakpoint condition. If fast traps are enabled, and if a fast breakpoint is encountered, the CNK does not notify the tool; instead, it calls the installed fast-trap signal handler.

Trap type conflicts

When a breakpoint is set in the shared text segment of a program and is consequentially tracked in the breakpoint table, the specific trap instruction that is placed at this breakpoint location cannot differ from the trap instruction set by any other process targeting the same virtual address. For example, if process A on a node sets a breakpoint at address X, and process B on the same node sets a breakpoint at address X, the trap instruction values provided in the set breakpoint commands must match (that is, cannot mix a fast trap with a normal trap instruction).

Command data provided by the sender:

- Thread identifier
- Address of the breakpoint
- Trap instruction: either Trap Word or Trap Word Immediate (D-form or X-form)

This command does not return any data.

Reset Breakpoint

The Reset Breakpoint command removes a breakpoint from the target processes address space. If the supplied instruction data does not match the original instruction data that existed at the time the breakpoint was set, the command returns a failure and the breakpoint is not removed. This additional testing is performed when the breakpoint has been recorded in the breakpoint table as described in "Set Breakpoint" on page 35.

Command data provided by the sender:

- Thread identifier
- Address of the breakpoint to be reset
- Instruction data to be restored into the breakpoint location

This command does not return any data.

Set Watchpoint

The watchpoint facility uses the DAC debug registers along with the debug control registers (DBCR0, DBCR1, DBCR2). The DAC registers are a limited resource within the core with reduced functionality:

- Four DAC registers are shared across the four threads within each core.
- A watched range requires two DAC registers.
- The watched range is defined by one DAC used as a base address and a second DAC used as a mask (not a start/end pair):
  - The watch range start address must be boundary aligned based on the size of the watch range.
  - The size of the watch range must be a power of 2.

Due to these limitations, it is difficult for tools to efficiently manage watchpoints by directly using the debug registers, especially when combined with breakpoints and stepping functions that actively use the thread scoped debug control registers. For this reason, the CNK provides a watchpoint facility that manages this resource across the cores and hardware threads. A watchpoint is scoped to a thread within the process. If the watchpoint is expected to be active for multiple threads in the process, each thread must be targeted. If multiple threads have the same watchpoint active, these multiple watchpoints do not contribute to the hardware resource constraints with a core.

Command data provided by the sender:

- Thread identifier
- Address where to set the watchpoint
- Number of bytes of the watched memory area
- Type of watchpoint: read, write, read or write

The following restrictions apply to the provided data:

- The address must be boundary aligned to the number of bytes provided. For example, if the size of the watch area is 16 bytes, the address must be aligned on a 16-byte boundary.
- The number of bytes must be a power of 2 (for example, 2, 4, 8,16).
- Hardware resources must not be over-committed. An example is attempting to set a watchpoint on a specific thread when two other watchpoints are already set across other threads running on the same core. If any of these restrictions are not met, an appropriate error return code in the command descriptor of the acknowledgment message is set.
- The address range for a watchpoint cannot overlap the address range of another watchpoint on the same core.

This command does not return any data.

Reset Watchpoint

The Reset Watchpoint command removes a watchpoint that was previously set using the Set Watchpoint command.

Command data provided by the sender:

- Thread identifier
- Address of the watchpoint to be reset

This command does not return any data.

Allocate Memory

This command allows the sender to allocate memory within the targeted processes address space. This memory is taken from the heap segment of the process using an anonymous mmap operation within the kernel. This memory has read/write/execute permissions.

Command data provided by the sender:

- Thread identifier
- Allocation size in number of bytes

This command returns the following data:

- Address of the allocated memory. This memory is initialized to zero and aligned on a 4096-byte boundary.

The command return code must be checked to determine if the allocation was successful.

Free Memory

This command allows the sender to deallocate memory within the targeted processes address space. This memory is returned to the process. Care must be taken to deallocate only what was allocated. The CNK does not track tool allocations separately from application allocations. Therefore, deallocating memory that was not allocated by the tool can cause unpredictable results.

Command data provided by the sender:

- Thread identifier
- Address of the allocation to be freed
- Allocation size to be freed in number of bytes

This command does not return any data.

