IBM Parallel Environment for AIX

Dynamic Probe Class Library Programming Guide

Version 3 Release 1

SA22-7420-00

IBM Parallel Environment for AIX

Dynamic Probe Class Library Programming Guide

Version 3 Release 1

SA22-7420-00

Note

Before using this information and the product it supports, read the information in “Notices” on page 209.

First Edition (April 2000)

This edition applies to Version 3, Release 1 of IBM Parallel Environment for AIX (product number 5765-D93) and to all subsequent releases and modifications until otherwise indicated in new editions.

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

IBM welcomes your comments. A form for readers' comments may be provided at the back of this publication, or you may address your comments to the following address:

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

FAX (United States & Canada): 1+914+432-9405
FAX (Other Countries):
Your International Access Code +1+914+432-9405

IBMLink (United States customers only): IBMUSM10(MHVRCFS)
IBM Mail Exchange: USIB6TC9 at IBMMAIL
Internet e-mail: mhvrcfs@us.ibm.com

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

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

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

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

Contents

Figures

Tables

About this book

This book describes the Dynamic Probe Class Library (DPCL). It shows how a C++ program can call DPCL functions to dynamically insert and remove instrumentation code patches, or "probes", into a running application.

This book is designed as a companion volume to the *IBM Parallel Environment for AIX: DPCL Class Reference*, SA22-7421-00. In contrast to the *IBM Parallel Environment for AIX: DPCL Class Reference* (which provides detailed, alphabetically-ordered, information about each DPCL function), this book describes DPCL at a higher level. While not as detailed as the reference, this book provides an overview of the tasks a program can perform using DPCL, and describes the DPCL classes and functions related to each task.

By focusing on the tasks that a program can perform, this book provides the context necessary to understand how the various classes and functions of DPCL work together to accomplish specific goals. Once you understand the tasks that your program can perform, and know which classes need to be instantiated and which functions need to be called in order to perform each task, you can refer to the *IBM Parallel Environment for AIX: DPCL Class Reference* for any additional, more specific, information you require.

Who should read this book

This book is intended for C++ application developers working in an AIX environment. You should, therefore, understand the C++ programming language and the AIX operating system before reading this book. Furthermore, if you plan to use the DPCL to instrument parallel programs, you should understand parallel-programming concepts, and if you plan to run these programs in the Parallel Environment, you should also understand the Parallel Environment and the Parallel Operating Environment. Where necessary, this book provides some background information relating to these issues. More commonly, this book refers you to the appropriate documentation.

How This Book is Organized

This book is organized into three main parts – the first part contains a general overview of basic terms and concepts, the second part provides instructions for performing standard DPCL programming tasks, and the third part provides instructions for performing additional, more-advanced and/or less-commonly performed, DPCL programming tasks. In addition to these three main parts, this book has a forth part consisting of appendixes — a sample DPCL application, a glossary of terms, and a detailed index.

Throughout this book, C++ code examples illustrate various features of DPCL. Please note that these examples do not perform rigorous error checking and are provided for illustration only.

Overview of Contents

This book contains the following information:

- "DPCL Concepts and Overview" on page 1 contains three chapters designed to provide you with a general introduction and overview of the Dynamic Probe Class Library. This section contains the following chapters:
  - Chapter 1, "What is DPCL?" on page 3 describes the basic, yet central, concepts you need to understand in order to use DPCL. It provides a high-level description of how a program can use the classes and member functions of DPCL to instrument another application. This chapter also introduces and defines the DPCL terminology that is used throughout this book, and, in doing so, describes the various parts of the DPCL software system and illustrates how these parts work together to instrument a program.
  - Chapter 2, "What are the DPCL classes?" on page 25 builds on the overview of the DPCL system contained in Chapter 1, "What is DPCL?" on page 3 by describing the DPCL classes that represent the elements of the DPCL system. In doing so, it describes the purpose, supporting data types, and functions of each class.
  - Chapter 3, "A DPCL hello world program" on page 61 uses the familiar "Hello World" example to illustrate how to instrument an application using DPCL. Unlike traditional "Hello World" programs, our example program does not itself print out the string "Hello World". Instead, our program serves as a DPCL "target application". We show how another program (what we call a DPCL "analysis tool") can insert an instrumentation code patch (called a "probe") into the target application. The probe, from within the target application, will send the "Hello World" string back to the analysis tool.
- "Standard DPCL Programming Tasks" on page 71 contains chapters describing the common tasks that all programs built on DPCL will need to perform. This section contains the following chapters:
  - Chapter 4, "Performing status error checking" on page 73 provides general information on DPCL error checking.
  - Chapter 5, "Initializing the analysis tool to use the DPCL system" on page 79 describes basic initialization tasks your program must perform. To use the DPCL system, your program must include DPCL header files, initialize the DPCL system, and enter the DPCL main event loop.
  - Chapter 6, "Connecting to or starting the target application processes" on page 83 describes how a program must connect to a target application if it is to later dynamically insert probes into that application. If the target application process(es) are already running, this chapter describes how the analysis tool can connect to the process(es). This chapter also describes how an analysis tool can use DPCL to create one or more target application processes. When an analysis tool creates a process in this way, the DPCL system also establishes the connection necessary for inserting probes into the process.
  - Chapter 7, "Controlling execution of target application processes" on page 105 describes how an analysis tool can attach itself to a target application process in order to control execution of the process. It describes

how an application can, once in this attached state, use DPCL function calls to suspend, resume, or kill one or more target application processes.

  - Chapter 8, “Creating probes” on page 115 describes the steps your program must follow to create the actual instrumentation code that will be executed within the target application. This chapter describes how to create simple instructions or sequences of instructions, called probe expressions, to serve as the instrumentation code. It also describes how a probe expression can optionally call a function written in C when more complicated instrumentation is needed.
  - Chapter 9, “Executing probes in target application processes” on page 143 describes the steps your program must follow to execute probes within one or more target application processes.
  - Chapter 10, “Creating data callback routines” on page 167 describes how an analysis tool can create a "data callback routine" to respond to data sent back to the analysis tool from probes executing within target application processes.
  - Chapter 11, “Entering and exiting the DPCL main event loop” on page 169 describes how an analysis tool can enter an event loop to interface asynchronously with the DPCL system.
  - Chapter 12, “Disconnecting from target application processes” on page 171 describes how an analysis tool can disconnect processes it has finished examining.
  - Chapter 13, “Compiling and linking the analysis tool and target application” on page 173 describes how to prelink your target application with the DPCL libraries and compile your analysis tool with the DPCL library and include files.

- “Additional DPCL Programming Tasks” on page 175 contains chapters describing the additional tasks that some programs built on the DPCL will need or want to perform. This section contains the following chapters:
  - Chapter 14, “Handling signals and file descriptors through the DPCL system” on page 177 describes how an application can monitor AIX signals and file descriptors through the DPCL system.
  - Chapter 15, “Overriding default system callbacks” on page 181 describes how an application can create callback routines to handle unexpected system events such as a DPCL daemon exiting or a target application process terminating. These callbacks then replace the default system callbacks which merely print out error messages.
  - Chapter 16, “Generating diagnostic logs” on page 183 describes how a program can, for troubleshooting and debugging purposes, create a log file that records the activities of the DPCL system.

Conventions and terminology used in this book

This book uses the following typographic conventions:

Convention	Usage
bold	**Bold** words or characters represent system elements that you must use literally, such as: command names, flag names, and PE component names (**pedb**, for example).
`constant width`	Subroutine names, examples, and information that the system displays appear in `constant-width` typeface.
italic	*Italicized* words or characters represent variable values that you must supply, file names, and path names. *Italics* are also used for book titles, for the first use of a glossary term, and for general emphasis in text.
[item]	Used to indicate optional items.
<**Key>**	Used to indicate keys you press.

In addition to the highlighting conventions, this manual often uses the C++ scope resolution operator (`::`) when referring to DPCL functions. It does this for the following two reasons:

- First of all, this manual often uses the scope resolution operator for the same reason a program might — to remove ambiguity when multiple classes contain identically-named member functions. For example, since both the `Application` class and the `Process` class have a member function `connect`, this manual refers to these functions as `Application::connect` and `Process::connect` to avoid ambiguity.
- Secondly, this manual also uses the scope resolution operator in a more general way to simply show the class that contains a particular member function; it does this even when the function name is unique within the entire class library. The ambiguity being addressed in these cases is within this manual's text only, and not within DPCL's programming interface. For example, even though the function name `add_process` is unique within the entire class library, this manual may, for clarity, still refer to it as `Application::add_process`.

Accessing AIX man pages for DPCL classes and functions

The book you are reading now provides a general overview of the DPCL classes and functions, but does not provide detailed reference information. For detailed reference information on any of the DPCL classes or functions described here, refer to this book's companion volume — the *IBM Parallel Environment for AIX: DPCL Class Reference*, SA22-7421-00. For your convenience, all of the reference information is also provided as AIX man pages and can be accessed by the AIX **man** command as described in the following table.

Table 1. Accessing AIX man pages for DPCL classes and functions

To access the AIX man page for:	**ENTER:**
a general overview of the DPCL classes:	**man dpcl**
a DPCL class:	**man** *class-name* Where *class-name* is the name of the DPCL class. For example, to access the man page for the `Process` class: **ENTER man process**
a DPCL function:	**man** *function-name* Where *function-name* is the name of the DPCL function. Since different DPCL classes often have functions of the same name, note that the man page may contain multiple function descriptions. For example, for information on the `Process::connect` function, you would: **ENTER man connect** • A man page summarizing both the `Process::connect` and the `Application::connect` functions is displayed. Also note that many of the DPCL classes have overloaded operator functions. These are described in the class' man page.

Accessing DPCL sample applications

For many users, the easiest way to learn DPCL will be by running and examining actual working DPCL applications (called "*DPCL analysis tools*") to see how they use DPCL functions to insert probes into another running application (called a "*DPCL target application*"). For this reason, a set of sample applications was copied to the directory `/usr/lpp/ppe.dpcl/samples/` when you installed DPCL. The following table summarizes the sample applications located in subdirectories under `/usr/lpp/ppe.dpcl/samples`; the order in which the sample directories are listed in this table indicates the recommended order in which you should try the examples. Each of the subdirectories in the following table contains a README file that provides additional information about the program. These subdirectories also contain makefiles for the sample target applications and analysis tools. Refer to Chapter 13, “Compiling and linking the analysis tool and target application” on page 173 for more information on running DPCL programs.

Table 2. Accessing DPCL sample applications	
This subdirectory of /usr/lpp/ppe.dpcl/samples/:	**Contains:**
hello	a "hello world" program in which a target application is instrumented to send a "hello world" string back to the analysis tool. This sample application is described in detail in Chapter 3, “A DPCL hello world program” on page 61.
listmod	an analysis tool that displays all of the *instrumentation points* (locations where probes can be inserted) in the target application.
mpitrace	an analysis tool that finds all MPI function call sites within the target application, and inserts probes before and after the calls. These installed probes will generate messages indicating when execution of the target application has reached the points before and after the calls.
testcov	an analysis tool that periodically prints the number of times each function within the target application has been called. This sample application is described in detail in Appendix A, “A DPCL test coverage tool” on page 189.
diag	an analysis tool that periodically prints the values of global variables within the target application that have changed.
dynamic	an analysis tool that prints the value of a specified global variable when execution reaches a particular location within the target application's code.
profiler	an analysis tool that provides, for a given function within the target application, both the individual and accumulated values for elapsed usage and time.
probe_module	an analysis tool similar to the one in the subdirectory *dynamic*, except that it loads a *probe module* (a compiled object file containing one or more functions written in C) into the target application.
chaotic	an analysis tool that demonstrates how to install a probe at a function entry point within the target application to keep track of how many times the function is called.
stencil	function call count, probe module, and tracing examples. Although similar to some other sample applications, these examples accomplish the same tasks in different ways.

How to send your comments

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

- Send your comments by e-mail to: mhvrcfs@us.ibm.com

  Be sure to include the name of the book, the part number of the book, the version of PE, and, if applicable, the specific location of the text you are commenting on (for example, a page number or table number).
- Fill out one of the forms at the back of this book and return it by mail, by fax, or by giving it to an IBM representative.

National language support (NLS)

For national language support (NLS), all PE components and tools display messages located in externalized message catalogs. English versions of the message catalogs are shipped with the PE program product, but your site may be using its own translated message catalogs. The AIX environment variable **NLSPATH** is used by the various PE components to find the appropriate message catalog. **NLSPATH** specifies a list of directories to search for message catalogs. The directories are searched, in the order listed, to locate the message catalog. In resolving the path to the message catalog, **NLSPATH** is affected by the values of

the environment variables **LC_MESSAGES** and **LANG**. If you get an error saying that a message catalog is not found, and want the default message catalog:

ENTER **export NLSPATH=/usr/lib/nls/msg/%L/%N**

export LANG=C

The PE message catalogs are in English and are located in these directories:

/usr/lib/nls/msg/C
/usr/lib/nls/msg/En_US
/usr/lib/nls/msg/en_US

If your site is using its own translations of the message catalogs, consult your system administrator for the appropriate value of **NLSPATH** or **LANG**. For additional information on NLS and message catalogs, see *IBM Parallel Environment for AIX: Messages*, GA22-7419 and *IBM AIX Version 4 General Programming Concepts: Writing and Debugging Programs*, SC23-2533.

What's new in PE 3.1?

New version of PE

This release represents a new version of IBM Parallel Environment for AIX: PE Version 3, Release 1. In addition to functional enhancements, PE 3.1 includes these changes:

- PE now supports AIX 4.3.3.
- PE Version 1 is no longer supported.
- The visualization tool (VT) has been removed.

New application program interfaces (APIs)

Dynamic probe class library (DPCL) parallel tools development API

This release of PE includes a new set of interfaces called the *dynamic probe class library (DPCL).* With DPCL, tool builders can define instrumentation that can be inserted and removed from an application as it is running. Because the task of generating instrumentation is simplified, designers can develop new tools quickly and easily using DPCL.

Parallel task identification API

PE 3.1 includes a new Parallel Operating Environment (POE) API that allows an application to retrieve the process IDs of all POE master processes running on the same node. This information can be used for accounting, or to get more detailed information about the tasks spawned by these POE processes.

Support for FORTRAN 95

PE 3.1 supports FORTRAN 95 by providing the following new compiler scripts:

- **mpxlf95**
- **mpxlf95_chkpt**
- **mpxlf95_r**

Support for Distributed Computing Environment (DCE) security

This release of PE introduces full support for the Distributed Computing Environment (DCE) through the SP Security Services of IBM Parallel System Support Programs for AIX.

Previous releases of PE provided limited support for DCE / Distributed File System (DFS) security.

The use of DCE security is optional.

MPI enhancements

Full support for MPI I/O

PE 3.1 provides full support for all of the MPI I/O interfaces. Previous releases of PE provided only partial support for MPI I/O.

Support for MPI one-sided communication

With this release, PE now provides full support for MPI one-sided communication. MPI one-sided communication allows one process to specify all communication parameters for the sending operation as well as the receiving operation.

Support for MPI shared memory message passing

In this release, PE introduces support for shared memory MPI message passing on symmetric multiprocessor (SMP) nodes, for the Internet Protocol (IP) library and for the User Space (US) library.

This support includes a new environment variable that lets you select the shared memory protocol. MPI programs may benefit from using shared memory to send messages between two or more tasks that are running on the same node.

Your applications do not need to be changed in any way to take advantage of this support.

Support for 4096 tasks

On a 512-node system, this release of PE supports a maximum of 4096 tasks per User Space job and a maximum of 2048 tasks per IP job, depending on the system and the library you are using.

Note: The **pedb** debugger supports a maximum of 32 tasks.

Removal of VT support

Beginning with this release, PE no longer includes the visualization tool (VT) function. As a result, these commands are no longer available:

- poestat
- vt

Commands no longer supported

Beginning with this release, PE no longer supports these commands:

- mpmkdir
- mprcp
- poeauth
- poestat
- vt

Change to softcopy documentation filesets

In past releases, the **ppe.pedocs** fileset contained all of the PE softcopy publications. Beginning with release 3.1, **ppe.pedocs** will be replaced by these filesets:

- **ppe.html** (HTML files)
- **ppe.man** (man pages)
- **ppe.pdf** (PDF files)

See *IBM Parallel Environment for AIX: Installation* for more information on migration and installation.

DPCL Concepts and Overview

This section contains a general introduction to the Dynamic Probe Class Library. This introduction is contained within three chapters.

- Chapter 1, “What is DPCL?” on page 3 describes, at a very high level, how a program can use the DPCL to dynamically insert instrumentation code patches, or "probes" into an executable program. It describes dynamic instrumentation, and discusses this technology's advantage over more traditional methods of software instrumentation. It then describes our particular application of dynamic instrumentation technology — the DPCL system — by describing its various parts and showing how they work together to instrument an application. This chapter defines terms used throughout the book, describes the three types of probes you can create, and discusses the specific advantages of building tools on the DPCL.
- Chapter 2, “What are the DPCL classes?” on page 25 builds on the overview of the DPCL system contained in Chapter 1, “What is DPCL?” on page 3 by describing the DPCL classes that represent the elements of the DPCL system. In doing so, it describes the purpose, supporting data types, and functions of each class.
- Chapter 3, “A DPCL hello world program” on page 61 contains this book's initial code example — a DPCL version of the familiar "Hello World" program (a simple program that prints out the string "Hello World"). While simple, this program nevertheless illustrates some key features of the DPCL's programming interface. Specifically, it illustrates how a program built on the DPCL system can:
  - initialize itself to use the DPCL system
  - connect to the target application
  - create a simple probe
  - execute the probe within the target application process.

  By illustrating these programming tasks, this description of our "Hello World" program leads directly into this manual's next section (“Standard DPCL Programming Tasks” on page 71), which discusses these same programming tasks in more detail.

Chapter 1. What is DPCL?

DPCL (Dynamic Probe Class Library) is a C++ class library whose application programming interface (API) enables a program to dynamically insert instrumentation code patches, or "probes", into an executing program. The program that uses DPCL calls to insert probes is called the "analysis tool", while the program that accepts and runs the probes is called the "target application".

The DPCL product is an asynchronous software system designed to serve as a foundation for a variety of analysis tools that need to dynamically instrument (insert probes into and remove probes from) target applications. In addition to its API, the DPCL system consists of daemon processes that attach themselves to the target application process(es) to perform most of the actual work, and an asynchronous communication and callback facility that connects the class library to the daemon processes.

This overview describes the various parts of the DPCL system and how they work together to enable analysis tools to instrument target application processes. Before you can understand the individual parts of the DPCL system, however, you need to have a general understanding of the system as a whole. Here is a very high-level description of how an analysis tool uses the DPCL system to instrument a serial or parallel target application. Say you have an executable DPCL program that has been compiled with the DPCL header files and linked with the DPCL library. When you start execution of this program, its code does the following to instrument a target application:

1. **The analysis tool initializes itself to use the DPCL system.** To do this, the analysis tool calls a DPCL initialization function. This enables the analysis tool to respond asynchronously to data sent, via a DPCL daemon, from probes installed in the target application.
2. **The analysis tool connects to, or creates, one or more target application processes.**
   - To connect to one or more target application processes that are already running, the analysis tool calls a function designed for this purpose. When called, this DPCL function starts a DPCL daemon process on the host machine(s) running the target application process(es). The daemon attaches itself to the process(es) and will now perform much of the actual work requested by the analysis tool through the DPCL function calls.
   - To create one or more target application processes running, the analysis tool calls a function designed for this purpose. When called, this DPCL function creates the new process(es) on the particular host as indicated. As with connecting, a DPCL daemon process attaches itself to the process(es) and will now perform much of the actual work requested by the analysis tool through DPCL function calls.
3. **The analysis tool creates snippets of executable instrumentation code (the "probes") to be inserted into the target application process(es).** To do this, the analysis tool uses a DPCL class that has overloaded common operators so that expressions written within the context of the class do not execute locally but instead create internal data structures (called "*probe expressions*") that represent the operation. In addition to these overloaded operators (which implicitly call functions to create probe expressions

representing operations), there are additional functions that your code can explicitly call to create probe expressions representing conditional logic, a sequence of other probe expressions, or a function call.

A *probe expression* is a specific type of data structure called an "*abstract syntax tree*". *Abstract syntax trees* (a term used mainly in compiler technology) are data structures that represent instructions removed from a specific syntactic representation; they are a sort of intermediary stage between source code and executable instructions. When the analysis tool calls DPCL functions to insert a probe expression into one or more target application processes, the DPCL system will manipulate these abstract syntax trees into executable instructions that will run as part of the target application process(es).

While an analysis tool can create probe expressions that perform some simple logic, the programmatic capabilities of probe expressions are rather limited. For this reason, a probe expression may optionally call functions. Specifically, a probe expression can call:

- a DPCL system-defined function for sending collected data back to the analysis tool.
- AIX functions (like **getrusage**, **times**, and **vtimes**).
- functions contained in the target application.
- A C function compiled into an object file called a "*probe module*". Certain DPCL functions enable you to load such modules into one or more target application processes. Once a probe module is loaded into a target application process, the analysis tool can direct the target process to call any of its functions. Note, however, that probe module functions must be written in C; other languages (such as C++) are not supported.

4. **The analysis tool inserts the probes into the target application for execution.** To do this, the analysis tool calls certain DPCL functions that, like many DPCL functions, send service request messages to a DPCL daemon. This DPCL daemon then translates the messages into the desired action.

   There are three general ways an analysis tool can execute probes within a target application process. These three general approaches correspond to the three types of probes — *point probes*, *phase probes*, and *one-shot probes*. The analysis tool can:

   - place probes at a specific locations in the target application code. Such probes are called "*point probes*", and, when activated by the analysis tool, will run whenever execution reaches their installed location in the code.
   - install probes that are executed within the target application process(es) upon the expiration of a timer regardless of what the target application is doing. Such probes are called "*phase probes*". The invocation of phase probes is governed by data structures called *phases*. Each phase defines that phase probe(s) to be invoked and the time interval between successive invocations of the phase probe(s).
   - explicitly execute a probe within the target application process(es) regardless of what the target application is doing. Such probes are called "*one-shot probes*".

5. **Enters the DPCL main event loop.** This enables the analysis tool to interface asynchronously with the DPCL system. To do this, the analysis tool calls a DPCL function that puts execution in a processing loop. This enables the

analysis tool to respond asynchronously to, for example, messages from DPCL daemons.

6. **The analysis tool responds to data sent, via a DPCL daemon, from its installed probes.** To do this, the analysis tool must have created a *DPCL callback routine* (also called simply a "*DPCL callback*") designed to respond to the probe data. The probe can send data in a message to the DPCL daemon. The daemon then passes this message back to the analysis tool along with information indicating which callback should be used to respond to this probe's data. The analysis tool, meanwhile, has been sitting in its main event loop waiting for just such an event; it responds to the message by calling the appropriate callback and supplying it with the collected probe data.

That is a very high-level view of how the analysis tool interacts with the DPCL system in order to instrument a target application. The rest of this overview will provide a more-detailed description of this process and the DPCL system. First, it describes dynamic instrumentation in general by answering the following questions:

- What is dynamic instrumentation?
- What are the advantages of dynamic instrumentation?

Next, this overview describes the basic architecture of the DPCL system by answering the questions:

- What is a DPCL target application?
- What is a DPCL analysis tool? In describing what a DPCL analysis tool is, this overview also answers the questions:
  - What is the DPCL API?
  - What are DPCL callbacks?
- What are the DPCL daemons?
- What is a probe? In describing what a probe is, this overview also answers the questions:
  - What is a probe expression?
  - What is a probe module?
  - What are the three types of probes?

Once these questions have been answered, you will have a clearer idea of the various parts of the DPCL system, and how they are organized. This overview then concludes by explaining why it is advantageous to build analysis tools on the DPCL system.

What is dynamic instrumentation?

Dynamic instrumentation refers to a specific type of software instrumentation. *Software instrumentation* refers to code that is inserted into a program to gather information regarding the program's run. As the instrumented application executes, the instrumented code then generates the desired information, which could include performance, trace, test coverage, diagnostic, or other data. Traditionally, software instrumentation has been inserted:

- manually by a programmer editing a program's source code
- automatically by a compiler designed to generate the instrumentation

- by linking in an instrumented version of a library
- directly into the executable by an application designed for this purpose

Dynamic instrumentation is distinct from these more traditional methods of software instrumentation because it can be added and removed while the application is running. The application does not need to be terminated and restarted from the beginning in order to add and remove instrumentation.

What are the advantages of dynamic instrumentation?

We chose to base the DPCL product on dynamic instrumentation technology, because dynamic instrumentation offers several key benefits that cannot be realized by traditional software instrumentation approaches. Specifically, a DPCL analysis tool's ability to add and remove instrumentation probes while the target application is running means that the analysis tool can perform run-time analysis. In other words, it can examine the target application's behavior without waiting for its execution to complete. This is especially useful for:

- examining programs, such as database servers, that do not normally terminate. Since analysis tools built on the DPCL system can insert instrumentation probes long after the target application has begun executing, instrumenting such programs is not a problem.
- examining long-running numerical programs, especially when the program is repetitive and the general execution structure can be obtained from a few early iterations.
- visualizing complex or long-running programs with a minimum of secondary storage consumption. Because it can visualize a target application's behavior as it actually runs, an analysis tool built on the DPCL system can avoid storing large volumes of trace or other collected data.
- enabling the user of the analysis tool to interactively tailor the type of data collected to match his or her needs. A user might want to do this, for example, in order to examine different hypotheses regarding the target application's performance. Since the user could alter the type of data collected without having to stop and restart the target application, this reduces the need to run the target application multiple times in order to collect the required data.

Additional advantages of dynamic instrumentation in general, as well as specific advantages of the DPCL system, are described in “Why is it advantageous to build analysis tools on the DPCL system?” on page 21. Before you can fully appreciate the advantages of building analysis tools on the DPCL system, you need to understand more about the DPCL system (as described next in “What is the DPCL system?”).

What is the DPCL system?

The DPCL system is an asynchronous software system whose client/server architecture enables analysis tools to connect to, and insert instrumentation probes into, one or more target application processes. What's more, the DPCL system encapsulates a parallel infrastructure, making it ideally suited for analyzing parallel programs. The DPCL system consists of several conceptual parts including:

- A class library whose Application Programming Interface (API) enables you to build analysis tools on the DPCL system.

- Daemon processes that connect to the target application process(es) to perform work requested by the analysis tool through the DPCL API function calls.
- Instrumentation probes that the analysis tool defines and inserts (via DPCL API function calls and daemons) into the target application to collect data.
- An asynchronous communication callback facility that connects the class library with the daemons. It is this callback facility that enables an analysis tool to respond to data collected and sent by its probes.

The following figure illustrates how the parts of the DPCL system work together to enable an analysis tool to instrument a serial target application.

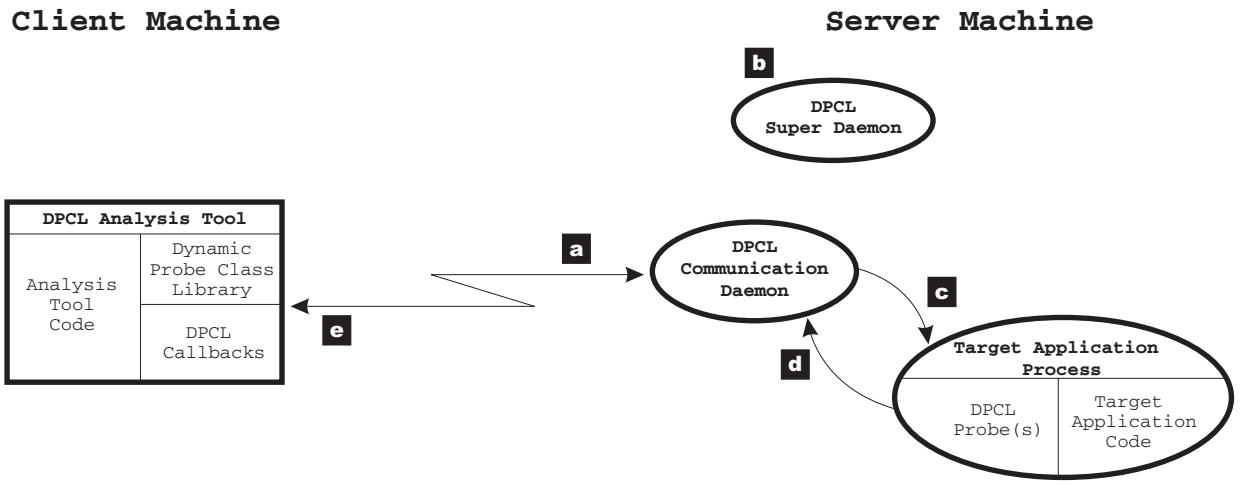

Figure 1. Instrumenting a serial target application

The preceding figure illustrates how the parts of the DPCL system work together to enable an analysis tool to instrument a serial target application. For additional explanation of this figure, refer to the following key:

a The analysis tool calls DPCL functions which request services from the DPCL communication daemon.

b The DPCL superdaemon coordinates the creation and removal of the DPCL communication daemon. It establishes the socket connection to the analysis tool, and then spawns (and transfers the socket connection to) the DPCL communication daemon.

c The DPCL communication daemon translates API function requests into the desired action (for example, the installation or removal of probes in the target application).

d Probes within the target application process send data to the DPCL communication daemon.

e The DPCL communication daemon forwards data collected by the probe(s) back to the analysis tool, triggering the appropriate callback routine in the analysis tool.

This next figure is similar to the preceding one, except that it shows how the parts of the DPCL system work together to enable an analysis tool to instrument a parallel target application. The preceding figure's key applies to this next figure as

well; refer to it for additional information. Note in this figure that only a single DPCL communication daemon runs, per user, on each server machine. If the analysis tool connects to multiple target application processes running on the same server machine, then that server machine's DPCL communication daemon will coordinate the communication with all of the processes.

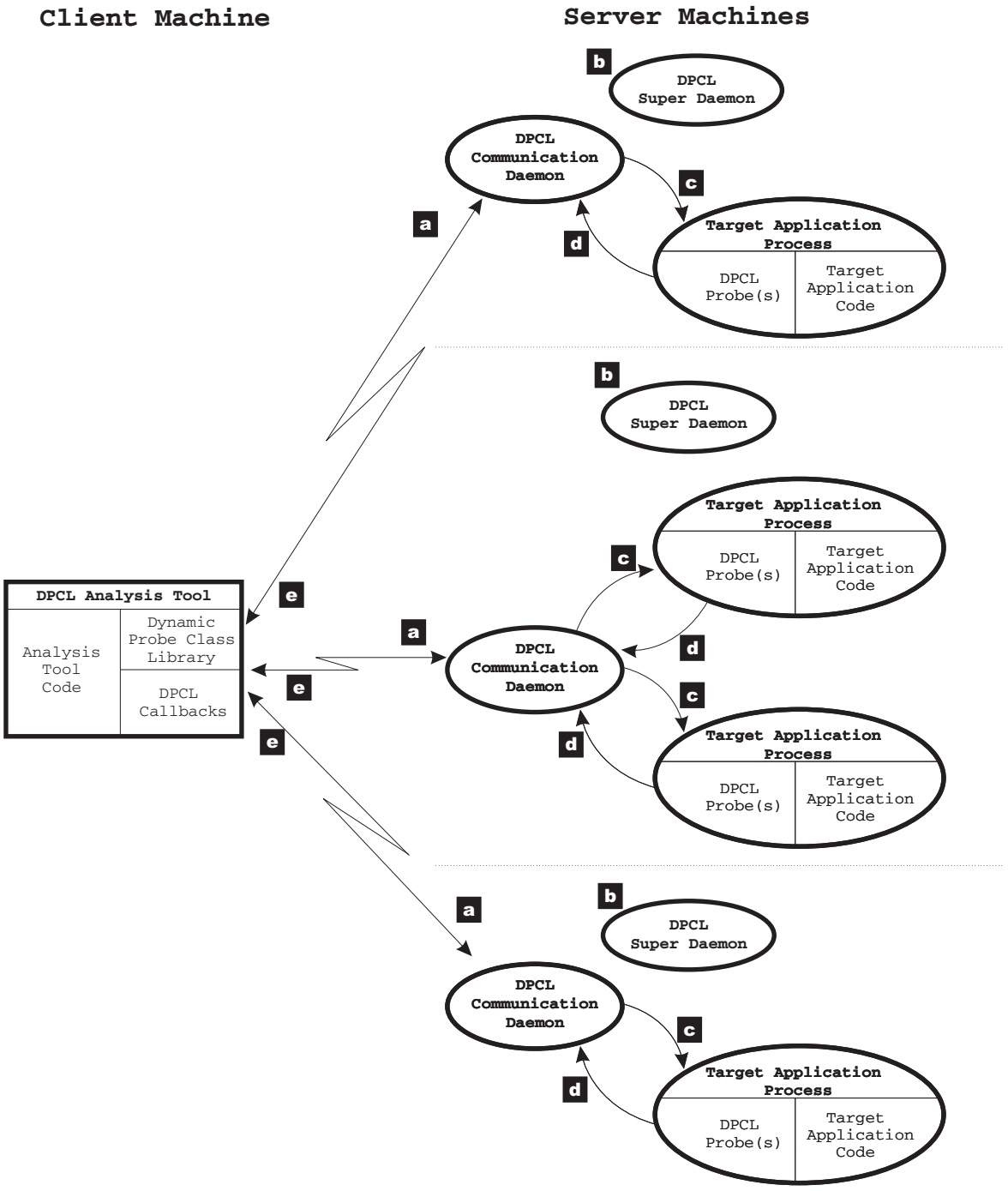

Figure 2. Instrumenting a parallel target application

The remainder of this section describes each part of the DPCL system (the target application, the analysis tool, the class library, the callbacks, the daemons, and the probes) in greater detail.

What is a DPCL target application?

A target application is an executable program into which the analysis tool inserts probes. A target application could be a serial or a parallel program. Furthermore, if the target application is a parallel program, it could follow either the Single Program Multiple Data (SPMD) or the Multiple Program Multiple Data (MPMD) model, and may be designed for either a message-passing or a shared-memory system.

What is a DPCL analysis tool?

An analysis tool (in the context of the DPCL product) is a C++ application that links in the DPCL library and uses the DPCL API calls to instrument (create probes and insert them into) one or more target application processes. In addition to containing code not related to accessing DPCL system functionality (such as defining the user interface), the analysis tool will also contain DPCL callback routines designed to respond to data sent back from the probes it has installed in the target application. Typically, an analysis tool is designed to measure program efficiency, confirm program correctness, or monitor program execution.

The DPCL system is designed to provide you, the creator of analysis tools, with a scalable general-purpose infrastructure for instrumenting target applications. In other words, the DPCL system concentrates on enabling your analysis tool to connect to the target application process(es), and then dynamically insert and remove probes as needed. You concentrate on creating the actual probes, and leverage the DPCL system's ability to insert them into one or more target application processes. What's more, the DPCL system encapsulates a parallel infrastructure, making it ideally suited for analyzing parallel programs. This design affords you a large degree of flexibility. For example, an analysis tool could be a complex and general-purpose tool like a debugger, or it might be a simple and specialized tool designed for only one particular program, user, or situation.

What is the DPCL API?

DPCL's Application Programming Interface (API) is the key means by which the analysis tool interacts with the DPCL system to effectively instrument a target application. Along with the DPCL callbacks (which enable an analysis tool to respond asynchronously to data sent from installed probes), the DPCL API is what enables the analysis tool to leverage the capabilities of the DPCL system. The DPCL API contains:

- three classes that the analysis tool can use to represent, and act upon, the target application. These are the `Process` class (used to represent a single AIX process), the `Application` class (used to represent a group of related AIX processes), and the `PoeAppl` class (a class derived from the `Application` class and used specifically to represent POE applications). Member functions of these classes enable an analysis tool to:
  - connect to target application processes
  - create new target application processes in a stopped state
  - start execution of target application processes
  - allocate and free memory for probes within target application processes
  - execute probes within target application processes
  - disconnect from target application processes
  - suspend and resume execution of target application processes.

  - terminate target application processes.

- a `ProbeExp` class that the analysis tool can use to build data structures called "probe expressions". These probe expressions represent code that the analysis tool can execute within one or more target application processes. The `ProbeExp` class overloads common operators so that expressions do not execute locally, but instead call member functions that create the probe expressions. Additional member functions of the `ProbeExp` class enable the analysis tool to create probe expressions representing conditional logic, a sequence of probe expressions, or a function call. The analysis tool can then insert these probe expressions into the target application process(es) for execution.
- Two classes that enable an analysis tool to navigate the source code structure of the target application, and identify locations where the analysis tool can safely install instrumentation probes. These classes are the:
  - `SourceObj` class. Objects of this class represent part of the source code structure, and a group of such objects provide the hierarchical representation of the target application's source code which the analysis tool can navigate.
  - `InstPoint` class. Objects of this class represent locations in the target application code where the analysis tool can install probes.

Be aware that the above is just a quick summary of some of the main classes and functional capabilities of the DPCL API. For more information about the DPCL classes, refer to Chapter 2, “What are the DPCL classes?” on page 25. For complete reference information on the DPCL API, refer to the *IBM Parallel Environment for AIX: DPCL Class Reference*.

What are blocking and nonblocking API calls?: There are two types of DPCL API calls — blocking calls (also referred to as pseudo-synchronous or semi-synchronous service requests) and nonblocking calls (also referred to as asynchronous service requests). Much of the functionality of the DPCL system is available in both blocking and nonblocking versions. For example, to connect to a target application, an analysis tool could call either a blocking function (`bconnect`) or a nonblocking function (`connect`). As this example implies, the naming convention for a blocking function is to prefix the letter "`b`" to the name of the nonblocking function.

To understand why the DPCL API includes both blocking and nonblocking versions of the same functionality, it is important to recall that the DPCL system is an asynchronous system that, by definition, acts upon events that, if they occur, will do so at an undetermined time and in an undetermined order. The nonblocking functions are designed to take advantage of the asynchronous nature of the DPCL system; calls to such functions return immediately without waiting for a response from the DPCL system. When an analysis tool calls an asynchronous, nonblocking function, it can specify the name of a callback routine that will respond to status returned by the DPCL system. The callback not only enables the analysis tool to perform status error checking, but also enables it to be structured in a more event-driven manner. When analyzing parallel programs, certain performance benefits can usually be realized by leveraging the event-driven nature of the asynchronous functions. The `Application` class, for example, is a grouping of related `Process` class objects that enables the analysis tool code to manipulate a set of related AIX processes as a single unit. For the asynchronous `Application` class functions, the callback routine that responds to the successful or unsuccessful completion of the operation will execute for each process individually. The callback

routine, in addition to performing error checking to ensure that the operation was successful for the particular process, could contain code for the next action to be performed on the process. In other words, the analysis tool could continue work on one process without waiting for the operation to complete on the other processes.

The blocking functions, on the other hand, are designed to hide the complexity of the asynchronous system; calls to such functions do not return control to the analysis tool until they either succeed or fail in carrying out the requested service. This means that callbacks are not needed; instead, the code to act upon the function return value can be placed right after to call to the function. Keep in mind, however, that these so-called "blocking" functions are not truly synchronous; instead they merely mimic a synchronous system while allowing the DPCL system to continue processing events not related to the blocking request. For example, data being sent from probes could still be processed by other callback routines while execution is supposedly "blocked" and waiting for the function to return. That is why the blocking functions are referred to as "pseudo-synchronous" or "semi-synchronous" service requests. We designed the blocking functions to be pseudo-synchronous in order to provide a simple blocking interface that helped avoid program deadlock by allowing the DPCL system to continue processing other requests while "blocked".

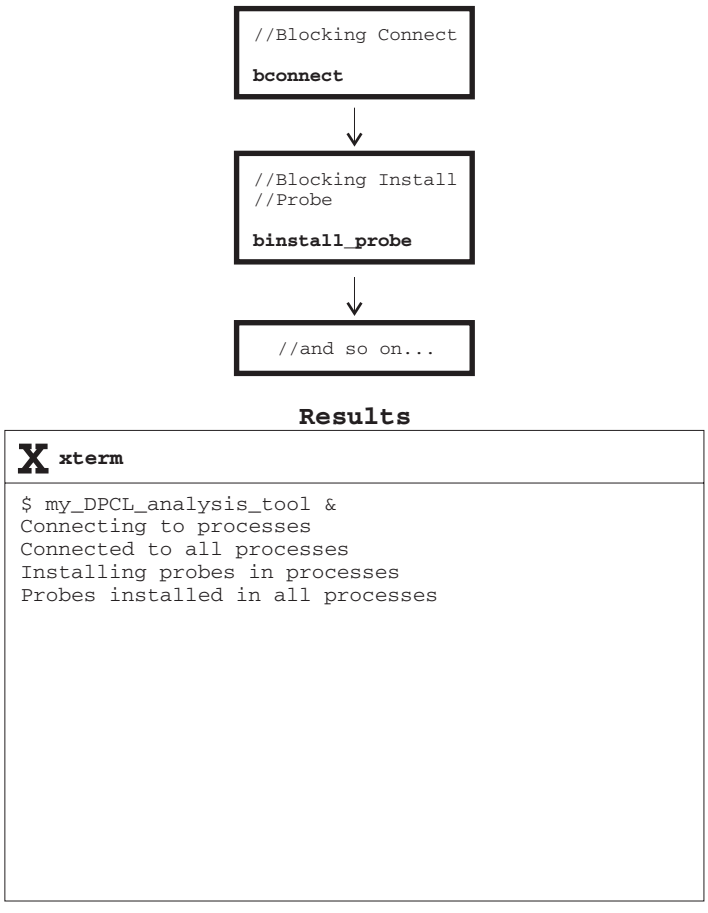

Figure 3. Blocking function calls (pseudo-synchronous service requests). Calls to blocking functions do not return control to the analysis tool until they either succeed or fail in carrying out the requested service. The blocking functions provide a simpler interface, and so are preferable for applications that don't need to take advantage of the finer level of control available in an asynchronous system. In particular, if the analysis tool is instrumenting a serial application, the blocking functions are probably preferable.

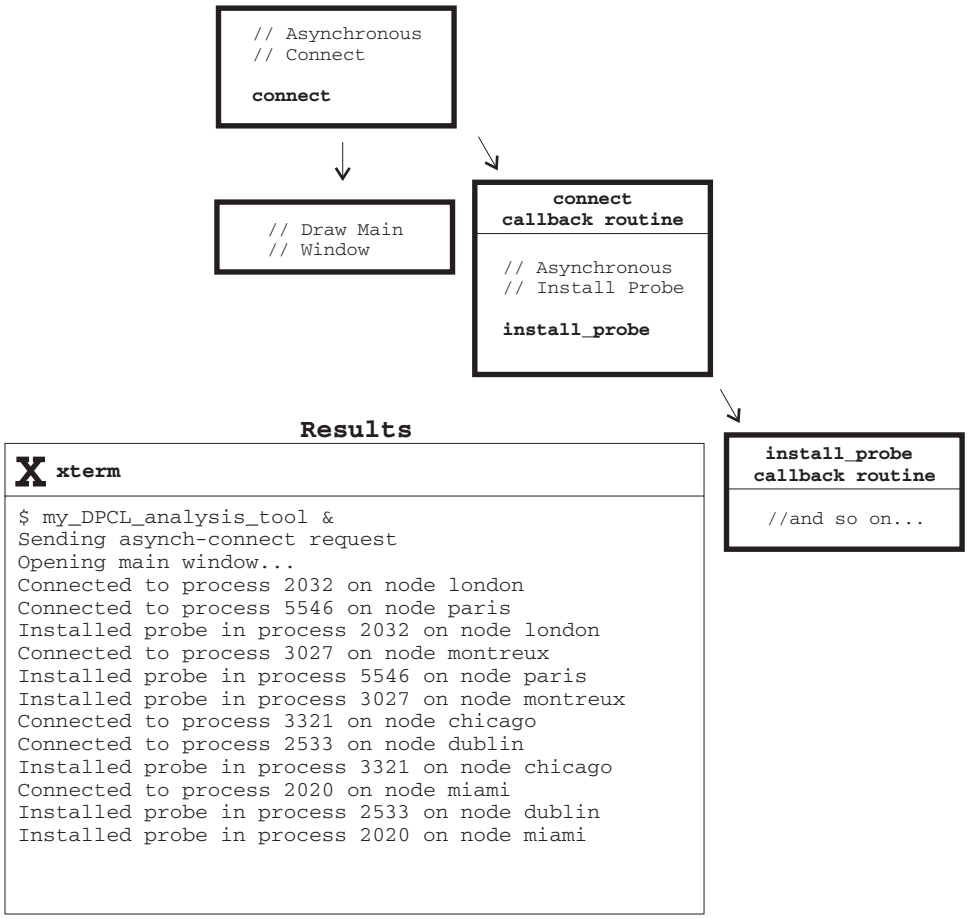

Figure 4. Nonblocking function calls (asynchronous service requests). The nonblocking functions are designed to take advantage of the asynchronous nature of the DPCL system. Calls to nonblocking functions return immediately upon issuing the service request, without waiting for a response to the request from the DPCL system. Instead, a callback routine responds to the return value from the system. The nonblocking functions can be harder to program, but enable the analysis tool to leverage the finer level of programmatic control.

What are DPCL callbacks?

DPCL callbacks are routines called by the DPCL system when certain messages arrive from a DPCL daemon. When an analysis tool initializes itself to use the DPCL system, one of the things it does is enter the DPCL main event loop so that it can interface asynchronously with the DPCL system. The DPCL main event loop listens to file descriptors and sockets for input; there will be one socket for each remote node to which the analysis tool is connected. Remember that the communication to and from each target application process is handled by a DPCL daemon. A DPCL daemon may send two types of messages to the analysis tool. A DPCL daemon may send a message:

- indicating whether a particular service requested by a DPCL function succeeded or failed.
- containing data from probes that the analysis tool has installed in the target application.

When the DPCL main event loop detects input on a file descriptor that is connected to a DPCL daemon, it calls a dispatch routine for the file descriptor or socket. If the

input is on a file descriptor representing a socket connection to a DPCL daemon, the message is examined and the appropriate callback for the message type is executed. Since there are two types of messages that can be sent from a DPCL daemon, there are two types of callbacks — *acknowledgment callbacks* and *data callbacks*. *Acknowledgment callbacks* are callbacks that respond to the success or failure of an asynchronous, nonblocking, function call. *Data callbacks* are callbacks that respond to probe data forwarded by the DPCL daemon.

All callback routines have the same parameters. These parameters include:

- a data structure with values indicating:
  - the socket or file descriptor over which the message was received
  - the message type
  - the size of the message sent
- a tag value supplied by the analysis tool when it installed the probe or called the asynchronous function.
- a pointer to the object that registered the callback routine. This would be the instance of the DPCL class object that installed the probe or called the asynchronous function.
- a raw byte stream containing the actual message — the probe data or the function return status.

What are the DPCL daemons?

There are two types of DPCL daemons — *DPCL communication daemons* and *DPCL superdaemons*.

- *DPCL superdaemons* are created the first time an analysis tool calls an API routine to connect to one or more target application processes on a given node. These superdaemons create the DPCL communication daemons, and are responsible for ensuring that only one such daemon exists on each remote host. They also perform user authentication on the remote host.
- *DPCL communication daemons* handle the communication between the analysis tool and target application processes. This is the daemon attached to the target application process that will perform much of the actual work requested, via DPCL function calls, by the analysis tool. This daemon also relays the data collected by instrumentation probes within the target application back to the analysis tool.

In order for the DPCL daemon processes to be as unobtrusive as possible, we have designed the DPCL system so that only one DPCL communication daemon per user and only one DPCL superdaemon will be running on each host machine at a time. Furthermore, we have ensured that these daemons are not persistent; they will terminate when the analysis tool issues a service request to disconnect from the target application process.

To better understand the purpose and life cycle of the two types of DPCL daemons, it is worthwhile to understand how they are created, used, and destroyed. First, an analysis tool will need to connect to the target application processes. To do this:

1. The analysis tool calls a DPCL function to connect to one or more target application processes.

2. The DPCL function creates a socket connection from the analysis tool to the `inetd` daemon running on the host where the target application process is running.
3. The `inetd` daemon spawns a new daemon process — a DPCL superdaemon process. This DPCL superdaemon inherits the analysis tool socket connection.
4. The DPCL superdaemon checks to see if there is already a DPCL superdaemon running on the host machine. If so, it transfers the analysis tool socket connection to the existing DPCL superdaemon and exits. If there is no other existing DPCL superdaemon on the host machine, then the DPCL superdaemon does not exit; it becomes the DPCL superdaemon for the host machine.
5. The DPCL superdaemon performs user authentication to ensure that the analysis tool user is authorized to run on the remote host.
6. The DPCL superdaemon checks to see if there is a DPCL communication daemon running for this user on the host machine.
   - If there is a DPCL communication daemon for this user already running, the DPCL superdaemon passes the analysis tool socket connection over to the DPCL communication daemon.
   - If there is no DPCL communication daemon for this user running on the host, the DPCL superdaemon spawns one. This DPCL communication daemon inherits the analysis tool socket connection and will handle communication between the analysis tool and the installed probes.

If the target application is a parallel application, similar connections and daemons need to be created for each host on which the target application processes are running. Once the analysis tool is connected via the DPCL communication daemon(s) to the target application process(es):

- The analysis tool may issue API function calls to, for example, install or remove probes. The DPCL functions send messages to the DPCL communication daemon(s), which translate the messages into the desired action.
- Probes installed within the target application process(es) may send collected data to the DPCL communication daemon(s), which will relay these messages back to the analysis tool.

Finally, when the analysis tool is done collecting data, it will need to disconnect from the target application processes. To do this:

1. The analysis tool calls a DPCL function to disconnect from one or more target application processes.
2. The DPCL function sends a disconnect message to the DPCL communication daemon. Provided the DPCL communication daemon is not connected to other analysis tools, it asks the DPCL superdaemon if it can exit.
3. If there are no analysis tools attempting to connect to one or more target application processes on this host, the DPCL superdaemon tells the DPCL communication daemon that it can exit.
4. The DPCL communication daemon exits.
5. If the DPCL communication daemon that exited was the last DPCL communication daemon running on this host, the DPCL superdaemon exits.

Since we have designed the DPCL system so that only one DPCL communication daemon per user will be running on a given host, this means that a single DPCL communication daemon may be coordinating the communication between multiple target application processes and/or multiple analysis tools. The following figure illustrates the possible connection variations that can exist for a single user on a single host machine.

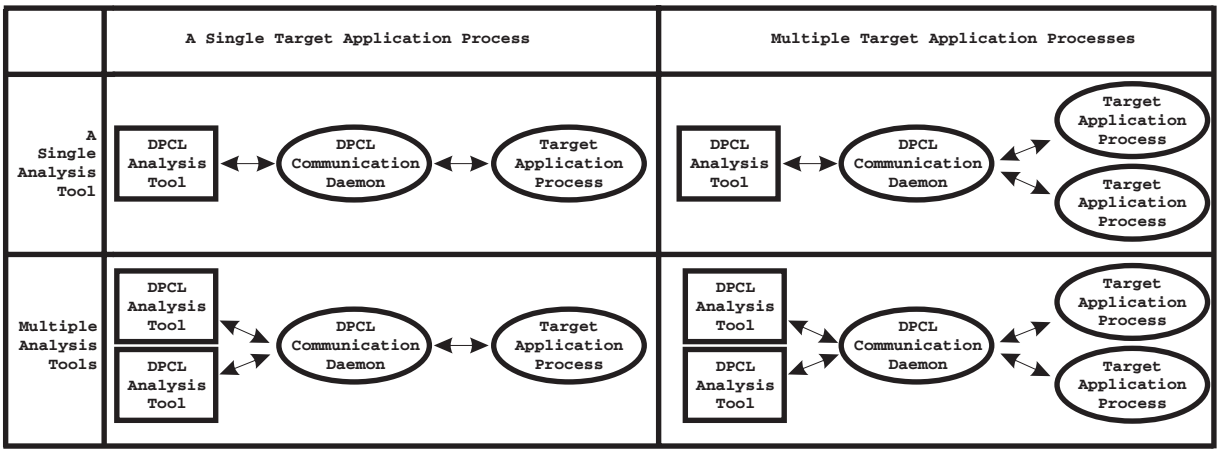

Figure 5. DPCL communication daemon. Each host machine has only one DPCL communication daemon per user. It may coordinate the communication between multiple target application processes and/or multiple analysis tools. Keep in mind that this figure illustrates a single user and a single host machine. Each user connected to a DPCL target application process on the host will have a separate DPCL communication daemon running. Likewise, each user will have a separate DPCL communication daemon for each host that is running target application process(es) to which he or she is connected.

What is a probe?

The term probe refers to the software instrumentation code patch that your analysis tool can insert into the target application. Probes are created by the analysis tool, and therefore are able to perform any work required by the tool. For example, depending on the needs of the analysis tool, probes could be inserted into the target application to collect and report performance information (such as execution time), keep track of pass counts for test coverage tools, or report or modify the contents of variables for debuggers.

Probes are created by the analysis tool using a combination of *probe expressions* and *probe modules* (described next in “What is a probe expression?” and “What is a probe module?” on page 17). For the purposes of this book, a probe is defined as "a probe expression that may optionally call functions".

What is a probe expression?

A *probe expression* is a simple instruction or sequence of instructions that represents the executable code to be inserted into the target application. Probe expressions are *abstract syntax trees* — data structures that represent the logic to be performed by the probe within the target application process(es).

The term "abstract syntax tree" is one we have borrowed from compiler technology. These data structures are called "abstract" because they are removed from the syntactic representation of the code. For example, an abstract syntax tree for the expression a + (b x c) is identical to the abstract syntax tree for the expression a +

b x c (where only precedence rules force the multiplication operation to be performed first).

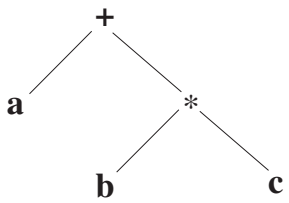

*Figure 6. Abstract syntax tree. This abstract syntax tree formed from either the expression a + (b * c) or the expression a + b * c.*

Compilers need to create abstract syntax trees from a program's source code as an intermediary stage before manipulating and converting the data structure into executable instructions. Since the DPCL system also needs to create executable instructions (for insertion into one or more target application processes), it also needs to create these abstract syntax trees. When the analysis tool inserts a probe expression into one of more target application processes, the DPCL system uses compilation techniques to manipulate these abstract syntax trees into executable instructions that will run as part of the target application process(es).

From the DPCL programmer's point of view, the procedure for creating a probe expression can be a "building block" task in which smaller probe expressions are eventually combined and sequenced into the full probe expression.

For example, the analysis tool can create probe expressions representing constant or variable values, and then combine these into more complex probe expressions representing simple operations on the values, or function calls that pass the values as parameters to the function. The analysis tool could then take two of these more complex probe expressions and combine them into a single probe expression that represents a sequence of the two existing expressions. Then the analysis tool could join two such sequences into a longer sequence or combine them into a conditional statement. This process of combining and sequencing smaller probe expressions into larger ones would continue, depending on the complexity of the probe logic, until the analysis tool has a single probe expression representing the full probe logic.

The class for creating probe expressions is the `ProbeExp` class. Constructors of this class enable your code to create probe expressions that represent temporary data variables. Functions of other DPCL classes enable your code to create probe expressions that represent persistent data variables. To create probe expressions to represent operations, the `ProbeExp` class has overloaded common operators so that expressions written within the context of the class do not execute locally, but instead call member functions designed to create a probe expression that represents the particular operation. Probe expressions to represent arithmetic, bitwise, logical, relational, assignment, and pointer operations can all be created in this way. These probe expressions can then in turn be used as subexpressions in forming other probe expressions — ones representing more complex operations. Other functions of the `ProbeExp` class (ones that must be called explicitly) enable your code to create a probe expression to represent a sequence of two existing probe expressions, a conditional statement, or a function call.

Although an analysis tool can create probe expressions to perform conditional control flow, integer arithmetic, and bitwise operations, the programmatic capabilities of probe expressions are rather limited. When more complicated probe logic is needed (such as iteration, recursion, and complex data structure manipulation), a probe expression can direct the target application to call a function in a *probe module*.

What is a probe module?

A *probe module* is a compiled object file containing one or more functions written in C. Once an analysis tool loads a particular probe module into a target application, a probe is able to call any of the functions contained in the module.

What are the three types of probes?

As already stated, a probe is a probe expression that may optionally call functions. There are three types of probes; they are differentiated by the manner in which their execution is triggered. The three types of probes are:

- *point probes* (which are installed at particular locations in the target application code and, when in an activated state, are triggered whenever execution reaches that location in the code).
- *phase probes* (which are triggered by expiration of a timer and executed regardless of what code the target application is executing).
- *one-shot probes* (which are executed once and immediately, regardless of what code the target application is executing).

Each probe type has different intended uses, and together are designed to enable an analysis tool to efficiently instrument a target application. By "efficiently instrument", we mean "to collect the necessary data and display it in a timely manner while minimizing the instrumentation's intrusion cost to the target application".

What is a point probe?: *Point probes* are probes that the analysis tool places at particular locations within one or more target application processes. When placed in an activated state by the analysis tool, a point probe will run as part of a target application process whenever execution reaches its installed location in the code. The fact that point probes are associated with particular locations within the target application code makes them markedly different from the other two types of probes (which are executed at a particular time regardless of what code the target application is executing).

To install a point probe within one or more target application processes, the analysis tool must navigate the source code structure of the target application to identify locations where it can safely install point probes. The analysis tool navigates the source code structure by means of source objects (represented by instances of the `SourceObj` class); the locations where point probes can be installed are called instrumentation points (represented by instances of the `InstPoint` class).

What are source objects?: Source objects provide a coarse, source-code-level, view of a target application process, and enable an analysis tool to display or navigate a hierarchical representation of a particular target application process. After connecting to a process, the analysis tool can get the top-level source object (called the "program object") for the process; the analysis tool does this by calling the member function `Process::get_program_object`. This function returns the top-level source object (an instance of the DPCL class `SourceObj`). Since

applications can be quite large, the initial source object provides only a very coarse view of the source structure; essentially, it is just a list of the modules (compilation units) contained in the target application process. Each of these modules is itself a source object and is considered a child of the program source object.

To navigate down into the source structure of a module, an analysis tool gets a reference to one of these module source objects (using the member function `SourceObj::child`) and expands it (using the `SourceObj::expand` or its blocking equivalent `SourceObj::bexpand`). Expanding a module source object returns the additional structure of the module — including data, functions, and instrumentation points.

It is important to keep in mind that the program object and all its child module objects reflect the source hierarchy associated with a particular process only. This means that, in some cases, the analysis tool will need to navigate multiple source hierarchies (as described in the following table).

If the target application is:	Then:
A serial program.	The analysis tool need only navigate the target application's single source hierarchy.
A parallel program that follows the Single Program Multiple Data (SPMD) model.	Each process in the target application has the same source and, therefore, the same source hierarchy. The analysis tool need only navigate a single source hierarchy. The analysis tool still has the option to either insert identical instrumentation in each of the processes, or else instrument the processes differently.
A parallel program that follows the Multiple Program Multiple Data (MPMD) model.	There are multiple programs, and, therefore, the analysis tool will need to navigate multiple source objects.

What are instrumentation points?: Instrumentation points are locations within a target application process where an analysis tool can install point probes. Instrumentation points are locations that the DPCL system determines are safe to insert new code. Such locations are:

- function entry
- function exit
- function call

Instrumentation points are obtained from source objects, at the function level, using the `SourceObj::exclusive_point` or `SourceObj::inclusive_point` functions. Both functions take an integer index value as an input value and return an instrumentation point as a result. The difference between the two is that the `SourceObj::exclusive_point` function gives the analysis tool access only to instrumentation points that are tied to that particular source object in the source object hierarchy, while the `SourceObj::inclusive_point` function gives the analysis tool access to all instrumentation points associated with the given source object and all of its lower level source objects in the source object hierarchy.

When should an analysis tool use a point probe?: An analysis tool should use a point probe when it needs to collect data associated with a particular location in the target application's code.

To better understand when it is useful, and when it is not useful, to use a point probe, consider the following hypothetical situation. Say your analysis tool is a

profiler that needs to measure the accumulation of floating point counts by function. Say also that this analysis tool is a Java™ client that needs to periodically refresh itself to display the newly-collected data to the user. Since the functions are specific locations in the target application code, you would need to use point probes to measure them effectively. In this particular example, you would set up two point probes for each function — one at the beginning of the function and one at the end. Each time the function starts executing, the first probe would determine the number of floating point instructions executed up to that point. Later, the second probe would determine the number of floating point instructions executed, and, by subtracting the first figure from the second, the number of floating point instructions executed within the function.

So in this example, you would use a set of point probes to collect the data. Keep in mind, however, that it is not enough to simply accumulate collected data within the target application process(es) as we have in this example. Remember that we also need to communicate this information back to the analysis tool. In this case, we have said that our hypothetical application needs to periodically refresh its Java client to display the newly-collected data. We could have the point probes themselves send their collected data back to the analysis tool, but this would not be an efficient solution. You would not, in this example, want to send data back to the analysis tool using the point probes because such probes located in frequently-executed functions could swamp the network with messages and, in doing so, take a valuable resource away from the target application. This solution would be unacceptable, as it would likely slow the target application appreciably; the instrumented version of the target application would no longer be representative of the actual, uninstrumented, version of the target application. While point probes are useful for collecting the data in this example, the actual communication of that data back to the analysis tool would be better handled using phase probes (as described in “What is a phase probe?”) or one-shot probes (as described in “What is a one-shot probe?” on page 20).

What is a phase probe?: *Phase probes* are probes that are executed periodically, upon expiration of a timer, regardless of what part of the target application's code is executing. A phase probe, unlike the other two types of probes, must call a probe module function; in other words, a phase probe cannot be a simple probe expression that does not call a probe module function. The control mechanism for invoking these time-initiated phase probes is called a *phase*.

What is a phase?: *Phases* are the control mechanism for invoking phase probes at set intervals. Represented by instances of the `Phase` class, phases enable your analysis tool code to specify the particular phase probe(s) to be invoked and the CPU-time interval at which their execution is triggered. The set interval at which a phase is activated to invoke its phase probes is called the *phase period*. Although the phase period is initially defined when the analysis tool first creates the phase, the analysis tool can later lengthen or shorten the phase period as desired.

A phase can, each time the phase period expires, call up to three phase probes. As already stated, a phase probe must call a probe module function, so the phase is actually triggering calls to up to three probe module functions — a *begin function*, a *data function*, and an *end function*. While the phase must, in order to be useful, call at least one of these functions, any one of them is optional. At the very least, an analysis tool will usually supply a data function.

When a phase is added to a target application process, it will, once the phase period expires, be activated by the DPCL system. (The DPCL system uses a SIGPROF signal to activate a phase, so be aware that target applications that themselves use the SIGPROF signal cannot be instrumented with phases.) Once the phase is activated, it will call the phase probe module functions that have been associated with it. The first phase probe it calls is the one identifying the begin function (provided one has been specified). Typically, the begin function will perform any setup tasks that may be required. When the begin function completes, the phase calls the phase probe that identifies the data function (provided one has been specified). The data function executes once per datum that the analysis tool will have previously allocated and associated with this phase. Executing once per datum enables the data function to perform the same actions on different data. Each datum, for example, could be a separate counter — each incremented by the same data function. If the analysis tool does not associate any data with the phase, then the data function will not execute. When the data function finishes executing for the last datum, the phase calls the phase probe that identifies the end function (provided one has been specified). Typically, the end function performs any clean up chores that may be required.

When should an analysis tool use phases to invoke phase probes?: An analysis tool should use phases and phase probes whenever necessary work is best done on a periodic basis. For example, in “When should an analysis tool use a point probe?” on page 18, we described a hypothetical situation in which an analysis tool needed to measure the accumulation of floating point counts by function. While we determined that point probes placed inside functions were the best way to collect data, we also determined that they were an impractical way to send that data back to the analysis tool. We determined point probes would be impractical because such probes located within frequently-executed functions would utilize too much of the available network communication resource, and so slow the target application unacceptably.

In this example, a phase that triggers one or more phase probes at a set interval would be an ideal way to communicate the data collected by the point probes back to the analysis tool. By using a phase, you would be able to tune how often you send updates back to the target application. While you cannot control how often the point probes are executed to gather the data, you can use a phase to govern how often a phase probe is triggered to send the collected data back to the analysis tool. What's more, since the analysis tool can modify the phase period to trigger execution of the phase probe more frequently or less frequently, the analysis tool could dynamically govern how often the data is sent. For example, our hypothetical analysis tool could also monitor network traffic or the target application's performance to determine if the intrusion cost of the data being sent from the phase probes is too great. If so, the analysis tool could modify the phase period so that the data is sent less often.

What is a one-shot probe?: A *one-shot probe* is a type of probe that is executed by the DPCL system immediately upon request, regardless of what the application happens to be doing.

When should an analysis tool use a one-shot probe?: An analysis tool should use a one-shot probe whenever it wants to explicitly and immediately execute code within the target application process on a one-time basis. Most commonly, analysis tools would use one-shot probes to:

- Perform setup or cleanup activities for other probes. For example, say an analysis tool has installed a set of point probes to write trace data to a file. Before data collection starts, the analysis tool could execute a one-shot probe to open the trace file. When data collection is complete, and the other probes are through writing to the trace file, the analysis tool could execute another one-shot probe to flush the file descriptor and close the trace file.
- Get a "snapshot" of a particular measure at a particular time. For example, the analysis tool could execute a one-shot probe to call AIX subroutines like `getrusage`, `times`, or `vtimes` to get performance and system-resource information for a target application process.

For example, in “When should an analysis tool use a point probe?” on page 18, we introduced a hypothetical analysis tool that, in order to measure the accumulation of floating point counts by function, installed a set of point probes to collect this data. We continued this same example in “When should an analysis tool use phases to invoke phase probes?” on page 20 by using a phase probe to minimize network traffic by only periodically sending the data back to the analysis tool to be displayed to the operator. Suppose now that, as the creator of this analysis tool, you wanted to add a "Refresh" button to the tool's graphical user interface so that the operator could force the tool to update itself with the most current information. To do this, the analysis tool could, whenever the operator clicks on the "Refresh" button, execute a one-shot probe to send the most recently collected data back to the analysis tool for display.

Why is it advantageous to build analysis tools on the DPCL system?

Our original motivation for creating the DPCL system came from the observation that customers were often asking for more application performance analysis tools than tool suppliers had the resources to build. High performance application developers were asking for tools that would provide detailed, accurate information about I/O usage, cache (and other memory usage), CPU and functional unit usage, message passing and synchronization, and operating system effects. Furthermore, they were asking for application profiles to identify problems, and event traces to determine the root causes of problems.

However, while programming tools were becoming more expensive to build and maintain, available tool development resources were shrinking rapidly. More tools were needed, but fewer tools could be created. So, in creating the DPCL system, our goals were to:

- **Reduce the cost of developing new tools.** The DPCL system accomplishes this goal by providing a scalable general-purpose infrastructure that enables analysis tools to instrument target applications. Its relatively simple application programming interface enables analysis tools to easily connect to target applications, and insert probes to perform typical tasks such as reading system counters and program variables. What's more, the DPCL system encapsulates a parallel infrastructure, making it ideally suited for analyzing parallel programs. To create an analysis tool without the benefit of the DPCL system would be a highly nontrivial job involving more complicated and time-consuming programming. For example, you would have to:
  - employ compilation techniques before you could insert the instrumentation probes into the executable you want to examine

- create careful locking mechanisms to ensure that the analysis tool and target application do not write to the same files
- set up sockets to enable the instrumentation probes that you place inside the target application to send collected data back to the analysis tool.
- set up some system of callbacks in the analysis tool to handle the data being sent back from the instrumentation probes in the target application.
- address scalability issues if you intend to use your tool to analyze scalable parallel programs.

Not only would you have to create an application from scratch to do all that, but, since analysis tools must be very careful not to adversely effect the target application's performance, you must manage to do all these things in such a way that the interference, or "*intrusion cost*", to the target application is minimal. This is essential, because if the intrusion cost is too great, then the data you're collecting from executing the instrumented version of the target application is no longer representative of the actual, uninstrumented program.

By building your analysis tool on top of the DPCL system, however, you are able to easily leverage its capabilities and thus can spare yourself the burdensome programming chores outlined above. What's more, by saving you the time and effort normally associated with developing analysis tools, the DPCL system effectively reduces the cost of developing new tools.

- **Reduce the intrusion cost of instrumentation.** As already stated, it is essential that the instrumented version of the target application is still representative of the actual, uninstrumented version of the application. The DPCL system is able to easily reduce the instrumentation intrusion cost that can be quite problematic for more traditional software instrumentation techniques. This is because the DPCL system is based on dynamic instrumentation, and so can add instrumentation probes to, and remove them from, the target application while it is running. That means that the instrumentation code need only reside in the target application for as long as it is needed to gather data, and that decisions regarding what data should be collected can be made and changed during the program's execution. For example, if a problem is suspected, an instrumentation probe can be inserted into the target application to gather just the data needed to verify if the suspected problem does in fact exist. If the probe does verify that the problem exists, it can be removed and replaced by another probe designed to ascertain the cause of the problem. If the original probe, however, concludes that the problem does not exist, then it could be removed and replaced by a probe designed to verify a different hypothesis.

  This ability to make and change data collection decisions during execution is unique to dynamic instrumentation. All other methods of instrumentation require you to make data collection decisions before running the program, and often before compiling or linking the program. Such restrictions often result in one choosing to gather more data than is actually needed, thus increasing the intrusion cost of the instrumentation.

- **Enable the creation of common tools across an organization or industry.** The DPCL system accomplishes this goal by its very nature; it is a general-purpose and reusable class library that provides a common architecture to all analysis tools that are built on it.

- **Enable greater flexibility and interoperability among tools.** Since the DPCL system provides a common architecture to all analysis tools that are built on it,

it is able to work with multiple analysis tools concurrently. This means you can use more than one analysis tool (for example, a test coverage tool and an application profiler) on the same target application at the same time.

- **Increase industry innovation in tool development, and, in doing so, increase the number and variety of programming tools.** A side benefit of reducing the cost of developing new tools is that it then becomes cost effective to experiment with more speculative analysis techniques. By building analysis tools on the DPCL system, novel and innovative ideas can be evaluated inexpensively, leading to a greater variety of tools available to the whole industry.

Chapter 2. What are the DPCL classes?

The following information is a summary of each of the DPCL classes that your analysis tool can use to instrument a target application. Before reading this information, you should first understand the concepts introduced in Chapter 1, "What is DPCL?" on page 3 (which provides a high-level description of the DPCL system and shows how its various parts work together to enable analysis tools to instrument one or more target application processes). The following DPCL class summary builds on this earlier description of the DPCL system by describing the DPCL classes that represent the elements of the DPCL system (target application processes, probes, instrumentation points, and so on). In doing so, it provides an overview of the purpose of each class, along with information on its supporting data types and functions. For complete reference information on any of the classes, data types, and functions summarized here, refer to DPCL's AIX man pages or the *IBM Parallel Environment for AIX: DPCL Class Reference*.

The following summary of the DPCL classes and functions is divided into the following sections:

- "What are the Process, Application, and PoeAppl classes?" describes the three classes that an analysis tool can use to represent, and act upon, target application process(es).
- "What are the ProbeExp, ProbeHandle, and ProbeModule classes?" on page 39 describes three classes that an analysis tool can use to represent, and act upon, probes.
- "What are the SourceObj and InstPoint classes?" on page 48 describes two classes that the analysis tool can use to examine the source code associated with a target application process and identify instrumentation points (locations where point probes can be installed).
- "What is the ProbeType class?" on page 55 describes the class that the analysis tool can use to represent the data type of an object within a target application process.
- "What is the Phase class?" on page 57 describes the class that the analysis tool can use to represent the control mechanism for invoking phase probes at set intervals.
- "What is the AisStatus class?" on page 58 describes the class used to store status information.

What are the Process, Application, and PoeAppl classes?

This section describes the three classes that an analysis tool can use to represent, and act upon, the target application process(es). These are the `Process` class (described in "What is the Process class?" on page 26 and used to represent a single AIX process), the `Application` class (described in "What is the Application class?" on page 32 and used to represent a group of related AIX processes), and the `PoeAppl` class (described in "What is the PoeAppl class?" on page 39, this class is derived from the `Application` class, and used to represent POE applications).

What is the Process class?

The `Process` class (defined in the header file `Process.h`) is the most fundamental of all the DPCL classes. Instances of this class represent a single target application process. The analysis tool can use objects of this class to connect to an existing target application process, or create a new process. First, the analysis tool creates an instance of the `Process` class. Depending on the constructor used, the `Process` object may, or may not, have values that identify a particular AIX process by host name and process ID.

- If the `Process` object is created with one of the non-default constructors, the analysis tool will have supplied a process ID (and if the process is not local, a host name) to identify a particular AIX process. The analysis tool can then call a member function to connect to the process.
- If the `Process` object is created with the default constructor, it will not have values identifying an AIX process. The analysis tool can call a member function to create a new process running. Alternatively, it can assign the host and process ID information using an assignment operator and call a member function to connect to the process.

Once connected to the process, member functions of this class enable the analysis tool to:

- get a `SourceObj` class object that represents the source code structure associated with this process. The `SourceObj` class is described in more detail in “What is the SourceObj class?” on page 48.
- install, activate, and remove point probes
- add, remove, and set the phase interval for, phase probes
- execute a one-shot probe within the process
- allocate and deallocate memory within the process for use by the probes.
- load and unload probe modules so that they may be called by point, phase, and one-shot probes.

For additional process control, an analysis tool can also attach to the process. By attaching to the process, the analysis tool can control its execution. Specifically, once attached to the process, the analysis tool can call member functions of the `Process` class to:

- suspend and resume execution of the process.
- destroy (terminate) the process.

In order to manipulate processes, you must understand the concept of process connect states. A `Process` object's connect state reflects the relationship between the `Process` class object in the analysis tool and the actual AIX process it represents. The various `Process` connect states are enumerated in the `ConnectState` type; the `Process::query_state` function returns one of the enumeration constants of the `ConnectState` type to indicate the `Process` object's connect state.

A `Process` object's connect state reflects what actions can be performed on the actual AIX process through DPCL function calls. For example, your analysis tool cannot execute probes within a target application process unless the analysis tool is "connected" or "attached" to that process. The "connected" state is represented

by the enumeration constant `PRC_connected` of the `ConnectState` enumeration type, and the "attached" state is represented by the constant `PRC_attached`. When the analysis tool calls certain functions to act upon a process, the DPCL system checks the `Process` object's connect state to determine if the action is allowed. If it is not allowed, the operation fails.

Certain member functions of the `Process` class serve to move the `Process` object from one state to another. For example, the analysis tool can establish a communication connection to an AIX process by calling the `Process::connect` function, or its blocking equivalent `Process::bconnect`. In addition to establishing the communication connection that allows the analysis tool to install probes into the target application process, these functions also move the `Process` object's connect state from `PRC_unconnected` to `PRC_connected`.

As already mentioned, the analysis tool can query a `Process` object's connect state by calling the `Process::query_state` function. This enables the analysis tool to check a process' state before making a DPCL service request that may not be allowed.

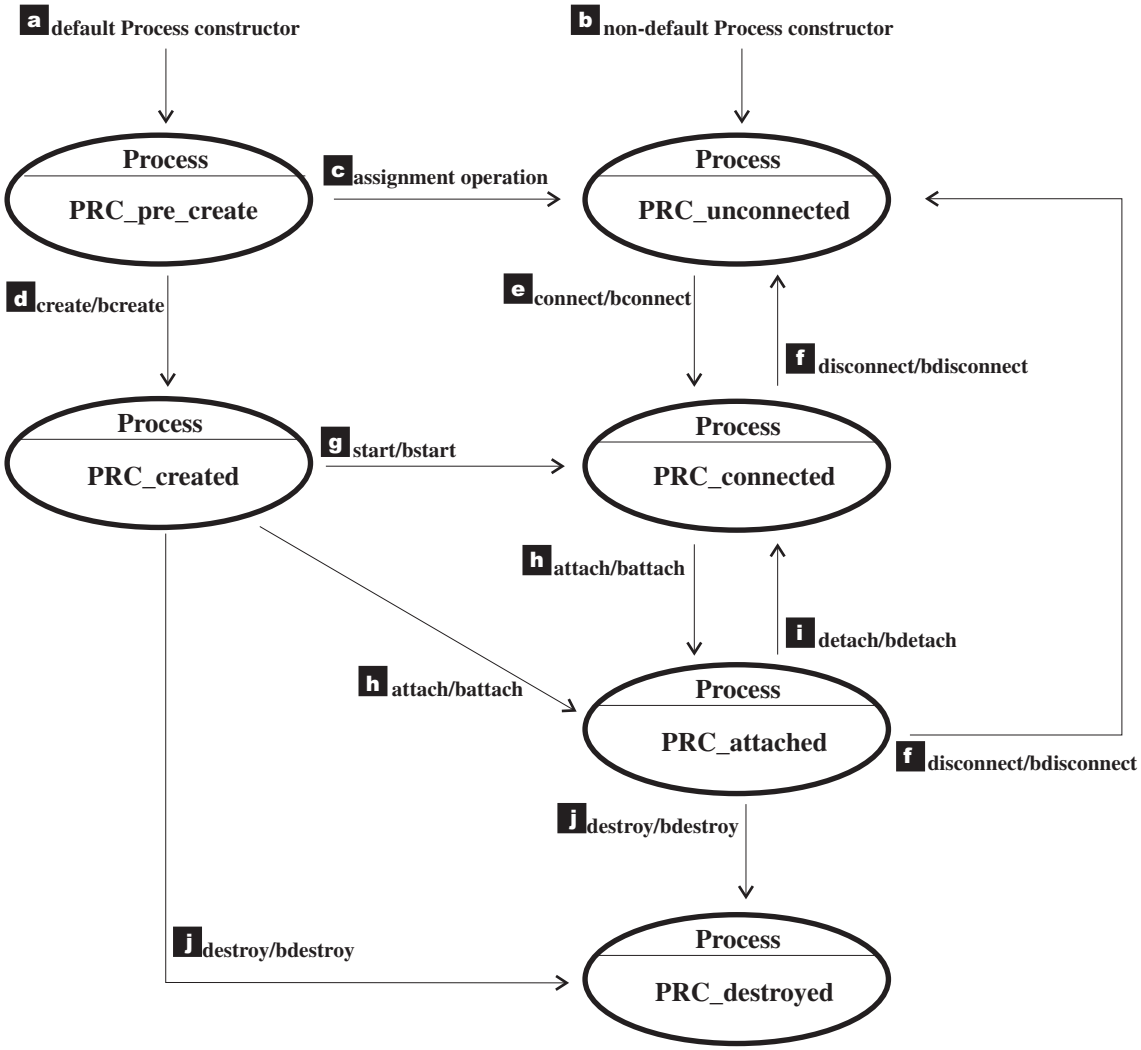

Figure 7. Process connect state diagram

The preceding figure illustrates the various process connect states and shows the enumeration constants of the ConnectState enumeration type. It also shows the constructors, operations, and functions that place a Process object into a particular connect state. For additional explanation of this figure, refer to the following key:

a If the analysis tool creates a Process class object using the default constructor, the Process object's initial connect state is PRC_pre_create. The PRC_pre_create connect state means that the Process object does not yet represent a particular process. In other words, it does not yet have values indicating a particular host name and process ID. The analysis tool can either assign these values to the Process object (**c**), or create a process (**d**)

b If the analysis tool creates a Process class object using the non-default constructor, the Process object's initial connect state is PRC_unconnected. The PRC_unconnected connect state means that, although the Process object has values indicating a particular host name and process ID, it is not yet connected to the process. In fact, the DPCL system does not at this point even know if the Process object's host and process ID values are valid. Now that it has a Process object that represents a particular AIX process, however, the analysis tool can connect to it (**e**).

c Using an assignment operator, the analysis tool can move a Process from the PRC_pre_create connect state to the PRC_unconnected connect state. The assignment operation assigns values to the Process object indicating a particular host name and process ID. The analysis tool can then attempt to connect to the process indicated (**e**)

d Using the create or bcreate function, the analysis tool can move a Process from the PRC_pre_create connect state to the PRC_created connect state. The create and bcreate functions create an AIX process on a particular host, but suspends execution of the process at its first executable instruction. The analysis tool can start the process running (**g**), attach to the process (**h**), or terminate the process (**j**).

e Using the connect or bconnect function, the analysis tool can move a Process from the PRC_unconnected connect state to the PRC_connected connect state. The connect and bconnect functions establish a connection to the actual AIX process represented by the Process object. In the PRC_connected state, the analysis tool is able to install and execute probes within the target application process. For additional process control (specifically, the ability to suspend, resume, and destroy the process), the analysis tool can attach to the process (**h**).

f Using the disconnect or bdisconnect function, the analysis tool can move a Process from the PRC_connected or PRC_attached connect state, back to the PRC_unconnected connect state. Although the Process object will have values indicating a particular host name and process ID, it will no longer be connected to the process. The analysis tool can, if desired, reconnect to the Process (**e**).

g Using the start or bstart function, the analysis tool can move a Process from the PRC_created connect state to the PRC_connected connect state. The start and bstart functions start execution of the target application process. Once in the PRC_connected connect state, the analysis tool is able to install and execute probes within the target application process. For additional process control (specifically, the

ability to suspend, resume, and destroy the process), the analysis tool can attach to the process (**h**).

h Using the `attach` or `battach` function, the analysis tool can move a `Process` from the `PRC_created` or `PRC_connected` connect state to the `PRC_attached` connect state. In the `PRC_attached` connect state, the analysis tool can install and execute probes within the target application process (as in the `PRC_connected` connect state), and can also control execution of the process. Moving to the `PRC_attached` connect state automatically suspends execution of the process. This enables the analysis tool to perform actions on the process (such as installing probes) before resuming its execution. The analysis tool can resume execution of the suspended process. Once resumed, the analysis tool can again suspend the process' execution. The analysis tool can also terminate (**j**) the process. Since only one analysis tool may be attached to a particular process at a time, your analysis tool may wish to detach itself from the process (**i**) when it is through controlling its execution.

i Using the `detach` or `bdetach` function, the analysis tool can move a `Process` from the `PRC_attached` connect state to the `PRC_connected` connect state. The analysis tool will no longer be able to directly control execution of the process, but in the `PRC_connected` connect state will still be able to install and execute probes within the process.

j When the `Process` object is in the `PRC_created` or `PRC_attached` connect state, the analysis tool can destroy (terminate) the process using the `destroy` or `bdestroy` function. This places the `Process` object in the `PRC_destroyed` state.

Table 3 on page 30 summarizes the various functions of the `Process` class. Since many of the actions carried out by the functions can occur on remote hosts, note that blocking and nonblocking versions of the same functionality are usually provided. As described in "What are blocking and nonblocking API calls?" on page 10, blocking calls do not return control to the analysis tool until they either succeed or fail in carrying out the requisite service, while the nonblocking calls return immediately, enabling the analysis tool to continue with other work. Since the nonblocking calls return immediately, however, that returned status indicates only whether or not the request was successfully made; it does not indicate whether or not the request succeeded. For this reason, when calling a nonblocking function, the analysis tool can specify an acknowledgment callback routine. Once the operation succeeds or fails, the DPCL system will trigger execution of this callback routine and pass it the status information in the form of an `AisStatus` object (described in "What is the AisStatus class?" on page 58). An acknowledgment callback not only enables the analysis tool to perform status error checking, but also enables the analysis tool to be structured in a more event-driven manner. For example, the acknowledgment callback for the `Process::connect` function could check that the operation succeeded, and then could call the `Process::attach` function to attach to the process. Similarly, the acknowledgment callback for the `Process::attach` function could contain code for the next piece of work. While the nonblocking calls enable you to structure your analysis tool program in this sort of event driven manner, the blocking calls enable you to structure your analysis tool program in a more traditional way that is more straightforward and, therefore, easier to code.

For complete information on any of the functions summarized in this table, refer to the function's AIX man page or the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Table 3 (Page 1 of 3). Process class function summary

Calling the function:	Does this:	In order for the DPCL system to carry out this service request, the Process object's connect state must be:
activate_probe bactivate_probe	activates one or more point probes that have been installed (by the install_probe or binstall_probe function) within the process. Activating a point probe will cause the probe to run when execution reaches its installed location in the code.	PRC_created, PRC_connected, or PRC_attached
add_phase badd_phase	Adds a new phase (the control mechanism for invoking phase probes at set intervals) to the process.	PRC_connected or PRC_attached
alloc_mem balloc_mem	allocates memory within the process for use by probes.	PRC_connected or PRC_attached
attach battach	Attaches to the process. This action changes the Process object's connect state to PRC_attached. Attaching to a process also suspends its execution in the same way that a call to the suspend or bsuspend function would. This enables the analysis tool to install probes in the process before resuming its execution with a call to the resume or bresume function.	PRC_connected or PRC_created
connect bconnect	Connects to the process. This action changes the Process object's connect state from PRC_unconnected to PRC_connected.	PRC_unconnected
create bcreate	creates a new process running on a specified host. This action changes the Process object's connect state from PRC_pre_create to PRC_created.	PRC_pre_create
deactivate_probe bdeactivate_probe	Deactivates one or more point probes that have been installed (by the install_probe or binstall_probe function) and activated (by the activate_probe or bactivate_probe function) within the process. Once deactivated, a probe will no longer run when execution reaches its installed location in the code.	PRC_created, PRC_connected, or PRC_attatched
destroy bdestroy	Destroys (terminates) the process. This action changes the Process object's connect state from PRC_created or PRC_attached to PRC_destroyed.	PRC_created or PRC_attached
detach bdetach	detaches the analysis tool from the process. This action changes the Process object's connect state from PRC_created or PRC_attached to PRC_connected.	PRC_created or PRC_attached
disconnect bdisconnect	disconnects the analysis tool from the process. This action changes the Process object's connect state to PRC_unconnected.	PRC_connected or PRC_attached
execute bexecute	executes a one-shot probe within the process.	PRC_created, PRC_connected, or PRC_attached
free_mem bfree_mem	deallocates memory within the process previously allocated by alloc_mem or balloc_mem.	PRC_created, PRC_connected, or PRC_attached
get_host_name	copies into a buffer a null-terminated string that represents the name of the host on which the process is running.	Not applicable. This is a local operation within the analysis tool process only.
get_host_name_length	returns the name length, including the terminating null byte, of the host machine on which the process is running. (Used to determine the maximum size of the buffer when calling the get_host_name function.)	Not applicable. This is a local operation within the analysis tool process only.

Table 3 (Page 2 of 3). Process class function summary

Calling the function:	Does this:	In order for the DPCL system to carry out this service request, the Process object's connect state must be:
get_pid	Returns the AIX process ID for the process.	Not applicable. This is a local operation within the analysis tool process only.
get_phase_period	returns the time duration, in seconds, between successive activations of a particular phase in the process.	PRC_created, PRC_connected, or PRC_attached
get_program_object	Returns the top-level source object associated with the process.	PRC_created, PRC_connected, or PRC_attached
get_task	Returns the task identifier associated with the process.	Not applicable. This is a local operation within the analysis tool process only.
install_probe binstall_probe	installs one or more probes as point probes within the process. Once activated (by the activate_probe or bactivate_probe function), a point probe will run when execution reaches its installed location in the code.	PRC_created, PRC_connected, or PRC_attached
load_module bload_module	Loads a probe module in the process.	PRC_created, PRC_connected or PRC_attached
operator =	Assigns values to the Process object.	Not applicable. This is a local operation within the analysis tool process only.
query_state	Returns the connect state of the Process object.	Not applicable. Queries the Process object's connect state.
remove_phase bremove_phase	Removes a phase (the control mechanism for invoking phase probes at set intervals) from the process. The phase will have been previously added to the process by the add_phase or badd_phase function.	PRC_created, PRC_connected, or PRC_attached
remove_probe bremove_probe	Removes one or more point probes from the process.	PRC_created, PRC_connected, or PRC_attached
resume bresume	resumes execution of the process (if it has been previously suspended by the suspend, bsuspend, attach, or battach functions).	PRC_attached
send_stdin	Provides text to be used as standard input to the process (when the analysis tool has created the process using the create or bcreate function).	PRC_created. This service request can also be carried out in the PRC_connected or PRC_attached states as long as the Process was once in the PRC_created state (in other words, created by the create or bcreate function).
set_phase_exit bset_phase_exit	Specifies a set of exit functions to be executed when a phase is removed.	PRC_created, PRC_connected, or PRC_attached
set_phase_period bset_phase_period	Changes the time interval between successive activations of a particular phase.	PRC_created, PRC_connected, or PRC_attached

Table 3 (Page 3 of 3). Process class function summary

Calling the function:	Does this:	In order for the DPCL system to carry out this service request, the Process object's connect state must be:
`start` `bstart`	Starts a process that has been created using the `create` or `bcreate` functions. Such a process will have been started, but suspended at its first executable instruction. This action changes the `Process` object's connect state from `PRC_created` to `PRC_connected`.	`PRC_created`
`suspend` `bsuspend`	Suspends execution of the process. The analysis tool can resume execution of the process by calling the `resume` or `bresume` function.	`PRC_attached`
`unload_module` `bunload_module`	unloads a probe module from the process.	`PRC_created`, `PRC_connected`, or `PRC_attached`

What is the Application class?

The `Application` class (defined in the header file `Application.h`) is a grouping of related `Process` class objects. By grouping a number of `Process` objects under an `Application` object, the analysis tool is able to manipulate a set of related AIX processes (represented by the `Process` objects) as a single unit. For example, say the target application is a parallel program that follows the Single Program Multiple Data (SPMD) model. To connect to its processes, your analysis tool could make a separate call to the `Process::connect` or `Process::bconnect` function for each process. A more convenient approach, however, is to group the `Process` objects under an `Application` object; the analysis tool can then connect to all the processes managed by the `Application` object by making a single call to the `Application::connect` or `Application::bconnect` function. The `Application` class function makes the individual service requests for each process managed by the `Application` object. The analysis tool uses the `Application::add_process` function to group processes under the `Application` object.

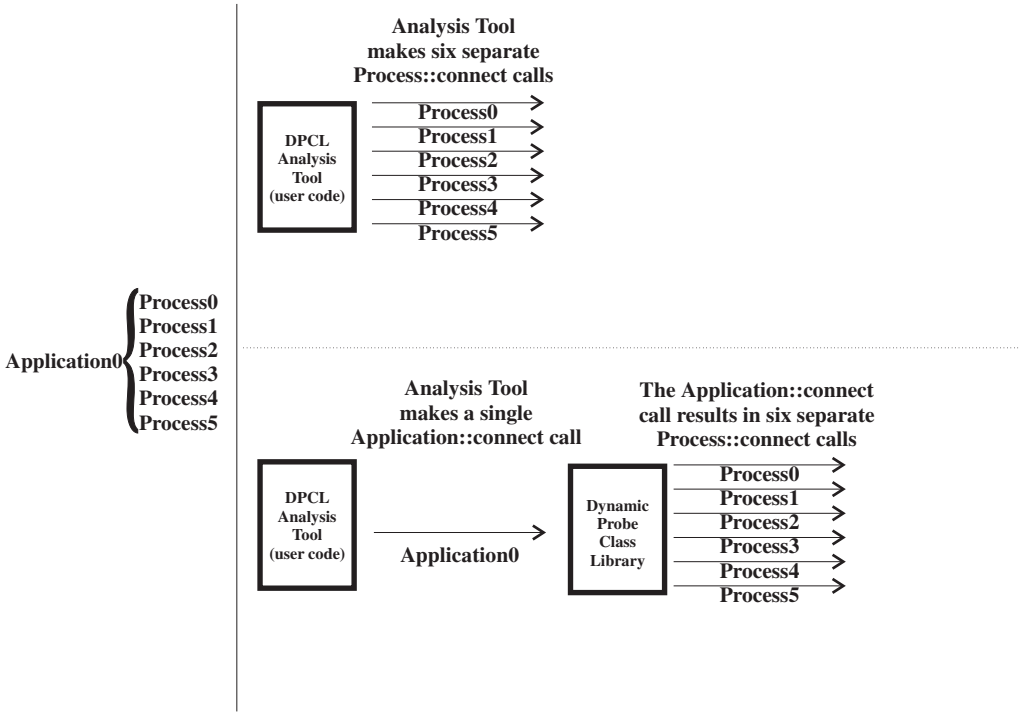

Figure 8. Process objects grouped under an Application object. In this figure, six `Process` *objects are grouped under an* `Application` *object. Here, the analysis tool is trying to connect to the AIX processes represented by the* `Process` *objects. It can do this either by making a separate* `Process::connect` *or* `Process::bconnect` *call for each process, or by making a single call to* `Application::connect` *or* `Application::bconnect`*.*

It is important to understand that an `Application` object can be any grouping of `Process` objects, and not necessarily all the processes in a parallel application. It may, for example, be only a subset of the parallel application's processes, or, in fact, **any** set of AIX processes. What's more, a single `Process` object may be grouped under several different `Application` objects. For example, say you have a parallel program that follows the Multiple Program Multiple Data (MPMD) model and solves a particular problem using functional parallelism. In this case, let's say that three separate programs were created to perform specific work; at run time, multiple processes for each executable program will run. For certain actions, it will be convenient to have a single `Application` object that manages all of the target application processes. This would enable the analysis tool to, using a single function call, perform certain actions on all processes (such as connecting to, suspending, resuming, terminating, or disconnecting from the processes). On the other hand, because the analysis tool is dealing with a target application composed from multiple programs, there are certain actions that it should not perform globally on all processes. It would not, for example, make sense to install a point probe in all the processes globally because a valid instrumentation point for some processes would not be valid for others. In this case, your analysis tool might organize the processes under four separate `Application` objects. One `Application` object could contain all the `Process` objects and be used by global operations. Each of the other three `Application` objects could contain just the `Process` objects associated with one of the three source programs. Figure 9 on page 34 illustrates this situation; to simplify the figure, only six processes are illustrated.

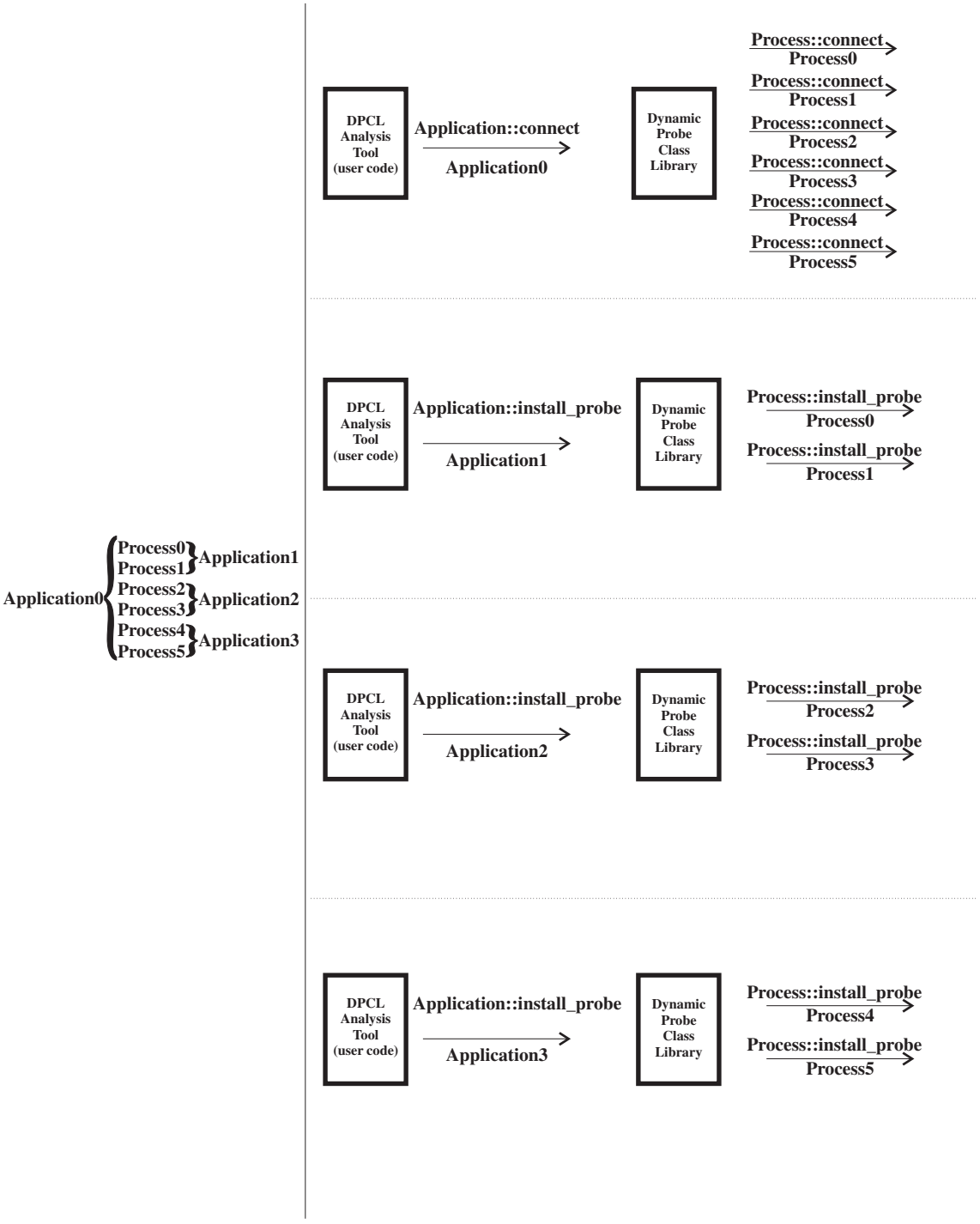

Figure 9. Process objects grouped under multiple Application objects. In this figure, the `Process` *objects are grouped under four separate* `Application` *objects. This enables the analysis tool to call an* `Application` *class function to manipulate all, or only a select subset, of the target application processes.*

By comparing Table 3 on page 30 and Table 4 on page 36, you can clearly see that most of the `Process` class functions have equivalent functions in the `Application` class. The difference between a `Process` and an `Application` version of a function is that the `Application` class version of the function carries out the request for all of the processes managed by the `Application` object. Therefore, member functions of the `Application` class enable the analysis tool to:

- connect to, and disconnect from, the processes. When connected to the processes, member functions of this class enable the analysis tool to:
  - install, activate, and remove point probes within the processes.
  - add, remove, and set the phase interval for, phase probes within the processes.
  - execute a one-shot probe within the processes.
  - allocate and deallocate memory within the processes for use by the probes.
  - load and unload probe modules so that their functions may be called by point, phase, or one-shot probes.
- Attach to, or detach from, the processes. By attaching to the processes, the analysis tool can control their execution. Specifically, once attached to the processes, the analysis tool can call member functions of the `Application` class to:
  - suspend and resume execution of the processes
  - destroy (terminate) the processes

Additional functions of the `Application` class enable the analysis tool to:

- add or remove a `Process` object from the set of processes managed by the `Application` object.
- ascertain the number of processes (`Process` objects) currently managed by the `Application` object.
- return a particular `Process` object from the set of those managed by the `Application` object.
- return the status of the most recent `Application` function call for a particular process managed by the `Application` object.

Table 4 on page 36 summarizes the various functions of the `Application` class. Since many of the actions carried out by these functions can occur on remote hosts, note that, as in the `Process` class, blocking and nonblocking versions of the same functionality is usually provided. Also note that the majority of the `Application` class functions carry out some service request for all of the `Process` objects managed by the `Application` object, and the action requested may succeed on certain processes while failing on others. This adds additional complexity to error status checking. Specifically:

- The blocking functions will return only after the operation has succeeded or failed for all processes managed by the `Application` object, but the status (`AisStatus` object) returned will indicate only whether or not the operation succeeded on **all** processes. If it did not, the analysis tool should use the `Application::status` function to determine the process(es) on which the operation failed.
- The nonblocking calls return immediately, but the returned status value indicates only whether or not the requests were successfully made; it does not indicate whether or not they succeeded. For this reason, when calling a nonblocking function, the analysis tool can specify an acknowledgment callback routine that will be called by the DPCL system upon the successful or unsuccessful completion of the operation for each process managed by the `Application` object. The DPCL system will call the acknowledgment callback routine and pass it the status (`AisStatus` object) and a pointer to the `Process`

object to which the status applies. Not only does this enable the analysis tool to perform status error checking for each process, but it also enables the analysis tool to be structured in a more event-driven manner. For example, the acknowledgment callback for the `Application::connect` function could check that the operation succeeded for each individual process in turn. If the operation did succeed, the callback could then call the `Process::attach` function to attach to the process.

The `AisStatus` object is described in "What is the AisStatus class?" on page 58. Error checking considerations for Application functions are described in Chapter 4, "Performing status error checking" on page 73.

For more information on the `Process` object connect state information shown in this table, refer to Figure 7 on page 27. For complete information on any of the functions summarized in this table, refer to the function's AIX man page or the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Table 4 (Page 1 of 3). Application class function summary

Calling the function:	**Does this:**	**In order for the DPCL system to carry out this service request for a particular process managed by the Application object, the Process object's connect state must be:**
`activate_probe` `bactivate_probe`	Within each process managed by this `Application` object, activates one or more point probes that have been installed (by the `install_probe` or `binstall_probe` function) within the process. Activating a point probe will cause the probe to run when execution reaches its installed location in the code.	`PRC_created`, `PRC_connected`, or `PRC_attached`
`add_phase` `badd_phase`	For each process managed by this `Application` object, adds a new phase (the control mechanism for invoking phase probes at set intervals) to the process.	`PRC_connected` or `PRC_attached`
`add_process`	Adds a process (`Process` object) to the set of processes managed by this `Application` object. The analysis tool can later remove this process from the set of processes managed by the application by calling the `Application::remove_process` function.	Not applicable. This is a local operation within the analysis tool process only.
`alloc_mem` `balloc_mem`	Within each process managed by this `Application` object, allocates memory for use by probes.	`PRC_connected` or `PRC_attached`
`attach` `battach`	For each process managed by this `Application` object, attaches the analysis tool to the process. This action changes each `Process` object's connect state to `PRC_attached`. Attaching to the processes managed by the `Application` object also suspends their execution in the same way that a call to the `suspend` or `bsuspend` function would. This enables the analysis tool to install probes in the processes before resuming their execution with a call to the `resume` or `bresume` function.	`PRC_connected` or `PRC_attached`
`connect` `bconnect`	For each process managed by this `Application` object, connects the analysis tool to the process. This action changes each `Process` object's connect state from `PRC_unconnected` to `PRC_connected`.	`PRC_unconnected`

Table 4 (Page 2 of 3). Application class function summary

Calling the function:	Does this:	In order for the DPCL system to carry out this service request for a particular process managed by the Application object, the Process object's connect state must be:
deactivate_probe bdeactivate_probe	For each process managed by this Application object, deactivates one or more point probes that have been installed (by the install_probe or binstall_probe function) and activated (by the activate_probe or bactivate_probe function) within the process. Once deactivated, a probe will no longer run when execution reaches its installed location in the code.	PRC_created, PRC_connected, or PRC_attached
destroy bdestroy	For each process managed by this Application object, destroys (terminates) the process. This action changes each Process object's connect state from PRC_created or PRC_attached to PRC_destroyed.	PRC_created or PRC_attached
detach bdetach	For each process managed by this Application object, detaches the analysis tool from the process. This action changes each Process object's connect state from PRC_created or PRC_attached to PRC_connected.	PRC_created or PRC_attached
disconnect bdisconnect	For each process object managed by this Application object, disconnects the analysis tool from the process. This action changes each Process object's connect state to PRC_unconnected.	PRC_connected or PRC_attached
execute bexecute	For each process managed by this application object, executes a one-shot probe within the process.	PRC_created, PRC_connected, or PRC_attached
free_mem bfree_mem	For each process managed by this application object, deallocates memory within the process previously allocated by the alloc_mem or balloc_mem function.	PRC_created, PRC_connected, or PRC_attached
get_count	Returns the number of processes (Process objects) currently managed by this application object.	Not applicable. This is a local operation within the analysis tool process only.
get_process	Returns a particular Process object from the set of those managed by this Application object. This enables the analysis tool to act upon the individual process independent of the other processes managed by this Application object.	Not applicable. This is a local operation within the analysis tool process only.
install_probe binstall_probe	For each process managed by this Application object, installs one or more probes as point probes within the process. Once activated (by the activate_probe or bactivate_probe function), a point probe will run when execution reaches its installed location in the code.	PRC_created, PRC_connected, or PRC_attached
load_module bload_module	For each process managed by this Application object, loads a probe module in the process.	PRC_created, PRC_connected, or PRC_attached
operator =	Assigns values to the Application object.	Not applicable. This is a local operation within the analysis tool process only.
remove_phase bremove_phase	For each process managed by this Application object, removes a phase (the control mechanism for invoking phase probes at set intervals) from the process. The phase will have been previously added to the application's processes by the add_phase or badd_phase functions.	PRC_created, PRC_connected, or PRC_attached

Table 4 (Page 3 of 3). Application class function summary

Calling the function:	Does this:	In order for the DPCL system to carry out this service request for a particular process managed by the Application object, the Process object's connect state must be:
remove_probe bremove_probe	For each process managed by this Application object, removes one or more point probes from the process.	PRC_created, PRC_connected, or PRC_attached
remove_process	Removes a particular process (Process object) from the set of those managed by this Application object.	Not applicable. This is a local operation within the analysis tool process only.
resume bresume	For each process managed by this Application object, resumes execution of the process (if it has been suspended by the suspend, bsuspend, attach, or battach functions).	PRC_attached
send_stdin	For each process managed by this Application object, provides text to be used as standard input to the process (when the analysis tool has created the process using the Process::create or Process::bcreate function).	PRC_created. This service request can also be carried out in the PRC_connected or PRC_attached connect state as long as the Process objects managed by this Application object were once in the PRC_created connect state (in other words, created by the create or bcreate function).
set_phase_exit bset_phase_exit	For each process managed by this Application object, specifies a set of exit functions to be executed when a phase is removed.	PRC_created, PRC_connected, or PRC_attached
set_phase_period bset_phase_period	For each process managed by this Application object, changes the time interval between successive activations of a particular phase.	PRC_created, PRC_connected, or PRC_attached
start bstart	For each process managed by this Application object, starts the process (which will have been created using the Process::create or Process::bcreate function). Such a process will have been started, but suspended at its first executable instruction. This action changes each Process object's connect state from PRC_created to PRC_connected.	PRC_created
status	Returns, for a particular Process object managed by this Application object, the status (AisStatus object) for the most recent Application function call. If an Application function call does not succeed for all processes managed by the Application object, this enables the analysis tool to check the status of each Process object individually.	Not applicable. This function is designed for error status checking and is not related to process connect states.
suspend bsuspend	For each process managed by this Application object, suspends execution of the process. The analysis tool can resume execution of the processes by calling the resume or bresume function.	PRC_attached
unload_module bunload_module	For each process managed by this Application object, unloads a probe module from the process.	PRC_created, PRC_connected, or PRC_attached

What is the PoeAppl class?

The PoeAppl class (defined in the header file PoeAppl.h) is derived from the Application class; its purpose is to provide additional convenience functions for connecting to or starting a job in the Parallel Operation Environment (POE). POE (which is fully described in the manual *IBM Parallel Environment for AIX: Operation and Use, Volume 1*) is an execution environment designed to hide, or at least smooth, the differences between serial and parallel execution. In POE, you execute a program from a "home node" (which can be any workstation on the LAN), and POE will, sometimes in conjunction with IBM LoadLeveler, allocate host machines on which the various processes of your POE job will run. A number of POE environment variables enable you to control such things as how the system resources are allocated and how standard I/O between the home node and the processes of your program should be handled.

Table 5 summarizes the various functions of the PoeAppl class. As already stated, the purpose of this class is simply to provide additional convenience functions for either connecting to or starting a POE job. Since this class is derived from the Application class, all of the Application class functions are available to objects of the PoeAppl class. For complete information on any of the functions summarized in the table, refer to the function's AIX man page or the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Table 5. PoeAppl class function summary

Calling the function:	Does this:
create bcreate	Creates a POE application in a suspended state. Arguments to these functions enable you to specify the POE home node, arguments for the **poe** command or the target application, environment variable settings that effect the POE execution environment, and files for remote *stdin*, *stdout*, and *stderr*. After this operation completes, the PoeAppl object will contain Process objects that represent the various processes of the POE application. Since this function creates the POE application in a suspended state, each process will have been suspended at its first executable instruction (as represented by the PRC_connected state). To start the processes running, the analysis tool can call the Application::start or Application::bstart function. To manipulate any of the processes on an individual basis (using Process class functions), the analysis tool can call the Application::get_process function to return a particular Process object.
init_procs binit_procs	Initializes an empty PoeAppl object to contain Process objects representing a particular POE target application's processes. To connect to these running processes, the analysis tool can call the Application::connect or Application::bconnect function.
send_stdin	Provides text to be used as input to the POE home node process for the *stdin* device (file descriptor 0). To use this function, the POE application must have been created using the PoeAppl::create or PoeAppl::bcreate function.

What are the ProbeExp, ProbeHandle, and ProbeModule classes?

This section describes the three classes that an analysis tool can use to represent, and act upon, probes. These are:

- the ProbeExp class, described in “What is the ProbeExp class?” on page 40, and used to represent probe expressions to be executed within one or more target application processes.
- the ProbeHandle class, described in “What is the ProbeHandle class?” on page 46, and used to represent identifying handles to installed point probes. When the analysis tool calls one of the DPCL functions that install one or more

probe expressions (`ProbeExp` objects) as point probes, the function called will return an array of `ProbeHandle` objects that identify the point probes.

- the `ProbeModule` class, described in “What is the ProbeModule class?” on page 47, and used to represent probe modules (compiled object files containing one or more functions written in C).

What is the ProbeExp class?

The `ProbeExp` class (defined in the header file `ProbeExp.h`) is used to represent probe expressions to be executed within one or more target application processes. As described in “What is a probe expression?” on page 15, probe expressions are "abstract syntax trees" — data structures that represent logic removed from any particular syntactic representation. The term "abstract syntax tree" is one we have borrowed from compiler technology. Compilers create abstract syntax trees from a program's source code as an intermediary stage before manipulating and converting them into executable instructions. The DPCL system needs to create these abstract syntax tree data structures for the same reason that compilers do — freed from a particular syntactic representation, they can be more easily converted into executable instructions. This is because a compiler (or, in this case, the DPCL system) only has to translate the abstract logic into executable instructions, and does not need to parse a particular language.

A probe expression could be a simple one that represents, for example, a persistent data value or a variable data value. In this case, the abstract syntax tree would be a very simple one consisting of a single node. These simple probe expressions, however, could be combined into more complex ones representing such things as operations, conditional statements, sequences of expressions, and function calls. For example, Figure 10 illustrates how an analysis tool could create the abstract syntax tree to represent the expression a + (b * c).