Install Trap Handler

This command allows a tool to install a signal handler for the SIGTRAP signal. The installed signal handler is expected to have the following signature, consistent with a standard sigaction handler:

```
void (*sa_sigaction)(int, siginfo_t *, void *);
```

The third parameter is a pointer to the ucontext.

Signal handlers are process scoped handlers. Therefore, any thread within the process can be specified as the target for the command.

Command data provided by the sender:

- Thread identifier
- Function pointer to the signal handler. This function pointer points to the .opd entry of the function and is not a pointer to the code itself. It is expected that within the .opd entry is a pointer to the code followed by a pointer to the TOC. The two pointers within the opd entry of the trap handler are cached by the kernel when this command is executed. Therefore, the .opd entry must be initialized before calling this command.

This command does not return any data.

Remove Trap Handler

The command uninstalls a trap handler that was installed with the Install Trap Handler command.

Command data provided by the sender:

- Thread identifier
- Function pointer to the signal handler

This command does not return any data.

Set Preferences

This command is used to set various tool preferences after obtaining Control authority.

Command data provided by the sender:

- Thread identifier.
- Watchpoint trap mode: Trap-on interrupt or trap-after interrupt. This preference controls whether the occurrence of a DAC interrupt is presented at the instruction causing the interrupt or presented after the instruction causing the interrupt. Default behavior is Trap-After interrupt.
- Enable or disable fast breakpoints. When fast breakpoints are enabled using this command and a trap condition occurs at a location that contains a fast-trap instruction (that is, a Trap Immediate instruction), the Notify message is not sent to the tool. Instead the normal signal delivery path is followed. It is expected that the tool previously set up a signal handler using the Install Trap Handler command. This signal handler is then called. When fast breakpoints are disabled, any occurrence of a fast breakpoint trap condition follows the standard trap handling flow for a tool, resulting in a Notify message being sent to the tool. This operation is process scoped. Therefore, any thread within the process can be specified as the target.
- Enable or disable fast watchpoints. When fast watchpoints are enabled using this command and a DAC interrupt condition occurs, the Notify message is not sent to the tool. Instead, the normal signal delivery path is followed. It is expected that the tool previously

set up a signal handler using the Install Trap Handler command. This signal handler is then called. This operation is a process scoped action. Therefore, any thread within the process can be specified as the target.

Release Control

If a tool no longer requires Control authority, or is giving up control authority to a higher priority tool, it must issue the Release Control command. This command satisfies a signal notification and continues the target thread with the continuation signal if set. For more information, see “Set Continuation Signal” on page 34. This command does not release any threads that might have been placed into the Hold state using the Hold Thread command. For more information, see “Hold Thread” on page 33. If a requesting tool previously conflicted with this tool, and if the requesting tool is the highest priority conflicting tool, the requesting tool is sent a Notify message indicating that the Control authority is now available. A tool might give up Control authority because a higher priority tool wants to gain control. This tool can set a request in the message to cause a Notify Available message to be sent to it when a subsequent tool releases Control authority. Thus, a tool can temporarily yield to a higher priority tool and then regain control. This must be the last command in the list of commands within the Update message. After this command is completed, no additional update messages are allowed by this tool until it reacquires Control authority. If any previous command targeted to the same thread failed, this command is not completed. For example, if a Set Continuation Signal command is packaged within the same Update Message and the Set Continuation Signal command fails, this Release Control Process command is not attempted and returns an error code.

The following information is provided by the tool for the Release Control message:

- Request the sending of a Control Available Notify message when the next tool releases Control authority.

The following error conditions are detected and reported in the Release Control acknowledgment message return code:

- Invalid job identifier was specified.
- An invalid rank number was specified.
- Additional commands were specified in the message command list.
- A previous command did not complete successfully.
- Additional pending signal notification(s) exist and must be handled before releasing control.

4.8 Abnormal job and tool exit considerations

When a tool is attached to a job, the tool has certain responsibilities that must be adhered to when abnormal termination conditions occur within the job or within the tool.