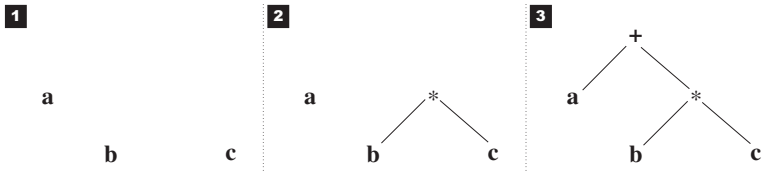

*Figure 10. Building an abstract syntax tree. This figure builds an abstract syntax tree to represent the expression a + (b * c). First, in 1, three separate probe expressions are created to represent the variable values a, b, and c. These will be the abstract syntax tree's end, or "terminal" nodes. Next, in 2, the probe expressions representing the variables b and c are combined into a single probe expression representing the multiplication operation b * c. Finally, in 3, the probe expression representing the variable x is combined with the probe expression representing the multiplication operation to form the probe expression representing the operation a + (b * c). Figure 11 on page 41 expands on this abstract syntax tree to create a probe expression representing conditional logic.*

The following figure builds further on the abstract syntax tree created in Figure 10 to form a more complex abstract syntax tree to represent the conditional expression:

```
if (x == a + (b * c))
     my_function();
```

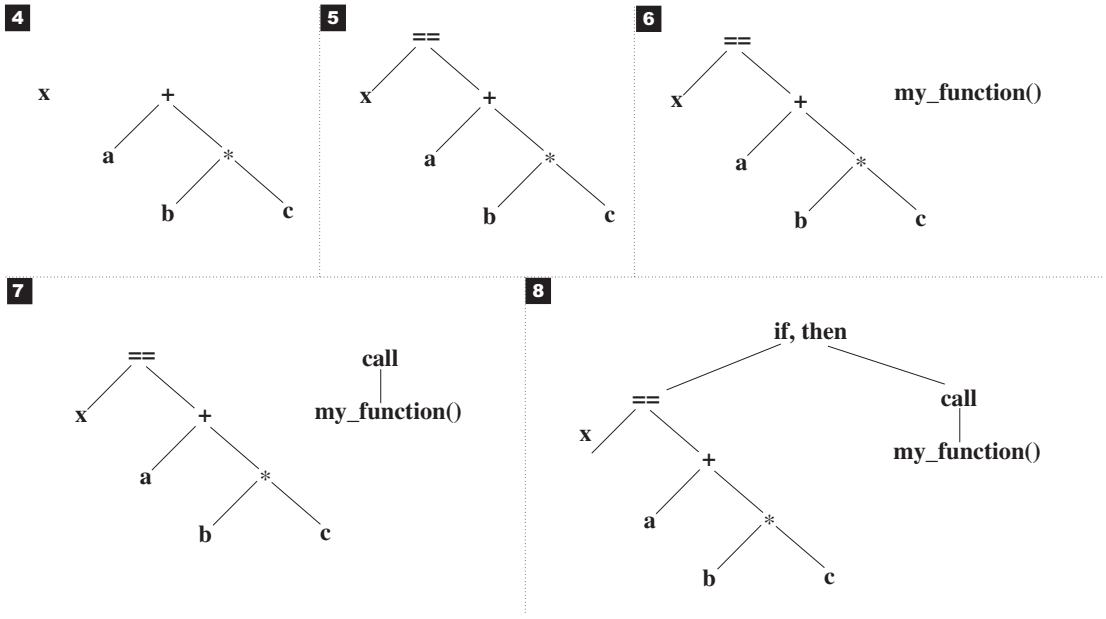

Figure 11. Building a more complex abstract syntax tree. This figure builds on the probe expression created in Figure 10 to create a more complex probe expression representing a conditional expression in which a function is called if the condition tests true. First, in **4** *and* **5** *, the existing probe expression is expanded to form the test condition x == a + (b * c). Then, in* **6** *and* **7** *, the probe expression representing the function call is built. Finally, in* **8** *, these two separate probe expressions are combined to form a probe expression representing the conditional statement.*

The member functions of the `ProbeExp` class are divided into two types — those that the analysis tool can use to create probe expressions and so build abstract syntax trees, and those that the analysis tool can use to query and return values from probe expressions, and so navigate an abstract syntax tree. In addition to the member functions of the `ProbeExp` class, certain functions of other DPCL classes (such as the `Process::alloc_mem`, `ProbeHandle::get_expression`, and `ProbeModule::get_reference` functions) also return probe expressions.

First, let's discuss how an analysis tool goes about creating probe expressions and building the abstract syntax tree. The simplest probe expressions are those that consist of a single abstract syntax tree node. These, if combined into a larger probe expression, are referred to as "terminal nodes" because they have no child probe expression. In DPCL, terminal nodes can represent a persistent data value, a variable data value, or a reference to a function.

The analysis tool can create a probe expression representing a:	**By:**
persistent data value	• using the `ProbeExp` class constructors
variable data value	• calling the `Process::alloc_mem`, `Process::balloc_mem`, `Application::alloc_mem`, or `Application::balloc_mem` function to allocate memory for the variable in the target application process(es). • calling the `SourceObj::reference` function to get a reference to a program variable. • calling the `ProbeType::get_actual` function to get the actual value of a function parameter in the target application.

The analysis tool can create a probe expression representing a:	By:
reference to a function	• calling the `ProbeModule::get_reference` function to get a reference to a probe module function. • calling the `SourceObj::reference` function to get a reference to a function in the target application. • using the predefined probe expression `Ais_send` (which references a function the analysis tool can use to send information from an installed probe back to the analysis tool.)

These simple probe expressions do not need to be combined into more complex ones. (Even a probe expression representing a reference to a probe module function does not need to be combined into a probe expression representing a function call since it can be specified as a phase probe when instantiating a `Phase` class object.) As already stated, however, these simple probe expressions can be combined into more complex ones. These more complex probe expressions can represent an operation, a function call, a sequence of instructions, and conditional logic.

The analysis tool can create a probe expression to represent:	By:
an operation	Using any of the regular operators (such as +, *, !=, and so on) which have been overloaded so that, within the context of a probe expression, they do not execute locally, but instead create a probe expression representing the operation. Since the assignment (=) and address (&) operators could not be easily overloaded, the `ProbeExp` class provides explicit member functions (`ProbeExp::assign` and `ProbeExp::address`) to perform these operations.
a function call	calling the `ProbeExp::call` function.
a sequence of instructions	calling the `ProbeExp::sequence` function.
conditional logic	calling the `ProbeExp::ifelse` function.

Now let's discuss the `ProbeExp` class functions that enable the analysis tool to query and return information from probe expressions. In order to understand these functions, you should understand how DPCL represents operations, function calls, instruction sequences, and conditional logic as abstract syntax trees. The following figures illustrate these abstract syntax trees. In these figures, note that, for each node in the tree, the node type is shown. A probe expression's node type (as represented by one of the enumeration constants of the `CodeExpNodeType` enumeration type) represents the various operators and operands that may be found in a probe expression. For example, the enumeration constant `CEN_mult_op` indicates that the probe expression node represents a multiplication operation, and the enumeration constant `CEN_call_op` indicates that the probe expression node represents a function call. An analysis tool can ascertain a probe expression node type by calling the `ProbeExp::get_node_type` function.

Figure 12 on page 43 shows probe expression abstract syntax trees representing the operations `8 * 2` and `10 + (8 * 2)`.

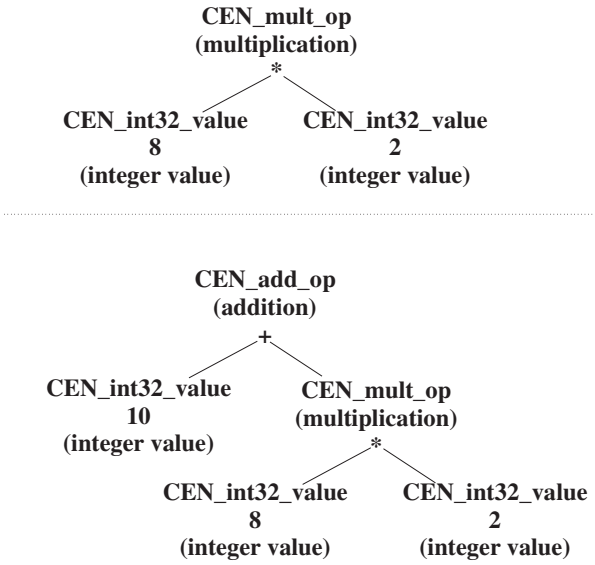

Figure 12. Probe expression abstract syntax trees representing operations

Figure 13 shows probe expression abstract syntax trees representing the function calls `my_function(x);` and `my_function(x, y, z);`. Note that the function call node (`CEN_call_op`) cannot have more than two children. When there are more than two function call arguments, the `CEN_list_item` nodes are used to organize them into a deeper tree.

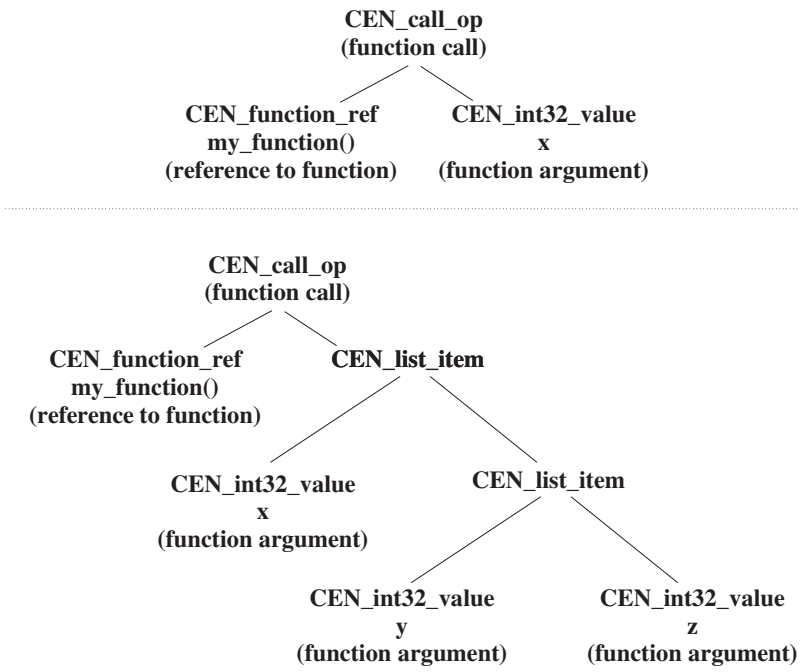

Figure 13. Probe expression abstract syntax trees representing function calls

Figure 14 on page 44 shows a probe expression abstract syntax tree that represents the instruction sequence `x++; my_function(x);`.

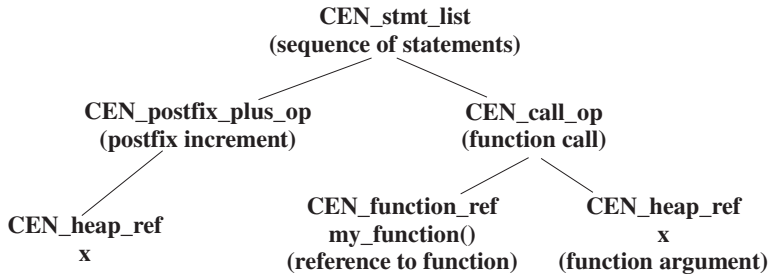

Figure 14. Probe expression abstract syntax tree representing an instruction sequence

Figure 15 shows probe expression abstract syntax trees that represent the conditional statements `if (x != 4) my_function(x);` and `if (x != 4) my_function(x); else my_other_function();`. Note that both the "if..." and if... else" conditional statements are created using the same function call — `ProbeExp::ifelse`. Note also that the "if... else" statement node (`CEN_if_else_stmt`) is the only node type that has three children.

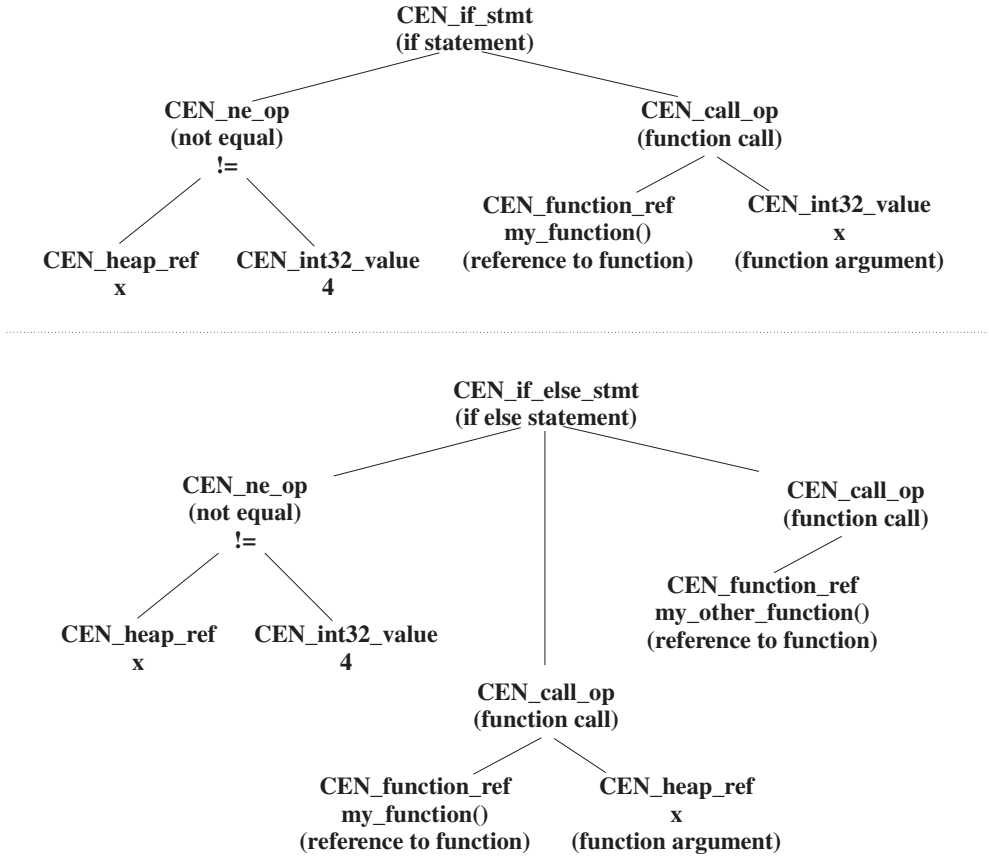

Figure 15. Probe expression abstract syntax trees representing conditional statements

To navigate and query a probe expression abstract syntax tree, the analysis tool uses member functions of the `ProbeExp` class designed for this purpose. The analysis tool can determine:

- the probe expression node's type (using the `ProbeExp::get_node_type` function).
- whether or not the probe expression's node has children (using the `ProbeExp::has_children` function), and, if so, how many (using the

ProbeExp::has_left, ProbeExp::has_right, and ProbeExp::has_center functions). If it does have any of these children, the analysis tool can get the actual probe expression for any one of them (using the ProbeExp::value_left, ProbeExp::value_right, and ProbeExp::value_center functions). By getting a child probe expression, the analysis tool essentially navigates down into the tree, and can now query and return values for the child probe expression's node.

- whether or not the probe expression node represents a datum of a particular type (for example, ProbeExp::has_int, ProbeExp::has_uint, and so on). If the probe expression does have such a value, the analysis tool can use other functions (for example ProbeExp::value_int, ProbeExp::value_uint, and so on) to get the value contained within the probe expression.

Table 6 summarizes the various functions of the ProbeExp class. For complete information on any of the functions summarized in the table, refer to the function's AIX man page or the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Table 6 (Page 1 of 2). ProbeExp class function summary

Calling the function:	Does this:
address	Creates a probe expression that represents the referencing of another probe expression. This function is provided because the & operator could not be easily overloaded since it is used in passing arguments to functions that use call-by-reference.
assign	Creates a probe expression that represents the assignment of a value represented by one probe expression into a storage location represented by another probe expression. This function is provided because the = operator could not be easily overloaded without causing simple expression manipulation to become unwieldy.
call	Creates a probe expression that represents a function call.
get_data_type	Returns the data type (ProbeType object) of this probe expression. (For more information on ProbeType objects, refer to “What is the ProbeType class?” on page 55.)
get_node_type	Returns the type of node at the root of this probe expression's abstract syntax tree.
has_int8 has_int16 has_int32 has_int64 has_int has_uint8 has_uint16 has_uint32 has_uint64 has_uint	Returns a Boolean value indicating whether or not this probe expression represents a datum with the data type in question.
has_string	Returns a Boolean value indicating whether or not the probe expression has a string data type.
has_children has_left has_right has_center	Returns a Boolean value indicating whether or not the probe expression's node has child probe expression nodes (has_children), has a left-hand child node (has_left), has a right-hand child node (has_right), or has a center node (has_center).
ifelse	Creates a probe expression that represents a conditional statement.
is_same_as	Compares two probe expressions for equivalence.

Table 6 (Page 2 of 2). ProbeExp class function summary

Calling the function:	Does this:
operator + (binary or unary) operator += operator - (binary or unary) operator -= operator * (binary or unary) operator *= operator / operator /= operator % operator %= operator == operator ! operator != operator < operator <= operator << operator <<= operator >> operator >>= operator >>>= operator & (binary or unary) operator &= operator && operator | operator |= operator || operator ^ operator ^= operator ~ operator []	Creates probe expressions to represent particular operations. These common operators have been overloaded so that, when used within the context of the ProbeExp class, they do not execute locally but instead create a probe expression that represent the operation. **Note:** Not all of these operator functions are compatible with all operator types. Refer to the *IBM Parallel Environment for AIX: DPCL Class Reference* for information on which types are valid for each overloaded operator function.
operator =	Assigns values to the ProbeExp object.
sequence	Creates a probe expression that represents the sequence of two probe expressions.
value_int8 value_int16 value_int32 value_int64 value_uint8 value_uint16 value_uint32 value_uint64	Returns the value, of the indicated type, contained within this probe expression.
value_left value_right value_center	Returns the value contained within a child probe expression of the calling probe expression.
value_text	Copies, into a specified buffer, a string representing the value contained within this probe expression node.
value_text_length	Returns the length of the text string contained within this probe expression node. This enables the analysis tool to determine the size of the buffer into which the value_text function will copy the text string.

What is the ProbeHandle class?

Objects of the ProbeHandle class (which is defined in the header file ProbeHandle.h) represent identifying handles to installed point probes. When the analysis tool installs one or more probes as point probes (using the Process::install_probe, Process::binstall_probe, Application::install_probe, or Application::binstall_probe function), the function called will return an output array containing ProbeHandle objects representing each of the installed point probes. Using these probe handles, the analysis tool can make subsequent DPCL

function calls to manipulate the installed point probes. Specifically, the analysis tool can:

- call the `Process::activate`, `Process::bactivate`, `Application::activate`, or `Application::bactivate` function to activate one or more installed point probes. An activated point probe will run when execution reaches its installed location in the code.
- call the `Process::deactivate`, `Process::bdeactivate`, `Application::deactivate`, or `Applicaion::bdeactivate` function to deactivate one or more installed point probes. A deactivated point probe will not execute.
- call the `Process::remove_probe`, `Process::bremove_probe`, `Application::remove_probe`, or `Application::bremove_probe` function to remove one or more installed point probes.

The following table summarizes the various functions of the `ProbeHandle` class. For complete information on any of these functions, refer to the function's AIX man page or the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Table 7. ProbeHandle class function summary

Calling the function:	Does this:
`get_expression`	Returns a copy of the probe expression (`ProbeExp` object) for the point probe represented by this `ProbeHandle` object. Using member functions of the `ProbeExp` object (described in "What is the ProbeExp class?" on page 40), the analysis tool can navigate this probe expression's abstract syntax tree to get information about the probe expression.
`get_point`	Returns the instrumentation point (`InstPoint` object) at which the point probe represented by this `ProbeHandle` object was installed. Using member functions of the `InstPoint` object (described in "What is the InstPoint class?" on page 52), the analysis tool can get information about the instrumentation point at which the point probe has been installed.
`operator =`	Assigns values to the `ProbeHandle` object.

What is the ProbeModule class?

Objects of the `ProbeModule` class (which is defined in the header file `ProbeModule.h`) represent probe modules that the analysis tool can load into one or more target application processes using the `Process::load_module`, `Process::bload_module`, `Application::load_module`, or `Application::bload_module` function. A probe module is a compiled object file containing one or more functions written in C. Once a probe module is loaded into a target application process, its functions are available to the analysis tool as if they were native to the target application.

To invoke a probe module function, the analysis tool can first get a reference to the function by calling the `ProbeModule::get_reference` function. This returns a probe expression (`ProbeExp` object) that represents a reference to the probe module function. This simple probe expression can be specified as a phase probe when instantiating or assigning values to a `Phase` class object (described in "What is the Phase class?" on page 57), or it can be combined (using the `ProbeExp::call` function) into a probe expression representing a function call. The probe expression representing a function call can then be installed as a point probe (using the `Process::install_probe`, `Process::binstall_probe`, `Application::install_probe`, or `Application::binstall_probe`) or executed as a one-shot probe (using the

Process::execute, Process::bexecute, Application::execute, or Application::bexecute function).

The following table summarizes the various functions of the ProbeModule class. For complete information on any of these functions, refer to the function's AIX man page or the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Table 8. ProbeModule class function summary

Calling the function:	Does this:
get_count	Returns the number of functions contained in this probe module.
get_name	Copies, into a specified buffer, the name of a particular function (as identified by a supplied index) in the module.
get_name_length	Returns the length of the mangled name of a particular function in the module. This enables the analysis tool to determine the size of the buffer into which the get_name function will copy the function name.
operator =	Assigns values to the ProbeModule object.
operator == operator !=	Compares two ProbeModule objects for equivalence.
get_reference	Returns a probe expression (ProbeExp object) that represents a reference to a particular function within this probe module. The analysis tool can then use this probe expression in forming a more complex probe expression. In particular, the analysis tool can call the ProbeExp::call function to create a probe expression that represents a call to the particular probe module function.

What are the SourceObj and InstPoint classes?

This section describes two classes that the analysis tool can use to examine the source code associated with a target application process and identify locations, called "*instrumentation points*", where point probes can be installed. These are the SourceObj class (described in “What is the SourceObj class?” and used to represent the source code structure of a target application process) and InstPoint class (described in “What is the InstPoint class?” on page 52 and used to represent instrumentation points).

What is the SourceObj class?

Objects of the SourceObj class (which is defined in the header file SourceObj.h) are called "source objects" and are used by the DPCL system to represent the source code structure associated with a target application process (Process object). The source code structure associated with a process is represented as a hierarchical tree of SourceObj objects — with each SourceObj object in the tree representing a particular type of source object — a program-level source object, a module-level source object, a function-level source object, or a data-level source object. The analysis tool can navigate and query this hierarchy of source objects to:

- display source code information to the analysis tool user.
- get a probe expression (ProbeExp object) that represents a reference to a program function or variable.
- identify an instrumentation point at which it can install point probes.

To get the top, program-level, source object associated with a process, the analysis tool calls the Process::get_program_object function. Since applications may be very large and the analysis tool (or the analysis tool user) may be interested only in

a particular module (compilation unit) within the program, the hierarchy of SourceObj objects under the initial program-level source object does not represent the full source code structure. Instead, for performance reasons, the hierarchy of SourceObj objects reaches down only as far as the module level. The module-level source objects are children of the program-level source object. By calling the SourceObj::child function for the program-level source object, the analysis tool can get any of its child (module-level) source objects, and so navigate one level down the source code hierarchy. To navigate further down into the source hierarchy to examine additional program structure, the analysis tool can then call the SourceObj::expand or SourceObj::bexpand function for the module-level source object. The SourceObj::expand or SourceObj::bexpand functions expand the tree to include the low-level source objects.

The following figure illustrates how an analysis tool can obtain a program-level source object and expand one of its module level source objects so that the source object hierarchy then contains SourceObj object nodes for functions and global data variables. This figure shows the enumeration constants of the SourceType enumeration type. These enumeration constants are returned by the SourceObj::src_type function and identify the particular SourceObj node in the source hierarchy as either a program-level, module-level, function-level, or data-level source object.

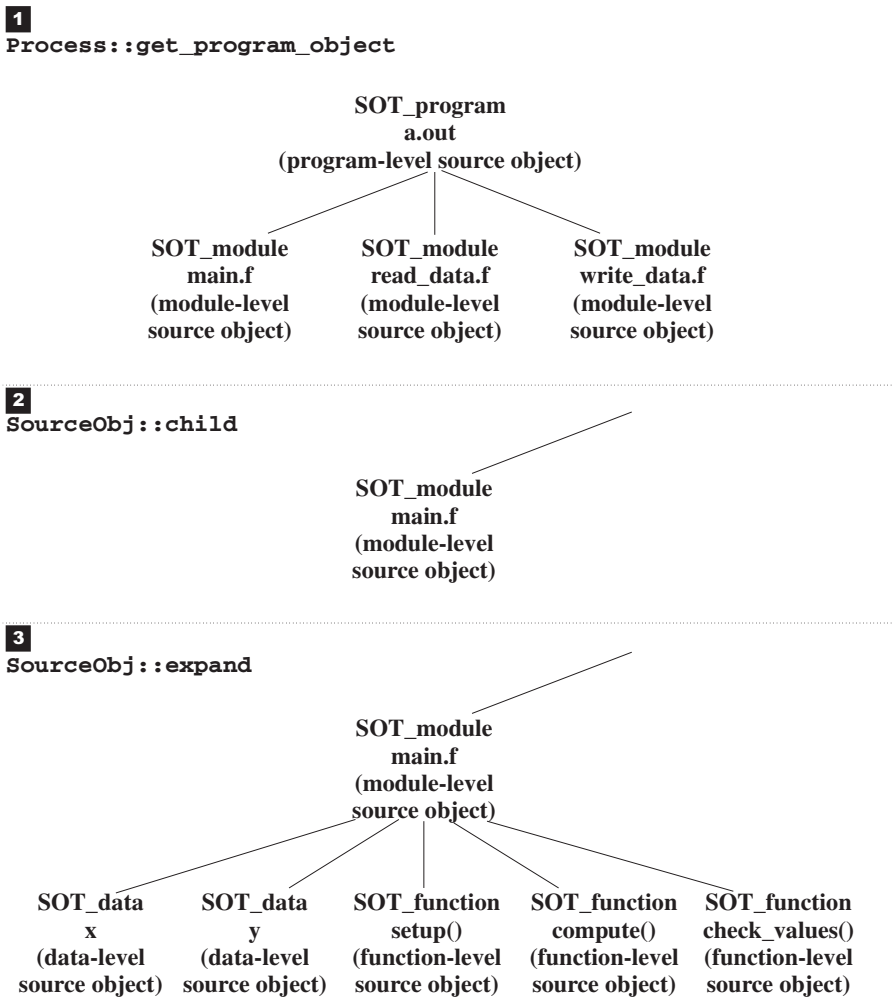

Figure 16. Navigating and expanding a source hierarchy. First, in **1** *, the analysis tool calls the* `Process::get_program_object` *function to get the initial source hierarchy down to the module level. Then, in* **2** *, the analysis tool calls the* `SourceObj::child` *function to navigate down one level in the source hierarchy. Then, in* **3** *, the analysis tool calls the* `SourceObj::expand` *function to expand the particular module's hierarchy of function-level and data-level source objects.*

Once the source hierarchy has been expanded below a particular module-level source object, the analysis tool can navigate further down the hierarchy to obtain information about the module's functions and global data variables. In particular, if the source object is a function-level or data-level source object, the analysis tool can call the `SourceObj::reference` function to get a probe expression that represents a reference to that function or global data variable. Note that the `SourceObj` object representing a global data variable will only be found under the `SourceObj` object representing the module where the global data variable was defined.

The analysis tool can also identify instrumentation points (by calling the `SourceObj::inclusive_point` or `SourceObj::exclusive_point` functions) at which it can install probe expressions as point probes. These functions are described in greater detail next in “What is the InstPoint class?” on page 52.

The following table summarizes the various functions of the SourceObj class. For complete information on any of these functions, refer to the function's AIX man page or the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Table 9 (Page 1 of 2). SourceObj class function summary

Calling the function:	Does this:
address_end	Returns the address of the last element associated with this source object.
address_start	Returns the address of the first element associated with this source object.
child	Returns a particular child source object of this source object.
child_count	Returns the number of child source objects for this source object.
exclusive_point	Returns a particular instrumentation point (InstPoint object) within this source object.
exclusive_point_count	Returns the number of instrumentation points (InstPoint objects) within this source object.
expand bexpand	Expands an unexpanded module. Expanding a module enables the analysis tool to navigate further down into the source hierarchy, examining additional program structure (such as data, functions, and instrumentation points) in the module.
get_data_type	When the source object represents a variable, returns a ProbeType object representing the data type of the variable.
get_demangled_name	When the source object represents a function, copies, into a specified buffer, the demangled name of that function.
get_demangled_name_length	When the source object represents a function, returns the length of the demangled name of the function. This enables the analysis tool to determine the size of the buffer into which the get_demangled_name function will copy the demangled function name.
get_mangled_name	When the source object represents a function, copies, into a specified buffer, the mangled name of that function.
get_mangled_name_length	When the source object represents a function, returns the length of the mangled name of the function. This enables the analysis tool to determine the size of the buffer into which the get_mangled_name function will copy the mangled function name.
get_program_type	Indicates whether the program is using the 32-bit or 64-bit address memory model. This function indicates this information by returning one of the enumeration constants (SOL_lp32 or SOL_lp64) of the LpModel enumeration type.
get_variable_name	When the source object represents a data variable, copies, into a specified buffer, the variable name.
get_variable_name_length	When the source object represents a data variable, returns the length of the variable name. This enables the analysis tool to determine the size of the buffer into which the get_variable_name function will copy the variable name.
inclusive_point	Returns a particular instrumentation point (InstPoint object) within this source object or any of its child source objects.
inclusive_point_count	Returns the number of instrumentation points (InstPoint objects) within this source object and all of its child source objects.
line_end	Returns the approximate source line number of the last line in this source object.
line_start	Returns the approximate source line number of the first line in this source object.
module_name	Copies, into a specified buffer, the file name and path of the module that contains this source object.
module_name_length	Returns the length of the file name and path of the module that contains this source object. This enables the analysis tool to determine the size of the buffer into which the module_name function will copy the module's file name and path.
obj_parent	Returns the parent source object for this source object.
operator =	Assigns values to the SourceObj object.
operator == operator !=	compares two SourceObj objects for equivalence.
program_name	When the source object represents a program, copies, into a specified buffer, the file name and path of the executable program.

Table 9 (Page 2 of 2). SourceObj class function summary	
Calling the function:	**Does this:**
program_name_length	When the source object represents a program, returns the length of the file name and path of the executable program. This enables the analysis tool to determine the size of the buffer into which the program_name function will copy the executable program's file name and path.
reference	When the source object represents a program function or variable, creates a probe expression (ProbeExp object) that represents a reference to that function or variable. The analysis tool can then use these probe expressions in forming more complex probe expressions. A probe expression representing a function can be combined by the analysis tool into a probe expression representing a function call. A probe expression representing a variable can be used by the analysis tool in creating a probe expression that modifies or otherwise uses that variable.
src_type	Returns the source object type of this source object. This function indicates this information by returning one of the enumeration constants of the SourceType enumeration type.

What is the InstPoint class?

Objects of the InstPoint class (which is defined in the header file InstPoint.h) represent instrumentation points — locations within the target process(es) where the analysis tool can install point probes by calling the Process::install_probe, Process::binstall_probe, Application::install_probe, or Application::binstall_probe functions. All four of these functions must, for each probe expression being installed as a point probe, be passed an InstPoint object that indicates where the probe should be installed. To get an InstPoint object, the analysis tool calls the SourceObj::exclusive_point or SourceObj::inclusive_point function. (Note that in order for these functions to return an InstPoint object, at least one of the module-level source objects in the source object hierarchy must have been expanded by the SourceObj::expand or SourceObj::bexpand function.) To determine the number of instrumentation points under a particular source object, the analysis tool can call the SourceObj::exclusive_point_count or SourceObj::inclusive_point_count function. The SourceObj::exclusive_point_count function returns the number of InstPoint objects associated with the particular source object, while the SourceObj::inclusive_point_count function returns the number of InstPoint objects associated with a particular source object and all of its child source objects.

Figure 17 on page 53 illustrates the difference between exclusive and inclusive point counts.

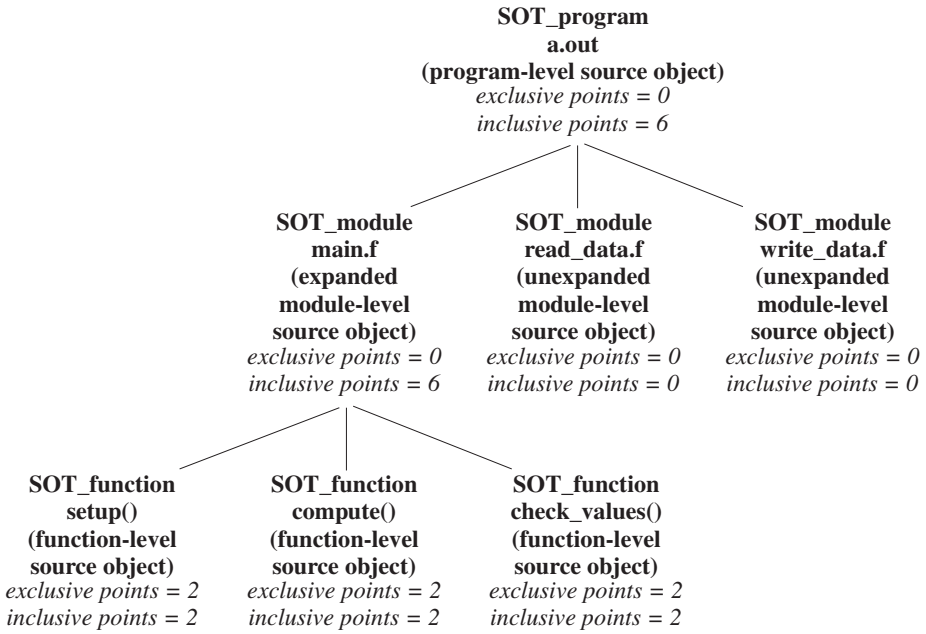

Figure 17. Exclusive and inclusive instrumentation point counts for source objects. Exclusive instrumentation points refer to those instrumentation points associated with the source object itself. Inclusive instrumentation points refer to those instrumentation points associated with the source object, and all of its children. Note that the two unexpanded module-level source objects show no instrumentation points. That is because the instrumentation point information is only available once the module-level source object has been expanded.

When calling either the `SourceObj::exclusive_point` or the `SourceObj::inclusive_point` function, the analysis tool identifies a particular `InstPoint` object by supplying the function with an index value. The `SourceObj::exclusive_point_count` and `SourceObj::inclusive_point_count` functions can be used to initialize a loop that will cycle though the `InstPoint` objects and examine each in turn to determine if it represents a suitable location at which to install a particular probe expression as a point probe.

The analysis tool can also get the `InstPoint` object for a point probe that has already been installed by calling the `ProbeModule::get_point` function.

Once the analysis tool has a particular `InstPoint` object, it can call `InstPoint` class functions to get information about the instrumentation point. If the analysis tool is examining the instrumentation point to determine if a point probe should be installed there, it will want to know whether the instrumentation point represents a function entry, function exit, or function call site. To get this information, it can call the `InstPoint::get_type` function, which returns one of the enumeration constants (`IPT_function_entry`, `IPT_function_exit`, or `IPT_function_call`) of the `InstPtType` enumeration type. If the instrumentation point represents a function call site, the analysis tool can also get the location where the point probe will be placed relative to the instrumentation point. To get this information, the analysis tool can call the `InstPoint::get_location` function which returns one of the enumeration constants (`IPL_before` or `IPL_after`) of the `InstPtLocation` enumeration type. Figure 18 on page 54 illustrates the various instrumentation point type and location information that the analysis tool can get by calling the `InstPoint::get_type` and `InstPoint::get_location` functions.

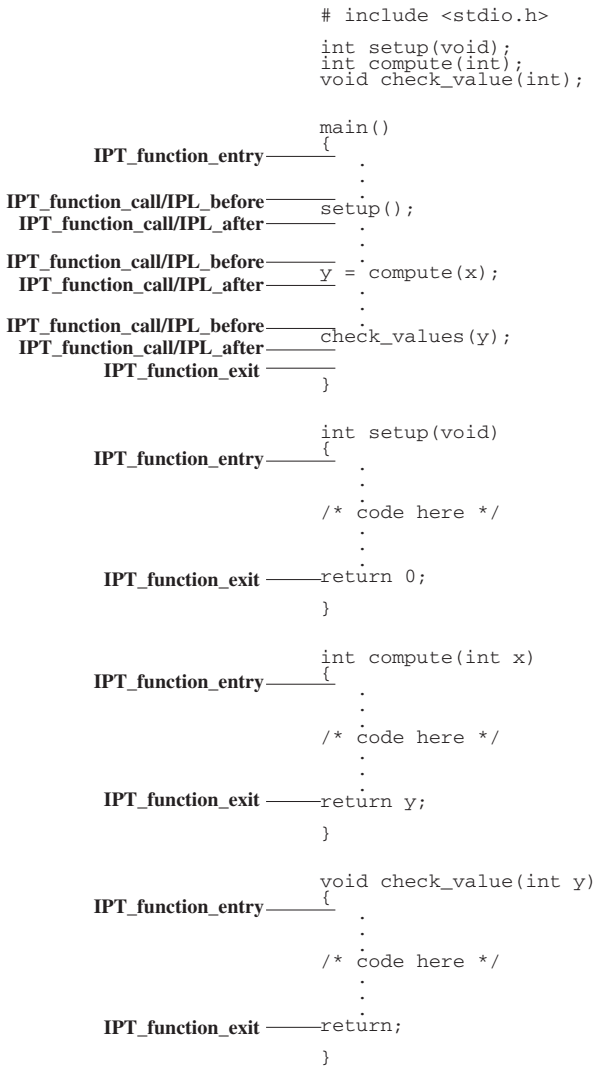

Figure 18. Instrumentation point types and locations

In addition to the instrumentation point type and location information, the analysis tool can:

- when the instrumentation point refers to a function call site, get the mangled or demangled name of the function being called.
- get the `SourceObj` object that contains the instrumentation point.
- get the approximate line number in the source code where the instrumentation point is located.

The following table summarizes the various functions of the `InstPoint` class. For complete information on any of these functions, refer to the function's AIX man page or the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Table 10 (Page 1 of 2). InstPoint class function summary

Calling the function:	Does this:
`get_address`	Returns the address of this instrumentation point within the target application program.
`get_container`	Returns the source object (`SourceObj` object) that contains this instrumentation point.

Table 10 (Page 2 of 2). InstPoint class function summary	
Calling the function:	**Does this:**
get_demangled_name	When the instrumentation point represents a function call site, copies, into a specified buffer, the demangled name of that function.
get_demangled_name_length	When the instrumentation point represents a function call site, returns the length of the demangled name of the function. This enables the analysis tool to determine the size of the buffer into which the get_demangled_name function will copy the demangled function name.
get_line	Returns the approximate line number in the source code where this instrumentation point occurs.
get_location	When the instrumentation point represents a function call site, indicates whether the point probe will execute before or after the function call. This function indicates this information by returning one of the enumeration constants (IPL_before or IPL_after) of the InstPtLocation enumeration type.
get_mangled_name	When the instrumentation point represents a function call site, copies, into a specified buffer, the mangled name of that function.
get_mangled_name_length	When the instrumentation point represents a function call site, returns the length of this function's mangled name. This enables the analysis tool to determine the size of the buffer into which the get_mangled_name function will copy the mangled function name.
get_type	Indicates whether this instrumentation point represents a function entry, function exit, or function call site. This function indicates this information by returning one of the enumeration constants (IPT_function_entry, IPT_function_exit, or IPT_function_call) of the InstPtType enumeration type.
operator =	Assigns values to the InstPoint object.
operator == operator !=	Compares two InstPoint objects for equivalence.

What is the ProbeType class?

Objects of the ProbeType class (which is defined in the header file ProbeType.h) represent data types associated with either a source object (SourceObj object), or a probe expression (ProbeExp object). The member and friend functions of this class are of two types — those that enable the analysis tool to create a ProbeType object to represent a data type, and those that enable the analysis tool to query information about a ProbeType object.

The functions that create ProbeType objects to represent a particular data type are used by the analysis tool primarily for allocating memory for probes. When allocating memory for use by probes (using the Process::alloc_mem, Process::balloc_mem, Application::alloc_mem, and Application::balloc_mem function), the analysis tool supplies a ProbeType object as a function parameter in order to indicate the type of memory to allocate.

The functions that query ProbeType objects enable the analysis tool to ascertain the data type associated with a particular source object (SourceObj object) or probe expression (ProbeExp object). The ProbeType object associated with a source object is returned by the SourceObj::get_data_type function, while the ProbeType object associated with a probe expression is returned by the ProbeExp::get_data_type function. The ability to get the data type of a source object or probe expression enables the analysis tool to use the source object or probe expression value correctly.

One particularly useful ability of the ProbeType class is to return actual integer and pointer values of function parameters. To do this, the analysis tool first creates a ProbeType object that represents the function prototype (by calling the ProbeType::function_type function). Then, it can create a probe expression by calling the ProbeType::get_actual function (an index value supplied to the ProbeType::get_actual function indicates the particular function parameter). When the probe expression returned by the ProbeType::get_actual function is executed within the target application process, it will get the actual parameter value. Note that the ProbeType::get_actual function can only return an actual parameter value if it represents a 32–bit integer or a pointer.

One important thing you should understand about ProbeType objects is that they are expression trees. While in many cases (such as ProbeType objects representing an integer) it is not necessary to represent the ProbeType object as a tree, we have structured ProbeType objects as trees in order to have ProbeType objects that represent pointers and function prototypes. (See Figure 19).

The data type information for each ProbeType object is represented by an enumeration constant of the DataExpNodeType enumeration type. The analysis tool can query this constant by calling the ProbeType::get_node_type function. The analysis tool can also determine how many child ProbeType objects are under a particular ProbeType object by calling the ProbeType::child_count function. To get a particular child ProbeType object of a parent ProbeType object, the analysis tool can call the ProbeType::child function. For example, if calling the ProbeType::get_node_type function returns the enumeration constant DEN_pointer_type (indicating a pointer to a pointee), the analysis tool can get the ProbeType object for the pointee by calling the ProbeType::child function.

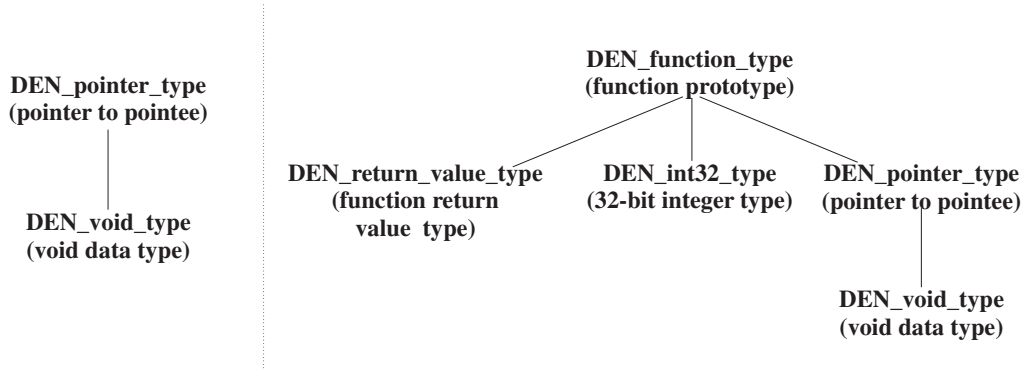

Figure 19. Probe type trees. This figure illustrates the two situations (data types representing pointers and function prototypes) in which a ProbeType *expression tree will be more complex than a single node. The analysis tool can call the* ProbeType::get_node_type *function to determine the type represented by the* ProbeType *object's node in the expression tree. To navigate down into the tree, the analysis tool can call the* ProbeType::child *function to get a* ProbeType *object that represents one of the child* ProbeType *object nodes of the current* ProbeType *object's node. In the case of a ProbeType object that represents a function prototype, its leftmost subtree will represent the function return type and its right subtrees will represent any function parameters.*

The following table summarizes the various functions of the ProbeType class. For complete information on any of these functions, refer to the function's AIX man page or the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Table 11. ProbeType class function summary	
Calling the function:	**Does this:**
child	Returns the sub-type of a data type (a child ProbeType object of this ProbeType object). The analysis tool can use this function to: • get the data type of the pointee (for a ProbeType object representing a pointer). • get the data type of a function return type or argument (for a ProbeType object representing a function).
child_count	Returns the number of sub-types associated with a data type (the number of child ProbeType objects for this ProbeType object).
function_type	A friend function. Creates a ProbeType object that represents a function prototype or type signature data type.
get_actual	When the ProbeType object represents a function prototype, returns a probe expression (ProbeExp object) representing the value of a particular function parameter. Note that the ProbeType::get_actual function can only return an actual parameter value if it represents an integer or a pointer.
get_node_type	Indicates the type of this ProbeType object's node in the data type expression tree. This function indicates this information by returning one of the enumeration constants of the DataExpNodeType enumeration type.
int32_type	A friend function. Creates a ProbeType object that represents a 32-bit integer.
operator =	Assigns values to the ProbeType object.
operator == operator !=	Compares two ProbeType objects for equivalence.
pointer_type	A friend function. Creates a ProbeType object that represents the data type of a pointer to a pointee.
unspecified_type	A friend function. Creates a ProbeType object that represents an unspecified data type.
void_type	A friend function. Creates a ProbeType object that represents a void data type.

What is the Phase class?

Objects of the Phase class (which is defined in the header file Phase.h) represent the control mechanism for invoking phase probes at set intervals. The phase probes themselves are probe expressions (ProbeExp objects) that represent references to probe module functions. Before instantiating a Phase object, the analysis tool will:

1. add the probe module containing the functions to be invoked as phase probes to one or more target application processes. The analysis tool can add the probe module to a single process by calling the Process::load_module or Process::bload_module function. The analysis tool can also add the probe module to all processes managed by an Application object by calling the Application::load_module or Application::bload_module function.
2. create a probe expression (ProbeExp object) representing a reference to each probe module function that will serve as a phase probe. To do this, the analysis tool calls the ProbeModule::get_reference function.

When instantiating the Phase class object, the analysis tool can specify up to three phase probes (ProbeExp objects that reference probe module functions) to be invoked and the CPU-time interval at which their execution will be triggered. The three phase probes that can be invoked represent a begin function, a data function, and an end function. While the phase must, in order to be useful, call at least one of these functions, any one of them is optional. At the very least, an analysis tool will usually supply a data function.

Once the analysis tool has instantiated a Phase class object, it can:

- add the Phase to one or more target application processes by calling the Process::add_phase, Process::badd_phase, Application::add_phase, or Application::badd_phase function.
- if the Phase specifies a data function, allocate and associate data with the Phase by calling the Process::alloc_mem, Process::balloc_mem, Application::alloc_mem, or Application::balloc_mem function. Each time the phase is triggered, its data function executes once per datum that the analysis tool has previously allocated and associated with the phase.
- specify a set of probe module functions (ProbeExp objects that represent a reference to those functions) to be executed when the phase is removed or the application terminates. To specify this set of "exit functions", the analysis tool can call the Process::set_phase_exit, Process::bset_phase_exit, Application::set_phase_exit, or Application::bset_phase_exit function.
- ascertain, by calling the Process::get_phase_period function, the CPU-time interval at which the phase is activated and its phase probes are invoked. If desired, the analysis tool can reset this interval by calling the Process::set_phase_period, Process::bset_phase_period, Application::set_phase_period, or Application::bset_phase_period function.
- remove the Phase from one or more target application process by calling the Process::remove_phase, Process::bremove_phase, Application::remove_phase, or Application::bremove_phase function.

As you can see from the preceding list, most of the functions that act upon a Phase class object are actually member functions of other classes. The Phase class itself contains member functions only for assignment and equivalence comparison as shown in the following table. For complete information on any of the functions described in the preceding list or the following table, refer to the function's AIX man page or the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Table 12. Phase class function summary

Calling the function:	Does this:
operator =	Assigns values to the Phase object.
operator == operator !=	Compares two Phase objects for equivalence.

What is the AisStatus class?

Objects of the AisStatus class (which is defined in the header file AisStatus.h) store status and severity codes (and, in some cases, data associated with the status) returned by other DPCL functions, and can also be used by the analysis tool to store status values for its own purposes. When a DPCL function call returns an AisStatus object, the analysis tool can check its status code by calling the AisStatus::status function, and its severity by calling the AisStatus::severity function. The AisStatus::status function returns one of the constants enumerated in the AisStatusCode enumeration type, and the AisStatus::severity function returns one of the constants enumerated in the AisSteverityCode enumeration type. For a complete listing and description of these constants, refer to the *IBM Parallel Environment for AIX: DPCL Class Reference*.

In some situations, the AisStatus object returned by the DPCL system will contain one or more data values associated with the particular status. To determine if an AisStatus object has any additional data values, the analysis tool can call the AisStatus::data_count function. If there are, it can use the AisStatus::data_value and AisStatus::data_value_length functions to get the value.

The analysis tool can also create AisStatus objects for its own status reporting. The AisStatus class constructor enables the analysis tool to specify a status and severity value and the AisStatus::add_data function enables it to associate data with the status.

The following table summarizes the various functions of the AisStatus class. For complete information on any of these functions, refer to the function's AIX man page or the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Table 13. AisStatus class function summary

Calling the function:	**Does this:**
add_data	Adds one data value to the list of data associated with this AisStatus object.
data_count	Returns the number of data values associated with this AisStatus object.
data_value	Copies, into a specified buffer, a particular data value associated with this AisStatus object.
data_value_length	Returns the length of a particular data value associated with this AisStatus object. This enables the analysis tool to determine the size of the buffer into which the data_value function will copy the data value.
operator =	Assigns values to the AisStatus object
operator AisStatusCode	Cast function that returns the status code reflected in this AisStatus object.
operator int	Cast function that returns the integer equivalent of the status code reflected in this AisStatus object.
severity	Returns the severity code reflected in this AisStatus object. The severity code is returned as one of the enumeration constants of the AisSeverityCode enumeration type.
status	Returns the status code reflected in the object. The status code is returned as one of the enumeration constants of the AisStatusCode enumeration type.
status_name	Returns the name of the status code reflected in the object. The name is in American English, and the string is stored in a constant array within the function. This function is intended only for limited diagnostic use during tool development.

Chapter 3. A DPCL hello world program

Since this is a programming guide, tradition dictates that our first code example should be a "Hello World" program (a simple program that prints out the string "Hello World"). Unlike, traditional "Hello World" programs, however, our program (`hello.c`) does not print out the "Hello World" string. Instead, all it does is put itself into an infinite loop. This will be the target application in our example; we will show how to instrument it so that it sends the "Hello World" string back to our analysis tool.

The source code for both the "Hello World" target application and the analysis tool were copied to the directory */usr/lpp/ppe.dpcl/samples/hello* when you installed DPCL. If you want to run these programs for illustration, refer to the instructions “Compiling, linking, and running the DPCL hello world program” on page 70.

The hello world target application

The code for our target application is shown below. The reason we have designed it to enter an infinite loop is so that the analysis tool we create in this example will have time to connect to it. Without the infinite loop, this target application would exit before our analysis tool had a chance to instrument it. We also put in some print and sleep statements so its execution will be more visible for this example.

```
#include <stdio.h>

void hello(void);

main()
{
    while (1)
    {
      hello();
    }
}

void hello(void)
{
      /* This is where our call to printf("Hello World"); would be */

   fprintf(stdout,".");   /* something to help us see what is going on */
   fflush(stdout);
   sleep(1);

   return;
}
```

The hello world analysis tool

Since the DPCL is a C++ class library, we will use C++ to build our analysis tool. We will call our analysis tool `eut_hello` and construct it in the file `eut_hello.C`. In order to instrument our target application to print out the "Hello World" string, our analysis tool needs to:

1. initialize itself to use the DPCL
2. connect to the target application
3. create a probe that will, once installed in the target application, send the "Hello World" string back to the analysis tool

4. install the probe into the target application
5. enter the DPCL main event loop

These are the basic steps that DPCL analysis tools will follow to instrument target applications. Note that since we are instrumenting a serial application in this example:

- we will use an instance of the DPCL `Process` class to represent the target application. If this were a parallel application, we would instead represent the target application as an instance of the `Application` class or the `PoeAppl` class. Keep in mind that, for all the `Process` class member functions illustrated in this example, there are equivalent `Application` and `PoeAppl` member functions.
- we will, when they are available, use the blocking API calls in this example. Since the target application in this example is not a parallel application, there is no advantage to using the nonblocking versions of these calls. Keep in mind, however, that the blocking calls we use in this example also have nonblocking equivalents.

Also be aware that this is a simple DPCL programming example that does not perform rigorous error checking. In general, actual DPCL programs you create should check the status of DPCL function calls and respond to error conditions. Here's the source code for our "Hello World" analysis tool. The rest of this chapter will describe this code in more detail.

```
#include <stdio.h>
#include <stdlib.h>    // for atoi() call
#include <libgen.h>    // for basename() call
#include <dpcl.h>

#define MODNAME "hello.c"
#define FCNNAME "hello"

void data_cb (GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg);

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

main(int argc, char *argv[])
{
    if (argc != 3) {
        printf("Usage: %s <hostname> <target_pid>\n", argv[0]);
        exit(99);
    }

    int apppid = atoi(argv[2]);

    Process  P(argv[1], apppid);

    Ais_initialize();

    printf("Connecting to process %d on node %s\n", apppid, argv[1]);

    AisStatus sts = P.bconnect();
    if (sts.status() != ASC_success) {
        printf("bconnect error %s\n", sts.status_name());
        printf("exiting...\n");
        return 99;
    }

    SourceObj myprog = P.get_program_object();

    SourceObj mymod;

    const int   bufSize = 1024;
    char        bufmname[bufSize];      // buffer for module_name(..)
```

```
printf("\n");
printf("module count = %d\n", myprog.child_count());
printf("looking for module '%s'\n", MODNAME);

int found = 0;    // flag for whether we find module we want

for (int c = 0; found == 0 && c < myprog.child_count(); c++) {
    mymod = myprog.child(c);
    mymod.module_name(bufmname, bufSize);
    if (strcmp(basename(bufmname), MODNAME) == 0)
        found = 1;
}

if ( !found ) {
    printf("cannot find module '%s'\n", MODNAME);
    printf("disconnecting and exiting...\n");
    P.bdisconnect();
    return 99;
}

printf("\n");
printf ("found module %s -- expanding....\n", MODNAME);

sts = mymod.bexpand(P);
if (sts.status() != ASC_success) {
    printf("bexpand failed: %s\n", sts.status_name());
    printf("disconnecting and exiting...\n");
    P.bdisconnect();
    return 99;
}

printf("\n");
printf("function count = %d\n", mymod.child_count());
printf("look for the function '%s'\n", FCNNAME);

SourceObj myfun;

char      bufdname[bufSize];   // buffer for get_demangled_name(..)

found = 0;
for ( c = 0; found == 0 && c < mymod.child_count(); c++) {
    myfun = mymod.child(c);
    myfun.get_demangled_name(bufdname, bufSize);
    if (strcmp(bufdname, FCNNAME) == 0)
        found = 1;
}

if ( !found ) {
    printf("cannot find function '%s'\n", FCNNAME);
    printf("disconnecting and exiting...\n");
    P.bdisconnect();
    return 99;
}

printf("\n");
printf("looking for an instrumentation point\n");

InstPoint  mypoint;

printf("point count = %d\n", myfun.exclusive_point_count());

found = 0;
for ( c = 0; found == 0 && c < myfun.exclusive_point_count(); c++) {
    mypoint = myfun.exclusive_point(c);
    if ( mypoint.get_type() == IPT_function_entry)
        found = 1;
}

if ( !found ) {
    printf("cannot find entry point for function '%s'\n", FCNNAME);
    printf("disconnecting and exiting...\n");
    P.bdisconnect();
```

```
        return 99;
    }

    ProbeExp parms[3];   // Create an array of expressions

    char * hello_message = "Hello World";

    parms[0] = Ais_msg_handle;
    parms[1] = ProbeExp(hello_message);
    parms[2] = ProbeExp (1 + strlen(hello_message));

    ProbeExp      myexp = Ais_send.call(3,parms);

    ProbeHandle   myph;

    GCBFuncType   mydcb = data_cb;

    GCBTagType    mytg = 0;

    printf("\n");
    printf("found instrumentation point -- installing probe\n");
    sts = P.binstall_probe(1, &myexp, &mypoint, &mydcb, &mytg, &myph);
    if (sts.status() != ASC_success) {
        printf("binstall_probe failed: %s\n", sts.status_name());
        printf("disconnecting and exiting...\n");
        P.bdisconnect();
        return 99;
    }

    sts = P.bactivate_probe(1, &myph);
    if (sts.status() != ASC_success) {
        printf("bactivate_probe failed: %s\n", sts.status_name());
        printf("disconnecting and exiting...\n");
        P.bdisconnect();
        return 99;
    }

    printf("\n");
    Ais_main_loop();
}

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void
data_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg)
{
    printf("%s\n", (char*) msg);
}

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Step 1: Initialize tool to use the DPCL system

In order to use the DPCL, all analysis tools must:

- include the header file for each DPCL class and function group it uses.

  ```
  #include <dpcl.h>
  ```

- call the DPCL initialization routine (`Ais_initialize`)

  ```
  Ais_initialize();
  ```

Step 2: Connect to the target application

The next step is to create a `Process` class object that represents the target application process, and then use this `Process` class object to connect to the target application process. The `Process` class constructor takes two parameters — one specifying the host name where the target application is running, and the other specifying its process ID. Our analysis tool will have the user supply this information as command-line arguments when starting the tool; the first argument will specify the host name, and second will be the process ID.

The following statement initializes an object of type `Process` using the information supplied in the command-line arguments.

```
Process  P(argv[1], apppid);
```

Once our analysis tool has a `Process` class object that identifies the target application process, it connects to that process using the `Process::bconnect` member function.

```
Ais_initialize();

printf("Connecting to process %d on node %s\n", apppid, argv[1]);

AisStatus sts = P.bconnect();
if (sts.status() != ASC_success) {
    printf("bconnect error %s\n", sts.status_name());
    printf("exiting...\n");
    return 99;
}
```

Step 3: Create hello world probe

Now that it's connected to the target application, our analysis tool needs to create a point probe that will send the string "Hello World" back to our analysis tool each time execution enters the target application's `hello` function. A simple probe expression will be sufficient to accomplish this; a probe module will not be necessary in this case. To send data back to the analysis tool, the DPCL provides a predefined probe expression (`Ais_send`). `Ais_send` is a probe expression representation of a function for sending data back to the analysis tool. The function represented by the `Ais_send` probe expression takes three parameters — a message handle for managing where the message is sent, the address of the data to send, and the size of the data being sent. If we were able to hand code this function call into our target application, it would look something like this:

```
Ais_send(Ais_msg_handle, "Hello World", 12);
```

Using the `ProbeExp` class in the DPCL, however, we have to use a slightly different approach to accomplish the same thing. That is because `Ais_send` is a probe expression representation of the actual function, and each parameter to the function also needs to be a probe expression. Then, all these individual probe expressions need to be combined into a single probe expression that represents the function call with parameters.

First our analysis tool needs to create an array of probe expressions, each representing one of the parameters to the `Ais_send` function. Note in the following code that `Ais_msg_handle` is another predefined probe expression supplied by the DPCL. It is specifically designed for the `Ais_send` function for managing where the message is sent.

```
ProbeExp parms[3];   // Create an array of expressions

char * hello_message = "Hello World";

parms[0] = Ais_msg_handle;
parms[1] = ProbeExp(hello_message);
parms[2] = ProbeExp (1 + strlen(hello_message));
```

Next the analysis tool can create a probe expression that calls `Ais_send` using the three parameters defined in the `parms` array.

```
ProbeExp      myexp = Ais_send.call(3,parms);
```

So now our analysis tool has a probe expression that, once installed as a point probe within the target application's `hello` function, will, each time execution enters the `hello` function, send the "Hello World" string back to the analysis tool.

What our analysis tool needs now is a callback routine that will handle those "Hello World" messages sent back from the point probe. In this example, our callback will simply print the string that it receives to standard output. The following code shows our callback definition:

```
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void
data_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg)
{
    printf("%s\n", (char*) msg);
}

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Step 4: Install and execute probe in the target application

In the preceding step (“Step 3: Create hello world probe” on page 65), we created a probe that will send the "Hello World" string back to the analysis tool. Now we need to identify the location, or "*instrumentation point*", within the target application where we can install our probe. When the analysis tool connected to the target application, the DPCL system gathered information about the target application's source structure down to the module (or file) level. Our analysis tool can examine, and expand, this structure to find an appropriate instrumentation point.

First the analysis tool needs to get a reference to the source structure of the target application process. This source structure is represented as a hierarchy of source objects (`SourceObj` class objects). The top level source object (the root of the hierarchy) is called the "program object". To get a reference to this object, our analysis tool uses the member function `Process::get_program_object`.

```
SourceObj myprog = P.get_program_object();
```

Next our analysis tool needs to search through the list of children contained within the program object. This list of children should be the set of modules contained within the target application. To find the `hello.c` module, our analysis tool uses the `SourceObj::child_count` function to determine the number of children (modules) in the program object. Once it knows the number of children, it uses a `for` loop to check the name of each child module source object of the program object. To get each child module source object of the program object, the analysis tool uses the `SourceObj::child` function. To check the name of each child, the analysis tool uses the `SourceObj::module_name` function.

```
SourceObj mymod;

const int   bufSize = 1024;
char        bufmname[bufSize];      // buffer for module_name(..)

printf("\n");
printf("module count = %d\n", myprog.child_count());
printf("looking for module '%s'\n", MODNAME);

int found = 0;    // flag for whether we find module we want

for (int c = 0; found == 0 && c < myprog.child_count(); c++) {
    mymod = myprog.child(c);
    mymod.module_name(bufmname, bufSize);
    if (strcmp(basename(bufmname), MODNAME) == 0)
        found = 1;
}

if ( !found ) {
    printf("cannot find module '%s'\n", MODNAME);
    printf("disconnecting and exiting...\n");
    P.bdisconnect();
    return 99;
}

printf("\n");
printf ("found module %s -- expanding....\n", MODNAME);

sts = mymod.bexpand(P);
if (sts.status() != ASC_success) {
    printf("bexpand failed: %s\n", sts.status_name());
    printf("disconnecting and exiting...\n");
    P.bdisconnect();
    return 99;
}
```

Note that once the analysis tool finds the `hello.c` module object, it must expand it. This is because, when the DPCL system connects to the target application, it retrieves the source hierarchy only down to the module level. To obtain additional information, our analysis tool must expand the specific module it is interested in. In the preceding code example, our analysis tool did this using the `SourceObj::bexpand` function.

```
sts = mymod.bexpand(P);
```

Now that the `hello.c` module object is expanded, our analysis tool can go deeper into the target application's source hierarchy. This next level contains the list of function and global data variables found within the module. In this next segment of code, our analysis tool looks through the list of the `hello.c` module's children looking for the `hello` function. To do this, the analysis tool uses the `SourceObj::child_count`, `SourceObj::child`, and `SourceObj::get_demangled_name` member functions.

```
printf("\n");
printf("function count = %d\n", mymod.child_count());
printf("look for the function '%s'\n", FCNNAME);

SourceObj myfun;

char      bufdname[bufSize];   // buffer for get_demangled_name(..)

found = 0;
for ( c = 0; found == 0 && c < mymod.child_count(); c++) {
    myfun = mymod.child(c);
    myfun.get_demangled_name(bufdname, bufSize);
    if (strcmp(bufdname, FCNNAME) == 0)
        found = 1;
}

if ( !found ) {
    printf("cannot find function '%s'\n", FCNNAME);
    printf("disconnecting and exiting...\n");
    P.bdisconnect();
    return 99;
}
```

Now that it has a reference to the `hello` function source object, our analysis tool needs to find an instrumentation point where it can place its point probe. In particular, the analysis tool is looking for an instrumentation point that represents the function entry point. To do this, the analysis tool uses a technique similar to the one used to find the `hello` function. Instead of the `SourceObj::child_count`, `SourceObj::type`, and `SourceObj::module_name` functions that it used to find the `hello` function, however, it now uses functions designed to identify instrumentation point information. These functions are the `SourceObj::exclusive_point_count`, `SourceObj::exclusive_point`, and `SourceObj::get_type` functions.

```
printf("looking for an instrumentation point\n");

InstPoint  mypoint;

printf("point count = %d\n", myfun.exclusive_point_count());

found = 0;
for ( c = 0; found == 0 && c < myfun.exclusive_point_count(); c++) {
    mypoint = myfun.exclusive_point(c);
    if ( mypoint.get_type() == IPT_function_entry)
        found = 1;
}

if ( !found ) {
    printf("cannot find entry point for function '%s'\n", FCNNAME);
    printf("disconnecting and exiting...\n");
    P.bdisconnect();
    return 99;
}
```

At this point in the analysis tool's code, it has all it needs to instrument the target application. It has:

- an object of type `Process` that represents the target application process
- an object of type `InstPoint` that represents the function entry point to the target application's `hello` function.
- a probe expression (`ProbeExp` object) that sends back "Hello World" strings to the analysis tool. The analysis tool will install this probe expression as a point probe at the entry point to the target application's `hello` function.

- a callback routine that will print the "Hello World" strings as they are sent from the installed point probe.

Now the analysis tool can actually install the probe expression as a point probe within the target application. The following code installs our probe using the member function `Process::binstall_probe` which takes, as parameters:

- the probe expression to install
- the instrumentation point that represents where to install it
- a tag parameter for passing optional information to the callback
- the callback for handling messages sent from the probe
- a probe handle that the analysis tool can use to reference the probe after it is installed.

```
ProbeHandle   myph;

GCBFuncType   mydcb = data_cb;

GCBTagType    mytg = 0;

printf("\n");
printf("found instrumentation point -- installing probe\n");
sts = P.binstall_probe(1, &myexp, &mypoint, &mydcb, &mytg, &myph);
if (sts.status() != ASC_success) {
    printf("binstall_probe failed: %s\n", sts.status_name());
    printf("disconnecting and exiting...\n");
    P.bdisconnect();
    return 99;
}
```

Now the point probe is installed, but it is not yet active. To activate our probe so that its code will execute as if it were part of the target application's code, our analysis tool calls the function `Process::bactivate_probe` as shown below. Note that this function takes, as a parameter, the probe handle `myph` (specified when the analysis tool installed the probe).

```
    sts = P.bactivate_probe(1, &myph);
    if (sts.status() != ASC_success) {
        printf("bactivate_probe failed: %s\n", sts.status_name());
        printf("disconnecting and exiting...\n");
        P.bdisconnect();
        return 99;
    }

    printf("\n");
    Ais_main_loop();
}
```

Step 5: Entering the DPCL main event loop

At the bottom of our `main` routine, note that the analysis tool calls the DPCL event handling function (`Ais_main_loop`). This is necessary because the DPCL system is asynchronous in the way it interfaces with the analysis tool. Calling the `Ais_main_loop` function places the analysis tool in an event loop for responding asynchronously to messages sent by the DPCL system.

```
Ais_main_loop();
```

Compiling, linking, and running the DPCL hello world program

Now that we have described the code contained in both our target application (`hello.c`) and our analysis tool (`eut_hello.C`), let's run these programs. When you installed DPCL (as described in *IBM Parallel Environment for AIX: Installation*), the source code for these two programs was copied to the directory */usr/lpp/ppe.dpcl/samples/hello*. To run these programs:

1. Copy the contents of the `hello` directory to a location where you can update the files. You must do this since root owns the `hello` directory, and you otherwise will not be able to run the makefiles provided.

   ```
   cp /usr/lpp/ppe.dpcl/samples/hello/* mydir
   ```

2. Change directories to the directory that contains the copied files.

   ```
   cd mydir
   ```

3. Compile and link the target application as you normally would. We have provided a makefile for this step.

   ```
   make -f Makefile.hello
   ```

4. Prelink your target application with a special DPCL library. We have provided a general-purpose shell script (`prelink`) to perform this step. This script is located in the directory */usr/lpp/ppe.dpcl/samples/prelink*. Assuming you have copied the script to the same directory as your *hello* executable, you would enter the following at the AIX command prompt.

   ```
   prelink hello
   ```

   The prelink script creates a prelinked output file with the extension `.DPCL`; in this example `hello.DPCL`.

5. Compile and link the analysis tool. We have provided a makefile for this step.

   ```
   make -f Makefile.eut_hello
   ```

6. Start the target application.

   ```
   hello.DPCL
   ```

7. In a separate xterm window, start the analysis tool, passing it the host name of the node running the target application and the target application's process ID. You can use the AIX command **ps** to ascertain the target application's process ID.

   ```
   eut_hello hostname process-ID
   ```

As the analysis tool runs, it prints out informational messages to show where it is in the process of instrumenting the target application. It installs the probe into the target application that, when it executes, sends the "Hello World" string back to the analysis tool. The callback routine for the probe, when activated, prints out the "Hello World" string.

Standard DPCL Programming Tasks

This section contains instructions for performing common tasks that all programs built on the DPCL system will need to perform. These instructions are divided into the following chapters.

- Chapter 4, “Performing status error checking” on page 73 provides general information on DPCL error checking.
- Chapter 5, “Initializing the analysis tool to use the DPCL system” on page 79 describes standard initialization tasks that your analysis tool must perform.
- Chapter 6, “Connecting to or starting the target application processes” on page 83 describes how your analysis tool can:
  - if the target application is already running, establish a connection to each process in the target application. The analysis tool must establish a connection to a target application process if it is to later dynamically insert probes into that process.
  - create one or more target application processes if the target application is not already running. When an analysis tool creates a process, the DPCL system also establishes the connection necessary for inserting probes into the process.
- Chapter 7, “Controlling execution of target application processes” on page 105 describes how an analysis tool can attach itself to a target application process in order to control execution of the process. It describes how an application can, once in this attached state, use DPCL function calls to suspend, resume, or kill one or more target application process.
- Chapter 8, “Creating probes” on page 115 describes how to create DPCL probes that can execute as part of a target application process. It describes how, once you have determined the logic you want the probe to perform, you can create the analysis tool code to build a probe expression to represent that logic. Since the programmatic capabilities of simple probe expressions are limited and not reusable among analysis tools, this chapter also describes how a probe expression can optionally call a function written in C and compiled into a probe module.
- Chapter 9, “Executing probes in target application processes” on page 143 describes the steps that your program must follow to insert a probe into one or more target application process for execution. The manner in which a probe is installed and executed within the target application process(es) distinguishes the probe as a particular probe type — either a point probe, a phase probe, or a one-shot probe. This chapter contains instructions for installing a probe as any one of these probe types.
- Chapter 10, “Creating data callback routines” on page 167 describes how an analysis tool can create a "data callback routine" to respond to data sent back to the analysis tool from probes executing within target application processes.
- Chapter 11, “Entering and exiting the DPCL main event loop” on page 169 describes how an analysis tool can enter an event loop to interface asynchronously with the DPCL system.

- Chapter 12, “Disconnecting from target application processes” on page 171 describes how an analysis tool can disconnect itself from processes it has finished examining, and exit from the DPCL main event loop.
- Chapter 13, “Compiling and linking the analysis tool and target application” on page 173 describes how to prelink your target application with the DPCL libraries and compile your analysis tool with the DPCL library and include files.

Instructions for other, more advanced and/or less-commonly performed, DPCL programming tasks (monitoring signals and file descriptors, overriding default system callbacks, and generating diagnostic logs) are contained in “Additional DPCL Programming Tasks” on page 175.

Chapter 4. Performing status error checking

Before we describe such standard DPCL programming tasks as initializing the analysis tool and connecting to a target application, a general discussion of DPCL error checking is in order. While many of the code examples in this book are for illustration only and therefore do not perform rigorous error checking, any real-world DPCL application will need to check the return status of each DPCL function it calls. As with any kind of error-checking situation, if the status returned by a DPCL function call indicates an error condition, the analysis tool should respond to the condition in some way. For example, if the returned status for a `Process::bconnect` call indicates that if was unable to connect to the process because of an invalid process ID, the analysis tool could respond by displaying the invalid process ID to the analysis tool user and prompt the user to supply a valid one. If error checking were not performed in this example, then subsequent DPCL calls made by the analysis tool (for, say, probe installation) would also fail since a connection to the process was never established.

As described in “What is the AisStatus class?” on page 58, status is returned by DPCL functions in the form of `AisStatus` objects which store status and severity codes. In some cases, the `AisStatus` object also contains data associated with the status. For example, in the situation described above in which the `AisStatus` object indicated that the process ID was invalid, the `AisStatus` object would also contain a data string to show the invalid process ID.

The following example code shows a single error checking routine that examines an `AisStatus` object and reports error conditions to the analysis-tool user. This simple function first calls the `AisStatus::status` function to see if it indicates the successful completion of whatever service was requested. If it does not, the function then calls the `AisStatus::status_name` function to get the name of the status code that was returned, and prints this name to standard output. Since the `AisStatus` object might also contain data associated with the error condition, the following example function also calls the `AisStatus::data_count` and `AisStatus::data_value` functions to get this information if it is available. The `AisStatus::data_count` function returns the number of data values associated with the `AisStatus` object and is used here to initialize a loop. The `AisStatus::data_value` function copies a particular data value associated with the `AisStatus` object into a specified buffer — in this case the buffer `buf`. For more information on the `AisStatus::status`, `AisStatus::data_count`, and `AisStatus::data_value` functions, refer to their AIX man pages or the *IBM Parallel Environment for AIX: DPCL Class Reference*.

```
#include <stdio.h>
#include <stdlib.h>
#include "dpcl.h"

#define BUFLEN 128

void check_status(AisStatus &sts, char *name) {
  char buf[BUFLEN];
  int i, dc;

  if (sts.status() != ASC_success) {
    printf("error from %s: %s\n", name, sts.status_name());
    dc = sts.data_count();
    for (i=0; i<dc; ++i) {
      sts.data_value(i, buf, BUFLEN);
      printf("  '%s'\n", buf);
    }
  }
}

main(int argc, char *argv[]) {
  AisStatus sts;
  char hostname[BUFLEN];
  int apppid;

  if (argc < 3) {
    printf("need two args\n");
    return 0;
  }

  strcpy(hostname, argv[1]);
  apppid = atoi(argv[2]);

  Process P(hostname, apppid);

  Ais_initialize();

  sts = P.bconnect();
  if (sts.status() != ASC_success) {
    check_status(sts, "bconnect");
  }

  return 0;
}
```

Asynchronous DPCL calls and `Application` class calls that perform operations for each process managed by the `Application` object introduce additional complexity to DPCL status error checking. Calls to asynchronous functions return immediately, but the returned `AisStatus` object will indicate only whether or not the DPCL service requested by the function was successfully sent, and **not** whether or not it succeeded. To determine the status of the operation when it actually succeeds or fails, the analysis tool can specify an acknowledgment callback routine when calling the asynchronous function. When the service requested by the asynchronous function either succeeds or fails, the DPCL system will trigger execution of the specified callback and will pass it the `AisStatus` object and a pointer to the `Process` object for which the requested service either succeeded or failed. When calling an asynchronous function, the analysis tool can also specify a tag value which will also be passed by the DPCL system to the acknowledgment callback.

The function prototype for a callback is:

```
void callback (
     GCBSysType sys,
     GCBTagType tag,
     GCBObjType obj,
     GCBMsgType msg);
```

Where: **is:**

`sys` a data structure defined as

```
struct GCBSysType {
     int msg_socket;
     int msg_type;
     int msg_size;
};
```

Where	Is
`msg_socket`	the socket or file descriptor from which the message was received.
`msg_type`	a message key or type value that represents the protocol or purpose behind the message. This is provided and used by the DPCL system in order to determine the callback routine to execute.
`msg_size`	the size of the message in bytes.

`tag` a value, large enough to contain a pointer, that is supplied by the analysis tool when the acknowledgment callback is identified. (In other words, when the asynchronous function is called.)

The `tag` parameter allows the analysis tool to use the acknowledgement callback routine for more than one purpose. For example, the analysis tool could create a general-purpose acknowledgment callback that could be used when calling any asynchronous function. The tag value in that case could contain information about which asynchronous function was called.

`obj` a pointer to the object that issued the request. In the case of the `Application` object, the requesting object will be the `Process` object managed by the `Application` object. The DPCL system supplies this information because, if an asynchronous `Application` function was called, the same acknowledgment callback will be triggered for each `Process` object managed by the `Application` object. By supplying a pointer to the invoking object, the DPCL system enables the callback to ascertain the `Process` for which status is being returned.

`msg` the AisStatus object.

Acknowledgment callback routines are particularly useful when error checking asynchronous `Application` class functions. Since these functions perform the requested operation on each `Process` object managed by the `Application` object, the DPCL system will trigger the callback routine for each `Process` object in turn. Since the operation may succeed on some processes while failing on others, the acknowledgment callback routine enables the analysis tool to check the status for each process.

The following example code calls the `Application::connect` function to connect to each process managed by the `Application` object. In this example, a generic error checking routine (`check_status`) is called to check the initial status returned by the `Application::connect` call, and is also called within the acknowledgment callback to check the status for each `Process`. Remember that the `AisStatus` object returned by the initial `Application::connect` call indicates only whether or not the request to

connect was successfully sent; it does not indicate whether or not any of the connect operations actually succeeded. As the connect request succeeds or fails for each particular `Process` object, the callback `connect_cb` is invoked and passed the `AisStatus` object, a pointer to the `Process` object, and, in this example, a pointer to the `Application` object (passed as the callback's `tag` value). Note that each time the callback is invoked, it increments the value of the integer `count`. The code uses this for comparison with the constant integer `NUM_PROCS` (defined outside the code shown) which indicates the number of `Process` objects managed by the `Application` object. When `count` is equal to `NUM_PROCS`, then all the processes managed by the `Application` object are connected, and the callback can continue with the next piece of work (in this case, calling the `Application::attach` function to attach to the processes).

```
//
// connect to the application
//
sts = A.connect(connect_cb, (GCBTagType) &A);
       check_status("A.connect(connect_cb, (GCBTagType) &A)", sts);


NUM_PROCS = A.get_count();
//
// callback to be called after the connect completes
//
void connect_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg)
{
         static int count = 0;
         count++;

    //
    // get the status from the msg parm
    //
    AisStatus *stsp = (AisStatus *)msg;
    check_status("connect_cb()", *stsp);

    //
    // get ptr to the process object from the obj parm
       //
    Process *p = (Process *)obj;
         printf("  %s: connect to pid: %d of Application\n", toolname, p->get_pid());

         if ( count >= NUM_PROCS) {
                  //
                  // get ptr to app object from the tag parm
                  //
      Application *a = (Application *)tag;
                  printf("  %s: connected to entire Application A\n", toolname);

      //
      // attach to the application
      //
      AisStatus sts = a->attach(attach_cb, tag);
                      check_status("a->attach(attach_cb, tag)", sts);
      }
}

//
// generic status-checking routine
// print status and exit when status is not success
//
void check_status (char *str, AisStatus sts)
{
    static char buf[256];

    if (sts.status() == ASC_success) {
        return;
    } else {
      printf("%s: Status for %s: %d, %s\n",  toolname, str, sts.status(), sts.status_name());
      for (int i=0; i<sts.data_count(); i++) {
```

```
            printf("  %s: data_value[%d]: %s\n",  toolname, i, sts.data_value(i, buf, sizeof(buf)));
         }
         }
      printf("  %s: exiting\n", toolname);
   fflush(stdout);
      exit(-1);
}
```

The preceding example illustrates how, when calling an asynchronous `Application` function, the analysis tool can use an acknowledgment callback to check the return status for each `Process` object managed by the `Application` object. When calling a blocking `Application` function, however, the analysis tool is not able to specify an acknowledgment callback. For blocking `Application` functions, acknowledgment callback routines are not necessary since the blocking calls do not return until the requested operation has either succeeded or failed for all of the `Process` objects managed by the `Application` object. To perform error status checking for a blocking `Application` function, the analysis tool should first examine the `AisStatus` object returned by the function — the `AisStatus` object will indicate whether or not the operation succeeded for all processes managed by the `Application` object. If it did not succeed for all processes, the analysis tool can call the `Application::status` function to determine the processes on which the operation failed. The `Application::status` function has one parameter — an index value that identifies the `Process` within the collection of those managed by the `Application` object. The `Application::get_count` function returns the number of `Process` objects managed by the `Application` object, and so can be used by the analysis tool to initialize a loop that calls the `Application::status` function to check each `Process` object's status in turn.

In the following example, a generic error checking routine (`check_status`) is called to check the status returned by the `Application::bconnect` call. If the returned status indicates that the connect operation failed on one or more processes, the same error checking routine is called to check the return status for each individual process.

```
#include <stdio.h>
#include <stdlib.h>
#include "dpcl.h"

#define BUFLEN 128

void check_status(AisStatus &sts, char *name) {
  char buf[BUFLEN];
  int i, dc;

  if (sts.status() != ASC_success) {
    printf("error from %s: %s\n", name, sts.status_name());
    dc = sts.data_count();
    for (i=0; i<dc; ++i) {
      sts.data_value(i, buf, BUFLEN);
      printf("  '%s'\n", buf);
    }
  }
}

main(int argc, char *argv[]) {
  AisStatus sts;
  char hostname[BUFLEN];
  char msg[BUFLEN];
  int i, pc;
    .
    .
    .
    // Code here to initialize application, create Process objects, organize
```

```
    // them under an Application object, and so on.
    .
    .
    .
  sts = A.bconnect();
  if (sts.status() != ASC_success) {
    check_status(sts, "A.bconnect");

    pc = A.get_count();
    for (i=0; i<pc; i++) {
      Process P = A.get_process(i);
      int tasknum = P.get_task();
      sprintf(msg, "bconnect on task %d", tasknum);
      sts = A.status(i);
      check_status(sts, msg);
    }

    return 0;
  }

  // rest of program
}
```

Chapter 5. Initializing the analysis tool to use the DPCL system

All DPCL analysis tools must, in order to use the DPCL system, perform the same initialization tasks. These tasks are the same whether the analysis tool is instrumenting a serial or parallel program. To initialize itself to use the DPCL system, the analysis tool must include the DPCL header files and initialize the DPCL system.

The following steps describe these three tasks in more detail. For sample code, see "Example: Initializing the analysis tool to use the DPCL system" on page 81.

Step 1: Include DPCL header file(s)

In order to have access to the functionality of the DPCL system, your analysis tool code must include the header file for each class, function, or function group that it will use. Your analysis tool can either include the header file `dpcl.h` (which includes all of the DPCL header files), or else it can include the individual header files that define just the classes, functions, or data types that it needs. To include all of the DPCL header files, use the preprocessor directive:

```
#include <dpcl.h>
```

If your analysis tool is a fairly simple application that does not use all the DPCL classes, you might, in order to minimize the size of your executable, want it to include only the header files for the classes, functions, and data types that it uses. The following table summarizes these individual header files. For more information on any of the classes or functions defined in these header files, refer to the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Table 14 (Page 1 of 2). DPCL header files

This Header File:	Explanation:
`AisGlobal.h`	Defines the global variables `Ais_msg_handle` and `Ais_send`. These global variables are used by probes when sending messages back to the analysis tool.
`AisHandler.h`	Contains the function prototypes (and a supporting data type definition) for the `AisHandler` function group. The functions in this function group are designed to enable the analysis tool to monitor signals and file descriptors through the DPCL system.
`AisInit.h`	Contains the function prototype for the `Ais_initialize` function. This function initializes the DPCL system.
`AisMainLoop.h`	Contains the function prototype for the `Ais_Main_Loop` and `Ais_end_main_loop` functions. The `Ais_Main_Loop` function puts execution in an endless event loop that enables the analysis tool to interface asynchronously with the DPCL system. The `Ais_end_main_loop` function breaks execution out of this event loop.
`AisStatus.h`	Defines the `AisStatus` class and contains the function prototypes (and supporting data type and constant definitions) for all functions in the class. Objects of the `AisStatus` class store status and severity codes returned by certain DPCL functions.
`Application.h`	Defines the `Application` class and contains the function prototypes for all functions in the class. Objects of this class represent related processes (`Process` class objects). Specifically, the target application can use objects of this class to represent a set of tasks in a parallel program.
`dpclExt.h`	Contains a prototype of the Ais_send function; used only with probe modules.
`GenCallBack.h`	Defines the `GenCallBack` data types for callback parameters.

Table 14 (Page 2 of 2). DPCL header files	
This Header File:	**Explanation:**
`InstPoint.h`	Defines the `InstPoint` class and contains the function prototypes (and supporting data type and constant definitions) for all functions in the class. Objects of this class represent instrumentation points in target application processes (where the analysis tool can install point probes).
`LogSystem.h`	Contains the function prototypes (and supporting data type and constant definitions) for the `LogSystem` function group. These functions enable the analysis tool to generate a diagnostic log.
`Phase.h`	Defines the `Phase` class and contains the function prototypes for all functions in the class. Objects of this class represent phase probes.
`PoeAppl.h`	Defines the `PoeAppl` class and contains the function prototypes for all functions in the class. Objects of this class (which is derived from the `Application` class) represent POE application processes. The `PoeAppl` class contains prototypes for convenience functions specific to manipulating POE applications (like starting a POE application in a stopped state, reading a POE configuration file, and supplying standard input text to the POE application).
`ProbeExp.h`	Defines the `ProbeExp` class and contains the function prototypes (and supporting data type and constant definitions) for all functions in the class. Objects of this class represent probe expressions to be executed within one or more target application processes.
`ProbeHandle.h`	Defines the `ProbeHandle` class and contains the function prototypes for all functions in the class. Objects of this class represent probe handles (references to probes that the analysis tool has installed in target application processes).
`ProbeModule.h`	Defines the `ProbeModule` class and contains the function prototypes for all functions in the class. Objects of this class represent probe modules to be loaded into one or more target application processes.
`ProbeType.h`	Defines the `ProbeType` class and contains the function prototypes (and supporting data type and constant definitions) for all functions in the class. Objects of this class represent a variable type (of a variable defined in a target application process or a probe).
`Process.h`	Defines the `Process` class and contains the function prototypes for all functions in the class. Objects of this class represent a target application process.
`SourceObj.h`	Defines the `SourceObj` class and contains the function prototypes (and supporting data type and constant definitions) for all functions in the class. Objects of this class represent part of the source code structure associated with a particular target application process.

Step 2: Initialize the DPCL system

All analysis tools built on the DPCL must initialize the DPCL system by calling the `Ais_initialize` function.

```
Ais_initialize();
```

The prototype for this function is contained in the header file `AisInit.h`.

Calling the `Ais_initialize` function enables the DPCL system to respond to unexpected system events such as a DPCL daemon exiting, or a target application process terminating. When such unexpected events occur, the DPCL system calls the appropriate system callback routine — either a default one provided by the DPCL system that simply prints out an error message, or one that you supply as part of the analysis tool code.

For general information about DPCL callbacks, refer to “What are DPCL callbacks?” on page 12. For information on overriding the default system callback routines with your own system callback routines, refer to Chapter 15, “Overriding default system callbacks” on page 181.

Example: Initializing the analysis tool to use the DPCL system

The following example code:

- includes the DPCL header files.
- calls the `Ais_initialize` function to initialize the DPCL system.
- calls the `Ais_main_loop` function to enter the DPCL main event loop. This enables the analysis tool to respond asynchronously to messages from DPCL communication daemons.

```
#include <dpcl.h>


main()
{
    Ais_initialize();

     .
     .    // Program Code Here
     .

}
```

Chapter 6. Connecting to or starting the target application processes

In order for your analysis tool to be able to dynamically insert probes into a target application process, it must have established a communication connection to that process. This "communication connection" is achieved though a socket connection from the analysis tool to the DPCL communication daemon, and shared-memory communication between the DPCL communication daemon and the target application process. There are two ways an analysis tool may establish such a connection to a process — either by calling a DPCL function to explicitly connect to the process, or by calling a DPCL function to actually create the process (which will implicitly establish the necessary connection). This chapter describes how an analysis tool can:

- if the target application is already running, establish a connection to each process in the target application. The analysis tool must establish a connection to a target application process if it is to later dynamically insert probes into that process. The first section of this chapter (“Connecting to the target application”) discusses the procedures for connecting to a serial application, a parallel (non-POE) application, and a POE application.
- create one or more target application processes if the target application is not already running. When an analysis tool creates a process, the DPCL system also establishes the connection necessary for inserting probes into the process. The second section of this chapter (“Starting the target application” on page 94) discusses the procedure for starting a serial application, a parallel (non-POE) application, and a POE application.

Connecting to the target application

Before an analysis tool can install and execute probes within a target application process, it must first establish a connection to that process. Connecting to a target application process on a particular host machine creates two DPCL daemon processes on that host — a *DPCL superdaemon process*, and a *DPCL communication daemon* process.

The DPCL superdaemon creates the DPCL communication daemon and is responsible for ensuring that only one communication daemon per user exists on the remote host. Although the DPCL superdaemon establishes the initial connection to the target application process, it passes this connection to the DPCL communication daemon. It is then the DPCL communication daemon that handles the communication between the analysis tool and the target application process. It is also the DPCL communication daemon that performs much of the actual work requested, via DPCL function calls, by the analysis tool. For more information on the DPCL daemon processes, refer to “What are the DPCL daemons?” on page 13.

The procedure for connecting to the target application differs depending on whether you are connecting to a serial application, a parallel (non-POE) application, or a parallel POE application.

If the target application is:	The analysis tool connects to it by:
A serial application	1. creating a `Process` object that identifies the target application process, and 2. calling the `Process::bconnect` or `Process::connect` function. For complete instructions, refer to “Connecting to a serial application” on page 84. For sample code, refer to “Example: Connecting to a serial application” on page 86.
A parallel application (non-POE)	1. Creating `Process` objects that identify the target application processes, 2. grouping the `Process` class objects under an `Application` object, and 3. calling the `Application::connect` or `Application::bconnect` function. For complete instructions, refer to “Connecting to a non-POE parallel application” on page 87. For sample code, refer to “Example: Connecting to a non-POE parallel application” on page 90.
A POE Application	1. Creating an empty `PoeAppl` object, 2. initializing the `PoeAppl` object to contain `Process` objects representing the POE target application processes, and 3. calling the `Application::connect` or `Application::bconnect` function. (The `PoeAppl` class is derived from the `Application` class, so these functions are available to the `PoeAppl` object.) For complete instructions, refer to “Connecting to a POE application” on page 90. For sample code, refer to “Example: Connecting to a POE application” on page 93.

Connecting to a serial application

To connect to a serial application, the analysis tool must:

1. instantiate a `Process` object that Identifies the target application process, and
2. call the `Process::bconnect` or `Process::connect` function.

The following steps describe these tasks in greater detail. For sample code, see “Example: Connecting to a serial application” on page 86.

Step 1: Instantiate a Process object that identifies the target application process

In order to connect to the target application process, the analysis tool must instantiate a `Process` object that represents the process. To do this, the analysis tool must:

1. Identify the host and process ID of the target application process, and
2. using this information, instantiate the `Process` object

The following substeps describe these tasks in greater detail.

Step 1a: Identify the host and process ID of the target application process: In order to instantiate a `Process` object that represents the target application process, the analysis tool must have some way of identifying the host running this process and the process ID on that host. The analysis tool could accomplish this in a number of ways; it could, for example, prompt the user to supply this information to standard input, or it could read a configuration file that contains the information.

Step 1b: Instantiate a Process object for the target application process: In order to connect to the target application process, the analysis tool must instantiate a `Process` object that represents the process. The `Process` class is defined in the header file `Process.h`. To assign the host name and process ID to a `Process` object, you can use a non-default constructor, a non-default constructor with a copy

constructor, or the default constructor with an assignment operator. Say the target application process is currently executing on a host machine whose IP host name is "myhost.xyz.edu", and the process ID is 12345.

Table 15. Instantiating a Process object

To instantiate a `Process` class object, the analysis tool can:	**For example:**
Use a default constructor and an assignment operator to assign values to the `Process` object.	`Process p;` `p = Process("myhost.xyz.edu", 12345);`
Use a non-default constructor to directly assign values to the `Process` object.	`Process p("myhost.xyz.edu", 12345);`
Use a non-default constructor and a copy constructor to assign values to the `Process` object.	`Process p = Process("myhost.xyz.edu", 12345);`

When you assign a host name and process ID to a `Process` object (as shown in the preceding table), the `Process` object is in an unconnected state. In other words, it is just an object allocated by the analysis tool process; it does not yet have a connection to the particular process indicated by its host and process ID values. In fact, the DPCL system does not at this point even know if the `Process` object's host and process ID values are valid. This unconnected state is represented by the enumeration constant `PRC_unconnected` of the `Process` class' `ConnectState` enumeration type. The analysis tool can query a `Process` object's state by calling the `Process::query_state` function.

For more information on the Process class constructor, refer to the *IBM Parallel Environment for AIX: DPCL Class Reference*

Step 2: Connect to the target application process

Once the analysis tool has a `Process` object that represents the target application, it can connect to it using either the blocking function `Process::bconnect`, or the asynchronous function `Process::connect`.

Table 16. Connecting to a target application process

To connect to a single target application process (`Process` object):	**Do this:**

Using the blocking function `bconnect`	`sts = P.bconnect();`
	`  check_status("P.bconnect()", sts);`
	`  printf("  %s: connected to pid:%d\n", toolname,  P.get_pid());`

Using the asynchronous function `connect`	`AisStatus sts = p->connect(connect_cb, GCBTagType(0));`
	`  check_status("p->connect(connect_cb, GCBTagType(0))", sts);`
	
	`//`
	`// callback to be called after the connect completes`
	`//`
	`void connect_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg)`
	`{`
	
	`  // code within callback routine`
	`  // can check the status of the`
	`  // operation and respond to its`
	`  // completion by, for example,`
	`  // continuing with other work`
	
	`}`

By using the Process::connect or Process::bconnect function, the analysis tool creates a connection to the Process indicated by the Process object's host and process ID values. When the DPCL system has established a connection between the analysis tool and a target application process, the analysis tool can instrument it with probes. This connected state is represented by the enumeration constant PRC_connected of the Process class' ConnectState enumeration type. The analysis tool can query a Process object's state by calling the Process::query_state function.

For more information on the Process::bconnect and Process::connect functions, refer to their AIX man pages, or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Example: Connecting to a serial application

The following example code:

- has the user identify the host name and process ID of the target application thorough command-line arguments.
- instantiates a Process object for the target application process using the host name and process ID supplied by the user.
- calls the bconnect function to create a DPCL daemon connection to the process represented by the Process object.

```
main(int argc, char *argv[])
{
    Process        P(argv[1], atoi(argv[2]));

    Ais_initialize();

    printf("Connecting to process %s on node %s\n", argv[2], argv[1]);

    AisStatus sts = P.bconnect();

    printf ("connect status  = %d\n", (int)sts);
    if ( (int)sts != 0) printf("%s\n", sts.status_name());
.
.
.
```

Connecting to a parallel application

The procedure for connecting to a parallel application differs depending on whether or not the application is designed to run in the Parallel Operating Environment.

If the target application is:	The analysis tool connects to it by:
A parallel application (non-POE)	1. instantiating Process objects that identify the target application processes 2. grouping the Process class objects under an Application object, and 3. calling the Application::connect or Application::bconnect function. For complete instructions, refer to “Connecting to a non-POE parallel application” on page 87. For sample code, refer to “Example: Connecting to a non-POE parallel application” on page 90.

If the target application is:	The analysis tool connects to it by:
A POE application	1. Creating an empty `PoeAppl` object, 2. initializing the `PoeAppl` object to contain `Process` objects representing the POE target application processes, and 3. calling the `Application::connect` or `Application::bconnect` function. (The `PoeAppl` class is derived from the `Application` class, so these functions are available to the `PoeAppl` object.) For complete instructions, refer to "Connecting to a POE application" on page 90. For sample code, refer to "Example: Connecting to a POE application" on page 93.

Connecting to a non-POE parallel application

To connect to a non-POE parallel application, the analysis tool must:

1. instantiate `Process` objects that identify the target application processes,
2. group the `Process` objects under an `Application` object, and
3. call the `Application::connect` or `Application::bconnect` function.

The following steps describe these tasks in greater detail. For sample code, see "Example: Connecting to a non-POE parallel application" on page 90.

Step 1: Instantiate Process objects that identify the target application processes: In order to connect to the target application processes in the parallel application, the analysis tool must instantiate `Process` objects that represent the processes. Later, the analysis tool will organize these `Process` objects under an `Application` object so that it may treat all the processes as if they were a single unit. To instantiate `Process` objects to represent the processes, the analysis tool must:

1. Identify the host and process ID of each target application process, and
2. instantiate a `Process` object for each process in the target application.

The following substeps describe these tasks in more detail.

Step 1a: Identify the host and process ID of each process in the target application: In order to instantiate `Process` objects that represent the target application processes, the analysis tool must have some way of identifying the host running, and the process ID for, each target application process. The analysis tool could accomplish this in a number of ways; it could, for example, prompt the user to supply this information to standard input, or it could read a configuration file that contains this information.

Step 1b: Instantiate a Process object for each process in the target application: In order to connect to the processes in the target application, the analysis tool must instantiate `Process` objects that represent the processes. The `Process` class is defined in the header file `Process.h`. To assign the host name and process ID to a `Process` object, the analysis tool can use a non-default constructor, a non-default constructor with a copy constructor, or the default constructor with an assignment operator. Say one of the target application processes is currently executing on a host machine whose IP host name is "myhost.xyz.edu", and the process ID is 12345.

Table 17. Instantiating Process objects for multiple target application processes	
To instantiate a Process class object, the analysis tool can:	**For example:**
Use a default constructor and an assignment operator to assign values to the Process object.	`Process p[128];` `p = Process("myhost.xyz.edu", 12345);`
Use a non-default constructor to directly assign values to the Process object.	`Process p[0]("myhost.xyz.edu", 12345);`
Use a non-default constructor and a copy constructor to assign values to the Process object.	`Process p[0] = Process("myhost.xyz.edu", 12345);`

When you assign a host name and process ID to a `Process` object (as shown in the preceding table), the `Process` object is in an unconnected state. In other words, it is just an object allocated by the analysis tool process; it does not yet have a connection to the particular process indicated by its host and process ID values. In fact, the DPCL system does not at this point even know if the `Process` object's host and process ID values are valid. This unconnected state is represented by the enumeration constant `PRC_unconnected` of the `Process` class' `ConnectState` enumeration type. The analysis tool can query a `Process` object's state by calling the `Process::query_state` function.

For more information on the `Process` class constructor, refer to the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Step 2: Group Process objects under an Application object: By grouping a set of `Process` objects under an `Application` object, the analysis tool is able to treat the set of `Process` objects as a single unit. In other words, it need only make a single call to an `Application` class member function to act upon all `Process` objects managed by that `Application` object. In this case, we are grouping all of the parallel target application's `Process` objects under the `Application` object. Note, however, that an `Application` object can be any grouping of `Process` objects that the analysis tool needs to manipulate as a single unit.

To group a set of `Process` objects under an `Application` object, the analysis tool must:

1. Instantiate an `Application` object, and
2. add the `Process` objects to the `Application` object.

The following substeps describe these tasks in greater detail.

Step 2a: Instantiate an Application object: In order to manipulate different processes in a parallel application, the analysis tool must group them into an `Application` object. The `Application` class is defined in the header file `Application.h`.

Table 18. Instantiating an Application object (for connecting to multiple processes)	
To instantiate a `Application` class object, the analysis tool can:	**For example:**
Use a default constructor:	`Application app1;`
Use a copy constructor:	`Application app2 = app1;`

For more information on the Application class constructor, refer to the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Step 2b: Add the Process objects to the Application object: The preceding step ("Step 2a: Instantiate an Application object" on page 88) showed how the analysis tool can use the `Application` class constructor to instantiate an empty `Application` object. In order to use this object to manipulate a set of processes as a single unit, the analysis tool now needs to add the `Process` objects to the `Application` object. It does this using the `Application::add_process` function. This function takes, as a parameter, the `Process` object to be added to the `Application` object. For example, the following line of code adds the `Process` object `p` to the `Application` object `app1`.

```
for (i=0; i<128; i++){

     app1.add_process(p[i]);

}
```

For more information on the `Application::add_process` function, refer its AIX man page, or its entry in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Step 3: Connect to the target application processes: At this point, the analysis tool has an `Application` object containing the `Process` objects that represent the target application processes. It can now connect to those processes using the blocking function `Application::bconnect`, or the asynchronous function `Application::connect`.

Table 19. Connecting to multiple target application processes

To connect to all processes in an application (all `Process` objects managed by an `Application` object):	**Do this:**

Using the asynchronous function `connect`	`AisStatus sts = a->connect(connect_cb, a);`
	`  check_status("a->connect(connect_cb, a)", sts);`
	
	`//`
	`// callback to be called after the connect completes`
	`//`
	`void connect_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg)`
	`{`
	
	`  // code within callback routine`
	`  // can check the status of the`
	`  // operation and respond to its`
	`  // completion by, for example,`
	`  // continuing with other work`
	
	`}`

Using the blocking function `bconnect`	`sts = A.bconnect();`
	`  check_status("A.bconnect()", sts);`
	`  printf("  %s: connected to Application A\n", toolname);`

By using the `Application::connect` or `Application::bconnect` function, the analysis tool creates connections to the processes indicated by the host and process ID values of the various `Process` objects managed by the `Application` object. These connections are achieved through a socket connection from the analysis tool to the DPCL communication daemon, and shared-memory communication between the DPCL communication daemon and the target

application process. When the DPCL system has established a connection to the processes represented by the Process objects managed by the Application object, the analysis tool can instrument those processes with probes. For each process, this connected state is represented by the enumeration constant PRC_connected of the Process class' ConnectState enumeration type. The analysis tool can query a Process object's state by calling the Process::query_state function.

For more information on the Application::connect and Application::bconnect functions, refer to their AIX man pages, or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Example: Connecting to a non-POE parallel application: The following example code:

- prompts the user to identify the host name and process ID for each of the target application processes.
- instantiates the Process objects to represent the target application processes using the host name and process ID information supplied by the user.
- instantiates an Application object and adds the Process objects to it.
- calls the Application::connect function to create a DPCL daemon connection to the target application processes.

```
printf("enter number of processes: ");
gets(s);
int np = atoi(s);
if (np <= 0) {
  printf("number of processes must be > 0\n");
  return 99;
}
else {
  A = new Application;
  for (int i=0; i<np; ++i) {
    printf("enter hostname: ");
    gets(hostname);
    printf("enter PID: ");
    gets(pidstr);
    int pid = atoi(pidstr);
    Process P(hostname, pid);
    A->add_process(&P);
  }
}

AisStatus sts = A->connect(connect_cb, A);
check_status("A->connect(connect_cb, A)", sts);

//
// callback to be called after the connect completes
//

void connect_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg) {
// code within callback routine can check the status of the operation and
// respond to its completion by, for example, continuing with other work
}
```

Connecting to a POE application

To connect to a Parallel Operating Environment (POE) application, the analysis tool must:

1. Instantiate an empty PoeAppl object,
2. initialize the PoeAppl object to contain Process objects representing the POE target application processes, and

3. call the `Application::connect` or `Application::bconnect` function. (The `PoeAppl` class is derived from the `Application` class, so these functions are available to the `PoeAppl` object.)

For more information on POE, refer to the *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment*.

Step 1: Instantiate a PoeAppl object: In order to connect to the target POE application, the analysis tool must instantiate a `PoeAppl` object that represents the POE application. Defined in the header file `PoeAppl.h`, the `PoeAppl` class is derived from the `Application` class, and provides additional convenience functions specific to POE applications. The following line of code illustrates how an analysis tool can instantiate a `PoeAppl` object using the default constructor.

```
PoeAppl app1;
```

The system response for the preceding line of code would be to create an empty `PoeAppl` object `app1`. For more information on the `PoeAppl` class constructor, refer to the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Step 2: Initialize the PoeAppl object to represent the POE target application: In order to connect to the POE target application processes, the analysis tool must now initialize the empty `PoeAppl` object so that it contains `Process` objects representing the POE target application processes. Doing so will enable the analysis tool to manipulate all the processes as though they were a single unit. Fortunately (as described in "Step 2b: Initialize the PoeAppl object"), the `PoeAppl` class provides convenience functions that enable the analysis tool to, in a single function call, instantiate `Process` objects and add them to the `PoeAppl` object. These convenience functions are the `PoeAppl::init_procs` function and its blocking equivalent `binit_procs`. In order to use either of these functions, however, the analysis tool must first identify the host and process ID of the POE home process as described in "Step 2a: Identify the host and process ID of the POE home process."

Step 2a: Identify the host and process ID of the POE home process: The next substep ("Step 2b: Initialize the PoeAppl object") describes how an analysis tool can initialize a `PoeAppl` object so that it contains `Process` objects representing all of the processes in the POE target application. In order to use the convenience functions that perform this initialization, however, the analysis tool must identify:

- the host name of the host where POE was invoked to start the POE application (called the POE home node).
- the process ID of the `poe` command invocation on the POE home node (called the POE home process).

The analysis tool could get this information in a number of ways; it could, for example, have the user supply this information as command line arguments.

For more information on POE, refer to the *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment*.

Step 2b: Initialize the PoeAppl object: In "Step 1: Instantiate a PoeAppl object," the analysis tool created an empty `PoeAppl` object. Now it can initialize this `PoeAppl` object so that it contains `Process` objects that represent the POE target application processes. This will enable the analysis tool to manipulate all of the POE application's processes as if they were a single entity. To initialize the `PoeAppl`

object, the analysis tool needs to call either the PoeAppl::init_procs function, or its blocking equivalent — the PoeAppl::binit_procs function.

Table 20. Initializing a PoeAppl object to contain Process class objects

To initialize the PoeAppl object to contain Process class objects representing the POE target application processes:	**Do this:**

Using the asynchronous function init_procs	`PoeAppl A;`
	`AisStatus sts = A.init_procs(hostname, pid, init_cb, NULL);`
	`check_status("A.init_procs()", sts);`
	
	`//`
	`// callback to be called after the connect completes`
	`//`
	`void init_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg)`
	`{`
	`  // code with callback routine can check the status of the operation and`
	`  // respond to its completion by, for example, continuing with other work`
	`}`

Using the blocking function binit_procs	`PoeAppl A;`
	`AisStatus sts = A.binit_procs(hostname, pid);`
	`check_status("A.binit_procs()", sts);`
	`printf("%s: Application has been initialized\n", toolname);`

Although the PoeAppl::init_procs and PoeAppl::binit_procs functions create the Process class objects (representing the processes in the target POE application) that the PoeAppl object will manage, note that these processes are, at this point, in an unconnected state. This unconnected state is represented by the enumeration constant PRC_unconnected of the Process class' ConnectState enumeration type. The analysis tool can query a Process object's state by calling the Process::query_state function.

For more information on the PoeAppl::init_procs and PoeAppl::binit_procs functions, refer to their AIX man pages or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*

Step 3: Connect to the POE target application's processes: At this point, the analysis tool should have a PoeAppl object that contains the Process objects that represent the POE target application processes. Since the PoeAppl class is derived from the Application class (and therefore has access to all of the Application class member functions), the analysis tool can now connect to the POE target application using either the blocking function Application::bconnect or the asynchronous function Application::connect.

Table 21. Connecting to multiple POE target application processes

To connect to all processes in an application (all `Process` objects managed by an `Application` object):	**Do this:**

Using the asynchronous function `connect`	`AisStatus sts = a->connect(connect_cb, a);`
	`  check_status("a->connect(connect_cb, a)", sts);`
	
	`//`
	`// callback to be called after the connect completes`
	`//`
	`void connect_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg)`
	`{`
	
	`  // code within callback routine`
	`  // can check the status of the`
	`  // operation and respond to its`
	`  // completion by, for example,`
	`  // continuing with other work`
	
	`}`

Using the blocking function `bconnect`	`sts = A.bconnect();`
	`  check_status("A.bconnect()", sts);`
	`  printf("  %s: connected to Application A\n", toolname);`

By using the `Application::connect` or `Application::bconnect` function, the analysis tool creates connections to the processes indicated by the host and process ID values of the various `Process` objects managed by the `Application` object.

When the DPCL system has established a connection to the processes represented by the Process objects managed by the Application object, the analysis tool can instrument those processes with probes. For each process, this connected state is represented by the enumeration constant `PRC_connected` of the `Process` class' `ConnectState` enumeration type. The analysis tool can query a `Process` object's state by calling the `Process::query_state` function.

For more information on the `Application::connect` and `Application::bconnect` functions, refer to their AIX man pages or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Example: Connecting to a POE application: The following example code:

- instantiates an empty `PoeAppl` object.
- prompts the user to identify the host name of the POE home node, and the process ID of the POE home process.
- initializes the `PoeAppl` object by calling the `PoeAppl::binit_procs` function, and passing it the POE home node and POE home process information obtained by the user.
- calls the `Application::connect` function to create a DPCL daemon connection to the target application process.

```
printf("enter hostname: ");
gets(hostname):
printf("enter pid: ");
gets(pidstr);
pid = atoi(pid);
```

```
PoeAppl A;
AisStatus sts = A.binit_procs(hostname, pid);
check_status("A.binit_procs()", sts);
printf("%s: Application has been initialized\n", toolname);

// connect to all the tasks in the POE application

sts = A.bconnect();
check_status("A.bconnect()", sts);
printf("%s: connected to application\n", toolname);
```

Starting the target application

The procedure for starting the target application process(es) depends on whether you are starting a serial application, a parallel (non-POE) application, or a parallel POE application.

If the target application is:	The analysis tool can start it by:
A serial application	1. Instantiating a `Process` class object to represent the process, 2. calling the `Process::create` or `Process::bcreate` function to create the process in a stopped state, and 3. calling the `Process::start` or `Process::bstart` function to start execution of the process. For complete instructions, refer to "Starting a serial application."
A parallel application (non-POE)	1. Instantiating `Process` class objects to represent the target application processes, 2. calling, for each `Process` object, the `Process::create` or `Process::bcreate` function to create the process in a stopped state, 3. grouping the `Process` objects under an `Application` object, and 4. calling the `Application::start` or `Application::bstart` function to start execution of the processes. For complete instructions, refer to "Starting a non-POE parallel application" on page 98.
A POE Application	1. Instantiating an empty `PoeAppl` object, 2. calling the `PoeAppl::create` or `PoeAppl::bcreate` function to create the POE application processes in a stopped state, and 3. calling the `Application::start` or `Application::bstart` function to start execution of the processes. (The `PoeAppl` class is derived from the `Application` class, so these functions are available to the `PoeAppl` object.) For complete instructions, refer to "Starting a POE application" on page 101.