4.8.1 Abnormal job exit

When one process in a job exits with a nonzero exit status and that exit status indicates an unhandled signal condition or a return code equal to one, the Control System forcibly terminates the remaining processes in a job by sending a SIGKILL to all other processes. If a tool has Control authority and if the tool has not continued the process from a Notify Signal message sent by the process, the SIGKILL signal remains pending, preventing the termination of the job. To address this situation, when the Control System sends the SIGKILL signals to the compute node processes, it also sends a SIGTERM signal to all tools currently associated with the job. The tool is then responsible for releasing control and satisfying all pending signal notifications, allowing the SIGKILL to continue and the job to terminate.

4.8.2 Abnormal tool exit

When a tool is started by the Control System, it is launched on all I/O nodes associated with the job. If a tool on one of the I/O nodes abnormally terminates, no explicit action is taken by the Control System. It is expected that the tool has a controlling process that is aware of the condition of its daemons running on the I/O nodes. For example, in the case of a debugger, it might be the debugger client monitoring the connections to each of the I/O node daemons and taking appropriate action if a failure occurs. The appropriate action might include the use of the **`end_tool`** command discussed in 2.3.2, "Stopping a tool" on page 10.

4.8.3 Message/command behavior

A process might enter a normal or abnormal exit flow while a tool is attached to a process. This does not occur if a tool is in control of a process and there is a Notify Signal message posted to the tool. If a process enters an exit flow and a message is sent from the tool to the compute node, a return code indicating that the process is exiting is returned. It is also possible that the exit flow might have been entered after the Query or Update message was started but before a command was processed. In this case, a command return code is provided to indicate this condition.

If a tool receives either the message return or the command return code indicating that the process is exiting, no additional actions directed to this process are required or expected from the tool. The CNK implicitly releases control and detaches the tool. Also note that a Process Exit Notify message is sent to the tool indicating that the process is exiting, which might drive additional tool cleanup actions.

4.9 libthread_db

The libthread_db library provides support for monitoring and manipulating threads. This library is contained in the GNU C libraries (GLIBC). The shared library object, libthread_db.so, is located within the lib64 directory. The associated header file, thread_db.h, is located in the sys-include directory.

The controlling process in the Blue Gene environment is the tool, and the target process is the attached compute node process. The CDTI commands provide a rich set of monitoring and control functions for threads. However, these controls are not cognizant of the library-level constructs associated with the threads. The libthread_db library adds this level of support. To enable the libthread_db, a set of callback functions must be implemented by the tool. These callback functions use the CDTI interface to access information in the target process and threads. The commands supported in the CDTI interface provide the necessary primitives to enable the base functionality of the libthread_db library.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this paper.

IBM Redbooks

The following IBM Redbooks publications provide additional information about the topic in this document. Note that some publications referenced in this list might be available in softcopy only.

- *IBM System Blue Gene Solution: Blue Gene/Q Application Development Manual*, SG24-7948-00
- *IBM System Blue Gene Solution: Blue Gene/Q Hardware Installation and Maintenance Guide*, SG24-7974-00
- *IBM System Blue Gene Solution: Blue Gene/Q Hardware Overview and Installation Planning*, SG24-7972-00
- *IBM System Blue Gene Solution: Blue Gene/Q Safety Considerations*, REDP-4656
- *IBM System Blue Gene Solution: Blue Gene/Q System Administration Manual*, SG24-7869-00

Other publications

This publication is also relevant as a further information source:

- *General Parallel File System HOWTO for the IBM System Blue Gene/Q Solution,* SC23-6939-00

How to get Redbooks

You can search for, view, download, or order these documents and other Redbooks, Redpapers, Web Docs, draft and additional materials, at the following website:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

IBM System Blue Gene Solution Blue Gene/Q Code Development and Tools Interface

Redpaper™

Compute Node Kernel, runjob, and CIOS tool interfaces

Tool messages

Tool commands

This book is one in a series of IBM publications written specifically for the IBM System Blue Gene supercomputer, Blue Gene/Q, which is the third generation of massively parallel supercomputers from IBM in the Blue Gene series. This IBM Redpaper publication helps you develop tools, such as a debugger, that can be used on the IBM Blue Gene/Q® platform to control and monitor application threads and processes.

REDP-4659-00