Starting a serial application

To start a serial application, the analysis tool must:

1. Instantiate a `Process` class object to represent the process,
2. call the `Process::create` or `Process::bcreate` function to create the process stopped at its first executable instruction, and
3. call the `Process::start` or `Process::bstart` function to start execution of the process.

The following steps describe these tasks in greater detail.

Step 1: Instantiate a Process object

The `Process` class contains member functions for creating an AIX process stopped at its first executable instruction, and for starting a process. These are the `create` and `bcreate` functions (described in more detail in "Step 2: Create target application Process"), and the `start` and `bstart` functions (described in more detail in "Step 3: Start the target application process" on page 96). In order to have access to these functions, the analysis tool must first instantiate a `Process` object. To instantiate a `Process` object `p`, the analysis tool code would be:

```
Process p;
```

Instantiating a `Process` object using the default `Process` class constructor creates a `Process` object in a "pre-created" state. Although the `Process` object is instantiated, the actual AIX process it will represent is not yet created. This pre-created state is represented by the enumeration constant `PRC_pre_create` of the `Process` class' `ConnectState` enumeration type. The analysis tool can query a `Process` object's state by calling the `Process::query_state` function.

Step 2: Create target application Process

In order to create a target application process, the analysis tool must:

1. Identify the executable to run, and the host on which to run it
2. call the `Process::create` or `Process::bcreate` function

The following substeps describe these tasks in more detail.

Step 2a: Identify the executable to run and the host on which the process will run: The next substep describes how the analysis tool can call the `Process::create` or `Process::bcreate` function to create a process on a particular host. In order to use either of these functions, however, the analysis tool must supply the absolute path to the executable, and the host name or IP address of the host machine where the executable will run. The analysis tool can get this information in a number of ways; it could, for example, prompt the user to supply this information as input, or it could read a configuration file that contains the information.

Step 2b: Create the target application process: The analysis tool can create a target application process using the function `Process::create` or its blocking equivalent `Process::bcreate`. In addition to specifying the full path to the executable, and the host machine on which it will run, you can also:

- supply arguments that the analysis tool will pass to the executable,
- supply environment variables to affect the executable's run, and
- specify files for redirecting standard input, standard output, and standard error.

Table 22. Creating a target application process

To create the target application process:	Do this:

Using the asynchronous function `create`	`sts = P.create(hostname, progname, create_argv, envp,`
	`stdout_cb, (GCBTagType) 0, stderr_cb, (GCBTagType) 0,`
	`create_cb, (GCBTagType) 0);`
	`check_status("P.create()", sts);`
	
	`//`
	`// callback to be called after the create completes`
	`//`
	`void create_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg)`
	`{`
	
	`// code within callback routine`
	`// can check the status of the`
	`// operation and respond to its`
	`// completion by, for example,`
	`// continuing with other work`
	
	`}`

Using the blocking function `bcreate`	`sts = P.bcreate(hostname, progname, create_argv, envp,`
	`stdout_cb, (GCBTagType) 0, stderr_cb, (GCBTagType) 0);`
	`check_status("P.bcreate()", sts);`
	`printf("  %s: created pid:%d\n", toolname,  P.get_pid());`

Using the `Process::create` or `Process::bcreate` function, the analysis tool creates an AIX process where execution has been stopped at the process' first executable instruction. In this "created" state, the DPCL system has established a connection that enables the analysis tool to install probes into the process. In DPCL, this state is represented by the enumeration constant `PRC_created` of the `Process` class' `ConnectState` enumeration type. The analysis tool can query a `Process` object's state by calling the `Process::query_state` function.

For more information on the `Process::create` or `Process::bcreate` functions, refer to the AIX man pages for these functions, or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Step 3: Start the target application process

Calling the `Process::create` or `Process::bcreate` function, as described in the proceeding step, will create a process but will stop its execution at the first executable instruction. This enables the analysis tool code to install probes into the process (as described in Chapter 9, “Executing probes in target application processes” on page 143) prior to starting the process. To start a process, the analysis can use the function `Process::start` or its blocking equivalent `Process::bstart`.

Table 23. Starting a target application process

To start the target application process:	Do this:

Using the asynchronous function `start`	`AisStatus sts = p->start(start_cb, GCBTagType(0));`
	`  check_status("p->start(start_cb, GCBTagType(0))", sts);`
	
	`//`
	`// callback to be called after the start completes`
	`//`
	`void start_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg)`
	`{`
	
	`  // code within callback routine`
	`  // can check the status of the`
	`  // operation and respond to its`
	`  // completion by, for example,`
	`  // continuing with other work`
	
	`}`

Using the blocking function `bstart`	`sts = P.bstart();`
	`  check_status("P.bstart()", sts);`
	`  printf("  %s: started pid:%d\n", toolname,  P.get_pid());`

Starting a target application process using the `Process::start` or `Process::bstart` functions starts execution of the process (which will then have been stopped at the process' first executable instruction). The `Process` state is now represented by the enumeration constant `PRC_connected` of the `Process` class' `ConnectState` enumeration type. The analysis tool can query a `Process` object's state by calling the `Process::query_state` function.

Starting a parallel application

The procedure for starting parallel application process(es) depends on whether or not the application was designed to run in the Parallel Operating Environment.

If the target application is:	The analysis tool can start it by:
A parallel application (non-POE)	1. Creating `Process` class objects to represent the target application processes, 2. calling, for each `Process` object, the `Process::create` or `Process::bcreate` function to create the process in a stopped state, 3. grouping the `Process` objects under an `Application` object, and 4. calling the `Application::start` or `Application::bstart` function to start execution of the processes. For complete instructions, refer to “Starting a non-POE parallel application” on page 98.
A POE Application	1. Creating an empty `PoeAppl` object, 2. calling the `PoeAppl::create` or `PoeAppl::bcreate` function to create the POE application processes in a stopped state, and 3. calling the `Application::start` or `Application::bstart` function to start execution of the processes. (The `PoeAppl` class is derived from the `Application` class, so these functions are available to the `PoeAppl` object.) For complete instructions, refer to “Starting a POE application” on page 101.

Starting a non-POE parallel application

To start a non-POE parallel application, the analysis tool must:

1. Create `Process` class objects to represent the target application processes,
2. call, for each `Process` object, the `Process::create` or `Process::bcreate` function to create the process stopped at its first executable instruction,
3. group the `Process` objects under an `Application` object, and
4. call the `Application::start` or `Application::bstart` function to start execution of the processes.

The following steps describe these tasks in greater detail.

Step 1: Instantiate Process objects: The `Process` class contains member functions for creating a process stopped at its first executable instruction. These are the `create` and `bcreate` functions and are described in more detail in "Step 2: Create target application." In order to have access to these functions, the analysis tool must first instantiate a `Process` object for each of the processes in the parallel application. The following code creates a `Process` object.

```
Process p;
```

Instantiating `Process` objects using the default `Process` class constructor creates the processes in a "pre-created" state. While the `Process` objects are themselves created, the actual processes they will represent are not. This pre-created state is represented by the enumeration constant `PRC_pre_create` of the `Process` class' `ConnectState` enumeration type. The analysis tool can query a `Process` object's state by calling the `Process::query_state` function.

Step 2: Create target application: In order to create target application processes, the analysis tool must:

1. Identify the executable file(s) to run, and the host machine(s) on which the processes will run.
2. For each `Process` object, call the `Process::create` or `Process::bcreate` function.

The following substeps describe these tasks in more detail.

Step 2a: Identify the executable file(s) to run and the host(s) on which the processes will run: The next substep describes how the analysis tool can call the `Process::create` or `Process::bcreate` function to create a process on a particular host. To create the target application processes, the analysis tool code will need to have each of the `Process` objects (created in "Step 1: Instantiate Process objects") invoke either the `create` or `bcreate` function. In order to use either of these functions, however, the analysis tool must supply the relative or absolute path to the executable, and the host name or IP address of the host machine where the executable will run. The analysis tool can get this information in a number of ways; it could, for example, prompt the user to supply this information to standard input, or it could read a configuration file that contains this information.

Step 2b: Create the target application processes: To create the target application processes, the analysis tool code must have each of the `Process` objects (created in "Step 1: Instantiate Process objects") invoke either the `create` or `bcreate`

function. In addition to specifying the full path to the executable, and the host machine on which it will run, you can also:

- supply arguments that the analysis tool will pass to the executable,
- supply environment variables to affect the executable's run, and
- specify files for redirecting standard input, standard output, and standard error.

Table 24. Creating multiple target application processes

To create the target application process:	Do this:

Using the asynchronous function `create`	`Process P;`
	`Application A;`
	
	`for (i=0; i<NUM_PROCS; ++i) {`
	`  AisStatus sts = P.create(hostname[i], progname[i], create_argv, envp,`
	`  stdout_cb, (GCBTagType) 0, stderr_cb (CBGTagType) 0,`
	`  create_cb, (GCBTagType) i);`
	`  check_status("P.create()", sts);`
	`  A.add_process(&P);`
	`}`
	
	`//`
	`// callback to be called after the create completes`
	`//`
	`void create_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg)`
	`{`
	`  // code within the callback routine can check the status of the operation and`
	`  // respond to its completion by, for example, continuing with other work`
	`}`

Using the blocking function `bcreate`	`//`
	`// create some programs on the specified host and add them to app A`
	`//`
	`Application A;`
	`Process P;`
	
	`for (i=0; i<NUM_PROCS; i++) {`
	`  sts = P.bcreate(hostname[i], progname[i], create_argv, envp,`
	`  stdout_cb, (GCBTagType) i, stderr_cb, (GCBTagType) i);`
	`  check_status("P.bcreate(hostname, progname, create_argv, envp...)",`
	`  sts);`
	`  printf("  %s: created pid[%d]:%d\n", toolname,  i, P.get_pid());`
	`  sts = A.add_process(&P);`
	`  check_status("A.add_process(&P)", sts);`
	`  printf("  %s: added pid[%d]:%d to Application A\n", toolname,  i,`
	`  P.get_pid());`
	`}`

Using the `Process::create` or `Process::bcreate` function, the analysis tool creates an AIX process where execution has been stopped at the process' first executable instruction. In this "created" state, the DPCL system has established a connection that enables the analysis tool to install probes into the process. In DPCL, this state is represented by the enumeration constant `PRC_created` of the `Process` class' `ConnectState` enumeration type. The analysis tool can query a `Process` object's state by calling the `Process::query_state` function.

For more information on the parameters to the `Process::create` or `Process::bcreate` functions, refer to the AIX man pages for these functions, or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Step 3: Group Process objects under an Application object: By grouping a set of `Process` objects under an `Application` object, the analysis tool is able to treat the set of `Process` objects as a single unit. In other words, it need only make a single call to an `Application` class member function to act upon all `Process` objects managed by that `Application` object. In this case, we are grouping all of the parallel target application's `Process` objects under the `Application` object. Note, however, that an `Application` object can be any grouping of `Process` objects that the analysis tool needs to manipulate as a single unit.

To group a set of `Process` objects under an `Application` object, the analysis tool must:

1. Create an `Application` object, and
2. add the `Process` objects to the `Application` object.

The following substeps describe these tasks in greater detail.

Step 3a: Instantiate an Application object: In order to manipulate different processes in a parallel application, the analysis tool must group them into an `Application` object. The `Application` class is defined in the header file `Application.h`.

Table 25. Instantiating an Application object (for starting multiple target application processes)

To instantiate a `Application` class object, the analysis tool can:	**For example:**
Use a default constructor:	`Application app1;`
Use a copy constructor:	`Application app2 = app1;`

For more information on the `Application` class constructor, refer to the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Step 3b: Add the Process objects to the Application object: The preceding step (“Step 3a: Instantiate an Application object”) showed how the analysis tool can use the `Application` class constructor to instantiate an empty `Application` object. In order to use this object to manipulate a set of processes as a single unit, the analysis tool now needs to add the `Process` objects to the `Application` object. It does this using the `Application::add_process` function. This function takes, as a parameter, the `Process` object to be added to the `Application` object. For example, the following line of code adds the `Process` object `p` to the `Application` object `app1`.

```
app1.add_process(p[i]);
```

For more information on the `Application::add_process` function, refer to its AIX man page, or its entry in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Step 4: Start the target application processes: The `Process::create` and `Process::bcreate` functions create a process stopped at its first executable instruction. This means that all the processes created by the analysis tool in “Step 2: Create target application” on page 98 will be stopped until the analysis tool explicitly starts them. This enables the analysis tool code to install probes into any or all of the processes (as described in Chapter 9, “Executing probes in target application processes” on page 143) prior to starting them. To start the processes, the analysis tool can use the function `Application::start` or its blocking equivalent `Application::bstart`.

Table 26. Starting multiple target application processes

To start the target application processes:	Do this:

Using the asynchronous function `start`	`sts = A.start(start_cb, (GCBTagType), &A);`
	`  check_status("P.start(start_cb, (GCBTagType), &A)", sts);`
	`  printf("  %s: started from Application A\n", toolname);`
	
	`//`
	`// callback to be called after the start completes`
	`//`
	`void start_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg)`
	`{`
	
	`  // code within callback routine`
	`  // can check the status of the`
	`  // operation and respond to its`
	`  // completion by, for example,`
	`  // continuing with other work`
	
	`}`

Using the blocking function `bstart`	`sts = A.bstart();`
	`  check_status("P.bstart()", sts);`
	`  printf("  %s: started Application A\n", toolname);`

Starting a group of processes using the `Application::start` or `Application::bstart` functions starts execution of the processes (which will have been stopped at the process' first executable instruction). The state of each `Process` object is now represented by the enumeration constant `PRC_connected` of the `Process` class' `ConnectState` enumeration type. The analysis tool can query a `Process` object's state by calling the `Process::query_state` function.

For more information on the `Application::start` or `Application::bstart` functions, refer to their AIX man pages or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*

Starting a POE application

To start a Parallel Operating Environment (POE) application, the analysis tool must:

1. instantiate an empty `PoeAppl` object,
2. call the `PoeAppl::create` or `PoeAppl::bcreate` function to create the POE application processes stopped at their first executable instructions, and
3. call the `Application::start` or `Application::bstart` function to start execution of the processes. (The `PoeAppl` class is derived from the `Application` class, so these functions are available to the `PoeAppl` object.)

For more information on POE, refer to the *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment*.

Step 1: Instantiate a PoeAppl object: The `PoeAppl` class has member functions (`create` and `bcreate`) that will create all the processes in a given POE application. In order to have access to these functions, however, the analysis tool must instantiate a `PoeAppl` object to represent the POE application. The `PoeAppl` class is derived from the `Application` class, and provides additional convenience functions specific to POE applications. The following line of code illustrates how an analysis tool can instantiate a `PoeAppl` object using the default constructor.

```
PoeAppl app1;
```

The system response for the preceding line of code would be to instantiate a `PoeAppl` object `app1`. For more information on the `PoeAppl` class constructor, refer to the *IBM Parallel Environment for AIX: DPCL Class Reference*

Step 2: Create POE target application processes: The `PoeAppl` class provides two convenience properties that enable the analysis tool to, in a single function call, create all the processes in a POE application. These two convenience functions (`PoeAppl::create` and `PoeAppl::bcreate`) also:

- create `Process` objects to represent each of the POE target application processes, and adds these `Process` objects to the `PoeAppl` object so that the analysis tool can manipulate them as a single unit.
- establish a connection to the target application processes so that the analysis tool can insert instrumentation probes into, and remove them from, the processes.

In order to use the DPCL to run POE jobs, you must understand the POE execution environment as described in *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment*. Briefly, POE is an execution environment designed to hide, or at least smooth, the differences between running parallel programs as opposed to serial ones. In addition to a command-line interface similar to the familiar AIX command prompt, POE provides a number of environment variables that you can set to influence the operation of POE and the execution of programs within it. These environment variables control such things as how machine resources are allocated, and how I/O is handled between the machine the program is launched from (called the POE home node) and the parallel tasks. Most of the POE environment variables also have associated command-line flags that enable you to temporarily override the environment variable value when invoking POE and your parallel program.

Before you can use the DPCL to run a POE program, the program must already be set up to run under POE. For information on how to do this, refer to *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment*; the information on running parallel programs using POE appears in "Chapter 2: Executing Parallel Programs" of that manual. The remainder of this step refers to POE terms, environment variables, and command-line flags introduced in this other manual.

Using the function `PoeAppl::create`, or its blocking equivalent `PoeAppl::bcreate`, the analysis tool specifies the POE home node from which the POE program will be launched, and provides the absolute path to the **poe** executable command (/usr/bin/poe). Your analysis tool must supply the absolute path to the **poe** command because, although POE supports running an MPI program by either specifying **poe** *mpi_prog* or, simply, *mpi_prog*, DPCL does not. DPCL supports only the long form (**poe** *mpi_prog*), and so requires the absolute path to the **poe** command.

To control POE's execution environment, the `PoeAppl::create` and `PoeAppl::bcreate` functions also have parameters that enable the analysis tool to set POE environment variables and command line flags. The following table illustrates the procedure for creating processes in a POE application.

Table 27. Creating POE target application processes

To create the target application processes for a POE application:	Do this:

Using the asynchronous function `create`	`PoeAppl P;`
	`char *poename = "/usr/bin/poe";`
	`char *argvp[] = {poename, appname, NULL};`
	`AisStatus sts = P.create(hostname, poename, argvp, envp,`
	`  stdout_cb, (GCBTagType) 0, stderr_cb, (GCBTagType) 0,`
	`  create_cb, (GCBTagType) &P);`
	`check_status("P.create()", sts);`
	
	`//`
	`// callback to be called after the create completes`
	`//`
	
	`void create_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg)`
	`{`
	`  // code within callback routine can check the status of the operation and`
	`  // respond to its completion by, for example, continuing with other work`
	`}`

Using the blocking function `bcreate`	`PoeAppl P;`
	`char *poename = "/usr/bin/poe";`
	`char *argvp[] = {poename, appname, NULL};`
	`AisStatus sts = P.bcreate(hostname, poename, argvp, envp,`
	`  stdout_cb, (GCBTagType) 0, stderr_cb, (GCBTagType) 0);`
	`check_status("P.bcreate()", sts);`
	`printf("%s: application has been created\n", toolname);`

Using the `PoeAppl::create` or `PoeAppl::bcreate` function, the analysis tool creates the processes, but suspends them at the process' first executable instruction. In this "created" state, the DPCL system has established connections to the processes that enable the analysis tool to install probes into them. In DPCL, this state is represented by the enumeration constant `PRC_created` of the `Process` class' `ConnectState` enumeration type. The analysis tool can query a `Process` object's state by calling the `Process::query_state` function.

For more information on the `PoeAppl::create` and `PoeAppl::bcreate` functions, refer to their AIX man pages, or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Step 3: Start the target application processes: The `PoeAppl::create` and `PoeAppl::bcreate` functions create the POE target application processes stopped at their first executable instructions. In other words, they create the processes but stop each of them at their first executable instruction. In DPCL, this state is represented by the enumeration constant `PRC_created` of the `Process` object's `ConnectState` enumeration type. The analysis tool can query the state of the `Process` object by calling the `Process::query_state` function.

The processes created by the `PoeAppl::create` and `PoeAppl::bcreate` functions will remain stopped until the analysis tool explicitly starts them. This enables the analysis tool code to install probes into any or all of the processes (as described in Chapter 9, “Executing probes in target application processes” on page 143) prior to starting them.

Since the `PoeAppl` class is derived from the `Application` class (and therefore has access to all of the `Application` class member functions), the analysis tool can

now start the POE target application processes using the function `Applicaiton::start` or its blocking equivalent `Application::bstart`.

Table 28. Starting POE target application processes

To start the target application processes:	**Do this:**

Using the asynchronous function `start`	`sts = A.start(start_cb, (GCBTagType), &A);`
	`  check_status("P.start(start_cb, (GCBTagType), &A)", sts);`
	`  printf("  %s: disconnected from Application A\n", toolname);`
	
	`//`
	`// callback to be called after the start completes`
	`//`
	`void start_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg)`
	`{`
	
	`  // code within callback routine`
	`  // can check the status of the`
	`  // operation and respond to its`
	`  // completion by, for example,`
	`  // continuing with other work`
	
	`}`

Using the blocking function `bstart`	`sts = A.bstart();`
	`  check_status("P.bstart()", sts);`
	`  printf("  %s: started Application A\n", toolname);`

Starting a group of processes using the `Application::start` or `Application::bstart` functions starts execution of the processes (which will have been stopped at the process' first executable instruction). The state of each `Process` object is now represented by the enumeration constant `PRC_connected` of the `Process` class' `ConnectState` enumeration type. The analysis tool can query a `Process` object's state by calling the `Process::query_state` function.

For more information on the `Application::start` or `Application::bstart` functions, refer to their AIX man pages or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*

Chapter 7. Controlling execution of target application processes

There are two relationships that an analysis tool can establish with a target application process in order to affect its execution. The analysis tool can "connect" to the process, and, once connected, can also "attach" itself to the process. Connection (described in detail in Chapter 6, “Connecting to or starting the target application processes” on page 83) establishes a communication channel to the host where the process resides and creates the environment within the process that allows the analysis tool to insert and remove instrumentation probes. Inserting instrumentation probes enables the analysis tool to indirectly influence the execution of a target application process.

An analysis tool can connect to processes either by explicitly connecting (by calling the `connect` or `bconnect` function of either the `Process` or `Application` class), or by creating them (by calling the `create` or `bcreate` function of either the `Process` or `PoeAppl` class). The `connect` and `bconnect` functions place the process(es) in a connected state represented by the enumeration constant `PRC_connected` of the `Process` class' `ConnectState` enumeration type. The `create` and `bcreate` functions place the process(es) in a created state represented by the enumeration constant `PRC_created` of the `Process` class' `ConnectState` enumeration type. A subsequent call to the `start` or `bstart` function will move the process(es) into the `PRC_connected` state. In either case, however, a connection has been established that enables the analysis tool to instrument the target application process(es) with probes.

When a process is connected (in either the `PRC_connected` or `PRC_created` state), the analysis tool can attach to the process using the `attach` or `battach` function of either the `Process` or `Application` class. Attaching to the process enables the analysis tool to control execution of the process directly. This direct process control is exclusive — no other analysis tool will be able to attach to the process. The following sections describe how an analysis tool can:

- attach to one or more connected processes by calling the `Process::attach`, `Process::battach`, `Applcation::attach`, or `Application::battach` function.
- suspend process execution by calling the `Process::suspend`, `Process::bsuspend`, `Application::suspend`, or `Application::bsuspend` function.
- resume execution of one or more suspended processes by calling the `Process::resume`, `Process::bresume`, `Application::resume`, or `Application::bresume` function.
- detach itself from one or more target application processes whose execution it no longer needs to control. Since only one analysis tool can be attached to a process at a time, it is important that your analysis tool detach itself from any process it no longer needs to control. To do this, the analysis tool uses the `Process::detach`, `Process::bdetach`, `Application::detach`, or `Application::bdetach` function.

The analysis tool can also terminate a process. If the process is in the `PRC_created` state, the analysis tool does not need to attach to it in order to terminate its execution. If the analysis tool has, on the other hand, not created but merely connected to a remote process, it must attach to the process in order to terminate

it. The analysis terminates one or more processes by calling the `Process::destroy`, `Process::bdestroy`, `Application::destroy`, or `Application::bdestroy` function.

Attaching to the target application process(es)

Attaching to a target application process enables an analysis tool to directly control the execution of the process. This includes resuming and suspending execution of the process (as described in “Resuming execution of the target application process(es)” on page 107 and “Suspending execution of the target application process(es)” on page 108), as well as terminating the process (as described in “Terminating target application processes” on page 109). Attaching to an application automatically suspends its execution. This enables the analysis tool to install probes in the process (as described in Chapter 9, “Executing probes in target application processes” on page 143) before resuming its execution (as described in “Resuming execution of the target application process(es)” on page 107).

In order to attach itself to one or more target application processes, the analysis tool must be connected to the process(es) as described in “Connecting to the target application” on page 83. In other words, the `Process` object must be in either the `PRC_connected` or `PRC_created` state; the analysis tool can query a `Process` object's state by calling the `Process::query_state` function.

To attach to a single process, the analysis tool can use the asynchronous function `Process::attach` or its blocking equivalent `Process::battach`. To attach to processes on an application-wide basis (for all `Process` objects managed by an `Application` object), the analysis tool can use the functions `Application::attach` or `Application::battach`.

Table 29. Attaching to one or more target application processes

To attach:	**To a single process (`Process` object)**	**To multiple processes (all of the `Process` objects managed by an `Application` object)**

Using the asynchronous function `attach`	`AisStatus sts = p->attach(attach_cb,`
	`  (GCBTagType) 0);`
	`check_status("p->attach(attach_cb,`
	`  (GCBTagType) 0)", sts);`
	
	`//`
	`// callback to be called after the attach`
	`// completes`
	`//`
	`void attach_cb(GCBSysType sys,`
	`  GCBTagType tag, GCBObjType obj,`
	`  GCBMsgType msg)`
	`{`
	
	`  // code within callback routine`
	`  // can check the status of the`
	`  // operation and respond to its`
	`  // completion by, for example,`
	`  // continuing with other work`
	
	`}`
		`AisStatus sts = a->attach(attach_cb, tag);`
		`  check_status("a->attach(attac_cbh,`
		`  tag)", sts);`
		
		`//`
		`// callback to be called after the attach`
		`// completes for each Process managed by`
		`// this Application object`
		`//`
		`void attach_cb(GCBSysType sys,`
		`  GCBTagType tag, GCBObjType obj,`
		`  GCBMsgType msg)`
		`{`
		
		`  // code within callback routine`
		`  // can check the status of the`
		`  // operation and respond to its`
		`  // completion by, for example,`
		`  // continuing with other work`
		
		`}`

Using the blocking function `battach`	`sts = P.battach();`
	`check_status("P.battach()", sts);`
	`printf("  %s: attached to pid:%d\n",`
	`  toolname,  P.get_pid());`
		`sts = A.battach();`
		`check_status("A.battach()", sts);`
		`printf("  %s: attached to A\n",`
		`  toolname);`

Attaching to a process puts the process in an attached state represented by the enumeration constant `PRC_attached` of the `Process` class' `ConnectState` enumeration type. The analysis tool can query a `Process` object's state by calling the `Process::query_state` function.

Once attached to one or more target application processes, the analysis tool can:

- resume execution of the process(es) as described in "Resuming execution of the target application process(es)."
- suspend execution of the process(es) as described in "Suspending execution of the target application process(es)" on page 108.
- terminate execution of the process(es) as described in "Terminating target application processes" on page 109.

Note that only one analysis tool can be attached to a particular process at a time. For this reason, an analysis tool may want to "detach" itself from a particular process when it no longer needs to directly control its execution. To do this, see "Detaching from target application processes" on page 110.

For more information on the `Process::attach`, `Process::battach`, `Application::attach`, and `Application::battach` functions, refer to their AIX man pages or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Resuming execution of the target application process(es)

When the analysis tool attaches to one or more target application processes (as described in "Attaching to the target application process(es)" on page 106), execution of the process(es) is temporarily suspended. By suspending a process when the analysis tool attaches to it, the DPCL system enables the analysis tool to perform actions on it before resuming its execution. If the analysis tool were a debugger, for example, this would enable it to examine the processes state. Also, an analysis tool could install probes into the target application (as described in Chapter 9, "Executing probes in target application processes" on page 143) before resuming its execution. Once the analysis tool has resumed execution of a process, it can explicitly suspend its execution again as described in "Suspending execution of the target application process(es)" on page 108.

To resume execution of a single suspended process, the analysis tool can use the asynchronous function `Process::resume` or its blocking equivalent `Process::bresume`. To resume execution of suspended processes on an application-wide basis (for all `Process` objects managed by an `Application` object), the analysis tool can use the `Application::resume` or `Application::bresume` functions.

Table 30. Resuming execution of one or more suspended target application processes

To resume execution:	Of a single process (`Process` object)	Of multiple processes (all of the `Process` objects managed by an `Application` object)

Using the asynchronous function `resume`	`AisStatus sts = p->resume(resume_cb, GCBTagType(0));`
	`check_status("p->resume(resume_cb, GCBTagType(0))", sts);`
	
	`//`
	`// callback to be called after the resume`
	`// completes`
	`//`
	`void resume_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg)`
	`{`
	
	`// code within callback routine`
	`// can check the status of the`
	`// operation and respond to its`
	`// completion by, for example,`
	`// continuing with other work`
	
	`}`
		`AisStatus sts = A.resume( resume_cb, GCBTagType(0) );`
		`if ( sts.status() != ASC_success ) {`
		`printf("A.resume(), status: %s\n", sts.status_name());`
		`return sts;`
		`}`
		
		`//`
		`// this is called after each process`
		`// resume request has completed`
		`//`
		`void resume_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg)`
		`{`
		`AisStatus *stsp = (AisStatus *)msg;`
		`if ( stsp->status() != ASC_success )`
		`printf("resume_cb, status: %s\n", stsp->status_name());`
		`}`

Using the blocking function `bresume`	`sts = P.bresume();`
	`check_status("P.bresume()", sts);`
	`printf("  %s: resumed pid:%d\n", toolname,  P.get_pid());`
		`AisStatus sts=A.bresume();`
		`return sts;`

Note that the process must be attached in order for the analysis tool to resume its execution. In other words, the `Process` object must be in the `PRC_attached` state; the analysis tool can query a `Process` object's state by calling the `Process::query_state` function.

For more information on the `Process::resume`, `Process::bresume`, `Application::resume`, or `Application::bresume` functions, refer to their AIX man pages, or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*

Suspending execution of the target application process(es)

When the analysis tool attaches to one or more target application processes (as described in “Attaching to the target application process(es)” on page 106), the DPCL system automatically suspends execution of the process(es). This enables the analysis tool to install probes in the process(es) (as described in Chapter 9, “Executing probes in target application processes” on page 143). When desired, the analysis tool then resumes execution of the processes as described in “Resuming execution of the target application process(es)” on page 107. After execution of a process has been resumed, the analysis tool can suspend it again.

To suspend execution of a single process, the analysis tool can use the asynchronous function `Process::suspend` or its blocking equivalent `Process::bsuspend`. To suspend execution of processes on an application-wide basis (for all `Process` objects managed by an `Application` object), the analysis tool can use the `Application::suspend` or `Application::bsuspend` functions.

Table 31. Suspending execution of one or more target application processes

To suspend execution of:	A single process (`Process` object)	Multiple processes (all of the `Process` objects managed by an `Application` object)

Using the asynchronous function `suspend`	`AisStatus sts = p->suspend(suspend_cb, GCBTagType(0));`
	`check_status("p->suspend(suspend_cb, GCBTagType(0))", sts);`
	
	`//`
	`// callback to be called after the suspend`
	`// completes`
	`//`
	`void suspend_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg)`
	`{`
	`// code within callback routine`
	`// can check the status of the`
	`// operation and respond to its`
	`// completion by, for example,`
	`// continuing with other work`
	`}`
		`AisStatus sts = A.suspend( suspend_cb, GCBTagType(0) );`
		`if ( sts.status() != ASC_success )`
		`return sts;`
		
		`//`
		`// this is called after each process's`
		`// suspend request is done`
		`//`
		`void suspend_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg)`
		`{`
		`Process *pp = (Process *)obj;`
		`printf("pid %d suspended\n", pp->get_pid());`
		`}`

Using the blocking function `bsuspend`	`sts = P.bsuspend();`
	`check_status("P.bsuspend()", sts);`
	`printf("  %s: suspended pid:%d\n", toolname,  P.get_pid());`
		`AisStatus sts=A.bsuspend();`
		
		`//  if app status bad, show status by process`
		`if ( sts.status() != ASC_success )`
		`for (int i; i<A.get_count; i++)`
		`printf("status for Process[%d]: %s\n", i, status(i).status_name);`

Note that the process must be attached in order for the analysis tool to suspend its execution. In other words, the `Process` object must be in the `PRC_attached` state; the analysis tool can query a `Process` object's state by calling the `Process::query_state` function.

For more information on the `Process::suspend`, `Process::bsuspend`, `Application::suspend`, and `Application::bsuspend` functions, refer to their AIX man pages or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*

Terminating target application processes

When attached to a target application process (in other words, when the `Process` is in the `PRC_attached` state), the analysis tool can terminate execution of the process. The analysis tool can also terminate a process if it has created the process (in other words, when the `Process` is in the `PRC_created state`). If it has created the process, the analysis tool does not need to attach to it in order to terminate its execution. If the analysis tool has, on the other hand, merely connected to a running process, it must attach to the process in order to terminate it. To determine if a process is in one of the required states (`PRC_created` or `PRC_attached`), the analysis tool can query its state by calling the `Process::query_state` function. To terminate a single target application process, the analysis tool can use the asynchronous function `Process::destroy` or its blocking equivalent `Process::bdestroy`. To terminate processes on an application-wide basis (for all `Process` objects managed by an `Application` object), the analysis tool can use the functions `Application::destroy` or `Application::bdestroy`. When using any of

these functions, however, the analysis tool should exercise the same caution it would use when calling the AIX command **kill**. Killing selected processes in a message-passing parallel program, for example, could result in program deadlock among the remaining processes.

Table 32. Terminating one or more target application processes

To terminate:	**A single process (`Process` object)**	**Multiple processes (all of the `Process` objects managed by an `Application` object)**

Using the asynchronous function `destroy`	`AisStatus sts = p->destroy(destroy_cb, GCBTagType(0));`
	`check_status("p->destroy(destroy_cb, GCBTagType(0))", sts);`
	
	`//`
	`// callback to be called after the`
	`// destroy completes`
	`//`
	`void destroy_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg)`
	`{`
	
	`// code within callback routine`
	`// can check the status of the`
	`// operation and respond to its`
	`// completion by, for example,`
	`// continuing with other work`
	
	`}`
		`AisStatus sts = a->destroy(destroy_cb, tag);`
		`check_status("a->destroy(destroy_cb, tag)", sts);`
		
		`//`
		`// callback to be called after the destroy`
		`// completes for each Process managed by`
		`// the Application object`
		`//`
		`void destroy_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg)`
		`{`
		
		`// code within callback routine`
		`// can check the status of the`
		`// operation and respond to its`
		`// completion by, for example,`
		`// continuing with other work`
		
		`}`

Using the blocking function `bdestroy`	`sts = P.bdestroy();`
	`check_status("P.bdestroy()", sts);`
	`printf("  %s: destroyed pid:%d\n", toolname,  P.get_pid());`
		`sts = A.bdestroy();`
		`check_status("A.bdestroy()", sts);`
		`printf("  %s: destroyed Application A\n", toolname);`

Destroying a one or more processes as shown in the preceding table places the `Process` object(s) in a destroyed state. This state is represented by the enumeration constant `PRC_destroyed` of the `Process` class' `ConnectState` enumeration type. The analysis tool can query a `Process` object's state by calling the `Process::query_state` function.

For more information on the `Process::destroy`, `Process::bdestroy`, `Application::destroy`, or `Application::bdestroy` functions, refer to their AIX man pages or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Detaching from target application processes

An analysis tool does not need to explicitly detach itself from the target application process. When either the analysis tool process or the target application process terminates, the attachment will, or course, be broken. However, it is also important to note that only one analysis tool can be attached to a particular process at a time. For this reason, an analysis tool may want to detach itself from the process when it no longer needs to control execution of the process. This enables another analysis tool to attach to the process.

To detach itself from a single process, the analysis tool can use the asynchronous function `Process::detach` or its blocking equivalent `Process::bdetach`. To detach itself from processes on an application-wide basis (for all `Process` objects managed

by an Application object), the analysis tool can use the functions Application::detach or Application::bdetach. Naturally the process must be attached in order for the analysis tool to detach it. In other words, the Process object must be in the PRC_attached state; the analysis tool can query a Process object's state by calling the Process::query_state function.

Table 33. Detaching from one or more target application processes

To detach:	From a single process (Process object)	From multiple processes (all of the Process objects managed by an Application object)

Using the asynchronous function detach	AisStatus sts = p->detach(detach_cb,
	GCBTagType(0));
	check_status("p->detach(detach_cb,
	GCBTagType(0))", sts);
	
	//
	// callback to be called after the
	// detach completes
	//
	void detach_cb(GCBSysType sys,
	GCBTagType tag, GCBObjType obj,
	GCBMsgType msg)
	
	// code within callback routine
	// can check the status of the
	// operation and respond to its
	// completion by, for example,
	// continuing with other work
	
	}
		AisStatus sts = A.detach(detach_cb,
		GCBTagType(0));
		if (sts.status() != ASC_success)
		return sts.status_name();
		
		//
		// this is called after each detach request is
		// complete
		//
		void detach_cb(GCBSysType sys, GCBTagType tag,
		GCBObjType obj, GCBMsgType msg)
		{
		AisStatus *stsp = (AisStatus *) msg;
		Process *pp = (Process *)obj;
		if (stsp->status() == ASC_success)
		printf("pid %d detached\n",
		pp->get_pid());
		else
		printf("pid %d not detached\n",
		pp->get_pid());
		}

Using the blocking function bdetach	sts = P.bdetach();
	check_status("P.bdetach()", sts);
	printf(" %s: detached from pid:%d\n",
	toolname, P.get_pid());
		AisStatus sts=A.bdetach();
		
		if (sts.status() != ASC_success)
		return sts;

For more information on the Process::detach, Process::bdetach, Application::detach, and Application::bdetach functions, refer to their AIX man pages or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*

Note that detaching from a target application process does not remove the analysis tool's connection to the process. The analysis tool will still be able to install and execute probes within the process.

Example: Controlling execution of target application processes

The following example illustrates how a target application can control execution of target application processes. Specifically, it illustrates the use of the Application::bsuspend, Application::bresume, Application::bdetach, and Application::bdestroy calls. The user passes this analysis tool the host name and process IDs of the target application processes. The analysis tool:

1. connects and attaches to the processes. Attaching to the processes suspends their execution.
2. calls a program function to insert point probes into the processes. (See Chapter 9, “Executing probes in target application processes” on page 143 for more information on inserting point probes.)

3. resumes execution of the processes.

   Once execution of the processes resumes, the installed point probes collect information until the user presses a key on the keyboard. The target application then:

4. suspends execution of the target application processes,
5. removes the point probes, and
6. if the program was compiled with the flag -DLET_RUN, detaches and disconnects from the target application processes. If the program was not compiled with -DLET_RUN, the analysis tool terminates the target application processes.

```
//
// Example DPCL program for bsuspend, bresume, bdetach
// and bdestroy calls.
// Compile with -DLET_RUN if you want the application to
// continue running after instrumenting it.
//

#include <stdlib.h>
#include <stdio.h>
#include <dpcl.h>

AisStatus instrument(Application A);
void insert_probes();
void remove_probes();
void wait_for_keystroke();

main(int argc, char *argv[])
{
      Application A;
      char hostname[128];
      int pid_list[128];
      int pid_count;

      if (argc <3) {
            printf("Usage: %s <hostname> <pid> [pid...]\n", argv[0]);
            exit(-1);
      }

      // get the hostname and pids from argv
      //
      strcpy(hostname, argv[1]);
      pid_count=argc-2;
      for (int i=0; i<pid_count; i++) {
            pid_list[i] = atoi(argv[i+2]);
      }

      Ais_initialize();

      // Call the Process constructor then
      // add the processes to the application
      //
      for (i=0; i<pid_count; i++) {
            Process p(hostname, pid_list[i]);
            A.add_process(&p);
      }

      // call this routine to instrument the application
      //
      AisStatus sts = instrument(A);

      // very basic error checking
      //
      if ( sts.status() != ASC_success ) {

            // print overall status for application
            //
            printf("Failed with overall status: %s\n", sts.status_name());
```

```
            // print individual status for processes in application
            //
            for (int i=0; i<A.get_count(); i++)
                  printf("status for Process[%d]: %s\n", i, A.status(i).status_name());
            exit(-1);
      }
}

AisStatus instrument(Application A)
{
      //  connect to the application
      //
      AisStatus sts=A.bconnect();
      if ( sts.status() != ASC_success ) return sts;

      //  attach to the application
      //
      sts=A.battach();
      if ( sts.status() != ASC_success ) return sts;

      //    Instrument the application
      //
      insert_probes();

      //  resume the application
      //
      sts=A.bresume();
      if ( sts.status() != ASC_success ) return sts;

      //    collect instrumentation until there is user input
      //
      wait_for_keystroke();

      //  suspend the application
      //
      sts=A.bsuspend();
      if ( sts.status() != ASC_success ) return sts;

      //    remove the instrumentation
      //
      remove_probes();

      //  resume the application
      //
      sts=A.bresume();
      if ( sts.status() != ASC_success ) return sts;

#ifdef LET_RUN

      //  detach from the application
      //
      sts=A.bdetach();
      if ( sts.status() != ASC_success ) return sts;

      //  let the application go on normally
      //
      sts=A.bdisconnect();
      if ( sts.status() != ASC_success ) return sts;

#else

      //  let the application go on normally
      //
      sts=A.bdestroy();
      if ( sts.status() != ASC_success ) return sts;

#endif // LET_RUN

      return sts;
}
```

```
// dummy function defs
void insert_probes() {}
void remove_probes() {}
void wait_for_keystroke() {}
```

Chapter 8. Creating probes

The term *probe* refers to a software instrumentation code patch that your analysis tool can insert into one or more target application processes. Probes are created by the analysis tool, and therefore are custom designed to perform whatever work is required by the tool. For example, depending on the needs of the analysis tool, probes could be inserted into the target application to collect and report performance information (such as execution time), keep track of pass counts for test coverage tools, or report or modify the contents of variables for debuggers. For an overview of probes, refer to “What is a probe?” on page 15.

For the purposes of this book, a probe is defined as "a probe expression that may optionally call functions". A *probe expression* (described in more detail in “What is a probe expression?” on page 15) is a simple instruction or sequence of instructions that represents the executable code to be inserted into the target application. An analysis tool can create probe expressions to perform conditional control flow, integer arithmetic, and bitwise operations. For instructions detailing how an analysis tool can create probe expressions, refer to “Creating probe expressions.”

When more complicated logic is needed (such as iteration, recursion, and complex data structure manipulation), a probe expression can call functions. Specifically, a probe expression can call:

- a DPCL system-defined function for sending collected data back to the analysis tool.
- AIX functions (like `getrusage`, `times`, and `vtimes`).
- functions contained in the target application.
- a C function compiled into an object file called a "*probe module*". A probe module (described in more detail in “What is a probe module?” on page 17) is a compiled object file containing one or more functions written in C. Once an analysis tool loads a particular probe module into a target application, a probe expression is able to call any of the functions contained in the module. For instructions detailing how an analysis tool can do this, see “Creating and calling probe module functions” on page 136.

The sections that follow (“Creating probe expressions” and “Creating and calling probe module functions” on page 136) describe how to create probes that can execute as part of the target application process. For instructions on actually executing the probes, refer to Chapter 9, “Executing probes in target application processes” on page 143.

Creating probe expressions

A probe expression (described in more detail in “What is a probe expression?” on page 15) is a simple instruction or sequence of instructions that represent the executable code to be inserted into one or more target application processes. A probe expression is a type of data structure called an abstract syntax tree (a term we have borrowed from compiler technology). These data structures are called "abstract syntax trees" (as opposed to simply "syntax trees") because they are removed from the syntactic representation of the code. Compilers need to create abstract syntax trees from a program's source code as an intermediary stage

before manipulating and converting the data structure into executable instructions. Since the DPCL system also needs to create executable instructions (for insertion into one or more target application processes), it also needs to create these abstract syntax trees. To create a probe expression, you need to:

1. Determine the basic logic for the probe expression. This is not a task that the analysis tool code preforms — instead, it is a task that you, the creator of the analysis tool code, should perform. Creating the probe expression that will execute in one or more target application processes can be a "building block" task that involves first creating simple probe expressions and then combining and sequencing them into a single probe expression that represents the complete code patch to be inserted into the target application processes. To do this, it can sometimes be helpful to have a C or pseudocode version of the basic logic you want to build into the probe expression. This C or pseudocode version can then serve as a map of the logic you need to create using probe expressions.
2. Use `ProbeExp` objects to represent the various parts of the probe expression logic — the individual "nodes" of the abstract syntax tree — and then combine and sequence these `ProbeExp` objects into a final `ProbeExp` object that represents the complete code patch to be inserted into the target application process.

The following steps describe these tasks in greater detail. For sample code, see “Example: Creating probe expressions” on page 135.

Step 1: Determine basic logic for the probe expression

The next step (“Step 2: Build the probe expression” on page 117) describes how an analysis tool can create a probe expression that it can later execute within a target application process. The procedure for creating a probe expression can be a "building block" task in which smaller probe expressions are eventually combined and sequenced into the full probe expression.

For example, the analysis tool can create probe expressions representing constant or variable values, and then combine these into more complex probe expressions representing simple operations on the values, or function calls that pass the values as parameters to the function. The analysis tool could then take two of these more complex probe expressions and combine them into a single probe expression that represents a sequence of the two existing expressions. Then the analysis tool could join two such sequences into a longer sequence or combine them into a conditional statement. And so on, depending on the complexity of the probe logic, until the analysis tool has a single probe expression representing the full probe logic.

While the procedure for building a probe expression is the topic of the next step (“Step 2: Build the probe expression” on page 117), you should at this point determine the basic logic that the probe expression will perform. To adequately do this, you should understand the programmatic capabilities of probe expressions. A probe expression can represent:

- **a variable or constant data type value.** Probe expressions can represent values of type `int` or `string`.

  The analysis tool can combine these probe expressions representing values into probe expressions representing operations or function calls.

- **an operation.** Probe expressions can represent arithmetic operations, bitwise operations, relational operations, logical operations, assignment operations, unary address operations, and dereferencing operations.

  The analysis tool can combine these probe expressions representing operations into probe expressions representing more complex operations, a sequence of instructions, or parts of a conditional statement.

- **a sequence of instructions.** Probe expressions can represent a sequence of two existing probe expressions.

  The analysis tool can combine these probe expressions into longer sequences or parts of a conditional statement.

- **a function call.** Probe expressions can represent a call to:
  - the DPCL system-defined function `Ais_send` (for sending data back to the analysis tool).
  - a function contained in a probe module.
  - a function that is already present in the application.
  - an AIX system call.

  The analysis tool can combine these probe expressions into probe expressions representing a sequence of instructions or a conditional statement.

- **a conditional statement.** Probe expressions can represent a conditional statement. The test condition, the code to execute if the condition tests true, and the code to execute if the condition tests false are all existing probe expressions.

  The analysis tool can combine these probe expressions representing conditional statements into more complex conditional statements or a sequence of instructions.

Since the procedure for creating a full probe expression can be a "building block" task in which simple probe expressions are combined to form more complex ones, you might want to sketch out the logic in C code or pseudocode. This will give you a map for building the probe expression (as described next in “Step 2: Build the probe expression”).

Step 2: Build the probe expression

Once you have determined the basic logic you want to build into the probe expression, you can create the analysis tool code that builds the probe expression. The analysis tool must first create `ProbeExp` objects to represent the various parts of the probe expression logic — the individual "nodes" of the abstract syntax tree — and combine and sequence these `ProbeExp` objects into a final `ProbeExp` object that represents the complete code patch to be inserted into the target application process. To build a probe expression, the analysis tool can:

- create probe expressions to represent data values. These can be:
  - constant values.
  - variable values (including a variable in the target application).
  - the actual value of a particular function parameter in the target application.
- create a probe expression to represent operations.

- combine existing probe expressions into a new probe expression representing a sequence of instructions.
- combine existing probe expressions into a new probe expression representing conditional logic.
- create probe expressions to represent function calls. Such a probe expression can represent a call to:
  - the DPCL system-defined function `Ais_send` (for sending data back to the analysis tool).
  - a function contained in a probe module.
  - a function that is already present in the application.
  - an AIX system call.

The following substeps describe these tasks in greater detail.

Step 2a: Create probe expressions to represent temporary or persistent data

Like most programming vehicles, probes require scratch space for both temporary and persistent data. The DPCL system automatically allocates probe temporary data each time a probe expression executes, and deallocates the data when the probe expression completes. Probe persistent data, on the other hand, must be explicitly allocated by the analysis tool code. "Creating probe expressions to represent probe temporary data" describes how an analysis tool can use `ProbeExp` class constructors to create probe expressions representing temporary data. "Creating probe expressions to represent persistent data" on page 120 describes how an analysis tool can explicitly allocate probe persistent data.

Creating probe expressions to represent probe temporary data: The DPCL system automatically allocates probe temporary data each time the probe expression is executed, and deallocates the data when the probe expression completes. Unlike probe persistent data (described next in "Creating probe expressions to represent persistent data" on page 120), the analysis tool does not need to explicitly allocate memory for the data. To create a probe expression to represent temporary data, the analysis tool uses the `ProbeExp` class constructor to specify the data type and initial value of the data.

Table 34. Creating probe expressions to represent temporary data

For example, to create a probe expression `pe` that represents this data type:	**And has an initial value of:**	**The analysis tool code would be:**
`int`	16	`ProbeExp pe32 = ProbeExp(16);`
`string`	"jason"	`ProbeExp pestr = ProbeExp("jason");`

For more information on the `ProbeExp` class constructor, refer to the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Creating probe expressions to represent temporary data In the target application: Using the `SourceObj::reference` function, the analysis tool is able to create a probe expression that references a global or static data variable in the target application. To do this, the analysis tool must navigate the target application's source code structure to identify the variable of interest. (If you are

unfamiliar with SourceObj objects and the concept of source hierarchies, you may want to refer to “What is the SourceObj class?” on page 48 before reading the following example code.

The following example code:

1. Calls the Process::get_program_object function to return the top-level source object (SourceObj object) associated with a process.
2. Identifies the program module that contains the variable. To do this, it uses the SourceObj::child_count function to initialize a for loop and then, within the for loop, uses the SourceObj::child and SourceObj::module_name functions to identify the target module.
3. Calls the SourceObj::bexpand function to expand the module. This enables the analysis tool to navigate further down the source hierarchy to examine additional program structure (including global data variables). An analysis tool could also expand a module using the asynchronous SourceObj::expand function.
4. Identifies the variable. To do this, the analysis tool code again uses the SourceObj::child_count function to initialize a for loop, and then, within the for loop, uses the SourceObj::child and SourceObj::get_variable_name functions to identify the target variable.
5. Calls the SourceObj::reference function to create a probe expression that represents the variable.

```
#include <dpcl.h>
#include <libgen.h>             // for basename()
// find the variable variable_name defined in the module module_name
// for a given Process p
// return: SourceObj contains the variable.
//         or src_type()==SOT_unknown_type if the variable not found
SourceObj find_variable(Process p,
                        char * module_name,
                        char * variable_name)
{
    // get the source object associated with a process
    SourceObj progobj = p.get_program_object();
    SourceObj ret;              // return variable
    SourceObj mod;              // module object

    // identifies the program module that contains the variable
    // we preallocate a buffer to hold the module name
    int mod_name_length=1024;
    char * mod_name = new char [mod_name_length];
    int mod_index;
    for(mod_index = 0;mod_index < progobj.child_count();mod_index++) {
      mod = progobj.child(mod_index);

      // enlarge the name buffer if necessary.
      if (mod.module_name_length() >= mod_name_length) {
        // double the length
        while(mod_name_length < mod.module_name_length()) mod_name_length*=2;
        delete [] mod_name;
        mod_name = new char [mod_name_length];
      } /* endif */

      // get the module name
      mod.module_name(mod_name,mod_name_length);

      // depending on how the target application was compiled, the module
      // name may contain path information; for purposes of illustration, we compare
      // against the basename only.
      if (0 == strcmp(basename(mod_name),module_name)) {        // found
        break;
```

```
      } /* endif */
    }
    delete [] mod_name;
    if(mod_index == progobj.child_count())      // module not found
      return ret;
    // now we need to expand the module's SourceObj if necessary.
    // since only one module needs to expand, bexpand() will be easier
    if(mod.child_count() == 0) {                //
      AisStatus sts = mod.bexpand(p);
      if(sts.status()!=ASC_success)             // expand failed
        return ret;
    }

    // preallocate the variable buffer to hold the variable
    int var_name_length=1024;
    char * var_name = new char [var_name_length];
    SourceObj var;
    for (int i=0; i<mod.child_count(); i++) {
      var = mod.child(i);
      if(var.src_type()!=SOT_data) continue;
      if (var.get_variable_name_length() >= var_name_length) {
        while(var_name_length < var.get_variable_name_length())
          var_name_length*=2;
        delete [] var_name;
        var_name = new char [var_name_length];
      } /* endif */
      var.get_variable_name(var_name,var_name_length);
      if(0 == strcmp(var_name,variable_name)) { // found the variable
        ret = var;
        break;
      }
    } /* endfor */
    delete [] var_name;
    return ret;
}

main() {
    Process p;
    // ...
    SourceObj var = find_variable(p,"hello.c","var");
    if(var.src_type() == SOT_unknown_type) {
      // error report
    }
    ProbeExp pevar = var.reference();
}
```

Creating probe expressions to represent persistent data: Unlike temporary data, probe persistent data must be explicitly allocated and deallocated by the analysis tool code. If your analysis tool code requires probe data to be persistent from one invocation of the probe to the next, it must allocate memory within the target application process(es). To create a probe expression to allocate persistent data in a single process, the analysis tool can use the `Process::alloc_mem` or its blocking equivalent `Process::balloc_mem`. To create a probe expression to allocate persistent data on an application-wide basis (for all `Process` objects managed by the `Application` object), the analysis tool can use the functions `Application::alloc_mem` or `Application::balloc_mem`.

Table 35. Allocating memory in one or more target application processes

To create a probe expression to allocate persistent data in:	**A single process (`Process` object)**	**Multiple processes (all of the `Process` objects managed by an `Application` object)**

Using the asynchronous function `alloc_mem`	`int initval=0;`
	`ProbeExp pe = P.alloc_mem(int32_type(),`
	`  &initval, malloc_cb,`
	`  (GCBTagType) 0, sts);`
	`if (sts.status() != ASC_success)`
	`  printf("alloc_mem error: %s\n",`
	`  sts.status_name());`
	`else`
	`  Ais_main_loop();`
	
	`void malloc_cb(GCBSysType s, GCBTagType t,`
	`  GCBObjType o, GCBMsgType m) {`
	`  AisStatus *stsp = (AisStatus *) m;`
	`  if (stsp->status() != ASC_success)`
	`  printf("malloc error: %s\n",`
	`  stsp->status_name());`
	
	`}`
		`int count = A.get_count();`
		`int initval=0;`
		`ProbeExp pe = A.alloc_mem(int32_type(),`
		`  &initval,app_cb,&count,sts);`
		`if(sts.status() != ASC_success){`
		`  printf("alloc_mem: Highest error is %s\n",`
		`  sts.status_name());`
		`}`
		`for(i = 0; i < A.get_count(); i++) {`
		`  if(A.status(i).status()!=ASC_success){`
		`  count -= 1;`
		`  }`
		`}`
		`if(count) Ais_main_loop();`
		
		`//...`
		
		`void app_cb(GCBSysType s, GCBTagType t,`
		`  GCBObjType o, GCBMsgType m) {`
		`  int* count = (int *) t;`
		`  AisStatus * stsp = (AisStatus *) m;`
		`  if(stsp->status() != ASC_success) {`
		`  // print error`
		`  }`
		`  *count -= 1;`
		`  if(*count == 0) Ais_end_main_loop();`
		`}`

Using the blocking function `balloc_mem`	`int initval=0;`
	`ProbeExp pe = P.balloc_mem(int32_type(),`
	`  &initval, sts);`
	`  if (sts.status() != ASC_success)`
	`  printf("balloc_mem error: %s\n",`
	`  sts.status_name());`
		`int initval=0;`
		`ProbeExp pe = A.balloc_mem(int32_type(),`
		`  &initval,sts);`
		`if(sts.status() != ASC_success) {`
		`  printf("balloc_mem: %s\n",sts.status_name());`
		`  for(int i=0;i<A.get_count();i++) {`
		`  if(A.status(i).status()!=ASC_success){`
		`  // print error`
		`  }`
		`  }`
		`}`

Keep in mind that, as with traditional programming, if your code allocates memory, it must later free that memory or it will create a memory leak. To create a probe expression that frees memory in a single process, the analysis tool can use the `Process::free_mem` function or its blocking equivalent `Process::bfree_mem`. To create a probe expression that frees memory on an application-wide basis (in all `Process` objects managed by an `Application` object), the analysis tool can use the functions `Application::free_mem` and `Application::bfree_mem`.

Table 36. Deallocating memory in one or more target application processes

To create a probe expression to deallocate persistent data in:	**A single process (`Process` object)**	**Multiple processes (all of the `Process` objects managed by an `Application` object)**

Using the asynchronous function `free_mem`	`int count = 1;`
	`sts = P.free_mem(pexp,proc_cb,&count);`
	`if(sts.status() != ASC_success){`
	`  printf("free_mem error %s\n",`
	`  sts.status_name());`
	`  count -= 1;`
	`}`
	`if(count) Ais_main_loop();`
	
	`// ...`
	
	`void proc_cb(GCBSysType s, GCBTagType t,`
	`  GCBObjType o, GCBMsgType m) {`
	`  int* count = (int *) t;`
	`  AisStatus * stsp = (AisStatus *) m;`
	`  if(stsp->status() != ASC_success) {`
	`  // print error`
	`  }`
	`  *count -= 1;`
	`  if(*count == 0) Ais_end_main_loop();`
	`}`
		`int count = A.get_count();`
		`AisStatus sts = A.free_mem(pexp,app_cb,&count);`
		`if(sts.status() != ASC_success){`
		`  printf("free_mem: error is %s\n",`
		`  sts.status_name());`
		`}`
		`for(i = 0; i < A.get_count(); i++) {`
		`  if(A.status(i).status()!=ASC_success){`
		`  count -= 1;`
		`  }`
		`}`
		`if(count) Ais_main_loop();`
		
		`// ...`
		
		`void app_cb(GCBSysType s, GCBTagType t,`
		`  GCBObjType o, GCBMsgType m) {`
		`  int* count = (int *) t;`
		`  AisStatus * stsp = (AisStatus *) m;`
		`  if(stsp->status() != ASC_success) {`
		`  // print error`
		`  }`
		`  *count -= 1;`
		`  if(*count == 0) Ais_end_main_loop();`
		`}`

Using the blocking function `bfree_mem`	`AisStatus sts = P.bfree_mem(pexp);`
	`if(sts.status() != ASC_success) {`
	`  printf("bfree_mem: %s\n",sts.status_name());`
	`}`
		`AisStatus sts = A.bfree_mem(pexp);`
		`if(sts.status() != ASC_success) {`
		`  printf("bfree_mem: %s\n",sts.status_name());`
		`  for(int i=0;i<A.get_count();i++) {`
		`  if(A.status(i).status()!=ASC_success){`
		`  // print error`
		`  }`
		`  }`
		`}`

For more information on the functions described in the tables above, refer to their AIX man pages or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Creating probe expressions to represent actual function parameter values in the target application: Using the `ProbeType::get_actual` function, the analysis tool can create a probe expression that represents the actual value of a particular function parameter in the target application. When executed within the target application, the probe expression will determine the actual value of the parameter at that point in time. If the analysis tool were a debugger, for example, it might want to display this information to the user. The probe expression representing the actual parameter value can then be combined into a more complex probe expression that determines the actual parameter value and sends that information back to the analysis tool which displays it to the user.

In order to use the `ProbeType::get_actual` function, the analysis tool must first create a `ProbeType` object (using the `ProbeType::function_type` function) that represents the prototype (or type signature) of the function. The DPCL system

needs to know this type information in order to correctly identify an actual parameter value for the function; if the ProbeType object created using the ProbeType::function_type function does not match the function prototype in the target application, the ProbeType::get_actual function will not return the correct information.

The following example code calls the ProbeType::function_type function to create a ProbeType object that represents a function prototype. This example then uses the ProbeType::get_actual function to create a probe expression that represents the actual parameter value of one of the function's parameters.

```
// create a prototype for a function that has an two arguments: and int
// and a pointer to int; and has no return value

ProbeType pr_args[2];
pr_args[0] = int32_type();
pr_args[1] = pointer_type(int32_type());
ProbeType proto = function_type(void_type(), 2, pr_args);

// create a probe expression representing the first parameter in a call
// to a function having this prototype

ProbeExp ap = proto.get_actual(0);
```

"Creating a probe expression to represent a call to the Ais_send function" on page 130 expands on this example to show how an analysis tool can create a probe to send the actual parameter information back to the analysis tool.

For more information on the ProbeType object, refer to "What is the ProbeType class?" on page 55. For more information on the ProbeType::function_type or ProbeType::get_actual function, refer to the function's AIX man page or the *IBM Parallel Environment for AIX: DPCL Class Reference*

Step 2b: Create probe expressions to represent operations

Writing the analysis tool code that creates probe expressions representing operations is a fairly straightforward task. This is because the ProbeExp class has overloaded common operators so that expressions written within the context of the class do not execute locally, but instead call member functions that create the probe expression. For example, the following line of code:

```
ProbeExp pe3 = pe1 + pe2;
```

creates the probe expression pe3 which represents the addition of probe expressions pe1 and pe2. With the exception of the simple assignment operator (=), and the unary address operator (&), the ProbeExp class has overloaded all of the C++ operators in this way. This includes arithmetic operators (+, -, *, /, %), bitwise operators (<<, >>, ~, ^, &, |), relational operators (<, >, ==, !=, <=, >=), logical operators (&&, ||, !), assignment operators (+=, -=, *=, /=, %=, <<=, >>=, ^=, &=, |=), and dereference operators (*, []). Whenever any of these operators are used within the context of a probe expression, they create a probe expression that represents the operation. The operands can either be objects in memory, or probe expressions that evaluate to values. This means that a probe expression representing an operation could itself be used as an operand when creating another probe expression.

As already mentioned, the two operators that are not overloaded by the ProbeExp class are the simple assignment operator (=) and the unary address operator (&). So these two operators retain their original semantics — "pe2 = pe1" performs the

assignment of probe expression `pe1` into probe expression `pe2` within the analysis tool, and "`&pe1`" takes the address of the probe expression `pe1` within the analysis tool. Instead of overloading the `=` and `&` operators, the `ProbeExp` class instead provides the member functions `assign` and `address`. For example, the following line of code:

```
ProbeExp pea = pe1.assign(pe2);
```

creates a probe expression that assigns the value computed by the probe expression `pe2` into the storage location indicated by `pe1`, and this next line of code:

```
ProbeExp peb = pe.address();
```

creates a probe expression `peb` that represents the address of the probe expression `pe`.

The following tables outline the various operations that can be represented in probe expressions. Be aware that not all of the operator functions summarized in these tables are compatible with all operator types. For more information on any of the functions listed in these tables (including information on which types are valid for each overloaded operator function), refer to the*IBM Parallel Environment for AIX: DPCL Class Reference*.

Table 37. Creating probe expressions to represent arithmetic operations

Operation:	**Operator:**	**For example, this code:**	**Creates a probe expression** `exp` **that represents the:**
Addition	+	`ProbeExp exp = lhs + rhs;`	addition of `lhs` and `rhs`.
Subtraction	-	`ProbeExp exp = lhs - rhs;`	subtraction of `rhs` from `lhs`.
Multiplication	*	`ProbeExp exp = lhs * rhs;`	multiplication of `lhs` and `rhs`.
Division	/	`ProbeExp exp = lhs / rhs;`	division of `lhs` by `rhs`.
Modulus	%	`ProbeExp exp = lhs % rhs;`	integer division of `lhs` by `rhs` in which the remainder rather than the dividend is returned.

Table 38. Creating probe expressions to represent bitwise operations

Operation:	**Operator:**	**For example, this code:**	**Creates a probe expression** `exp` **that represents the:**
Bitwise AND	&	`ProbeExp exp = lhs & rhs;`	bitwise AND of `lhs` and `rhs`.
Bitwise Inclusive OR	|	`ProbeExp exp = lhs | rhs;`	bitwise inclusive OR of `lhs and rhs`.
Bitwise Exclusive OR	^	`ProbeExp exp = lhs ^ rhs;`	bitwise exclusive OR of `lhs` and `rhs`.
Bitwise Left Shift	<<	`ProbeExp exp = lhs << rhs;`	bitwise left shift of `lhs` by `rhs` places.
Bitwise Right Shift	>>	`ProbeExp exp = lhs >> rhs;`	bitwise right shift of `lhs` by `rhs` places.
Complement (bitwise inversion)	~	`ProbeExp exp = ~ rhs;`	bitwise inversion of `rhs`.

Table 39. Creating probe expressions to represent logical operations

Operation:	Operator:	For example, this code:	Creates a probe expression `exp` that represents the:
Logical AND	&&	`ProbeExp exp = lhs && rhs;`	logical AND of `rhs` and `lhs`.
Logical OR	||	`ProbeExp exp = lhs || rhs;`	logical OR of `lhs` and `rhs`.
Logical Negation	!	`ProbeExp exp = ! rhs;`	logical negation of `rhs`.

Table 40. Creating probe expressions to represent relational operations

Operation:	Operator:	For example, this code:	Creates a probe expression `exp` that represents the:
Equality Comparison	==	`ProbeExp exp = lhs == rhs;`	comparison of `lhs` and `rhs` where 1 will be returned if the values are equal and 0 will be returned if they are not.
Inequality Comparison	!=	`ProbeExp exp = lhs != rhs;`	comparison of `lhs` and `rhs` where 0 will be returned if the values are equal and 1 will be returned if they are not.
Greater Than Comparison	>	`ProbeExp exp = lhs > rhs;`	relative size comparison of `lhs` and `rhs` where 1 is returned if `lhs` is greater than `rhs` and 0 is returned otherwise.
Less Than Comparison	<	`ProbeExp exp = lhs < rhs;`	relative size comparison of `lhs` and `rhs` where 1 will be returned if `lhs` is less that `rhs` and 0 will be returned otherwise.
Greater Than or Equal To Comparison	>=	`ProbeExp exp = lhs >= rhs;`	relative size comparison of `lhs` and `rhs` where 1 will be returned if `lhs` is greater than or equal to `rhs`, and 0 will be returned otherwise.
Less Than or Equal To Comparison	<=	`ProbeExp exp = lhs <= rhs;`	relative size comparison of `lhs` and `rhs` where 1 will be returned if `lhs` is less than or equal to `rhs`, and 0 will be returned otherwise.

Table 41 (Page 1 of 2). Creating probe expressions to represent assignment operations

Operation:	Operator:	For example, this code:	Creates a probe expression `exp` that:
Assignment	None. The = operator could not be overloaded without causing simple expression manipulation to become unwieldy. Instead, use the `assign` function	`ProbeExp exp = pe1.assign(pe2);`	represents the assignment of the value represented by `pe2` into the storage location indicated by `pe1`.

Table 41 (Page 2 of 2). Creating probe expressions to represent assignment operations

Operation:	Operator:	For example, this code:	Creates a probe expression `exp` that:
Addition Assignment	+=	`ProbeExp exp = lhs += rhs;`	represents the addition of `lhs` and `rhs` and the subsequent assignment of the result into the storage location indicated by `lhs`.
Subtraction Assignment	-=	`ProbeExp exp = lhs -= rhs;`	represents the subtraction of `rhs` from `lhs` and the subsequent assignment of the result into the storage location indicated by `lhs`.
Multiplication Assignment	*=	`ProbeExp exp = lhs *= rhs;`	represents the multiplication of `lhs` and `rhs` and the subsequent assignment of the result into the storage location indicated by `lhs`.
Division Assignment	/=	`ProbeExp exp = lhs /= rhs;`	represents the division of `lhs` by `rhs` and the subsequent assignment of the result into the storage location indicated by `lhs`.
Modulus Assignment	%=	`ProbeExp exp = lhs %= rhs;`	represents the integer division of `lhs` by `rhs` in which the remainder rather than the dividend is returned and is subsequently assigned into the storage location indicated by `lhs`.
Bitwise AND and Assignment	&=	`ProbeExp exp = lhs &= rhs;`	represents a bitwise AND of `lhs` and `rhs`, and the subsequent assignment of the result into the storage location indicated by `lhs`.
Bitwise Inclusive OR and Assignment	|=	`ProbeExp exp = lhs |= rhs;`	represents a bitwise inclusive OR of `lhs` and `rhs`, and the subsequent assignment of the result into the storage location indicated by `lhs`.
Bitwise Exclusive OR and Assignment	^=	`ProbeExp exp = lhs ^= rhs;`	represents a bitwise exclusive OR of `lhs` and `rhs`, and the subsequent storage of the result into the storage location indicated by `lhs`.
Left Shift and Assignment	<<=	`ProbeExp exp = lhs <<= rhs;`	represents a bitwise left shift of `lhs` by `rhs` places, and the subsequent storage of the result into the storage location indicated by `lhs`.
Right Shift and Assignment	>>=	`ProbeExp exp = lhs >>= rhs;`	represents a bitwise right shift of `lhs` by `rhs` places, and the subsequent storage of the result into the storage location indicated by `lhs`.

Table 42. Creating probe expressions to represent pointer operations

Operation:	Operator:	For example, this code:	Creates a probe expression `exp` that represents the:
Address	None. The & operator could not be overloaded because it is used in passing arguments to functions that use call by reference. Instead, use the `address` function.	`ProbeExp exp = pe.address();`	referencing of the probe expression `pe`.
Dereference	*	`ProbeExp exp = * rhs;`	dereferencing of the pointer value `rhs`.
Index and Dereference	[]	`ProbeExp exp = lhs [ rhs ];`	addition of `rhs` to `lhs`, and the subsequent dereferencing of the result.

Once the analysis tool has created a probe expression that represents an operation, it can:

- Use it as a subexpression in another probe expression representing an operation.
- Combine the probe expression with another to form a single probe expression that represents a sequence of the two expressions. For more information, see "Step 2c: Create probe expressions to represent a sequence of instructions."
- Combine the probe expression into a probe expression representing a conditional statement. The probe expression representing the operation could, in the conditional statement, represent the test condition, the code that executes if the condition tests true, or the code that executes if the condition tests false. For more information, see "Step 2d: Create probe expressions to represent conditional logic" on page 128.

Step 2c: Create probe expressions to represent a sequence of instructions

The preceding substep ("Step 2b: Create probe expressions to represent operations" on page 123) illustrates how an analysis tool can create probe expressions that represent operations. This substep now shows how an analysis tool can combine two probe expressions into a single probe expression that represents a sequence of the two operations. Such sequence probe expressions can then themselves be combined to form even longer sequences of instructions. To combine two probe expressions into a sequence, the analysis tool uses the `ProbeExp::sequence` function. For example, say the logic that an analysis tool wants to build into a probe expression can be represented in C code as:

```
pe1 = pe1 + 1;
fcn(pe1);
pe2 = pe1;
send(pe2);
```

To mimic this logic in a single probe expression, the analysis tool first creates probe expressions that represent the four basic operations:

```
ProbeExp stmt1 = pe1.assign(pe1 + ProbeExp(1));
ProbeExp stmt2 = fcn.call(1, &pe1);
ProbeExp stmt3 = pe2.assign(pe1);
args[0] = Ais_msg_handle;
args[1] = pe2.address();
args[2] = ProbeExp(4);
ProbeExp stmt4 = Ais_send.call(3, args);
```

Next, these four probe expressions are combined into two probe expressions — each representing the sequence of two operations.

```
ProbeExp seq1 = stmt1.sequence(stmt2);
ProbeExp seq2 = stmt3.sequence(stmt4);
```

Finally, these two sequence probe expressions are combined into a longer sequence representing the entire probe logic.

```
ProbeExp seqall = seq1.sequence(seq2);
```

Once the analysis tool has created a sequence probe expression, it can:

- Repeat this step to combine the sequence probe expression into a longer sequence.
- Combine the sequence probe expression into a probe expression representing a conditional statement. The sequence probe expression could be the test condition, the code that executes if the condition tests true, or the code that executes if the condition tests false. For more information, see “Step 2d: Create probe expressions to represent conditional logic.”

Step 2d: Create probe expressions to represent conditional logic

This substep shows how an analysis tool can create a probe expression to perform conditional logic. To do this the analysis tool:

1. creates probe expressions to represent the test condition, the code to execute if the condition tests true, and, optionally, the code to execute if the condition tests false.
2. uses the `ProbeExp::ifelse` function to combine the three probe expressions into a probe expression representing a conditional statement.

Using the `ProbeExp::ifelse` function, the analysis tool is able to mimic an `If` or `If/Else` expression in C.

For example, say the logic that an analysis tool wants to build into a probe expression can be represented in C code as the `If` expression:

```
if (pe1 > 0) send(pe1);
```

To mimic this logic in a probe expression, the analysis tool first creates probe expressions to represent the test condition and the code to execute if the condition tests true.

```
// first the test condition

ProbeExp ce = pe1 > ProbeExp(0);

// now the then clause

args[0] = Ais_msg_handle;
args[1] = pe1.address();
args[2] = ProbeExp(4);
ProbeExp te = Ais_send.call(3, args);
```

Next, the probe expression representing the test condition calls the `ifelse` function. The probe expression to execute if the condition tests true is supplied as a parameter to the function. The `ifelse` function returns a single probe expression that represents the entire conditional statement.

```
ProbeExp exp = ce.ifelse(te);
```

So in the above example, if the probe expression `ce` evaluates to a non-zero value, the probe expression `te` executes. If the probe expression `ce` evaluates to zero, however, `te` does not execute. Instead, execution continues past the conditional statement. In the above example, this entire conditional logic is stored in the `ProbeExp` object `exp`.

The above example illustrates how a probe expression can mimic an `If` statement. The analysis tool can also use the `ProbeExp::ifelse` function to create a probe expression that mimics an `If/Else` statement. All it needs to do is supply an additional parameter to the `ifelse` function — one representing the code to execute if the test condition tests false. For example, say the logic that an analysis tool wants to build into a probe expression can be represented in C code as the `If/Else` expression:

```
if (pe1 > 0)
   send(pe1);
else
   send(pe2);
```

To mimic this logic in a probe expression, the analysis tool first creates probe expressions to represent the test condition, the code to execute if the condition tests true, and the code to execute if the condition tests false.

```
// first the test condition

ProbeExp ce = pe1 > ProbeExp(0);

// now the then clause

args[0] = Ais_msg_handle;
args[1] = pe1.address();
args[2] = ProbeExp(4);
ProbeExp te = Ais_send.call(3, args);

// and the else clause

args[0] = Ais_msg_handle;
args[1] = pe2.address();
args[2] = ProbeExp(4);
ProbeExp ee = Ais_send.call(3, args);
```

Next, the probe expression representing the test condition calls the `ifelse` function. The probe expression to execute if the condition tests true, and the probe expression to execute if the condition tests false are both supplied as parameters to the function. The `ifelse` function returns a single probe expression that represents the entire conditional statement.

```
ProbeExp exp = ce.ifelse(te, ee);
```

So in the above example, if the probe expression `ce` evaluates to a non-zero value, the probe expression `te` executes. If the probe expression `ce` evaluates to zero, the probe expression `ee` executes. In the above example, this entire conditional logic is stored in the `ProbeExp` object `exp`.

Once the analysis tool has created a conditional probe expression, it can:

- Repeat this step to include the conditional probe expression as part of a more complex conditional statement.
- Combine the probe expression with another to form a single probe expression that represents a sequence of the two expressions. For more information, see "Step 2c: Create probe expressions to represent a sequence of instructions" on page 127.

Step 2e: Create probe expressions to represent function calls

This substep shows how an analysis tool can create a probe expression that represents a function call. The analysis tool can create such a probe expression to represent a call to:

- the DPCL system-defined function `Ais_send`. The ability to call this function enables a probe to send data back to the analysis tool.
- a function contained in a probe module (a compiled object file containing one or more functions written in C) that has been loaded into the target application process. For more information on probe modules, refer to "What is a probe module?" on page 17 and "Creating and calling probe module functions" on page 136.
- a function that is already present in the target application.
- an AIX function like `getrusage`, `times`, or `vtimes`. The ability to call AIX functions enables a probe to get performance and system-resource information for a target application process.

In all four of these situations, the analysis tool uses the `ProbeExp::call` function to create a probe expression that represents a function call. For more information on the `ProbeExp::call` function, refer to its AIX man page, or its entry in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Once the analysis tool has created a probe expression that represents a function call, it can:

- Combine the probe expression with another to form a single probe expression that represents a sequence of the two expressions. For more information, see "Step 2c: Create probe expressions to represent a sequence of instructions" on page 127.
- Combine the probe expression into a probe expression representing a conditional statement. The probe expression representing the function could be the code that executes if the condition tests true, or the code that executes if the condition tests false. For more information, see "Step 2d: Create probe expressions to represent conditional logic" on page 128.

Creating a probe expression to represent a call to the Ais_send function:

The `Ais_send` function enables a probe to send data back to the analysis tool. The `Ais_send` function takes three parameters — a message handle for managing where the data is sent, the address of the data to send, and the size of the data being sent. To send the data located at the address `&pcount` back to the analysis tool, the call to the `Ais_send` function, if written in C code, would be:

```
Ais_send(handle, &pcount, 4);
```

To mimic this function call in a probe expression, the analysis tool first creates an array of probe expressions, with each expression in the array representing one of the parameters to the `Ais_send` function. As described in "Step 2b: Create probe

expressions to represent operations" on page 123, the ProbeExp class did not overload the unary address operator (&). Note that the following code calls the ProbeExp::address function to mimic the unary address operator used in the preceding C code.

```
ProbeExp parms[3];
parms[0] = Ais_msg_handle;
parms[1] = pcount.address();
parms[2] = ProbeExp (4);
```

Next the analysis tool creates a probe expression that represents the call to Ais_send. The parameter values intended for the Ais_send function are passed as the probe expression array parms to the ProbeExp::call function. The first parameter to the ProbeExp::call function indicates the number of probe expressions in the array. In this example, there are three probe expressions in the array.

```
ProbeExp sendexp = Ais_send.call(3, parms);
```

"Creating probe expressions to represent actual function parameter values in the target application" on page 122 contains example code that shows how the analysis tool can create a probe expression that, when executed within a target application process, will determine the actual value of a function parameter at that point in time. The following example code expands on this earlier example to show how the analysis tool can create a probe expression that, when executed within a target application process, will call the Ais_send function to send this actual parameter information back to the analysis tool.

```
// allocate an integer variable

ProbeExp v = P.balloc_mem(int32_type(), NULL, sts);

// create a prototype for a function that has an two arguments: and int
// and a pointer to int; and has no return value

ProbeType pr_args[2];
pr_args[0] = int32_type();
pr_args[1] = pointer_type(int32_type());
ProbeType proto = function_type(void_type(), 2, pr_args);

// create a probe expression representing the first parameter in a call
// to a function having this prototype

ProbeExp ap = proto.get_actual(0);

// create a probe expression to copy the value of the actual parameter to
// the variable we allocated and then send that value back

ProbeExp p1 = v.assign(ap);

ProbeExp sendargs[3];
sendargs[0] = Ais_msg_handle;
sendargs[1] = v.address();
sendargs[2] = ProbeExp(4);
ProbeExp p2 = Ais_send.call(3, sendargs);

ProbeExp p3 = p1.sequence(p2);
```

For more information on the Ais_send function, refer to its AIX man page, or its entry in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Creating a probe expression to represent a call to an AIX function: The analysis tool can also use the ProbeExp::call function to create a probe expression that represents a call to an AIX function like getrusage, times, or

vtimes. The ability to call AIX functions enables a probe to get performance and system-resource information for a target application process. For example, say that, in order to get system resource usage for a target application process, the analysis tool needs to create a probe expression that calls the AIX function getrusage. In C code, a call to this function would look like:

```
#include <sys/resources.h>
struct rusage ru;
getrusage(RUSAGE_SELF, &ru);
```

To mimic this call in a probe expression, the analysis tool first creates an array of probe expressions, with each expression in the array representing one of the parameters to the getrusage function.

```
#include <sys/resources.h>
ProbeExp buf = P.balloc_mem(
     unspecified_type(sizeof(struct rusage)), NULL, sts);
if (sts.status() != ASC_success)
  printf("balloc_mem error: %s\n", sts.status_name();
else {
  ProbeExp params[2];
  params[0] = ProbeExp(RUSAGE_SELF);
  params[1] = buf.address();
}
```

Next the analysis tool creates a probe expression that represents a call to the getrusage function. In this example, moduleobj is the module in the source tree that contains the getrusage function, and i is the index of the getrusage function in that module.

```
SourceObj getrusage_fcn = moduleobj.child(i);
ProbeExp getrusage_ref = getrusage_fcn.reference();
ProbeExp the_call = getrusage_ref.call(2, params);
```

For more information about the AIX function getrusage, refer to its AIX man page.

Creating a probe expression to represent a call to a probe module function: The analysis tool can also use the ProbeExp::call function to create a probe expression that represents a call to a function contained in a probe module (a compiled object file containing one or more functions written in C) that has been loaded into a target application process. Once an analysis tool loads a particular probe module into a target application process, a probe expression is able to call any of the functions contained in the module. See “Creating and calling probe module functions” on page 136 for more information on creating probe modules.

Say that a probe module you have created contains a function count that is designed to count the number of times the subroutine is called. This probe module function takes one parameter — the predefined global variable Ais_msg_handle (which is used by the DPCL system when the probe sends data back to the analysis tool). In C code, a call to this function would look like:

```
count(handle);
```

To mimic this call in a probe expression, the analysis tool:

1. calls the ProbeModule::get_reference function to create a probe expression that represents a reference to the count function. The analysis tool supplies the ProbeModule::get_reference function with the index of the count function within the probe module; the ProbeModule::get_reference function returns a probe expression representing a reference to the count function. To identify the count

function within the probe module, the analysis tool can use the ProbeModule::get_count and ProbeModule::get_name functions.

```
char name[128];

ProbeModule pm("my_module");
int fcncount = pm.get_count();
int found = 0;
for (int i=0; !found && i<fcncount; ++i) {
  pm.get_name(i, name, 128);
  if ( strcmp(name, "count") == 0)
    found = 1;
  }

  if ( found == 0 )
    printf("function 'count' not found in probe module\n");
  else {
    ProbeExp count_ref = pm.get_reference(i);
```

For more information on the ProbeModule::get_reference, ProbeModule::get_count, and ProbeModule::get_name function, refer to their AIX man pages, or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

2. creates an array of probe expressions, with each expression in the array representing one of the parameters to the function. Since the count function has only one input parameter, this array has only one expression.

```
ProbeExp parms[1];
parms[0] = Ais_msg_handle;
```

3. combines these probe expressions (which represent a reference to the probe module function count and the parameters to supply to that function) into a single expression that represents the function call. The parameter value intended for the count function is passed as the probe expression array args to the ProbeExp::call function. The first parameter to the ProbeExp::call function indicates the number of probe expressions in the array. In this example, there is only one probe expression in the array.

```
ProbeExp the_call = count_ref.call(1, parms);
```

Creating a probe expression to represent a call to a target application function:

Using the SourceObj::reference function, the analysis tool is able to create a probe expression that represents a function in the target application. This probe expression can then be combined by the analysis tool into a probe expression representing a call to that function. To create a probe expression that represents a target application function, however, the analysis tool must first navigate the target application's source code structure to identify the function of interest. If you are unfamiliar with SourceObj objects and the concept of source hierarchies, you may want to refer to “What is the SourceObj class?” on page 48 before reading the following example code.

The following example code:

1. Calls the Process::get_program_object function to return the top-level source object (SourceObj object) associated with a process.
2. Identifies the program module that contains the function. To do this, it uses the SourceObj::child_count function to initialize a for loop and then, within the for

loop, use the SourceObj::child and SourceObj::module_name functions to identify the target module.

3. Calls the SourceObj::bexpand function to expand the module. This enables the analysis tool to navigate further down the source hierarchy to examine additional program structure (including functions). The analysis tool code could also expand a module using the asynchronous SourceObj::expand function.
4. Identifies the function. To do this, the analysis tool code again uses the SourceObj::child_count function to initialize a for loop, and then, within the for loop, uses the SourceObj::child and SourceObj::get_demangled_name functions to identify the target function.
5. Calls the SourceObj::reference function to create a probe expression that represents the function.
6. Uses the ProbeExp::call function to combine the probe expression representing the function into a more complex probe expression representing a function call.

```
#include <dpcl.h>
#include <libgen.h>             // for basename()
// find the function function_name defined in the module module_name
// for a given Process p
// return: SourceObj contains the function.
//         or src_type()==SOT_unknown_type if the function is not found
SourceObj find_function(Process p,
                        char * module_name,
                        char * function_name)
{
    // get the source object associated with a process
    SourceObj progobj = p.get_program_object();
    SourceObj ret;              // return object
    SourceObj mod;              // module object

    // identifies the program module that contains the function
    // we preallocate a buffer to hold the module name
    int mod_name_length=1024;
    char * mod_name = new char [mod_name_length];
    int mod_index;
    for(mod_index = 0;mod_index < progobj.child_count();mod_index++) {
      mod = progobj.child(mod_index);

      // enlarge the name buffer if necessary.
      if (mod.module_name_length() >= mod_name_length) {
        // double the length
        while(mod_name_length < mod.module_name_length()) mod_name_length*=2;
        delete [] mod_name;
        mod_name = new char [mod_name_length];
      } /* endif */

      // get the module name
      mod.module_name(mod_name,mod_name_length);

      // depending on how the target application was compiled, the module
      // name may contain path information; for purposes of illustration, we compare
      // against the basename only.
      if (0 == strcmp(basename(mod_name),module_name)) {        // found
        break;
      } /* endif */
    }
    delete [] mod_name;
    if(mod_index == progobj.child_count())      // module not found
      return ret;
    // now we need to expand the module's SourceObj if necessary.
    // since only one module needs to expand, bexpand() will be easier
    if(mod.child_count() == 0) {                //
      AisStatus sts = mod.bexpand(p);
      if(sts.status()!=ASC_success)             // expand failed
```

```
        return ret;
    }

    // preallocate the function buffer to hold the function name
    int fun_name_length=1024;
    char * fun_name = new char [fun_name_length];
    SourceObj fun;
    for (int i=0; i<mod.child_count(); i++) {
      fun = mod.child(i);
      if(fun.src_type()!=SOT_function) continue;
      if (fun.get_demangled_name_length() >= fun_name_length) {
        while(fun_name_length < fun.get_demangled_name_length())
          fun_name_length*=2;
        delete [] fun_name;
        fun_name = new char [fun_name_length];
      } /* endif */
      fun.get_demangled_name(fun_name,fun_name_length);
      if(0 == strcmp(fun_name,function_name)) { // found the function
        ret = fun;
        break;
      }
    } /* endfor */
    delete [] fun_name;
    return ret;
}

main() {
    Process p;
    // ....
    SourceObj fun = find_function(p,"hello.c","foo");
    if(fun.src_type() == SOT_unknown_type) {
      // error report
    }
    ProbeExp foo = fun.reference();
    ProbeExp parms[1];
    // parms[0]=...
    ProbeExp the_call=foo.call(1,parms);
}
```

Example: Creating probe expressions

The following sample code creates a probe expression pass counter. To do this, it:

1. Calls the `Application::balloc_mem` function to allocate a variable in all processes in the application. The result of this call is a probe expression `pcount` that represents the variable.
2. Combines the probe expression `pcount` into a more complex probe expression `addexp` that represents the operation `pcount = pcount + 1;`.
3. Creates another probe expression that represents an `Ais_send` function call that sends the value of `pcount` back to the analysis tool.
4. Combines the two probe expressions into a single probe expression that represents a sequence of the two expressions.

Since this example uses the `Ais_send` function, note that it also provides a data callback routine for handling the data sent. Refer to Chapter 10, “Creating data callback routines” on page 167 for more information on data callback routines.

```
// define a pass-counter probe

  ProbeExp pcount = A.balloc_mem(int32_type(), NULL, sts);
  if (sts.status() != ASC_success)
    printf("error from balloc_mem: %s\n", sts.status_name());
  else {
    ProbeExp addexpr = pcount.assign(pcount + ProbeExp(1));
    ProbeExp parms[3];
    parms[0] = Ais_msg_handle;
```

```
    parms[1] = pcount.address();
    parms[2] = ProbeExp(4);
    ProbeExp send_call = Ais_send.call(3, parms);
    ProbeExp pass_ctr = addexpr.sequence(send_call);

    // install the probe

    GCBFuncType cbarr[1];
    GCBTagType tagarr[1];
    ProbeHandle ph;

    cbarr[0] = count_cb;
    tagarr[0] = (GCBTagType) 0;

    // assume that point has already been set

    sts = A.binstall_probe(1, &pass_ctr, &point, cbarr, tagarr, &ph);
    if (sts.status() != ASC_success)
      printf("error from binstall_probe: %s\n", sts.status_name());

    .
    .
    .
} // end of program

// the callback function

void count_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg)
{
  Process *P = (Process *) obj;
  int *count = (int *) msg;
  int task = P->get_task();

  printf("task %d count = %d\n", task, *count);
}
```

Creating and calling probe module functions

A probe module is a compiled object file containing one or more functions written in C. Once an analysis tool loads a particular probe module into a target application, a probe expression is able to call any of the functions contained in the module. It is often preferable for a probe expression to call a probe module function rather than try to create the same probe logic with probe expressions alone. This is because probe modules:

- can be reused by other analysis tools.
- enable you to code the probe logic in a straightforward way using C. For complicated probe logic this can be easier than the process of building an abstract syntax tree of probe expressions (as described in “Creating probe expressions” on page 115).
- can contain more complicated probe logic than is possible is a simple probe expression.

In addition to these advantages, be aware that, if the probe is to be executed as a phase probe, than its logic must be contained in a probe module function. This is because phase probes, unlike point probes and one-shot probes, cannot be a simple probe expression that does not call a probe module function. To create a probe module and call one of its functions, you must:

1. Create the probe module function.
2. Compile the probe module.

3. Instantiate a `ProbeModule` class object to represent the probe module.
4. Load the probe module into one or more processes.
5. Create a probe expression to call the probe module function(s).
6. Create a data callback function to respond to any data message that can be sent by the probe.

The following steps describe these tasks in more detail. For sample code, see "Example: Creating and calling a probe module function" on page 140.

Step 1: Create probe module function

The first step in creating and calling a probe module function is to write the actual function. To send data back to the analysis tool, the probe module can call the built-in DPCL function `Ais_send`. The `Ais_send` function takes three parameters — a message handle for managing where the data is sent, the address of the data to send, and the size of the data being sent. In order to make the prototype of the `Ais_send` function available to the compiler, your analysis tool code should include the header file `dpclExt.h`.

For example, the following probe module function is a generic pass counter that, when installed within a subroutine in a target application process, will count the number of times the subroutine is called. Each time the counter is incremented 10 times, the probe will call the `Ais_send` function to send a message back to the analysis tool.

```
#include <dpclExt.h>

count(void *msg_handle)
{
  static int pcount = 0;
  char msg[100];

  pcount++;

  if ((pcount % 10) == 0)
  {
    sprintf(msg, "I have been called %d times\n", pcount);
    Ais_send(msg_handle, (void *) msg, 1 + strlen(msg));
  }
}
```

If your probe module function calls the `Ais_send` function as in this example, your analysis tool code will need to include a data callback routine to respond to data sent by the probe module. Refer to Chapter 10, "Creating data callback routines" on page 167 for more information on data callback routines. For more information on the `Ais_send` function, refer to its AIX man page, or its entry in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Step 2: Compile the probe module

Once you have written your probe module function, you'll need to compile it into an object file. This object file is the probe module. Before compiling the probe module, you'll need to create an export file to export its functions. What's more, if the probe module function calls any functions outside of the probe module, you'll need to create an import file. For example, in the preceding step, we encapsulated a simple pass counter into a function called `count`. Here's our export file:

```
* Any line started with a '*' is a comment
* We have a single function to export
count
```

Also, since the count function calls the built-in DPCL function `Ais_send`, we also create an import file:

```
#! .
Ais_send
```

Finally, we compile the file. Our export file is named `count.exp`, and our import file is named `count.imp`.

```
cc -o count count.c -bE:count.exp -bI:count.imp -bnoentry -I/usr/lpp/ppe.dpcl/include
```

Step 3: Instantiate a ProbeModule class object to represent the probe module

In order to load a probe module into one or more target application processes, the analysis tool must create a `ProbeModule` class object that represents the probe module. The probe module class is defined in the header file `ProbeModule.h`. To assign a probe module file name to a `ProbeModule` class object, you can use a non-default constructor, a non-default constructor with a copy constructor, or the default constructor with an assignment operator.

Table 43. Instantiating a ProbeModule object

To create a `ProbeModule` class object, the analysis tool can:	**For example:**
Use a default constructor and an assignment operator to assign the file name of the probe module to the `ProbeModule` object.	`ProbeModule my_probe_mod;` `my_probe_mod = ProbeModule("count");`
Use a non-default constructor to directly assign the file name of the probe module to the `ProbeModule` object.	`ProbeModule my_probe_mod("count");`
Use a non-default constructor and a copy constructor to assign the file name of the probe module to the `ProbeModule` object.	`ProbeModule my_probe_mod = ProbeModule("count");`

For more information on the probe module class and its constructors, refer to the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Step 4: Load probe module into Process class object(s)

In order for a probe expression to call a probe module function, the probe module function must be loaded into the same target application process(es) within which the probe expression will execute. To load a probe module on a single process basis, the analysis tool can use the asynchronous function `Process::load_module` or its blocking equivalent `Process::bload_module`. To load a probe module on an application-wide basis (for all `Process` objects managed by an `Application` object), the analysis tool can use the functions `Application::load_module` or `Application::bload_module`.

Table 44. Loading a probe module into one or more target application processes

To load a probe module:	In a single process (`Process` object)	In multiple processes (all of the `Process` objects managed by an `Application` object)

Using the asynchronous function `load_module`	`int count = 1;`
	`sts = P.load_module(&pmodule,proc_cb,&count);`
	`if(sts.status() != ASC_success){`
	`  printf("load_module error %s\n",`
	`  sts.status_name());`
	`  count -= 1;`
	`}`
	`if(count) Ais_main_loop();`
	
	`// ...`
	
	`void proc_cb(GCBSysType s, GCBTagType t,`
	`  GCBObjType o, GCBMsgType m) {`
	`  int* count = (int *) t;`
	`  AisStatus * stsp = (AisStatus *) m;`
	`  if(stsp->status() != ASC_success) {`
	`  // print error`
	`  }`
	`  *count -= 1;`
	`  if(*count == 0) Ais_end_main_loop();`
	`}`
		`int count = A.get_count();`
		`AisStatus sts = A.load_module(&pmodule,app_cb,`
		`  &count);`
		`if(sts.status() != ASC_success){`
		`  printf("load_module: error is %s\n",`
		`  sts.status_name());`
		`}`
		`for(i = 0; i < A.get_count(); i++) {`
		`  if(A.status(i).status()!=ASC_success){`
		`  count -= 1;`
		`  }`
		`}`
		`if(count) Ais_main_loop();`
		
		`void app_cb(GCBSysType s, GCBTagType t,`
		`  GCBObjType o, GCBMsgType m) {`
		`  int* count = (int *) t;`
		`  AisStatus * stsp = (AisStatus *) m;`
		`  if(stsp->status() != ASC_success) {`
		`  // print error`
		`  }`
		`  *count -= 1;`
		`  if(*count == 0) Ais_end_main_loop();`
		`}`

Using the blocking function `bload_module`	`AisStatus sts = P.bload_module(&pmodule);`
	`if(sts.status() != ASC_success) {`
	`  printf("bload_module: %s\n",`
	`  sts.status_name());`
	`}`
		`AisStatus sts = A.bload_module(&pmodule);`
		`if(sts.status() != ASC_success) {`
		`  printf("bload_module: %s\n",sts.status_name());`
		`  for(int i=0;i<A.get_count();i++) {`
		`  if(A.status(i).status()!=ASC_success){`
		`  // print error`
		`  }`
		`  }`
		`}`

For more information on the `Process::load_module`, `Process::bload_module`, `Application::load_module`, and `Application::bload_module` functions, refer to their AIX man pages, or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Step 5: Create probe expression to reference or call the probe module function

To execute a probe module function as a probe, your analysis tool code must create a probe expression that calls or references the probe module function.

- If it is going to execute the probe module function as a point probe or a one-shot probe, the analysis tool uses the `ProbeExp::call` function to create a probe expression representing a function call. See “Creating a probe expression to represent a call to a probe module function” on page 132 for more information.
- If it is going to execute the probe module function as a phase probe, the analysis tool uses the `ProbeModule::get_reference` function to create a probe expression that represents a reference to the function. See “Executing phase probes” on page 153 for more information.

For additional information on the ProbeExp::call and ProbeModule::get_reference functions, refer to their AIX man pages, or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Step 6: Create data callback function to respond to messages from the probe

As described in "Step 1: Create probe module function" on page 137, a probe module function can send data back to the analysis tool by calling the DPCL system-defined function Ais_send. If your probe module function does call the Ais_send function, then the analysis tool code must contain a data callback function that will handle the data that the Ais_send function will send. See Chapter 10, "Creating data callback routines" on page 167 for more information on how to do this.

Example: Creating and calling a probe module function

The following example code:

- encapsulates a general-purpose pass counter into the function count in the file count.c,.
- compiles count.c into a probe module count (using the count.exp and count.imp files to export and import functions as needed,
- creates a ProbeModule object to represent the probe module count, and loads the module into process P,
- gets a reference to, and creates a probe expression to call, the count function, and
- handles data sent by the Ais_send function using a data callback routine. If your probe module uses the Ais_send function to send data back to the analysis tool, then the analysis tool code must contain a data callback routine. Refer to Chapter 10, "Creating data callback routines" on page 167 for more information on data callback routines.

count.c

```
#include <dpclExt.h>
count(void *msg_handle)

{
  static int pcount = 0;
  char msg[100];

  pcount++;

  if ((pcount % 10) == 0)
  {
    sprintf(msg, "I have been called %d times\n", pcount);
    Ais_send(msg_handle, (void *) msg, 1 + strlen(msg));
  }
}
```

count.exp

```
* Any line started with a '*' is a comment
* We have a single function to export
count
```

count.imp

```
#! .
Ais_send
```

compilation command:

```
cc -o count count.c -bE:count.exp -bI:count.imp -bnoentry -I/usr/lpp/ppe.dpcl/include
```

analysis tool code:

```
#include <dpcl.h>
void count_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg);

main(int argc, char *argv[]) {

    Process         P("phantom.pok.ibm.com", 12345);

    Ais_initialize();

    ProbeModule       my_probe_mod("count");

    AisStatus sts = P.bconnect();

    sts = P.bload_module(&my_probe_mod);

    // look for a specific function object in the probe module
    ProbeExp  count_fun;
    const int bufSize = 128;
    char      bufname[bufSize];   // buffer for module_name(..)

    for (int i=0; i < my_probe_mod.get_count(); i++)
    {
        char *funct_name = my_probe_mod.get_name(i, bufname, bufSize);

        // is this function count ??
        if ( strcmp(funct_name, "count") == 0 )
        {
            count_fun = my_probe_mod.get_reference(i);
            break; // yes.
        }
    }

    ProbeExp args[1];
    args[0] = Ais_msg_handle;
    ProbeExp count_call = count_fun.call(1, args);

    // now search the source obj tree for a particular function
    SourceObj  my_program = P.get_program_object();

    // Look for the correct Module
    SourceObj my_module;
    char      bufmname[bufSize];      // buffer for module_name(..)

    for (i=0; i < my_program.child_count(); i++)
    {
        my_module = my_program.child(i);

        char *mod_name = my_module.module_name(bufmname,bufSize);

        if ( strcmp(mod_name, "stencil.f") == 0 )
        {
            // expand the module
            sts = my_module.bexpand(P);
            break;
         }
    }

    InstPoint  one_point;
    ProbeHandle   phandle;
    char buffname[bufSize];
    GCBFuncType cbarr[1];
    GCBTagType tagarr[1];
```

```
    cbarr[0] = count_cb;
    tagarr[0] = 0;

    for ( i=0; i < my_module.inclusive_point_count(); i++)
    {
        one_point = my_module.inclusive_point(i);

        if ( (one_point.get_type() == IPT_function_call) &&
             (one_point.get_location() == IPL_before)     )
        {
            char *funct_name = one_point.get_demangled_name(buffname, bufSize);

               if ( strcmp(funct_name, "compute_stencil") == 0 )
            {
                // Install the trace probe at the function call site
                   sts = P.binstall_probe( 1, &count_call, &one_point,
                                           cbarr, tagarr, &phandle);

               //activate the probe
               sts = P.bactivate_probe(1, &phandle);
                }
        }
    }

    Ais_main_loop();
}

// collect data from the callback
void count_cb(GCBSysType sys, GCBTagType tag, GCBTagType obj, GCBMsgType msg)
{

    // the message being send is a string

    char * count_message = (char *) msg;

    printf("%s\n", count_message);

}
```

Chapter 9. Executing probes in target application processes

Chapter 8, “Creating probes” on page 115 describes how to define a probe that can execute as part of a target application process. The manor in which the probe is installed and executed within one or more target application processes distinguishes the probe as a particular probe type — either a point probe, a phase probe, or a one-shot probe.

- point probes are installed at particular locations in the target application code, and, when in an activated state, are triggered whenever execution reaches that location in the code. See “Installing and activating point probes” for instructions on executing probes of this type. For an overview of point probes, see “What is a point probe?” on page 17.
- Phase probes are triggered by expiration of a timer and are executed regardless of what code the target application is executing. See “Executing phase probes” on page 153 for instructions on executing probes of this type. For an overview of phase probes, see “What is a phase probe?” on page 19.
- One-shot probes are executed immediately and regardless of what code the target application is executing. See “Executing one-shot probes” on page 163 for instructions on executing probes of this type. For an overview of one-shot probes, see “What is a one-shot probe?” on page 20.

Installing and activating point probes

Point probes are probes installed at particular locations in the target application code that, when in an activated state, are triggered whenever execution reaches that location in the code. To install and activate a point probe in one or more target application processes, the analysis tool must:

1. Define the probe as described in Chapter 8, “Creating probes” on page 115.
2. Navigate the application source structure (represented by `SourceObj` objects) to identify the instrumentation point (`InstPoint` object) where the probe will be installed.
3. Install the probe at the instrumentation point within one or more target application processes.
4. Activate the probe so that it will execute as part of the target application process whenever execution reaches its installed location in the target application code.

The following steps describe these tasks in greater detail. For sample code, see “Example: Installing and activating a point probe” on page 151.

Step 1: Create point probe

A probe is a probe expression that may optionally call functions. The first step in installing and activating a point probe is to build the actual probe expression that will serve as the point probe. See Chapter 8, “Creating probes” on page 115 for detailed instructions on how to do this.

Step 2: Navigate application source structure to get instrumentation point

Instrumentation points (`InstPoint` objects) are locations within a target application process where an analysis tool can install point probes. Before the analysis tool can install a point probe in a target application process, it must get a reference to the `InstPoint` object where it wishes to place the probe. To get such a reference, the analysis tool must navigate the target application's source structure by means of source objects (`SourceObj` objects). Each `SourceObj` object represents part of the source code structure associated with the target application process, and a group of such objects provide the hierarchical representation of the source code structure which the analysis tool can navigate. For an overview of instrumentation points, see "What are instrumentation points?" on page 18. For an overview of source objects, see "What are source objects?" on page 17.

To navigate an application's source structure to get an instrumentation point, the analysis tool must:

1. Get a reference to a target application process (`Process` object) associated with the source code structure to navigate. If the analysis tool is instrumenting a serial application, then this is simply the `Process` object that represents the target application. If the analysis tool is instrumenting a parallel application, however, this would be just one of the `Process` class objects managed by the `Application` class object. Note that while this initial source code navigation must be performed using a single `Process` object, the analysis tool can later use member functions of the `Application` class to install, activate, and remove the point probe in all processes managed by the application (assuming these are like processes compiled from the same source as in an SPMD application).
2. Get a reference to the top-level source object (called the "program object") for the process. The analysis tool does this by calling the `Process::get_program_object` function.
3. Navigate one level down into the source hierarchy to get a reference to the source object representing the module where the analysis tool will install the point probe.
4. Expand the target module source object so the analysis tool can navigate further down into its source hierarchy.
5. Navigate one level further down into its source hierarchy to get a reference to the source object that represents the function where the analysis tool will install the point probe.
6. Navigate one level further down into the source hierarchy to get a reference to the instrumentation point where the analysis tool will install the point probe.

Step 2a: Get target process object

If the analysis tool is instrumenting a serial application, it doesn't need to perform this step, and you can skip ahead to "Step 2b: Get program object" on page 145. If the analysis tool is instrumenting a parallel application, however, it must first get a reference to a single process (`Process` object) that is managed by the `Application` class object. This is because a source hierarchy is associated with a particular process only. If multiple processes in the parallel application were compiled from the same source code (such as a SPMD program), the analysis tool can later use member functions of the `Application` class to install, activate, and remove the point probe for all the `Process` objects managed by the `Application`. To navigate the

source hierarchy, however, the analysis tool must identify a single representative Process in the Application. To do this, the analysis tool uses the Application::get_process function. The following line of code, for example, returns the first Process object in the Application object app1.

```
Process p = app1.get_process(0);
```

For more information on the Application::get_process function, refer to its AIX man page, or its entry in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Step 2b: Get program object

To navigate the source code structure associated with a particular process, the analysis tool first needs to get a reference to the source object (SourceObj object) that represents the top of the process source hierarchy. This top level SourceObj object is called the "program object" and is returned by the Process::get_program_object function. The following line of code stores the program object in a new SourceObj object named myprog.

```
SourceObj myprog = P.get_program_object();
```

For more information on the Application::get_program_object function, refer to its AIX man page, or its entry in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Step 2c: Identify target module where point probe will be installed

Once the analysis tool has a reference to the program object, it needs to navigate one level down into the source hierarchy to get a reference to the SourceObj object that represents the module where the analysis tool will install the point probe. To do this, the analysis tool can use the SourceObj::child_count, SourceObj::child, and SourceObj::module_name functions

The SourceObj::child_count function returns the number of child SourceObj objects associated with the SourceObj object; in the case of a program object, these child SourceObj objects represent the program modules. The SourceObj::child function returns a specific child SourceObj. Once it has a reference to the module level SourceObj object, the analysis tool can use the SourceObj::module_name function to identify the name of the module.

This code example uses the SourceObj::child_count function to initialize a for loop, and then, within the for loop, uses the SourceObj::child and SourceObj::module_name functions to identify the target module. This sample code assumes a specialized analysis tool designed for a particular program, and so the name of the target module is already known. For a general-purpose analysis tool, where the target application is not known at design time, these same functions could be used, for example, to populate a scrolled list to show the module names. Here's the specialized analysis tool example that identifies hello.c as the target module.

```
for (int c = 0; c < myprog.child_count(); c++)
{
      mymod = myprog.child(c);

      if (strcmp(mymod.module_name(bufmname, bufSize), "hello.c") ==0)
      {
            // CODE HERE
```

```
        }
}
```

For more information on the `SourceObj::child_count`, `SourceObj::child`, and `SourceObj::module_name` functions, refer to their AIX man pages, or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Step 2d: Expand target module

Once the analysis tool identifies the target module source object, it must expand it in order to navigate further down into the source hierarchy. To do this, the analysis tool can use either the blocking function `SourceObj::bexpand`, or the asynchronous function `SourceObj::expand`.

Table 45. Expanding a module-level source object

To use:	For example:

The blocking function
`SourceObj::bexpand`
	`for (int c = 0; c < myprog.child_count(); c++)`
	`{`
	`  mymod = myprog.child(c);`
	
	`  if (strcmp(mymod.module_name(bufmname, bufSize), "hello.c") ==0)`
	`  {`
	`  printf ("found module hello.c. expanding....\n");`
	
	`  sts = mymod.bexpand(P);`
	
	`  printf ("expand status  module # %d = %d\n", c, (int)sts);`
	`  if ( (int)sts != 0) printf("%s\n", sts.status_name());`
	
	`  break;`
	`  }`
	`}`

The asynchronous function
`SourceObj::expand`
	`void expand_cb(GCBSysType sys, GCBTagType tag, GCBTagType obj,`
	`  GCBMsgType msg);`
	
	`// ...`
	
	`for (c = 0; c < myprog.child_count(); c++)`
	`{`
	`  mymod = myprog.child(c);`
	`  name = mymod.module_name(buffer, bufsize);`
	`  if (strcmp(name, "integral_c.c") == 0)`
	`  {`
	`  // found the target module, expand it`
	`  async_done = 0;`
	`  sts = mymod.expand(P, (GCBFuncType)expand_cb, (GCBTagType)2);`
	`  printf("expand is submitted, status = %d\n", (int)sts);`
	`  if (sts.status() != ASC_success) exit(1);`
	`  break;`
	`  }`
	`}`
	
	`// ...`
	
	`void`
	`expand_cb(GCBSysType sys, GCBTagType tag, GCBTagType obj, GCBMsgType msg)`
	`{`
	`  AisStatus *sts = (AisStatus *)msg;`
	
	`  printf("expand is completed, status = %d\n", (int)*sts);`
	`  async_done = 1;`
	`  Ais_end_main_loop();`
	`}`

For more information on the SourceObj::bexpand and SourceObj::expand functions, refer to their AIX man pages, or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*

Step 2e: Identify target function

Once the analysis tool has expanded the target module, it can navigate one more level down into the source hierarchy to get a reference to the SourceObj that represents the function where the analysis tool will install the point probe. Once again, the analysis tool can navigate down into the hierarchy using the SourceObj::child_count and SourceObj::child functions. The SourceObj::child_count function returns the number of child SourceObj objects associated with the SourceObj object; in the case of a module source object, these child SourceObj objects represent the module's functions. The SourceObj::child function returns a specific child SourceObj object. Once it has a reference to the function level SourceObj object, the analysis tool can use the SourceObj::get_demangled_name function (to identify the demangled name of the function) or the SourceObj::get_mangled_name function (to identify the mangled name of the function).

This code example uses the SourceObj::child_count function to initialize a for loop, and then, within the loop, uses SourceObj::child and SourceObj::get_demangled_name functions to identify the target function. This sample code assumes a specialized analysis tool designed for a particular program, and so the name of the target function is already known. For a general purpose analysis tool, where the target application is not known at design time, these same functions could be used, for example, to populate a scrolled list to show the function names. Here's the specialized example that identifies the module hello as the target function.

```
SourceObj myfun;

char      bufdname[bufSize];   // buffer for get_demangled_name(..)

printf("function count = %d\n", mymod.child_count());

for ( c = 0; c < mymod.child_count(); c++)
{
     myfun = mymod.child(c);

     char *name = myfun.get_demangled_name(bufdname, bufSize);

     if (strcmp(name, "hello") ==0)
     {
            printf("function hello found.\n");
            break;
     }
}
```

For more information on the SourceObj::child_count, SourceObj::child, SourceObj::get_demangled_name, and SourceObj::get_mangled_name functions, refer to their AIX man pages or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Step 2f: Get reference to instrumentation point within target application

Once the analysis tool identifies the target function source object, it can identify the actual instrumentation point (`InstPoint` object) where it will install the point probe. To do this, the analysis tool can use the `SourceObj::exclusive_point_count`, `SourceObj::exclusive_point`, and `InstPoint::get_type` functions. The `SourceObj::exclusive_point_count` function returns the number of instrumentation points in the function, and the `SourceObj::exclusive_point` function returns a specific `InstPoint` object. The `InstPoint::get_type` function enables the analysis tool to determine whether the instrumentation point represents a function entry, function exit, or function call site.

This code example uses the `SourceObj::exclusive_point_count` function to initialize a `for` loop, and then, within the `for` loop, uses the `SourceObj::exclusive_point` and `InstPoint::get_type` functions to identify the function entry site.

```
InstPoint  mypoint;

printf("point count = %d\n", myfun.exclusive_point_count());

for ( c = 0; c < myfun.exclusive_point_count(); c++)
{
     mypoint = myfun.exclusive_point(c);

     if ( mypoint.get_type() == IPT_function_entry)
     {
          printf("function entry point found.\n");
          break;
     }
}
```

For more information on the `SourceObj::exclusive_point_count`, `SourceObj::exclusive_point`, and `InstPoint::get_type` functions, refer to their AIX man pages or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Step 3: Install probe at instrumentation point

Once the analysis tool has defined its probes (as described in Chapter 8, “Creating probes” on page 115) and identified an instrumentation point (as described in “Step 2: Navigate application source structure to get instrumentation point” on page 144), it can install the probe at the instrumentation point. To do this on a single process basis, the analysis tool can use the asynchronous function `Process::install_probe` or its blocking equivalent `Process::binstall_probe`. To do this on an application-wide basis, the analysis tool can use the functions `Application::install_probe` or `Application::binstall_probe`. Note that all of these functions accept an array of probes. This means that the analysis tool can install multiple point probes using a single call.

Table 46. Installing a point probe in one or more target application processes

To install a probe:	**In a single process (`Process` object)**	**In multiple processes (all of the `Process` objects managed by an `Application` object)**

Using the asynchronous function `install_probe`	`void install_probe_cb(GCBSysType sys,`
	`  GCBTagType tag, GCBTagType obj,`
	`  GCBMsgType msg);`
	
	`// ...`
	
	`// install the probe in function entry point`
	`points[0] = mypoint;`
	`cbs[0] = data_cb;`
	`tags[0] = (GCBTagType)1;`
	`async_done = 0;`
	`sts = P.install_probe(1, &fullexp, points,`
	`  cbs, tags,`
	`  (GCBFuncType)install_probe_cb,`
	`  (GCBTagType)3, phds);`
	`printf("install_probe is submitted, \`
	`status = %d\n", (int)sts);`
	`if (sts.status() != ASC_success) exit(1);`
	
	`// ...`
	
	`void`
	`install_probe_cb(GCBSysType sys,`
	`  GCBTagType tag, GCBTagType obj,`
	`  GCBMsgType msg)`
	`{`
	`  Process *p = (Process *)obj;`
	`  AisStatus *sts = (AisStatus *)msg;`
	
	`  printf("task %d install_probe is \`
	`completed, status = %d\n", p->get_task(),`
	`  (int)*sts);`
	`  async_done = 1;`
	`  Ais_end_main_loop();`
	`}`
		`void install_probe_cb(GCBSysType sys,`
		`  GCBTagType tag, GCBTagType obj,`
		`  GCBMsgType msg);`
		
		`// ...`
		
		`// install the probe in function entry point`
		`points[0] = mypoint;`
		`cbs[0] = data_cb;`
		`tags[0] = (GCBTagType)1;`
		`async_done = 0;`
		`sts = A.install_probe(1, &fullexp, points,`
		`  cbs, tags, (GCBFuncType)install_probe_cb,`
		`  (GCBTagType)3, phds);`
		`printf("install_probe is submitted, \`
		`status = %d\n", (int)sts);`
		`if (sts.status() != ASC_success) exit(1);`
		
		`// ...`
		
		`void`
		`install_probe_cb(GCBSysType sys, GCBTagType tag,`
		`  GCBTagType obj, GCBMsgType msg)`
		`{`
		`  static int count = 0;`
		`  Process *p = (Process *)obj;`
		`  AisStatus *sts = (AisStatus *)msg;`
		
		`  printf("task %d install_probe is \`
		`completed, status = %d\n", p->get_task(),`
		`  (int)*sts);`
		`  count++;`
		`  if (count >= num_procs)`
		`  {`
		`  async_done = 1;`
		`  Ais_end_main_loop();`
		`  }`
		`}`

Using the blocking function `binstall_probe`	`// install the probe in function entry point`
	`points[0] = mypoint;`
	`cbs[0] = data_cb;`
	`tags[0] = (GCBTagType)1;`
	`sts = P.binstall_probe(1, &fullexp, points, cbs,`
	`  tags, phds);`
	`printf("install_probe is done, status = %d\n",`
	`  (int)sts);`
	`if (sts.status() != ASC_success) exit(1);`
		`// install the probe in function entry point`
		`points[0] = mypoint;`
		`cbs[0] = data_cb;`
		`tags[0] = (GCBTagType)1;`
		`sts = A.binstall_probe(1, &fullexp, points, cbs,`
		`  tags, phds);`
		`printf("install_probe is done, status = %d\n",`
		`  (int)sts);`
		`if (sts.status() != ASC_success) exit(1);`

For more information on the `Process::install_probe`, `Process::binstall_probe`, `Application::install_probe`, or `Application::binstall_probe`, refer to their AIX man pages or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*

Step 4: Activate probe

Once you have installed a point probe in one or more target application processes, you must activate it so that it will execute as part of the target application process whenever execution reaches its installed location in the target application code. To do this on a single process basis, the analysis tool can use the asynchronous function `Process::activate_probe` or its blocking equivalent `Process::bactivate_probe`. To do this on an application-wide basis (for all `Process` objects managed by an `Application` object), the analysis tool can use the function

Application::activate_probe or Application::bactivate_probe. Note that all of these functions accept an array of probe handles. This means that the analysis tool can activate more than one point probe using a single call.

Table 47. Activating a point probe in one or more target application processes

To activate a probe:	In a single process (Process object)	In multiple processes (all of the Process objects managed by an Application object)

Using the asynchronous function activate_probe	`void activate_probe_cb(GCBSysType sys,`
	`  GCBTagType tag, GCBTagType obj,`
	`  GCBMsgType msg);`
	
	`// ...`
	
	`// activate the probe`
	`async_done = 0;`
	`sts = P.activate_probe(1, phds,`
	`  (GCBFuncType)activate_probe_cb,`
	`  (GCBTagType)4);`
	`printf("activate_probe is submitted, \`
	`status = %d\n", (int)sts);`
	`if (sts.status() != ASC_success) exit(1);`
	
	`// ...`
	
	`void`
	`activate_probe_cb(GCBSysType sys,`
	`  GCBTagType tag, GCBTagType obj,`
	`  GCBMsgType msg)`
	`{`
	`  Process *p = (Process *)obj;`
	`  AisStatus *sts = (AisStatus *)msg;`
	
	`  async_done = 1;`
	`  printf("task %d activate_probe is \`
	`completed, status = %d\n", p->get_task(),`
	`  (int)*sts);`
	`  Ais_end_main_loop();`
	`}`
		`void activate_probe_cb(GCBSysType sys,`
		`  GCBTagType tag, GCBTagType obj,`
		`  GCBMsgType msg);`
		
		`// ...`
		
		`// activate the probe`
		`async_done = 0;`
		`sts = A.activate_probe(1, phds,`
		`  (GCBFuncType)activate_probe_cb,`
		`  (GCBTagType)4);`
		`printf("activate_probe is submitted, \`
		`status = %d\n", (int)sts);`
		`if (sts.status() != ASC_success) exit(1);`
		
		`// ...`
		
		`void`
		`activate_probe_cb(GCBSysType sys,`
		`  GCBTagType tag, GCBTagType obj,`
		`  GCBMsgType msg)`
		`{`
		`  static int count = 0;`
		`  Process *p = (Process *)obj;`
		`  AisStatus *sts = (AisStatus *)msg;`
		
		`  printf("task %d activate_probe is \`
		`completed, status = %d\n", p->get_task(),`
		`  (int)*sts);`
		`  count++;`
		`  if (count >= num_procs)`
		`  {`
		`  async_done = 1;`
		`  Ais_end_main_loop();`
		`  }`
		`}`

Using the blocking function bactivate_probe	`// activate the probe`
	`sts = P.bactivate_probe(1, phds);`
	`printf("activate_probe is done, \`
	`status = %d\n", (int)sts);`
	`if (sts.status() != ASC_success) exit(1);`
	`// enter main loop`
	`Ais_main_loop();`
	`printf("----- The End -----\n");`
		`// activate the probe`
		`sts = A.bactivate_probe(1, phds);`
		`printf("activate_probe is done, \`
		`status = %d\n", (int)sts);`
		`if (sts.status() != ASC_success) exit(1);`
		`// enter main loop`
		`Ais_main_loop();`
		`printf("----- The End -----\n");`

For more information on the Process::activate_probe, Process::bactivate_probe, Application::activate_probe, or Application::bactivate_probe functions, refer to their AIX man pages or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Example: Installing and activating a point probe

The following example code:

1. navigates the source structure of a parallel application to identify an instrumentation point.
2. installs a point probe at the instrumentation point.
3. activates the point probe.

```
#include <dpcl.h>
void data_cb   (GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg);

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

main(int argc, char *argv[])
{

    PoeAppl        A; // A represents a POE application

    Ais_initialize();

    AisStatus sts = A.binit_procs(argv[1], atoi(argv[2]));
    printf ("read config status = %d\n", (int)sts);

    sts = A.bconnect();

    printf ("connect status = %d\n", (int)sts);

    Process  P = A.get_process(0);

    SourceObj myprog = P.get_program_object();

    SourceObj mymod;

    const int   bufSize = 128;
    char        bufmname[bufSize];      // buffer for module_name(..)

    for (int c = 0; c < myprog.child_count(); c++)
    {
        mymod = myprog.child(c);

        char * modname = mymod.module_name(bufmname,bufSize);

        if (strcmp(modname, "chaotic.f") ==0)
        {
            printf("Module chaotic.f found... expanding\n");

            sts = mymod.bexpand(P);

            printf ("expand status = %d\n", (int)sts);

            break;
        }
    }

    SourceObj myfun;

    char        bufdname[bufSize];   // buffer for get_demangled_name(..)

    for ( c = 0; c < mymod.child_count(); c++)
    {
        myfun = mymod.child(c);

        char * funname = myfun.get_demangled_name(bufdname,bufSize);

        if (strcmp(funname, "exchange") ==0)
        {
            printf("function exchange found... \n");

            break;
```

```
        }
    }

    InstPoint  mypoint;

    for ( c = 0; c < myfun.exclusive_point_count(); c++)
    {
        mypoint = myfun.exclusive_point(c);

        if ( mypoint.get_type() == IPT_function_entry)
        {
            printf("Found function entry point\n");

            break;
         }
    }

    // malloc pcount;

    int           val = 0;
    ProbeExp pcount = A.balloc_mem( int32_type(), &val, sts);

    printf ("malloc  pcount status = %d\n", (int)sts);

    ProbeExp addexp = pcount.assign(pcount + ProbeExp(1));

                              // Create the expression parameters
    ProbeExp parms[3];

    parms[0] = Ais_msg_handle;
    parms[1] = pcount.address();
    parms[2] = ProbeExp (4);

    ProbeExp sendexp = Ais_send.call(3,parms);

    ProbeExp fullexp = addexp.sequence(sendexp);

    ProbeHandle   myph;
    GCBFuncType   cbs[] = {data_cb};
    GCBTagType    tags[] = {(GCBTagType)0};


    sts = A.binstall_probe(1, &fullexp, &mypoint,
                           cbs, tags,
                           &myph);

    printf ("install status = %d\n",(int)sts);

    sts = A.bactivate_probe(1, &myph);

    printf ("activate status = %d\n", (int)sts);


    Ais_main_loop();
}

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void
data_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg)
{
   Process *p = (Process *)obj;
   int *i = (int *)msg;

   printf("Task %d sent the count %d\n", p->get_task(),*i);

   // terminate the program after 10 messages are received

   if (*i == 10) {
       Ais_end_main_loop();
   }
}
```

```
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Executing phase probes

Phase probes are probes that are executed periodically upon execution of a timer, regardless of what part of the target application's code is executing. The interval at which the probes are executed is measured in CPU time (as opposed to wall-clock time), and is called the *phase period*. The control mechanism for invoking phase probes at these set CPU-time intervals is called a "*phase*". Represented by instances of the `Phase` class, phases enable your analysis tool code to specify the particular phase probe(s) to be invoked and the interval at which their execution is triggered. Note that the DPCL system uses a SIGPROF signal to activate a phase; target applications that themselves use the SIGPROF signal cannot be instrumented with phases. For more information on phase probes and phases, refer to “What is a phase probe?” on page 19.

To add a phase to one or more target application processes in order to have it call its associated phase probes (probe module functions) at set CPU-time intervals, the analysis tool:

1. Creates one or more probe modules containing the function(s) to be triggered by the phase. A phase can, each time its interval expires, call up to three probe module functions — a phase begin function, a phase data function, and a phase end function. In addition to these probes module functions that are executed each time the phase is activated, the analysis tool can also create:
   - an initialization function to be executed only when the phase is first added to a target application process.
   - Up to three phase exit functions (the phase exit begin function, the phase exit data function, and the phase exit end function) to be executed when the phase is removed from the target application process. Upon phase removal, the phase exit begin function will execute, followed by the phase exit data function (which will execute once for each datum associated with the phase), followed by the phase exit end function.
2. If using a data function, allocates and associates data with the phase's data function. Each time the phase is triggered, the data function executes once per datum that the analysis tool has previously allocated and associated with the phase.
3. For each of the phase probes (probe module functions) to be used by the phase, creates a probe expression that represents a reference to the desired function. This not only includes the phase begin function, phase data function, and phase end function, but also any initialization function or phase exit functions.
4. Creates a `Phase` class object that defines the phase probe(s) (probe module function(s)) to be executed by the phase, and the interval between successive invocations of the phase.
5. Calls a function to add the phase to one or more target application processes. If you want an initialization function to be executed when the phase is first added to the target application process, you supply the function that adds the phase with a probe expression that represents a reference to the initialization function.

6. Calls a function that specifies up to three exit functions (begin, data, and end) to be executed when the phase is removed from the target application process. You supply this function with probe expressions representing the desired phase exit functions.
7. If desired, modifies the phase period (the CPU-time interval at which the installed phase is activated to execute its probes).

Step 1: Create probe module(s)

The first step in creating a phase structure to execute phase probes is to create one or more probe modules that contain the functions to be triggered by the phase. A phase can, each time its interval expires, call up to three probe module functions — a begin function, a data function, and an end function. While the phase must, in order to be useful, call at least one of these functions, any one of them is optional. At the very least, an analysis tool will usually supply a data function.

Once the phase is activated, it will call the phase probe module functions that have been associated with it. The first phase probe it calls is the one identifying the begin function (provided one has been specified). Typically, the begin function will perform any setup tasks that may be required. When the begin function completes, the phase calls the phase probe that identifies the data function (provided one has been specified). The data function executes once per datum that the analysis tool will have previously allocated and associated with this phase. When the data function finishes executing for the last datum, the phase calls the phase probe that identifies the end function (provided one has been specified). Typically, the end function performs any clean up chores that may be required.

In addition to these probes module functions that are executed each time the phase is activated, the analysis tool can also create:

- an initialization function to be executed only when the phase is first added to a target application process.
- Up to three phase exit functions (the phase exit begin function, the phase exit data function, and the phase exit end function) to be executed when the phase is removed from the target application process. These functions will be executed when:
  - the analysis tool explicitly sends a request to have the phase removed (using the `Process::remove_phase`, `Process::bremove_phase`, `Application::remove_phase`, or `Application::bremove_phase` function).
  - the analysis tool disconnects from the target application process (using the `Process::disconnect`, `Process::bdisconnect`, `Application::disconnect`, or `Application::bdisconnect` function) without first explicitly requesting to have the phase removed.
  - The target application process finishes executing while the indicated phase is still active.

  When the phase exit functions are triggered, the first phase probe it calls is the one identifying the phase exit begin function (provided one has been specified). Like the phase begin function, the phase exit begin function will perform any setup tasks that may be required for the data and end functions to follow. When the phase exit begin function completes, the DPCL system executes the phase probe that identifies the phase exit data function (provided one has been specified). The phase exit data function, like the phase data function, executes

once per datum that the analysis tool will have previously allocated and associated with this phase. When the phase exit data function finishes executing for the last datum, the DPCL system calls the phase probe that identifies the phase exit end function (provided one has been specified). Typically, the end function performs any final clean up chores that may be required. Like the phase functions, however, any of these phase exit functions is optional.

The following example code shows a probe module that defines the phase begin function, the phase data function, and the phase end function. It also defines an initialization function to be executed when the phase is first added to a target application process. In order to execute this module's functions in one or more target application process, the module will need to be compiled and then loaded into the target application process. For more information on how to do this, refer to “Creating and calling probe module functions” on page 136.

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/param.h>
#include <dpclExt.h>

static int   visit = 0;
static char msg[MAXPATHLEN];
static char msg_loc[MAXPATHLEN];
static char msg_time[MAXPATHLEN];
static int   msg_size;

void
init_func(void *handle)
{
   sprintf(msg, "init_func() started\n");
   msg_size = strlen(msg) + 1;
   Ais_send(handle,msg,msg_size) ;
}

void
begin_func(void *handle)
{
  int         rc;

    sprintf(msg, "begin_func() invoked");
    msg_size = strlen(msg) + 1;

    if ((rc = Ais_send(handle, msg, msg_size)) != 0) {
        printf("begin_func(): ERROR, Ais_send()=%d\n", rc);
    }

    printf("begin_func() called\n");
}



void
data_func(void *handle, void *_data)
{
    int     rc;

    int *data = (int *)_data;

    *data = *data + 1;


    sprintf(msg, "data_func(): data = %d", *data);
    msg_size = strlen(msg) + 1;

    if ((rc = Ais_send(handle, msg, msg_size)) != 0) {
         printf("data_func(): ERROR, Ais_send()=%d\n", rc);
    }
```

```
    printf("data_func() called\n");
}

void
end_func(void *handle)
{
    int            rc;

    sprintf(msg, "end_func() invoked\n");
    msg_size = strlen(msg) + 1;

    if ((rc = Ais_send(handle, msg, msg_size)) != 0) {
         printf("end_func(): ERROR, Ais_send()=%d\n", rc);
    }

    printf("end_func() called\n");
}
```

Step 2: Create probe expression(s) to reference the probe module function(s)

When creating an instance of the `Phase` class (as described next in "Step 3: Create phase" on page 157), the analysis tool must, for each phase probe (module function) that the `Phase` will trigger, create a probe expression that represents a reference to the function. In order to have loaded the probe module into the target application process(es), the analysis tool will have already created a `ProbeModule` class object to represent the probe module. (For more information, see "Creating and calling probe module functions" on page 136.) To create a probe expression that represents a reference to one of the probe module's functions, the analysis tool can call the `Probe_Module::get_reference` function. The analysis tool code provides this function with the index of the function within the probe module. To determine the number of functions in the module, the analysis tool can call the `ProbeModule::get_count` function. To determine the name of a particular function, the analysis tool can call the `ProbeModule::get_name` function.

This following code example uses the `ProbeModule::get_count`, `ProbeModule::get_name`, and `ProbeModule::get_reference` functions to create probe expressions that represent references to all the probe module functions shown in the example code in "Step 1: Create probe module(s)" on page 154. The phase begin function (`begin_func`), the phase data function (`data_func`) and the phase end function (`end_func`) will all be used when instantiating the Phase class in the next step. The initialization function (`init_func`) is the one to be executed only when the phase is first added to a target application process; the probe expression that represents a reference to this function will be used as a function parameter to the `Process::add_phase`, `Process::badd_phase`, `Application::add_phase`, or `Application::badd_phase`. These functions are described in "Step 4: Add phase to the target application process(es)" on page 157.

```
                              // look for functions in the loaded module
                              // that are to be used in phase
    for (int j = 0; j < load_mod.get_count(); j++) {
        if (strcmp("init_func",
           load_mod.get_name(j, buffer, sizeof(buffer))) == 0)
               {
                 init_func = load_mod.get_reference(j);
               }
        else if (strcmp("begin_func",
                load_mod.get_name(j, buffer, sizeof(buffer))) == 0)
                   {
                     begin_func = load_mod.get_reference(j);
                    }
        else if (strcmp("data_func",
                load_mod.get_name(j, buffer, sizeof(buffer))) == 0)
```

```
                {
                  data_func = load_mod.get_reference(j);
                 }
        else if (strcmp("end_func",
                load_mod.get_name(j, buffer, sizeof(buffer))) == 0)
                {
                  end_func = load_mod.get_reference(j);
                }
    }
```

For more information on the `ProbeModule::get_count`, `ProbeModule::get_name`, and `ProbeModule::get_reference` functions, refer to their AIX man pages or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Step 3: Create phase

A phase structure (`Phase` class object) defines the phase probe(s) (probe module function(s)) to be executed and the interval between successive invocations of these probes. The `Phase` class is defined in the header file `Phase.h`. This following example code creates a phase object that specifies the phase should be activated every second of CPU time to execute the phase begin function (`begin_func`), the phase data function (`data_func`) and the phase end function (`end_func`).

```
// create a phase with one second period
float period = 1.0;
myphase = Phase(period,
              begin_func, (GCBFuncType)data_cb, (GCBTagType)1,
              data_func, (GCBFuncType)data_cb, (GCBTagType)2,
              end_func, (GCBFuncType)data_cb, (GCBTagType)3);
```

Although the phase period (the CPU-time interval at which the phase is activated to execute its probes) is initially set when the analysis tool defines the phase, note that the analysis tool can later lengthen or shorted this interval as desired. Refer to “Step 7: Modify phase period” on page 159 for more information.

For more information on the `Phase` class and its constructors, refer to the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Step 4: Add phase to the target application process(es)

Once the analysis tool has created the `Phase` object (as described in “Step 3: Create phase”), it can add it to one or more target application processes. To add a phase on a single process basis, the analysis tool can use the asynchronous function `Process::add_phase` or its blocking equivalent `Process::badd_phase`. To add a phase on an application-wide basis (for all `Process` class objects managed by an `Application` object), the analysis tool can use the functions `Application::add_phase` or `Application::badd_phase`. Optional parameters of these four functions enable you to specify an initialization function to be executed when the phase is added to a target application process, as well as a data callback routine and callback tag for handling message data generated by the initialization function. The following code example uses the `Process::badd_phase` function to add the phase we created in “Step 3: Create phase” to a single target application process. The initialization function shown in the probe module in “Step 1: Create probe module(s)” on page 154 will execute when the `Phase` is added to the process. The probe expression `init_func` represents a reference to the probe module function; we created this probe expression in “Step 5: Create probe expression(s) to allocate and associate data with the phase” on page 158.

```
// add the phase to application
sts = P.badd_phase(myphase,
          init_func, (GCBFuncType)data_cb, (GCBTagType)4);
if (sts.status()==ASC_success){
   printf("badd_phase() was successful\n");
} else {
   printf("badd_phase() FAILED.. %s\n",sts.status_name());
   exit(0);
}
```

For more information on the `Process::add_phase`, `Process::badd_phase`, `Application::add_phase`, or `Application::badd_phase` functions, refer to their AIX man pages or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Step 5: Create probe expression(s) to allocate and associate data with the phase

If using a phase data function, the analysis tool code will need to create a probe expression to allocate and associate data with the phase. Each time the phase is triggered, the data function executes once per datum that the analysis tool has previously allocated and associated with the phase. Executing once per datum enables the data function to perform the same actions on the different data. Each datum, for example, could be a separate counter — each incremented by the same data function. If the analysis tool does not associate any data with the phase, then the data function will not execute.

"Creating probe expressions to represent persistent data" on page 120 describes how you can use the `Process::alloc_mem`, `Process::balloc_mem`, `Application::alloc_mem`, and `Application::balloc_mem` to allocated memory in target application processes. What this earlier section did not state, however, is the an optional parameter of these functions enables the analysis tool to associate the data allocated in the process(es) with a particular phase. In this following example of the `Process::balloc_mem` function, the probe expression `phase_da` is created to represent a persistent integer variable with the initial value of 1000. This probe expression also associates the allocated data with the `Phase` object `myphase` created in "Step 3: Create phase" on page 157. Each time this phase is activated, this data value will be passed to the phase data function `data_func`. Since there is only this one datum associated with the phase, the phase data function will execute only once each time the phase is activated.

```
// create variable to be used in the phase
int init_value = 1000;
ProbeExp phase_da = P.balloc_mem(int32_type(), (void *)&init_value,
                                 myphase, sts);
if (sts.status()==ASC_success){
   printf("balloc_mem() was successful\n");
} else {
   printf("balloc_mem() FAILED.. %s\n",sts.status_name());
   exit(0);
}
```

For more information on the `Process::alloc_mem`, `Process::balloc_mem`, `Application::alloc_mem`, and `Application::balloc_mem` functions, refer to their AIX man pages or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Step 6: Specify phase exit functions

If the analysis tool code created one or more phase exit functions in “Step 1: Create probe module(s)” on page 154, it needs to associate them with the `Phase` on one or more target application processes. To do this, the analysis tool code calls the `Process::set_phase_exit`, `Process::bset_phase_exit`, `Application::set_phase_exit`, or `Application::bset_phase_exit` function. The analysis tool code must supply these functions with probe expressions that represent references to the actual probe module functions. The analysis tool code must, therefore, have already created probe expressions for the exit functions as described in “Step 5: Create probe expression(s) to allocate and associate data with the phase” on page 158. The `set_phase_exit` and `bset_phase_exit` functions also allow the analysis tool to specify a callback routine and callback tag for each of the phase exit functions. The following code uses the `Process::bset_phase_exit` function call to associate a phase exit begin function, a phase exit data function, and a phase exit end function with the `Phase` object `phase1` on one particular target application process. The three phase exit functions are represented by the probe expressions `exit_begin_func`, `exit_data_func`, and `exit_end_func` previously created by calling the `ProbeModule::get_reference`. The procedure for creating a probe expression using the `ProbeModule::get_reference` function is described in “Step 5: Create probe expression(s) to allocate and associate data with the phase” on page 158.

```
sts = P.bset_phase_exit(myphase,
          exit_begin_func, (GCBFuncType)msg_cb, (GCBTagType)6,
          exit_data_func, (GCBFuncType)msg_cb, (GCBTagType)7,
          exit_end_func, (GCBFuncType)msg_cb, (GCBTagType)8);
```

For more information on the `Process::set_phase_exit`, `Process::bset_phase_exit`, `Application::set_phase_exit`, or `Application::bset_phase_exit` functions, refer to their AIX man pages or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Step 7: Modify phase period

A phase period specifies the interval of CPU time at which a phase is activated. When the phase is activated, it then executes its phase probes (phase begin, phase data, and phase end functions). The analysis tool initially sets the phase period when defining the `Phase` class object as described in “Step 3: Create phase” on page 157. The analysis tool can also, once the `Phase` class object has been added to one or more target application processes, modify the phase period so that the phase is activated at longer or shorter intervals of CPU time. The analysis tool can reset a phase period within a single target application process by calling the `Process::set_phase_period` function or its blocking equivalent — `Process::bset_phase_period`. The analysis tool can also reset a phase period on an application-wide basis (for each `Process` object managed by an `Application` object) by calling the `Application::set_phase_period` or `Application::bset_phase_period` function.

One use of the `Process::set_phase_period`, `Process::bset_phase_period`, `Application::set_phase_period`, and `Application::bset_phase_period` functions is to control when a phase will first execute. For example, say you do not want a phase to be activated until after the analysis tool has allocated and associated data with the phase. Unfortunately, you have to add the phase to one or more processes (as described in “Step 4: Add phase to the target application process(es)” on page 157) before you can allocate data for it (as described in “Step 5: Create

probe expression(s) to allocate and associate data with the phase" on page 158). In the example code below, the analysis tool adds a phase whose phase period is 999.0 to a particular process. The long phase period effectively places the phase in a suspended state while the analysis tool allocates and associates data with the phase. Once the data allocation step is complete, the analysis tool calls the `Process::set_phase_period` function to set the phase period to the intended CPU-time interval of 0.1 seconds

```
                        // create a phase object with a large period
                        // to effectively suspend its execution
period = 999.0;
try
{
    phase1 = Phase(period,
                begin_func, (GCBFuncType)msg_cb, (GCBTagType)1,
                   data_func, (GCBFuncType)msg_cb, (GCBTagType)2,
                end_func, (GCBFuncType)msg_cb, (GCBTagType)3);
}
catch (AisStatus excp)
{
    printf("new Phase failed with sts=%s\n", excp.status_name());
    exit(1);
}
                        // add the phase to target application
sts = P.badd_phase(phase1,
                init_func, (GCBFuncType)msg_cb, (GCBTagType)4);
printf("add_phase is done, status = %d\n", (int)sts);
if (sts.status() != ASC_success) exit(1);
                        // set the phase's exit functions
sts = P.bset_phase_exit(phase1,
                     exit_begin_func, (GCBFuncType)msg_cb, (GCBTagType)6,
                    exit_data_func, (GCBFuncType)msg_cb, (GCBTagType)7,
                    exit_end_func, (GCBFuncType)msg_cb, (GCBTagType)8);
printf("set_phase_exit is done, status = %d\n", (int)sts);
if (sts.status() != ASC_success) exit(1);
                        // allocate data variables for the phase
value = 0;
pcount = P.balloc_mem(int32_type(), &value, phase1, sts);
printf("alloc_mem is done, status = %d\n", (int)sts);
if (sts.status() != ASC_success) exit(1);

value = 100;
pcount2 = P.balloc_mem(int32_type(), &value, phase1, sts);
printf("alloc_mem is done, status = %d\n", (int)sts);
if (sts.status() != ASC_success) exit(1);
                        // set the phase to its true execution interval
                        // to resume its normal operation
period = 0.1;
sts = P.bset_phase_period(phase1, period);
printf("set_phase_period is done, status = %d\n", (int)sts);
if (sts.status() != ASC_success) exit(1);
                        // query the phase's execution interval
interval = P.get_phase_period(phase1, sts);
printf("phase period = %f\n", interval);
```

In addition to being able to set a phase period, the analysis tool can also ascertain the value of a phase period by calling the `Process::get_phase_period` function. For more information on the `Process::set_phase_period`, `Process::bset_phase_period`, `Application::set_phase_period`, `Application::bset_phase_period`, or `Process::get_phase_period` functions, refer to their AIX man pages, or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Example: Executing phase probes

The following example code:

1. Loads a probe module into a target application process.
2. Creates probe expressions that reference functions in the probe module.
3. Creates a phase using the probe expressions that represent probe module functions. The phase period is initially set to a high number (999.0) to effectively suspend activation of the phase until data has been allocated for it.
4. Adds the phase to the target application process.
5. Allocates data variables for the phase.
6. Resets the phase period so that the phase is executed every 0.1 second of CPU time.

```
#include <stdio.h>
#include <stdlib.h>

#include <dpcl.h>

void msg_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg);

void
main(int argc, char *argv[])
{
    Process P;
    AisStatus sts;
    int c;
    int value;
    ProbeExp pcount;
    ProbeExp pcount2;
    ProbeExp init_func;
    ProbeExp begin_func;
    ProbeExp data_func;
    ProbeExp end_func;
    ProbeExp exit_begin_func;
    ProbeExp exit_data_func;
    ProbeExp exit_end_func;
    ProbeModule load_1;
    float period;
    float interval;
    Phase phase1;
    const int bufsize = 256;
    char buffer[bufsize];
    char *name;

                            // initialize DPCL environment
    Ais_initialize();
                            // construct a valid Process object
    P = Process(argv[1], atoi(argv[2]));
                            // connect to the target application
    sts = P.bconnect();
    printf("connect is done, status = %d\n", (int)sts);
    if (sts.status() != ASC_success) exit(1);
                            // load probe module
    load_1 = ProbeModule("./load_1");
    sts = P.bload_module(&load_1);
    printf("load_module is done, status = %d\n", (int)sts);
    if (sts.status() != ASC_success) exit(1);
                            // look for all the phase related functions
    for (c = 0;c < load_1.get_count(); c++)
    {
        name = load_1.get_name(c, buffer, bufsize);
        if (strcmp("init_func", name) == 0)
        {
          init_func = load_1.get_reference(c);
          printf("found init_func\n");
      }
```

```
  else if (strcmp("begin_func", name) == 0)
    {
      begin_func = load_1.get_reference(c);
      printf("found begin_func\n");
  }
  else if (strcmp("data_func", name) == 0)
    {
      data_func = load_1.get_reference(c);
      printf("found data_func\n");
  }
  else if (strcmp("end_func", name) == 0)
    {
      end_func = load_1.get_reference(c);
      printf("found end_func\n");
  }
  else if (strcmp("exit_begin_func", name) == 0)
    {
      exit_begin_func = load_1.get_reference(c);
      printf("found exit_begin_func\n");
  }
  else if (strcmp("exit_data_func", name) == 0)
    {
      exit_data_func = load_1.get_reference(c);
      printf("found exit_data_func\n");
  }
  else if (strcmp("exit_end_func", name) == 0)
    {
      exit_end_func = load_1.get_reference(c);
      printf("found exit_end_func\n");
  }
}
                        // create a phase object with a large period
                        // to effectively suspend its execution
period = 999.0;
try
{
    phase1 = Phase(period,
                begin_func, (GCBFuncType)msg_cb, (GCBTagType)1,
                    data_func, (GCBFuncType)msg_cb, (GCBTagType)2,
                end_func, (GCBFuncType)msg_cb, (GCBTagType)3);
}
catch (AisStatus excp)
{
    printf("new Phase failed with sts=%s\n", excp.status_name());
    exit(1);
}
                        // add the phase to target application
sts = P.badd_phase(phase1,
               init_func, (GCBFuncType)msg_cb, (GCBTagType)4);
printf("add_phase is done, status = %d\n", (int)sts);
if (sts.status() != ASC_success) exit(1);
                        // set the phase's exit functions
sts = P.bset_phase_exit(phase1,
                    exit_begin_func, (GCBFuncType)msg_cb, (GCBTagType)6,
                  exit_data_func, (GCBFuncType)msg_cb, (GCBTagType)7,
                  exit_end_func, (GCBFuncType)msg_cb, (GCBTagType)8);
printf("set_phase_exit is done, status = %d\n", (int)sts);
if (sts.status() != ASC_success) exit(1);
                        // allocate data variables for the phase
value = 0;
pcount = P.balloc_mem(int32_type(), &value, phase1, sts);
printf("alloc_mem is done, status = %d\n", (int)sts);
if (sts.status() != ASC_success) exit(1);

value = 100;
pcount2 = P.balloc_mem(int32_type(), &value, phase1, sts);
printf("alloc_mem is done, status = %d\n", (int)sts);
if (sts.status() != ASC_success) exit(1);
                        // set the phase to its true execution interval
                        // to resume its normal operation
period = 0.1;
sts = P.bset_phase_period(phase1, period);
```

```
    printf("set_phase_period is done, status = %d\n", (int)sts);
    if (sts.status() != ASC_success) exit(1);
                            // query the phase's execution interval
    interval = P.get_phase_period(phase1, sts);
    printf("phase period = %f\n", interval);
                            // remove the phase from target application
    sts = P.bremove_phase(phase1);
    printf("remove_phase is done, status = %d\n", (int)sts);
    if (sts.status() != ASC_success) exit(1);

    //Ais_main_loop();
    printf("----- The End -----\n");
}

void
msg_cb(GCBSysType sys, GCBTagType tag, GCBTagType obj, GCBMsgType msg)
{
    static int count = 0;

    count++;
    char *chp = (char *)msg;
    printf("msg_cb received the msg(%d)=\"", count);
    for (int i = 0; i < sys.msg_size; ++i)
    {
        printf("%c", chp[i]);
    }
    printf("\"\n");
}
```

Executing one-shot probes

A one-shot probe is a type of probe that is executed by the DPCL system immediately upon request, regardless of what the application happens to be doing. To execute a one-shot probe in one or more target application processes, the analysis tool must:

1. define the probe as described in Chapter 8, “Creating probes” on page 115.
2. execute the probe.

Step 1: Create one-shot probe

A probe is a probe expression that may optionally call functions. The first step in executing a one-shot probe is to build the actual probe expression that will serve as the one-shot probe. Since a one-shot probe is executed immediately upon request, regardless of what the target application happens to be doing, your probe expression should be "signal safe".

For detailed instructions on creating probe expressions, refer to Chapter 8, “Creating probes” on page 115.

Step 2: Execute the one-shot probe

Once the analysis tool has defined the probe expression that will serve as the one-shot probe, it can execute it in one or more target application processes. To execute a one-shot probe in a single process, the analysis tool can use the asynchronous function `Process::execute` or its blocking equivalent `Process::bexecute`. To execute a one-shot probe on an application-wide basis (for all `Process` objects managed by an `Application` object), the analysis tool can use the function `Application::execute` or `Application::bexecute`.

Table 48. Executing a one-shot probe in one or more target application processes

To execute a one-shot probe:	**In a single process (`Process` object)**	**In multiple processes (all of the `Process` objects managed by an `Application` object)**

Using the asynchronous function `execute_probe`	`void execute_cb(GCBSysType sys, GCBTagType tag,`
	`  GCBTagType obj, GCBMsgType msg);`
	
	`// ...`
	
	`addexp = pcount.assign(pcount + ProbeExp(1));`
	
	`// create a probe expression to send the result`
	`// back to DPCL program`
	`parms[0] = Ais_msg_handle;`
	`parms[1] = pcount.address();`
	`parms[2] = ProbeExp(4);`
	`sendexp = Ais_send.call(3, parms);`
	`fullexp = addexp.sequence(sendexp);`
	
	`// issue an one-shot probe`
	`sts = P.execute(fullexp, (GCBFuncType)data_cb,`
	`  (GCBTagType)1, (GCBFuncType)execute_cb,`
	`  (GCBTagType)2);`
	`printf("execute is submitted, status = %d\n",`
	`  (int)sts);`
	`if (sts.status() != ASC_success) exit(1);`
	
	`// enter main loop`
	`Ais_main_loop();`
	`printf("----- The End -----\n");`
	
	`// ...`
	
	`void`
	`execute_cb(GCBSysType sys, GCBTagType tag,`
	`  GCBTagType obj, GCBMsgType msg)`
	`{`
	`  Process *p = (Process *)obj;`
	`  AisStatus *sts = (AisStatus *)msg;`
	
	`  printf("task %d execute is completed, \`
	`status = %d\n", p->get_task(),`
	`  (int)*sts);`
	`  Ais_end_main_loop();`
	`}`
		`void execute_cb(GCBSysType sys, GCBTagType tag,`
		`  GCBTagType obj, GCBMsgType msg);`
		
		`// ...`
		
		`addexp = pcount.assign(pcount + ProbeExp(1));`
		
		`// create a probe expression to send the result`
		`// back to DPCL program`
		`parms[0] = Ais_msg_handle;`
		`parms[1] = pcount.address();`
		`parms[2] = ProbeExp(4);`
		`sendexp = Ais_send.call(3, parms);`
		`fullexp = addexp.sequence(sendexp);`
		
		`// issue an one-shot probe`
		`sts = A.execute(fullexp, (GCBFuncType)data_cb,`
		`  (GCBTagType)1, (GCBFuncType)execute_cb,`
		`  (GCBTagType)2);`
		`printf("execute is submitted, status = %d\n",`
		`  (int)sts);`
		`if (sts.status() != ASC_success) exit(1);`
		
		`// enter main loop`
		`Ais_main_loop();`
		`printf("----- The End -----\n");`
		
		`// ...`
		
		`void`
		`execute_cb(GCBSysType sys, GCBTagType tag,`
		`  GCBTagType obj, GCBMsgType msg)`
		`{`
		`  static int count = 0;`
		`  Process *p = (Process *)obj;`
		`  AisStatus *sts = (AisStatus *)msg;`
		
		`  count++;`
		`  printf("task %d execute is completed, \`
		`status = %d\n", p->get_task(),`
		`  (int)*sts);`
		`  if (count >= num_procs)`
		`  {`
		`  Ais_end_main_loop();`
		`  }`
		`}`

Using the blocking function `bexecute_probe`	`// create an assignment statement`
	`addexp = pcount.assign(pcount + ProbeExp(1));`
	
	`// create a probe expression to send the result`
	`// back to DPCL program`
	`parms[0] = Ais_msg_handle;`
	`parms[1] = pcount.address();`
	`parms[2] = ProbeExp(4);`
	`sendexp = Ais_send.call(3, parms);`
	`fullexp = addexp.sequence(sendexp);`
	
	`// issue an one-shot probe`
	`sts = P.bexecute(fullexp, (GCBFuncType)data_cb,`
	`  (GCBTagType)1);`
	`printf("execute is done, status = %d\n",`
	`  (int)sts);`
	`if (sts.status() != ASC_success) exit(1);`
	
	`printf("----- The End -----\n");`
		`// create an assignment statement`
		`addexp = pcount.assign(pcount + ProbeExp(1));`
		
		`// create a probe expression to send the result`
		`// back to DPCL program`
		`parms[0] = Ais_msg_handle;`
		`parms[1] = pcount.address();`
		`parms[2] = ProbeExp(4);`
		`sendexp = Ais_send.call(3, parms);`
		`fullexp = addexp.sequence(sendexp);`
		
		`// issue an one-shot probe`
		`sts = A.bexecute(fullexp, (GCBFuncType)data_cb,`
		`  (GCBTagType)1);`
		`printf("execute is done, status = %d\n",`
		`  (int)sts);`
		`if (sts.status() != ASC_success) exit(1);`
		
		`printf("----- The End -----\n");`

For more information on the `Process::execute`, `Process::bexecute`, `Application::execute`, and `Application::bexecute` functions, refer to their AIX man pages, or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Example: Executing a one-shot probe

The following example code builds a probe expression and executes it as a one-shot probe in a single application process.

```
#include <stdio.h>
#include <stdlib.h>

#include <dpcl.h>

void data_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg);

void
main(int argc, char *argv[])
{
    Process P;
    AisStatus sts;
    int c;
    int value;
    ProbeExp pcount;
    ProbeExp addexp;
    ProbeExp sendexp;
    ProbeExp fullexp;
    ProbeExp parms[5];

                              // initialize DPCL environment
    Ais_initialize();
                              // construct a valid Process object
    P = Process(argv[1], atoi(argv[2]));
                              // connect to the target application
    sts = P.bconnect();
    printf("connect is done, status = %d\n", (int)sts);
    if (sts.status() != ASC_success) exit(1);
                              // allocate a data variable with init value = 0
    value = 0;
    pcount = P.balloc_mem(int32_type(), &value, sts);
    printf("alloc_mem is done, status = %d\n", (int)sts);
    if (sts.status() != ASC_success) exit(1);
                              // create an assignment statement
    addexp = pcount.assign(pcount + ProbeExp(1));
                              // create a probe expression to send the result
                              // back to DPCL program
    parms[0] = Ais_msg_handle;
    parms[1] = pcount.address();
    parms[2] = ProbeExp(4);
    sendexp = Ais_send.call(3, parms);
    fullexp = addexp.sequence(sendexp);
                              // issue an one-shot probe
    sts = P.bexecute(fullexp, (GCBFuncType)data_cb, (GCBTagType)1);
    printf("execute is done, status = %d\n", (int)sts);
    if (sts.status() != ASC_success) exit(1);

    printf("----- The End -----\n");
}

void
data_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg)
{
    Process *p = (Process *)obj;
    int *i = (int *)msg;
    printf("task %d sent the count %d\n", p->get_task(), *i);
}
```

Chapter 10. Creating data callback routines

As described in “Step 2e: Create probe expressions to represent function calls” on page 130, a probe expression can send data back to the analysis tool by calling the DPCL system-defined function `Ais_send`. “Step 1: Create probe module function” on page 137 also describes this built-in function, and shows how it can be called within a probe module function. If the probe expression created by the analysis tool calls the `Ais_send` function (or calls a probe module function that in turn calls `Ais_send`), then the analysis tool code must contain a data callback function that will handle the data that the `Ais_send` function will send.

At run time, the callback function will be invoked once for each data message sent from the probe. What's more, if the same probe is executed from within multiple processes, each data message from each process will trigger the same data callback.

The function prototype for a callback is:

```
void callback (
     GCBSysType sys,
     GCBTagType tag,
     GCBObjType obj,
     GCBMsgType msg);
```

Where: **is:**

`sys` a data structure defined as

```
struct GCBSysType {
     int msg_socket;
     int msg_type;
     int msg_size;
};
```

Where	Is
`msg_socket`	the socket or file descriptor from which the message was received.
`msg_type`	a message key or type value that represents the protocol or purpose behind the message. This is provided and used by the DPCL system in order to determine the callback routine to execute.
`msg_size`	the size of the message in bytes.

`tag` a value, large enough to contain a pointer, that is supplied by the analysis tool when the callback associated with the probe is identified. If the probe is executed by the analysis tool as:

- a point probe, then the callback and its associated tag are identified (as described in “Installing and activating point probes” on page 143) when the analysis tool calls the `Process::install_probe`, `Process::binstall_probe`, `Application::install_probe`, or `Application::binstall_probe` function.
- a phase probe, then the callback and its associated tag are identified (as described in “Executing phase probes” on page 153)

when the analysis tool uses a constructor to create the Phase class object.

- a one-shot probe, then the callback and its associated tag are identified (as described in “Executing one-shot probes” on page 163 when the analysis tool calls the Process::execute, Process::bexecute, Application::execute, or Application::bexecute function.

The tag parameter allows the analysis tool to use the callback function for more than one purpose. For example, the tag may contain a pointer to relevant data that changes from one execution of the callback to another.

obj
: a pointer to the object that issued the request. In the case of the Application object, the requesting object will be the Process object managed by the Application object. The DPCL system supplies this information because each data message from a probe triggers the same data callback, even when the probe is executed from within multiple processes. By supplying a pointer to the invoking object, the DPCL system enables the callback to respond differently depending on the invoking object.

msg
: a pointer to the message content. The actual content of the message is presented as a raw byte stream. The size of the message is stored in sys.msg_size. The format of the message is determined by the probe, and contains whatever data the probe supplied to the Ais_send function.

For example, the following code defines a callback for a simple pass counter. The installed probe calls Ais_send, supplying it with the count. This callback then identifies the program task (by using the pointer to the invoking Process and the Process::get_task function), and prints the task and count information to standard output.

```
void count_cb(GCBSystType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg)
{
  Process *P = (Process *) obj;
  int *count = (int *) msg;
  int task = P->get_task();

  printf("task %d count = %d\n", task, *count);
}
```

For more information on the Ais_send or Process::get_task functions, refer to their AIX man pages, or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Chapter 11. Entering and exiting the DPCL main event loop

In order to interface asynchronously with the DPCL system, the analysis tool must enter the DPCL main event loop. It does this by calling the `Ais_main_loop` function. This function puts execution in the event loop and generally does not return control to the analysis tool.

```
Ais_main_loop();
```

Calling the `Ais_main_loop` function puts execution in a processing loop that monitors file descriptors that represent socket connections to the DPCL daemons. Monitoring these file descriptors enables the analysis tool to receive, and execute user callbacks for, messages from connected DPCL daemons. The `Ais_main_loop` function also (as described in Chapter 14, “Handling signals and file descriptors through the DPCL system” on page 177) monitors file descriptors and signals that you register with the DPCL system.

Note: If your analysis tool needs to monitor file descriptors for its own purposes unrelated to DPCL, it can either do this in another program thread, or it can instruct the DPCL system to monitor these additional file descriptors.

If the analysis tool does monitor these additional file descriptors in a separate program thread, be aware that this other thread cannot call DPCL functions or reference DPCL data structures. DPCL is not thread safe; all DPCL function calls must be made from within the same program thread.

For more information on instructing the DPCL system to monitor these additional file descriptors, refer to Chapter 14, “Handling signals and file descriptors through the DPCL system” on page 177.

The call to `Ais_main_loop` is usually placed at the end of the analysis tool's `main` function because it generally does not return. If your analysis tool code needs to execute code placed after the call to `Ais_main_loop`, however, it can break out of the DPCL main event loop by calling the `Ais_end_main_loop` function from an analysis-tool supplied callback routine or signal handler.

```
Ais_end_main_loop();
```

Be aware that, once the analysis tool breaks out of its main event loop, it is, for the most part, no longer capable of responding to events asynchronously. The exception to this is the fact that, after a blocking call is issued and the DPCL system is working on the response to this call, it will process any other asynchronous events that occur. In general, however, non-blocking DPCL calls should not be used after breaking out of the DPCL main event loop, unless the analysis tool code will later reenter the DPCL main event loop by calling the `Ais_main_loop` function again.

The prototype for the `Ais_main_loop` and `Ais_end_main_loop` functions are contained in the header file `AisMainLoop.h`.

For more information on the `Ais_main_loop` and `Ais_end_main_loop` functions, refer to their AIX man pages or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Example: Entering and exiting the DPCL main event loop

In this example, the analysis tool enters the DPCL main event loop:

```
Ais_main_loop();
```

While this function generally does not return, the analysis tool code in this example also contains a signal handler that will break processing out of the DPCL main event loop. (The analysis tool has registered this signal handler by calling the `Ais_add_signal` function as described in “Handling signals through the DPCL system” on page 177.)

This signal handler (for the SIGINT signal) responds to an interrupt signal initiated by the user pressing <**Control>-c**.

```
int sighandler(int s){

     Ais_end_main_loop();

 }
```

Since the preceding signal handler breaks the analysis tool out of the DPCL main loop, execution of the main routine can now continue past its call to the `Ais_main_loop` function. The code following the `Ais_main_loop` function attaches to, and kills, the AIX process represented by the `Process` object `P`. Note in the following code, that blocking calls (`Process::battach` and `Process::bdestroy`) are used; the analysis tool cannot, since it has exited the DPCL main event loop, process events asynchronously.

```
Ais_main_loop();

AisStatus sts=P.battach();
if (sts.status()==ASC_success){
  printf("battach() was successful\n");
} else {
  printf("battach() FAILED.. %s\n",sts.status_name());
  exit(0);
}
sts=P.bdestroy();
if (sts.status()==ASC_success){
  printf("bdestroy() was successful\n");
  } else {
  printf("bdestroy() FAILED.. %s\n",sts.status_name());
  exit(0);
}
```

Chapter 12. Disconnecting from target application processes

When an analysis tool connects to a target application process, the DPCL system establishes a connection between the analysis tool process and the target application process. The DPCL system also creates the environment within the process that allows the analysis tool to insert and remove instrumentation probes. If the analysis tool actually creates one or more target application processes (using the `Process::bcreate`, `Process::create`, `PoeAppl::bcreate`, or `PoeAppl::create` functions as described in "Starting the target application" on page 94), the analysis tool will automatically be connected to the process(es). If the analysis tool did not create the process(es), however, it must explicitly connect in order to install and execute probes within the process(es). It does this using the `Process::connect`, `Process::bconnect`, `Application::connect`, or `Application::bconnect` functions as described in "Connecting to the target application" on page 83.

It is important to note that, once connected to a target application process, the analysis tool does not need to explicitly disconnect from that process. When either the analysis tool process or the target application process terminates, the connection will, of course, be broken. There are times, however, when an analysis tool may want to explicitly disconnect from processes. For example, if an analysis tool was done examining one set of processes, it may wish to disconnect from that set of processes before connecting to another.

Disconnecting from a process not only removes the connection to the process, but also removes any probes that the analysis tool has installed in the process. To disconnect from a single process, the analysis tool code can use the asynchronous function `Process::disconnect` or its blocking equivalent `Process::bdisconnect`. To disconnect from processes on an application-wide basis (for all `Process` objects managed by an `Application` object), the analysis tool can use the functions `Application::disconnect` or `Application::bdisconnect`.

Table 49 (Page 1 of 2). Disconnecting from one or more target application processes

To disconnect:	**From a single process (`Process` object)**	**From multiple processes (all of the `Process` objects managed by an `Application` object)**

Using the asynchronous function `disconnect`	`AisStatus sts = p->disconnect(disconnect_cb, GCBTagType(0));`
	`  check_status("p->disconnect(disconnect_cb, GCBTagType(0))", sts);`
	
	`//`
	`// callback to be called after the`
	`// disconnect completes`
	`//`
	`void disconnect_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg)`
	`{`
	
	`  // code within callback routine`
	`  // can check the status of the`
	`  // operation and respond to its`
	`  // completion by, for example,`
	`  // continuing with other work`
	
	`}`
		`AisStatus sts = a->disconnect(disconnect_cb, GCBTagType(0));`
		`  check_status("a->disconnect(disconnect_cb, GCBTagType(0))", sts);`
		
		`//`
		`// callback to be called after the`
		`// disconnect completes`
		`//`
		`void disconnect_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg)`
		`{`
		
		`  // code within callback routine`
		`  // can check the status of the`
		`  // operation and respond to its`
		`  // completion by, for example,`
		`  // continuing with other work`
		
		`}`

Table 49 (Page 2 of 2). Disconnecting from one or more target application processes

To disconnect:	**From a single process (`Process` object)**	**From multiple processes (all of the `Process` objects managed by an `Application` object)**

Using the blocking function `bdisconnect`	`sts = P.bdisconnect();`
	`  check_status("P.bdisconnect()", sts);`
	`  printf("  %s: disconnected from pid:%d\n",`
	`  toolname,  P.get_pid());`
		`sts = A.bdisconnect();`
		`  check_status("A.bdisconnect()", sts);`
		`  printf("  %s: disconnected from A\n",`
		`  toolname);`

For more information on the `Process::disconnect`, `Process::bdisconnect`, `Application::disconnect`, or `Application::bdisconnect` functions, refer to their AIX man pages or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Chapter 13. Compiling and linking the analysis tool and target application

Before running your DPCL analysis tool and target application, you need to:

- prelink your target application with the DPCL libraries
- compile your analysis tool with the DPCL library and include files.

In addition to the tasks described in this chapter, you should also verify that your *.rhosts* file or your system's */etc/hosts.equiv* file has been set up correctly on the host where your target application will run. If neither of these files has been set up correctly, you will encounter the error `ASC_failed_rhost_check` when you attempt to run your DPCL analysis tool. For more information, refer to *IBM Parallel Environment for AIX: Installation*.

Step 1: Prelink target application with DPCL library

Before you can instrument a target application, you should prelink it with the DPCL libraries. For this purpose, we have provided a generic script file named `prelink`. When you installed DPCL, this script file was copied to the directory `/usr/lpp/ppe.dpcl/samples/prelink`. To prelink your compiled target application, simply supply its name as an argument to the `prelink` script. Assuming you have copied the `prelink` script to the directory containing the target application executable "`foo`", you would enter the following at the AIX command prompt.

```
$ prelink foo
```

The prelink script creates a prelinked output file with the extension .DPCL; in this example `foo.DPCL`.

Step 2: Compile the analysis tool with DPCL library and include files

In order to run a DPCL analysis tool, you will need to compile and link it with the DPCL library and include files. The following example code shows a makefile for a DPCL analysis tool `eut_hello`. Note that:

- the DPCL include files are not located in `/usr/include`, but instead are installed in `/usr/lpp/ppe.dpcl/include`.
- the DPCL library file is named `libdpcl.a` and is located in the directory `/usr/lpp/ppe.dpcl/lib`. A link to `libdpcl.a` is also created in `/usr/lib`, so the compiler option `-L /usr/lpp/ppe.dpcl/lib` (here assigned to the variable `LIBLOC`) is not necessary.

```
.SUFFIXES:      .C

INCDIR      = /usr/lpp/ppe.dpcl/include
INC   = -I$(INCDIR)
LIBLOC      = -L /usr/lpp/ppe.dpcl/lib
LIB   = -ldpcl
CCFLAGS     = -g

all:  eut_hello

.C.o:
      $(CCC) $(CCFLAGS) $(INC) -c $(<)

eut_hello:  eut_hello.o
      $(CCC) -o eut_hello eut_hello.o $(LIB) $(LIBLOC)

clean:
      /bin/rm -rf eut_hello
      /bin/rm -rf eut_hello.o
```

Additional DPCL Programming Tasks

This section contains instructions for performing additional, more-advanced and/or less commonly performed DPCL programming tasks. These additional DPCL programming tasks are described in three chapters.

- Chapter 14, “Handling signals and file descriptors through the DPCL system” on page 177 describes how an application can monitor AIX signals and file descriptors through the DPCL system.
- Chapter 15, “Overriding default system callbacks” on page 181 describes how an application can create callback routines to handle unexpected system events such as a DPCL daemon exiting or a target application process terminating. These callbacks then replace the default system callbacks which merely print out an error message.
- Chapter 16, “Generating diagnostic logs” on page 183 describes how a program can, for troubleshooting and debugging purposes, create a log file that records the activities of the DPCL system.

Chapter 14. Handling signals and file descriptors through the DPCL system

The DPCL system manages a variety of signals and file descriptors in order to enable analysis tools to effectively instrument one or more target application processes. Socket connections from the analysis tool to the various DPCL communication daemons, for example, are represented by file descriptors that the DPCL system monitors using the AIX system call `select` in order to detect when a message has arrived from a DPCL communication daemon. Similarly, certain signals are handled by the DPCL system in order to implement features such as phase probes and one-shot probes. As a programming convenience for you, the developer of DPCL analysis tools, the DPCL system enables you to add additional signals and file descriptors to the lists of those it monitors. When these additional signals occur, or when these additional file descriptors are ready to be read, the DPCL system will call a function handler that you have specified. The function for adding signals to the list of those managed by the DPCL system is the `Ais_add_signal` function and is described in more detail in "Handling signals through the DPCL system." The function for adding file descriptors to the list of those managed by the DPCL system is the `Ais_add_fd` function and is described in more detail in "Handling file descriptors through the DPCL system" on page 178.

Handling signals through the DPCL system

In DPCL analysis tools, signal handlers that you set up using the AIX `sigaction` function will not be able to call DPCL functions. This restriction is in addition to the standard restrictions regarding functions that cannot be called within a `sigaction` signal handler as outlined in the `sigaction` man page. Using the DPCL `Ais_add_signal` function, however, you can specify a signal handler that will be called by the DPCL system rather than by the `sigaction` function. Signal handlers set up using the `Ais_add_signal` function have none of the `sigaction` function's restrictions on what functions they may call. The prototype for the `Ais_add_signal` function is contained in the DPCL header file `AisHandler.h`.

Signal handlers set up using `Ais_add_signal` and so executed by the DPCL system will be called at safe points so as not to interfere with the DPCL system calls. The DPCL system will pass the signal handler a single parameter value — the signal number. If the DPCL system is already processing a request, a signal handler registered by the `Ais_add_signal` function will be executed only after processing of the current request has completed. You should be aware that, depending on the processing required for any particular request, there may be some delay in processing the signal. If delays in processing are not acceptable in your program, and your signal handler does not call any DPCL function, then you should consider using the `sigaction` function to establish the signal handler. Setting up signal handlers by issuing both the `sigaction` and `Ais_add_signal` functions on the same signal within the same DPCL program is allowed. The last one called will override the first. The only restriction is that, if `Ais_add_signal` is used and a subsequent `sigaction` is issued against the same signal, the old action returned by `sigaction` should not be chained to the signal handler set up by `sigaction`. This is because, by chaining the handlers together, the function handler specified by `Ais_add_signal` would be called when the signal is raised, and not at safe points as it would be with the DPCL handler.

The following example code uses two calls to the Ais_add_signal function to specify signal handlers for the SIGINT and SIGALRM signals. In this example, the SIGINT signal will be handled by the sighandler function, and the SIGALRM signal will be handled by the sighandler_ALRM function.

```
Ais_add_signal(SIGINT,sighandler);
Ais_add_signal(SIGALRM,sighandler_ALRM);

.
.
.

int sighandler(int sig_num){
  Ais_end_main_loop();

  FILE *fileout=fopen(FILEOUT,"w");
  int num_tested=0;
  fprintf(fileout,"Source file \"%s\":\n",filename);
  if (fun_count == 0){
     fprintf(fileout," No function found!!\n");
  } else {
     fprintf(fileout,"\nNumber of times each function was executed:\n");
     for (int i=0; i< num_installed; i++){
         fprintf(fileout," %s\t\t%d\n",fun_arr[i].funname,fun_arr[i].pcount);
         if (fun_arr[i].pcount >0) num_tested++;
     }
     fprintf(fileout," Test coverage = %d %%\n",(num_tested*100)/fun_count);
  }
  fclose(fileout);
  printf("*Please check \"%s\" for the final result.\n",FILEOUT);

  return(0);
}
int sighandler_ALRM(int signal){
   print_fun_info();
   alarm(15);
   return(0);
}
```

Once your analysis tool code has used the Ais_add_signal function to add a signal and signal handler to the list of signals managed by the DPCL system, it can, if it no longer needs to handle the signal, call the Ais_remove_signal function. The Ais_remove_signal function removes the signal and signal handler from the list of signals managed by the DPCL system. An analysis tool can also call the Ais_query_signal function to get a pointer to the signal handler function for a specified signal. For more information on the Ais_add_signal, Ais_remove_signal, and Ais_query_signal functions, refer to their AIX man pages or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Handling file descriptors through the DPCL system

In order to monitor messages from DPCL communication daemons to the DPCL analysis tool, the DPCL system uses the AIX select subroutine to monitor file descriptors representing the socket connections from the analysis tool to the various DPCL communication daemons running on the remote hosts. The select subroutine sleeps on these file descriptors until a message arrives from one of the daemons. When the select loop detects that a file descriptor is ready for reading, execution breaks out of the loop and the DPCL system reads and otherwise handles the message.

If your analysis tool needs to monitor file descriptors for its own purposes unrelated to DPCL, it can either do this in another program thread, or it can instruct the DPCL

system to monitor these additional file descriptors. If the analysis tool does monitor these additional file descriptors in a separate program thread, be aware that this other thread cannot call the DPCL functions or reference DPCL data structures. DPCL is not thread safe; all DPCL function calls must be made from within the same program thread.

In order to easily accommodate analysis tools that also use file descriptors to represent files, sockets, and other I/O devices such as a Motif display variable, the DPCL system enables an analysis tool to add a file descriptor and input handler to the list of those it manages. To instruct the DPCL system to monitor an additional file descriptor, the analysis tool code calls the `Ais_add_fd` function, supplying it with the file descriptor to be managed and the function handler that the DPCL system should call when the file descriptor is ready for reading. The DPCL system will pass the function handler a single parameter value — the file descriptor. The prototype for the `Ais_add_fd` function is contained in the DPCL header file `AisHandler.h`.

The following example code uses the `Ais_add_fd` function to instruct the DPCL system to monitor input on the standard input (stdin) file descriptor.

```
#include <stdio.h>
#include <dpcl.h>

int stdin_handler(int);

main() {
  Ais_initialize();

  printf("example of handler for stdin\n");
  printf("Type any text.  Type 'exit' or 'quit' to end program.\n");

  Ais_add_fd(0, stdin_handler);

  Ais_main_loop();

  Ais_remove_fd(0);
}

int stdin_handler(int fd) {
  char s[256];
  gets(s);
  printf("stdin: '%s'\n", s);

  if (!strcmp(s, "exit") || !strcmp(s, "quit"))
    Ais_end_main_loop();
  return 0;
}
```

Once your analysis tool code has used the `Ais_add_fd` function to add a file descriptor and input handler to the list of file descriptors managed by the DPCL system, it can, if it no longer needs to handle the file descriptor, call the `Ais_remove_fd` function. The `Ais_remove_fd` function removes the file descriptor and input handler from the list of file descriptors managed by the DPCL system. For more information on the `Ais_add_fd` and `Ais_remove_fd` functions, refer to their AIX man pages or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Chapter 15. Overriding default system callbacks

As described in “What are DPCL callbacks?” on page 12, most DPCL callbacks are initiated by analysis tool requests. Acknowledgment callbacks, for example, respond to the success/failure of an asynchronous function. These success/failure messages are sent from one or more DPCL communication daemons running on one or more hosts. Similarly, data callbacks respond to message data sent by the DPCL communication daemon(s) — either messages forwarded from installed probes, or diagnostic messages. In all of these cases, however, the messages sent from the DPCL communication daemon to the analysis tool are the result of a direct request from the analysis tool — either an asynchronous request, a request to install a probe, or a request to turn diagnostic logging on. Furthermore, in all these cases, the analysis tool code will have specified, upon making the request, the name of a callback routine to handle the messages from the DPCL communication daemon(s).

System callbacks, on the other hand, are different than these user-initiated requests. They are not the result of a request from the analysis tool, but are instead used to respond to unexpected events (possibly due to system error) such as the DPCL communication daemon exiting. The default system callbacks respond to these unexpected events simply by printing an error or information message describing the event. However, if you would like your analysis tool to respond to any of these events in another way, you can use the `Ais_override_default_callback` function to override the default system callback and call your own callback instead.

To override a default system callback using the `Ais_override_default_callback` function, you supply a message key that indicates the type of system message you want to handle in your callback routine. The message types are:

AIS_DEFAULT_MSG
: If you specify this message type when calling the `Ais_override_default_callback` function, your callback will be invoked when the DPCL communication daemon receives a message that otherwise has no associated callback routine. This callback will be invoked only in cases of an internal DPCL error.

AIS_EXIT_MSG
: If you specify this message type when calling the `Ais_override_default_callback` function, your callback will be invoked when a DPCL communication daemon unexpectedly exits. The message that activates this callback contains the socket handle for the connection to the daemon that exited.

AIS_PROC_TERMINATE_MSG
: If you specify this message type when calling the `Ais_override_default_callback` function, your callback will be invoked when a target application process terminates. The message is an integer containing the process ID of the process that terminated. The host on which the terminated process was running can be inferred by examining the `msg_socket` field of the `GCBSysType` parameter passed to your function and then matching up the socket and `Process` values.

The following example code overrides the default system callback for handling termination of the DPCL daemon. In this example, the new callback is `my_exit_cb`.

```
Ais_override_default_callback(AIS_EXIT_MSG, my_exit_cb, (GCBTagType) 0,
             &prev_cb, &prev_tag);
```

For more information about the `Ais_override_default_callback` function, refer to its AIX man page or its entry in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Chapter 16. Generating diagnostic logs

Each DPCL communication daemon is capable of generating diagnostic messages that enable you or the IBM Support Center to identify problems that your analysis tool may encounter while interacting with the DPCL system. These messages can be turned on separately for each DPCL communication daemon that your analysis tool is interacting with. Since, on each host machine, there is one DPCL communication daemon running per DPCL user, this means that you can generate diagnostic messages that detail your analysis tool's interaction with the DPCL system on each host machine that is running target application processes to which your analysis tool is connected. Be aware, however, that you cannot have multiple analysis tools for the same user attempting to control logging at the same time.

To turn logging on for a particular host machine, your analysis tool code can call either the `Ais_log_on` function or its blocking equivalent — `Ais_blog_on`. The prototypes for these functions (in addition to supporting data type and constant definitions) are contained in the header file `LogSystem.h`. There are five levels of diagnostic messages that may be generated. When calling either the `Ais_log_on` or `Ais_blog_on` function, your analysis tool can specify the logging level using one of the enumeration constants of the `LoggingLevel` enumeration type. The `LoggingLevel` enumeration constants are:

`LGL_fatal` Diagnostic messages will be generated by the DPCL communication daemon only if the next DPCL system action is to crash.

`LGL_severe` In addition to messages indicating fatal conditions, the DPCL communication daemon will generate messages for other serious errors.

`LGL_warning` In addition to messages indicating fatal and severe error conditions, the DPCL communication daemon will also generate warning messages.

`LGL_trace` In addition to fatal, severe and warning messages, the DPCL communication daemon will also generate function entry/exit trace information.

`LGL_detail` The most detailed level of diagnostic messages will be generated by the DPCL communication daemon. In addition to the severe, error, warning, and function entry/exit trace information, the DPCL communication daemon will generate other, more general, information. Be aware that this setting will result in a large number of generated messages.

In addition to being able to specify the level of diagnostic logging, you also specify whether the DPCL communication daemon should:

- Save messages to a file on the host machine.
- send each message back to the analysis tool to be handled by a callback, or
- save messages to a file on the host machine, and also send each message back to the analysis tool to be handled by a callback.

If you elect to have the DPCL communication daemon save the messages to a file, it will save the file to the directory `/tmp`, and will name it by concatenating the string

"dpclsd" with the DPCL communication daemon's AIX process ID. The string "dpclsd" and the AIX process ID are separated by a period (.). For example, if the AIX process ID of the DPCL communication daemon is 3231, the generated file would be `/tmp/dpclsd.3231`.

While it is easiest to simply have the DPCL communication daemon save the diagnostic information to a log file on the host machine, you might want to manipulate the messages using a callback in order to have more control over the log file your analysis tool then generates. For example, you could create a single log from the diagnostic messages generated by several host machines, or you could intersperse additional diagnostic messages specific to your tool along with the DPCL communication daemon messages.

When calling either the `Ais_log_on` or `Ais_blog_on` function, your analysis tool can specify this "logging destination" using one of the enumeration constants of the `LoggingDest` enumeration type. The `LoggingDest` enumeration constants are:

`LGD_client` The DPCL communication daemon sends each message back to the analysis tool for processing within a specified callback routine. The messages sent to the logging callback are null-character ('\0') terminated ASCII strings.

`LGD_daemon` The DPCL communication daemon saves the messages to a log file on the host machine.

`LGD_both` The DPCL communication daemon sends each message back to the analysis tool for processing within a specified callback routine and also saves the messages to a log file on the host machine. If processing the messages in a callback, be aware that the messages sent to the logging callback are null-character ('\0') terminated ASCII strings.

`LGD_neither` Turns logging off. Messages are not sent to the analysis tool, nor are they saved to a log file on the host.

The following table illustrates how to turn logging on using either the `Ais_log_on` or `Ais_blog_on` function. In this example, warning-level messages will be generated and saved to a log file on the host. The NULL values in the following examples represent where we could potentially specify a callback routine or a callback tag. If we had chosen to have the DPCL communication daemon send each message back to the analysis tool, we would have specified the callback routine that would process these messages.

To start the target application processes:	**Do this:**
Using the asynchronous function `Ais_log_on`	`sts = Ais_log_on(host, LGL_warning, LGD_daemon, NULL, NULL, NULL, NULL);`
Using the blocking function `Ais_blog_on`	`sts = Ais_blog_on(host, LGL_warning, LGD_daemon, NULL, NULL);`

Once your analysis tool code has turned logging on, it can modify the logging level or the logging destination by simply calling the `Ais_log_on` or `Ais_blog_on` functions again, specifying the new logging level or destination. You can turn logging off by specifying enumeration constant `LGD_neither` as the logging destination. You can also turn logging off by calling the `Ais_log_off` or `Ais_blog_off` functions.

For more information about the `Ais_log_on`, `Ais_blog_on`, `Ais_log_off`, or `Ais_blog_off` functions, refer to the AIX man pages for these functions, or their entries in the *IBM Parallel Environment for AIX: DPCL Class Reference*.

Appendixes

Appendix A. A DPCL test coverage tool

The following sample program (*eut_testcov.C*) was copied to the directory (*/usr/lpp/ppe.dpcl/samples/testcov*) when you installed DPCL. It illustrates how you can use the DPCL system to create a simple test coverage tool that periodically reports the frequency of function calls in a particular module.

First let's look at how this DPCL test coverage tool appears from the end user's point of view, and then we will examine its source code in more detail. First of all, depending on the program arguments, you can:

- Connect to a running process.

  ```
  eut_testcov <host name> pid <process ID>
  ```

- Start a process running and connect to it.

  ```
  eut_testcov <host name> path </full path/executable>
  ```

- Connect to all the processes in a POE application.

  ```
  eut_testcov <host name> poe_pid <poe process ID>
  ```

- Start a POE process running and connect to its process(es).

  ```
  eut_testcov <host name> poe_path </full path/poe> </full path/executable>
  ```

Here's a sample run of the `eut_testcov` program. In this sample run, the arguments specify that the `eut_testcov` program should start a particular POE application running as 4 processes. Since the program is designed to show the test coverage for a particular module in a particular process, the user is first prompted to select a particular process.

```
Enter the process number to probe (0 to 3)

0

Using process 0
```

Once the particular process is entered, the user is then prompted to select a module in the process.

```
Enter the source file name to check test coverage.
  type !Help to list source files

prod_cons.c

Enter the name of the final output file.

cov_out
```

Once the module is selected, the DPCL test coverage tool will then display the frequency of function calls in that module at intervals of 15 seconds until the end user interrupts this by pressing <**Control**>**-c**. When the user presses <**Control**>**-c**, the program then prints the final test coverage information to a file. Here is some sample output showing the kind of information displayed. Note here that some of the output is coming from the test coverage tool, while the produce/consume messages are coming from our sample target application. For clarity, the following sample output shows the analysis-tool messages in bold.

```
bcreate() was successful.
Using Process 25648 on host pe02.pok.ibm.com...
Module prod_cons.c found... expanding
bexpand() was successful.

function "f2" found
function "produce" found
function "consume" found
function "main" found
function "alarm_handler" found
function "consume_data" found
function "domath" found
function "produce_data" found

*Display interval set at every 15 seconds*
*Enter <ctrl>-c to exit*
bstart() was successfulCompute 3: checking in
produce 3 0
Control 0: #nodes = 4
Control: expect to receive 1200 messages
Compute 1: checking in
produce 1 0
Compute 2: checking in
produce 2 0
consume 3 0
produce 3 1
produce 2 1
produce 1 1
consume 2 0
produce 3 2
produce 2 2
produce 1 2
consume 1 0
produce 3 3
produce 2 3
** Number of times each function was executed:
** f2       0
** produce         0
** consume         1
** main            1
** alarm_handler         0
** consume_data          3
** domath          3
** produce_data          0

** Test coverage = 44 %
produce 1 3
consume 3 1
produce 3 4
produce 2 4
produce 1 4
consume 2 1
produce 3 5
produce 2 5
produce 1 5
produce 3 6
consume 1 1
produce 2 6
produce 1 6
produce 3 7
consume 3 2
produce 2 7
produce 1 7

<ctrl>-c

** Number of times each function was executed:
** f2       0
** produce         0
** consume         1
** main            1
** alarm_handler         0
```

```
** consume_data          7
** domath          7
** produce_data          0

** Test coverage = 44 %
*Please check "/u/cywong/public/eut_testcov.res" for the final result.
produce 3 8
consume 2 2
battach() was successful
bdestroy() was successful
```

Now let's look at the source code for the DPCL test coverage tool. First, let's take a look at the complete source, then we'll describe its parts in more detail.

```
// IBM_PROLOG_BEGIN_TAG
// This is an automatically generated prolog.
//
//
//
// Licensed Materials - Property of IBM
//
// Restricted Materials of IBM
//
// (C) COPYRIGHT International Business Machines Corp. 1999
// All Rights Reserved
//
// US Government Users Restricted Rights - Use, duplication or
// disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
// IBM_PROLOG_END_TAG
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
/*

    Program: eut_testcov
    Program function:
       Periodically shows the frequency of function calls in a module
    Usage:
       eut_testcov <host name> <pid> <module name>
          OR
       eut_testcov <host name> poe_pid <poe pid>
          OR
       eut_testcov <host name> path </full path/executable>
          OR
       eut_testcov <host name> poe_path </full path/poe> </full path/executable>

       - specify "d" for <host name> to use default host

*/
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
const int INTERVAL=15;      //time interval for display information
const int BUFLEN=80;        //size of buffer for storing file and module name
char FILEOUT[128];
// ----------------------------------------------------------------------------
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <AisGlobal.h>
#include <AisInit.h>
#include <AisMainLoop.h>
#include <PoeAppl.h>
#include <Process.h>
#include <InstPoint.h>
#include <ProbeExp.h>
#include <Application.h>
#include <ProbeType.h>
#include <ProbeHandle.h>
#include <InstPoint.h>
#include <SourceObj.h>
#include <AisStatus.h>
#include <LogSystem.h>
#include <AisHandler.h>
```

```
#include <iostream.h>
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Process     P;
PoeAppl     A;
struct fun_info{
    char funname[256];
    int pcount;
};

int fun_count=0;
int fun_num;
int lflag=0;    //Ais_main_loop() started flag

fun_info *fun_arr = NULL;

int num_installed = 0;
char filename[BUFLEN];

void write_results(void);
void ck_process(void);
void print_fun_info(void);
void get_Process(void);
void data_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg);
void stdout_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg);
void stderr_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg);
int sighandler(int signal);
int sighandler_ALRM(int signal);
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
main(int argc, char *argv[])
{
    const char USAGE1[]="\nUSAGE: eut_testcov <host name> pid <pid>\n";
    const char USAGE2[]="\t\tOR\n\teut_testcov <host name> poe_pid <poe pid>\n";
    const char USAGE3[]="\t\tOR\n\teut_testcov <host name> path </full path/executable\n";
    const char USAGE4[]="\t\tOR\n\teut_testcov <host name> poe_path </full"
      "path/poe> </full path/executable>\n";
    const char USAGE5[]="\t*specify \"d\" for <host name> to use default host\n";

    const int bufSize = 128;
    int hostname_length;

    char        hostname[bufSize];      // buffer for get_host_name()
    if (argc < 4){
      cout<<"\n**Incorrect numbers of argument"
      "entered**"<<USAGE1<<USAGE2<<USAGE3<<USAGE4<<USAGE5<<endl;
      exit(0);
    }

    char **argvv=argv+3;

    Ais_initialize();
    Ais_add_signal(SIGINT,sighandler);
    Ais_add_signal(SIGALRM,sighandler_ALRM);

    if(strcmp(argv[1],"d")==0){
      gethostname(hostname,BUFLEN);
      printf("  *Running on \"%s\"\n",hostname);
    } else
      strcpy(hostname,argv[1]);

    if (strcmp("poe_pid",argv[2]) == 0) {   //POE

      AisStatus sts = A.binit_procs(hostname, atoi(argv[3]));
      if (sts.status() != ASC_success){
        printf("binit_procs() was not successful...  %s\n",sts.status_name());
        exit(0);
      } else {
        printf("binit_procs() was successful.\n");
        sts = A.bconnect();
        if (sts.status() != ASC_success){
          printf("bconnect() was not successful...  %s\n",sts.status_name());
          exit(0);
```

```
    } else {
      printf("bconnect() was successful.\n");
    }
    get_Process();
  }
  sts=A.battach();   //stop the application
  if (sts.status() != ASC_success){
     printf("battach() was not successful...  %s\n",sts.status_name());
     exit(0);
  } else {
     printf("battach() was successful.\n");
 }


} else if (strcmp("pid",argv[2]) == 0){     //Not POE
  P = Process(hostname, atoi(argv[3]));
  AisStatus sts = P.bconnect();
  if (sts.status() != ASC_success){
    printf("bconnect() was not successful...  %s\n",sts.status_name());
    exit(0);
  } else {
    printf("bconnect() was successful.\n");
  }
  sts=P.battach();   //stop the application
  if (sts.status() != ASC_success){
    printf("battach() was not successful...  %s\n",sts.status_name());
    exit(0);
  } else {
    printf("battach() was successful.\n");
  }


} else if (strcmp("path",argv[2]) == 0){      //start a new appl
  AisStatus sts = P.bcreate(hostname,argv[3],argvv,NULL,
                  stdout_cb,NULL, stderr_cb,NULL);
  if (sts.status() != ASC_success){
    printf("bcreate() was not successful...  %s\n",sts.status_name());
    exit(0);
  } else {
    printf("bcreate() was successful.\n");
  }

 } else if (strcmp("poe_path",argv[2]) == 0){
  AisStatus sts=A.bcreate(hostname,argv[3],argvv,NULL,
        stdout_cb, NULL, stderr_cb, NULL);
  if (sts.status() != ASC_success){
    printf("bcreate() was not successful...  %s\n",sts.status_name());
    exit(0);
  } else {
    printf("bcreate() was successful.\n");
  }

  get_Process();

 } else {
  cout<<USAGE1<<USAGE2<<USAGE3<<USAGE4<<USAGE5<<endl;
  exit(0);
}


printf("Using Process %d",P.get_pid());

P.get_host_name(hostname,bufSize);
printf(" on host %s...\n",hostname);

SourceObj myprog = P.get_program_object();
SourceObj mymod;
char        bufmname[bufSize];      // buffer for module_name(..)

char name[BUFLEN];
int found=0;
```

```
    while(found==0)
    {
      cout<<"Enter the source file name to check test coverage."<<endl;
      cout<<"type !Help to list source files"<<endl;

      cin>>filename;

      if(strcmp(filename,"!Help")==0)
      {
        //  loop through source files to list
        cout<<"Source files:"<<endl;
        int count=myprog.child_count();
        for(int x=0;x<count;x++)
        {
          mymod=myprog.child(x);
          mymod.module_name(name,BUFLEN);
          cout<<x<<" "<<name<<endl;
        }
      } else
       {
        for (int c = 0; c < myprog.child_count(); c++)
        {
          mymod = myprog.child(c);
          const char * modname = mymod.module_name(bufmname,bufSize);

        if (strcmp(modname, filename) ==0)
          {
            printf("Module %s found... expanding\n",filename);
          AisStatus sts = mymod.bexpand(P);
            if (sts.status() != ASC_success)
            {
              printf("bexpand() was not successful.. %s\n",sts.status_name());
              exit(0);
            }
          else
          {
              printf("bexpand() was successful.\n");
            }
            found=1;
          break;
        }
        }
      //}
      if (found == 0)
           cout<<"\""<<filename<<"\" not found.\n";
      }

    }                          //end while

    cout<<"Enter the name of the final output file."<<endl;
    cin>>FILEOUT;

    ProbeExp parms[3];
    parms[0]=Ais_msg_handle;
    parms[1]=ProbeExp("xxx");    //send dummy information to application
    parms[2]=ProbeExp(4);
    ProbeExp mysend=Ais_send.call(3,parms);

    SourceObj myfun;
    char        bufdname[bufSize];   // buffer for get_demangled_name(..)

#ifdef DEBUG
    printf("mymod.child_count() = %d\n",mymod.child_count());
#endif

    fun_count=mymod.child_count();
    fun_arr = new fun_info[fun_count];

    printf("\n");
    for ( int c = 0; c < fun_count; c++)
    {
      myfun = mymod.child(c);
```

```
      const char * funname = myfun.get_demangled_name(bufdname,bufSize);
        if (funname == NULL) continue;
        printf("function \"%s\" found\n",funname);

        strcpy(fun_arr[fun_num].funname,funname);
        fun_arr[fun_num].pcount=0;
        fun_num++;

        InstPoint  mypoint;

        for ( int d = 0; d < myfun.exclusive_point_count(); d++)
        {
           mypoint = myfun.exclusive_point(d);

          if ( mypoint.get_type() == IPT_function_entry)
          {

#ifdef DEBUG
            printf("  Found function entry point.\n");
#endif

            ProbeHandle   myph;
            GCBFuncType   mydcb = data_cb;
            GCBTagType    mytg = (GCBTagType) (fun_num-1);

            AisStatus sts = P.binstall_probe(1, &mysend, &mypoint,
                          &mydcb, &mytg,
                          &myph);
            if (sts.status() != ASC_success){
            printf("  binstall_probe() was not successful.. "
            "%s\n",sts.status_name());
              exit(0);
            } else {
              ++num_installed;

#ifdef DEBUG
              printf("  binstall_probe() was successful\n");
#endif

            }


            sts = P.bactivate_probe(1, &myph);
            if (sts.status() != ASC_success){
              printf("  bactivate_probe() was not successful.. %s\n",sts.status_name());
              exit(0);
            } else {

#ifdef DEBUG
              printf("  bactivate_probe() was successful\n");
#endif

            }

            break;
          }
        }
    }

    printf("\n*Display interval set at every %d seconds*\n",INTERVAL);
    printf("*Enter <CTRL>-c to exit*\n");
    sleep(3);

    if(strcmp(argv[2],"path")==0){
      AisStatus sts=P.bstart();
      if (sts.status()==ASC_success){
        printf("bstart() was successful\n");
      } else {
        printf("bstart() FAILED.. %s\n",sts.status_name());
        exit(0);
      }
    }
```

```
    if (strcmp(argv[2],"poe_path")==0)
    {
      AisStatus sts=A.bstart();
      if (sts.status()==ASC_success){
        printf("bstart() was successful\n");
      } else {
        printf("bstart() FAILED.. %s\n",sts.status_name());
        exit(0);
      }
    } else if(strcmp(argv[2],"poe_pid")==0){
      AisStatus sts=A.bresume();
      if (sts.status()==ASC_success){
        printf("bresume() was successful\n");
      } else {
        printf("bresume() FAILED.. %s\n",sts.status_name());
        exit(0);
      }
    } else if(strcmp(argv[2],"pid")==0){
      AisStatus sts=P.bresume();
      if (sts.status()==ASC_success){
        printf("bresume() was successful\n");
      } else {
        printf("bresume() FAILED.. %s\n",sts.status_name());
        exit(0);
      }
    }


    alarm(INTERVAL);
    lflag=1;
    Ais_main_loop();

    if(strcmp(argv[2],"path")==0){
      AisStatus sts=P.battach();
      if (sts.status()==ASC_success){
        printf("battach() was successful\n");
      } else {
        printf("battach() FAILED.. %s\n",sts.status_name());
        exit(0);
      }
      sts=P.bdestroy();
      if (sts.status()==ASC_success){
        printf("bdestroy() was successful\n");
      } else {
        printf("bdestroy() FAILED.. %s\n",sts.status_name());
        exit(0);
      }
    }

    if (strcmp(argv[2],"poe_path")==0)
    {
      AisStatus sts=A.battach();
      if (sts.status()==ASC_success){
        printf("battach() was successful\n");
      } else {
        printf("battach() FAILED.. %s\n",sts.status_name());
        exit(0);
      }
      sts=A.bdestroy();
      if (sts.status()==ASC_success){
        printf("bdestroy() was successful\n");
      } else {
        printf("bdestroy() FAILED.. %s\n",sts.status_name());
        exit(0);
      }
    }


}
```

```
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
void print_fun_info(void)
{
  int num_tested=0;
  if (fun_count == 0){
      printf(" No function found!!\n");
  } else {
     printf("\n** Number of times each function was executed:\n");
     for (int i=0; i< num_installed; i++){
         printf("** %s\t\t%d\n",fun_arr[i].funname,fun_arr[i].pcount);
         if (fun_arr[i].pcount >0) num_tested++;
     }
     printf("\n** Test coverage = %d %%\n",(num_tested*100)/num_installed);
     fflush(stdout);
  }
}

void
data_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg)
{
        Process *p = (Process *)obj;

        int     i = (int) tag;
#ifdef DEBUG
        printf("Task %d sent the function number %d\n", p->get_task(),i);
#endif
        fun_arr[i].pcount++;

}
int sighandler(int s){
  if (lflag == 1){
    Ais_end_main_loop();
    write_results();
  } else {
    printf("*\"<CTRL>-c\" was entered.  Exiting..\n");
    exit(0);
  }
  return(0);
}
int sighandler_ALRM(int signal){
   ck_process();

   print_fun_info();
   sleep(5);
   alarm(15);
   return(0);
}
void
stdout_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg) {
    //  callback to receive standard out from application and display

    char *p = (char *) msg;
    printf("stdout_cb: ");
    for (int i=0; i<sys.msg_size; i++) {
        printf("%c",*p);
        p++;
    }
    printf("\n");
    fflush(stdout);
}
void
stderr_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg) {
    //  callback to receive standard error from application and display

    char *p = (char *) msg;
    printf("stder_cb: ");
    for (int i=0; i<sys.msg_size; i++) {
        printf("%c",*p);
        p++;
    }
    printf("\n");
```

```
    fflush(stdout);
}
void get_Process(void){
  char procname[256];
  int procno,proccnt;
  proccnt = A.get_count() -1;
  cout << "Enter the process number to probe (0 to "<<proccnt<<")"<<endl;
  cin>>procname;
  procno=atoi(procname);
  cout<<"Using process "<<procno<<endl;

  P = A.get_process(procno);
}
void ck_process(void){
   //check to see if the process was destroyed

   ConnectState *state=new ConnectState;
    AisStatus sts = P.query_state(state);
    if (sts.status() == ASC_success){
      if (*state == PRC_destroyed){
          cout<<"Process terminated.  Exiting...\n";
          write_results();
          exit(0);
      }
    }

}
void write_results(void){
    FILE *fileout=fopen(FILEOUT,"w");
    int num_tested=0;
    fprintf(fileout,"Source file \"%s\":\n",filename);
    if (fun_count == 0){
      fprintf(fileout," No function found!!\n");
    } else {
      fprintf(fileout,"\nNumber of times each function was executed:\n");
      for (int i=0; i< num_installed; i++){
         fprintf(fileout," %s\t\t%d\n",fun_arr[i].funname,fun_arr[i].pcount);
         if (fun_arr[i].pcount >0) num_tested++;
      }
      fprintf(fileout," Test coverage = %d %%\n",(num_tested*100)/num_installed);
    }
    fclose(fileout);
    printf("*Please check \"%s\" for the final result.\n",FILEOUT);
}
```

Now let's examine this program in greater detail. The first thing it does is initialize itself to use the DPCL system. Among other, general, initialization tasks (such as declaring variables), our program:

1. includes the DPCL header files.

   ```
   #include <AisGlobal.h>
   #include <AisInit.h>
   #include <AisMainLoop.h>
   #include <PoeAppl.h>
   #include <Process.h>
   #include <InstPoint.h>
   #include <ProbeExp.h>
   #include <Application.h>
   #include <ProbeType.h>
   #include <ProbeHandle.h>
   #include <InstPoint.h>
   #include <SourceObj.h>
   #include <AisStatus.h>
   #include <LogSystem.h>
   #include <AisHandler.h>
   ```

2. calls the DPCL initialization routine (`Ais_initialize`) to initialize the DPCL system.

   ```
   Ais_initialize();
   ```

3. calls the `Ais_add_signal` function twice to specify that the DPCL system should add the SIGINT and SIGALRM signals to the list of signals it manages. When one of these signals occurs, the DPCL system will call its respective handler. The handler for the SIGINT signal will respond to the end user pressing <**Control>-c** by printing the final coverage information to a file. Handling the SIGALRM signal enables our program to print the coverage information to the screen at 15 second intervals; the SIGALRM signal will be detected by the DPCL system at those intervals and it will call the appropriate handler.

```
Ais_add_signal(SIGINT,sighandler);
Ais_add_signal(SIGALRM,sighandler_ALRM);
```

Next, our program will either connect to or create the target application process(es) based on the arguments that the user passed to the program.

If the user wants to:	Then our analysis tool:
connect to a running POE application	1. calls the `PoeAppl::binit_procs` function to initialize the `PoeAppl` object `A` to contain `Process` objects that represent the POE application's processes. 2. calls the `Application::bconnect` function to connect to the POE processes represented by the `Process` objects managed by the `PoeAppl` object `A`.
connect to a single running process	1. initializes the `Process` object `P` with information (host name and process ID) that identifies the AIX process. 2. calls the `Process::bconnect` function to connect to the process.
start a single-process application running	calls the `Process::bcreate` function to create the process. The process is created in a "stopped state"; its execution is stopped before the first executable instruction. The analysis tool will later, after point probes are installed, start this process running by calling the `Process::bstart` function.
start the multiple processes of a POE application running	calls the `PoeAppl::bcreate` function to create the processes. The processes are created in a "stopped state"; their execution is stopped before the first executable instruction. The analysis tool will later, after point probes are installed, start these processes running by calling the `Application::bstart` function.

```
if(strcmp(argv[1],"d")==0){
  gethostname(hostname,BUFLEN);
  printf("  *Running on \"%s\"\n",hostname);
} else
  strcpy(hostname,argv[1]);

if (strcmp("poe_pid",argv[2]) == 0) {   //POE

  AisStatus sts = A.binit_procs(hostname, atoi(argv[3]));
  if (sts.status() != ASC_success){
    printf("binit_procs() was not successful...  %s\n",sts.status_name());
    exit(0);
  } else {
    printf("binit_procs() was successful.\n");
    sts = A.bconnect();
    if (sts.status() != ASC_success){
      printf("bconnect() was not successful...  %s\n",sts.status_name());
      exit(0);
    } else {
      printf("bconnect() was successful.\n");
    }
    get_Process();
  }
  sts=A.battach();   //stop the application
  if (sts.status() != ASC_success){
     printf("battach() was not successful...  %s\n",sts.status_name());
     exit(0);
  } else {
     printf("battach() was successful.\n");
 }
```

```
    } else if (strcmp("pid",argv[2]) == 0){     //Not POE
      P = Process(hostname, atoi(argv[3]));
      AisStatus sts = P.bconnect();
      if (sts.status() != ASC_success){
        printf("bconnect() was not successful...  %s\n",sts.status_name());
        exit(0);
      } else {
        printf("bconnect() was successful.\n");
      }
      sts=P.battach();   //stop the application
      if (sts.status() != ASC_success){
        printf("battach() was not successful...  %s\n",sts.status_name());
        exit(0);
      } else {
        printf("battach() was successful.\n");
      }


    } else if (strcmp("path",argv[2]) == 0){      //start a new appl
      AisStatus sts = P.bcreate(hostname,argv[3],argvv,NULL,
                     stdout_cb,NULL, stderr_cb,NULL);
      if (sts.status() != ASC_success){
        printf("bcreate() was not successful...  %s\n",sts.status_name());
        exit(0);
      } else {
        printf("bcreate() was successful.\n");
      }

     } else if (strcmp("poe_path",argv[2]) == 0){
      AisStatus sts=A.bcreate(hostname,argv[3],argvv,NULL,
            stdout_cb, NULL, stderr_cb, NULL);
      if (sts.status() != ASC_success){
        printf("bcreate() was not successful...  %s\n",sts.status_name());
        exit(0);
      } else {
        printf("bcreate() was successful.\n");
      }
```

If our analysis tool has connected to or created a POE application, note that, in the preceding code, it calls the `get_Process` program function to prompt the user to select a single process in the POE application to instrument. The program function `get_Process`:

1. calls the `Application::get_count` function to determine the number of processes in the POE application represented by the `PoeAppl` object `A`.
2. calls the `Application::get_process` function to get the `Process` object that represents the process identified by the user. The `Process` object returned by the `Application::get_process` function is assigned to the `Process` object variable `P`.

```
void get_Process(void){
  char procname[256];
  int procno,proccnt;
  proccnt = A.get_count() -1;
  cout << "Enter the process number to probe (0 to "<<proccnt<<")"<<endl;
  cin>>procname;
  procno=atoi(procname);
  cout<<"Using process "<<procno<<endl;

  P = A.get_process(procno);
}
```

From this point on, the target application is mainly concerned with a single process represented by the `Process` object `P`. The analysis tool prints the process ID and the name of the host machine running the process to standard output. It gets this

information by calling the Process::get_pid and Process::get_host_name functions.

```
printf("Using Process %d",P.get_pid());

P.get_host_name(hostname,bufSize);
printf(" on host %s...\n",hostname);
```

Next, our analysis tool needs to get a reference to the source structure of the selected target application process. This source object is represented as a hierarchy of source objects (SourceObj class objects). The top level source object (the root of the hierarchy) is called the "program object", and our analysis tool gets a reference to it using the Program::get_program_object function. The SourceObj object returned by the Process::get_program_object function is assigned to the SourceObj object variable myprog. An additional SourceObj object (mymod) is instantiated to represent an individual module in the source hierarchy.

```
SourceObj myprog = P.get_program_object();
SourceObj mymod;
```

Since our analysis tool is designed to show the test coverage for a particular module only, it next prompts the user to select a module (source file). The user can either enter a source file name, or type !Help and press <**Enter>** to list the source files for the process. To list the source files, our analysis tool uses the SourceObj::child_count function to determine the number of modules objects under the program object. Using this number to initialize a for loop, the program then assigns each module object in turn to the SourceObj object variable mymod, and identifies the module's name using the SourceObj::module_name function. If the user enters a module name, the analysis tool code uses a similar loop to assign the appropriate module-level source object to the SourceObj object mymod, and then expands the module using the SourceObj::bexpand function. The analysis tool must expand the module because, when the DPCL system connects with a target application process, it retrieves the source hierarchy only down to the module level. Expanding the module with the SourceObj::expand or, in this case, the blocking function SourceObj::bexpand enables our analysis tool to go deeper into the source hierarchy.

```
while(found==0)
{
  cout<<"Enter the source file name to check test coverage."<<endl;
  cout<<"type !Help to list source files"<<endl;

  cin>>filename;

  if(strcmp(filename,"!Help")==0)
  {
    //  loop through source files to list
    cout<<"Source files:"<<endl;
    int count=myprog.child_count();
    for(int x=0;x<count;x++)
    {
      mymod=myprog.child(x);
      mymod.module_name(name,BUFLEN);
      cout<<x<<" "<<name<<endl;
    }
  } else
   {
    for (int c = 0; c < myprog.child_count(); c++)
    {
      mymod = myprog.child(c);
      const char * modname = mymod.module_name(bufmname,bufSize);

    if (strcmp(modname, filename) ==0)
      {
```

```
          printf("Module %s found... expanding\n",filename);
        AisStatus sts = mymod.bexpand(P);
          if (sts.status() != ASC_success)
          {
            printf("bexpand() was not successful.. %s\n",sts.status_name());
            exit(0);
          }
        else
        {
            printf("bexpand() was successful.\n");
          }
          found=1;
        break;
      }
      }
    //}
    if (found == 0)
         cout<<"\""<<filename<<"\" not found.\n";
    }

  }                        //end while
```

Now that is has a particular module to work with, our analysis tool can get to its real work — instrumenting the target application to periodically show the frequency of function calls in the module. Basically, our analysis tool will do this by installing a point probe at the function entry point of each function in the module. Each time execution enters a function the probe will execute, sending a message back to the analysis tool. The probe's data callback will process these incoming messages, keeping track of how many times each of the functions was called.

At 15-second intervals, a SIGALRM signal handler will take coverage information collected by the probe's data callback routine and will print it to standard output. The SIGALRM signal was, as you may recall, one of the signals that the analysis tool added (using the `Ais_add_signal` function) to the list of signals managed by the DPCL system. When the SIGALRM signal is received, the DPCL system will call the handler specified by the analysis tool when it called the `Ais_add_signal` function. The other signal that the analysis tool added to the list of those managed by the DPCL system was the SIGINT signal. When the user presses <**Control>-c**, the DPCL system will detect the signal and will call another signal handler in the analysis tool. This one will print a final report of the information collected by the probe's data callback routine.

That's basically how our analysis tool will instrument the target application; now let's look at the specifics. First of all, the analysis tool needs to create the probe expression that, when installed as a point probe at the various function entry points, will send a message back to the analysis tool. To send data back to the analysis tool, DPCL provides a predefined probe expression (`Ais_send`). `Ais_send` is a probe expression representation of a function for sending data back to the analysis tool. The function represented by the `Ais_send` probe expression takes three parameters — a message handle for managing where the message is sent, the address of the data to send, and the size of data being sent. In our case, the contents of the message is not important since a tag value passed to the data callback will track which function was called. Since the content of the message is not important, our probe will send the string "xxx". If we were able to hand code this function call into our target application, it would look like this:

```
Ais_send(Ais_msg_handle, "xxx", 4);
```

Using the `ProbeExp` class in DPCL, however, we have to use a slightly different approach to accomplish the same thing. That is because `Ais_send` is a probe

expression representation of the actual function, and each parameter to the function also needs to be a probe expression. Then, all these individual probe expressions need to be combined into a single probe expression that represents the function call with parameters.

First our analysis tool needs to create an array of probe expressions, each representing one of the parameters to the Ais_send function. Note in the following code that Ais_msg_handle is another predefined probe expression supplied by DPCL. It is specifically designed for the Ais_send function for managing where the message is sent.

```
ProbeExp parms[3];
parms[0]=Ais_msg_handle;
parms[1]=ProbeExp("xxx");    //send dummy information to application
parms[2]=ProbeExp(4);
```

Next the analysis tool can create the probe expression that calls the Ais_send function using the three parameters defined in the parms array.

```
ProbeExp mysend=Ais_send.call(3,parms);
```

Now that the analysis tool has created the probe expression, it can install it as point probes at the function entry point for each function within the selected module. To do this, our analysis tool uses a series of nested loops as shown below. First it dives one level deeper into the module's source hierarchy to identify the function-level SourceObj objects within the module-level SourceObj object mymod. It does this by initializing the first for loop to the number of child SourceObj objects in the SourceObj object mymod. To get the number of child SourceObj objects in mymod, the analysis tool calls the SourceObj::child_count function. For each child SourceObj object in mymod, the analysis tool then uses the Source::get_demangled_name function to determine if the SourceObj object represents a function in the source hierarchy. If the SourceObj object does not represent a function, the SourceObj::get_demangled_name function will return 0, and the for loop will ignore this child SourceObj object and continue with its next iteration. If the SourceObj object does represent a function, it uses a for loop to identify the instrumentation point representing the function entry point. The analysis tool uses the SourceObj::exclusive_point_count function to identify the number of instrumentation points in the function-level SourceObj object. It then uses the SourceObj::exclusive_point function to get a reference to a particular instrumentation point for the current iteration of the loop and uses the InstPoint::get_type reference to determine if the instrumentation point represents the function entry site. (The IPT_function_entry enumeration constant of the InstPtType enumeration type indicates that the point is the function entry site.) When the analysis tool identifies a function entry site, it installs and activates its probe expression as a point probe at that instrumentation point using the Process::install_probe and Process::activate_probe functions.

```
    SourceObj myfun;
    char        bufdname[bufSize];   // buffer for get_demangled_name(..)

#ifdef DEBUG
    printf("mymod.child_count() = %d\n",mymod.child_count());
#endif

    fun_count=mymod.child_count();
    fun_arr = new fun_info[fun_count];

    printf("\n");
    for ( int c = 0; c < fun_count; c++)
    {
      myfun = mymod.child(c);
```

```
      const char * funname = myfun.get_demangled_name(bufdname,bufSize);
        if (funname == NULL) continue;
        printf("function \"%s\" found\n",funname);

        strcpy(fun_arr[fun_num].funname,funname);
        fun_arr[fun_num].pcount=0;
        fun_num++;

        InstPoint  mypoint;

        for ( int d = 0; d < myfun.exclusive_point_count(); d++)
        {
           mypoint = myfun.exclusive_point(d);

          if ( mypoint.get_type() == IPT_function_entry)
          {

#ifdef DEBUG
            printf("  Found function entry point.\n");
#endif

            ProbeHandle   myph;
            GCBFuncType   mydcb = data_cb;
            GCBTagType    mytg = (GCBTagType) (fun_num-1);

            AisStatus sts = P.binstall_probe(1, &mysend, &mypoint,
                        &mydcb, &mytg,
                        &myph);
            if (sts.status() != ASC_success){
            printf("  binstall_probe() was not successful.. "
            "%s\n",sts.status_name());
              exit(0);
            } else {
              ++num_installed;

#ifdef DEBUG
              printf("  binstall_probe() was successful\n");
#endif

            }


            sts = P.bactivate_probe(1, &myph);
            if (sts.status() != ASC_success){
              printf("  bactivate_probe() was not successful.. %s\n",sts.status_name());
              exit(0);
            } else {

#ifdef DEBUG
              printf("  bactivate_probe() was successful\n");
#endif

            }

            break;
          }
        }
    }
```

In the preceding example, consider for a moment the fact that this probe expression is potentially going to be installed and activated at multiple function entry points within this same process. When execution reaches the activated probe, it will send its message back to the analysis tool where its data callback routine is triggered. Here the same data callback routine will be triggered for all the installed point probes. The data callback needs some way of knowing which function entry point has been reached. To identify the function that triggers the data callback, our analysis tool code uses the data callback's `tag` and the program variable `fun_num`. The data callback can then use this `tag` value to identify the function that was

called. Here's the data callback code; it uses the array `fun_arr` to keep track of the number of times each function was called.

```
void
data_cb(GCBSysType sys, GCBTagType tag, GCBObjType obj, GCBMsgType msg)
{
        Process *p = (Process *)obj;

        int     i = (int) tag;
#ifdef DEBUG
        printf("Task %d sent the function number %d\n", p->get_task(),i);
#endif
        fun_arr[i].pcount++;

}
```

Getting back to the `main` routine, we see that the analysis tool has a few final actions to make before entering the DPCL main loop (using the `Ais_main_loop` function) to process events asynchronously. It instructs the end user to press <**Control>-c** to exit, and, if it created one or more processes (using the `Process::create` or `PoeAppl::create` function), it now starts them (using the `Process::start` or `Application::start` function). It also schedules a SIGALRM signal to be delivered to the analysis tool process after 15 seconds.

```
    printf("\n*Display interval set at every %d seconds*\n",INTERVAL);
    printf("*Enter <CTRL>-c to exit*\n");
    sleep(3);

    if(strcmp(argv[2],"path")==0){
      AisStatus sts=P.bstart();
      if (sts.status()==ASC_success){
        printf("bstart() was successful\n");
      } else {
        printf("bstart() FAILED.. %s\n",sts.status_name());
        exit(0);
      }
    }

    if (strcmp(argv[2],"poe_path")==0)
    {
      AisStatus sts=A.bstart();
      if (sts.status()==ASC_success){
        printf("bstart() was successful\n");
      } else {
        printf("bstart() FAILED.. %s\n",sts.status_name());
        exit(0);
      }
    } else if(strcmp(argv[2],"poe_pid")==0){
      AisStatus sts=A.bresume();
      if (sts.status()==ASC_success){
        printf("bresume() was successful\n");
      } else {
        printf("bresume() FAILED.. %s\n",sts.status_name());
        exit(0);
      }
    } else if(strcmp(argv[2],"pid")==0){
      AisStatus sts=P.bresume();
      if (sts.status()==ASC_success){
        printf("bresume() was successful\n");
      } else {
        printf("bresume() FAILED.. %s\n",sts.status_name());
        exit(0);
      }
    }


    alarm(INTERVAL);
    lflag=1;
    Ais_main_loop();
```

Once in the Ais_main_loop, the analysis tool process will respond to events asynchronously. As the target process runs and the installed probes execute, they will send data back to the analysis tool to trigger the data callback. Each message will indicate that a function was called, and, as already shown, the callback will use its tag value and an array to keep track of this information. After 15 seconds, the SIGALRM signal is delivered to the analysis tool process; it is detected by the DPCL system which calls the following signal handler (sighandler_ALRM) as specified by the analysis tool when it earlier called the Ais_add_signal function. The signal handler calls the program functions ck_process and print_fun_info. The program function ck_process checks to see if the target application process is still running. The program function print_fun_info prints the coverage information collected by the data callback, and schedules another SIGALRM signal to be delivered to the analysis tool process after another 15 seconds. Here's the signal handler followed by the functions that it calls.

```
int sighandler_ALRM(int signal){
   ck_process();

   print_fun_info();
   sleep(5);
   alarm(15);
   return(0);
}

void ck_process(void){
   //check to see if the process was destroyed

   ConnectState *state=new ConnectState;
    AisStatus sts = P.query_state(state);
    if (sts.status() == ASC_success){
      if (*state == PRC_destroyed){
          cout<<"Process terminated.  Exiting...\n";
          write_results();
          exit(0);
      }
    }

}

void print_fun_info(void)
{
  int num_tested=0;
  if (fun_count == 0){
      printf(" No function found!!\n");
  } else {
     printf("\n** Number of times each function was executed:\n");
     for (int i=0; i< num_installed; i++){
         printf("** %s\t\t%d\n",fun_arr[i].funname,fun_arr[i].pcount);
         if (fun_arr[i].pcount >0) num_tested++;
     }
     printf("\n** Test coverage = %d %%\n",(num_tested*100)/num_installed);
     fflush(stdout);
  }
}
```

The preceding signal handler and function will be called at 15 second intervals in order to print the coverage information collected by the data callback to standard output. If the end user presses <**Control>-c**, the DPCL system detects the SIGINT signal which, like the SIGALRM signal, has been added to the list of signals it monitors. In the case of the SIGINT signal, the DPCL system calls the analysis tool's signal handler sighandler. This signal handler breaks the analysis tool out of the DPCL main event loop (using the Ais_end_main_loop function), and writes the final coverage information to a file by calling the program function write_results.

```
int sighandler(int s){
  if (lflag == 1){
    Ais_end_main_loop();
    write_results();
  } else {
    printf("*\"<CTRL>-c\" was entered.  Exiting..\n");
    exit(0);
  }
  return(0);
}


void write_results(void){
    FILE *fileout=fopen(FILEOUT,"w");
    int num_tested=0;
    fprintf(fileout,"Source file \"%s\":\n",filename);
    if (fun_count == 0){
      fprintf(fileout," No function found!!\n");
    } else {
      fprintf(fileout,"\nNumber of times each function was executed:\n");
      for (int i=0; i< num_installed; i++){
         fprintf(fileout," %s\t\t%d\n",fun_arr[i].funname,fun_arr[i].pcount);
         if (fun_arr[i].pcount >0) num_tested++;
      }
      fprintf(fileout," Test coverage = %d %%\n",(num_tested*100)/num_installed);
    }
    fclose(fileout);
    printf("*Please check \"%s\" for the final result.\n",FILEOUT);
}
```

Since the preceding signal handler breaks the analysis tool out of the DPCL main loop, execution of the `main` routine can now continue past its call to the `Ais_main_loop` function. The analysis tool will then, since the end user has finished collecting the coverage information, attach to and kill any processes that it created and started. Processes that it merely connected to are allowed to continue running. To kill the single process or multiple processes of the POE application that it started, the analysis tool uses either the `Process::battach` and `Process::bdestroy` functions or the `Application::battach` and `Application::bdestroy` functions.

```
    if(strcmp(argv[2],"path")==0){
      AisStatus sts=P.battach();
      if (sts.status()==ASC_success){
        printf("battach() was successful\n");
      } else {
        printf("battach() FAILED.. %s\n",sts.status_name());
        exit(0);
      }
      sts=P.bdestroy();
      if (sts.status()==ASC_success){
        printf("bdestroy() was successful\n");
      } else {
        printf("bdestroy() FAILED.. %s\n",sts.status_name());
        exit(0);
      }
    }

    if (strcmp(argv[2],"poe_path")==0)
    {
      AisStatus sts=A.battach();
      if (sts.status()==ASC_success){
        printf("battach() was successful\n");
      } else {
        printf("battach() FAILED.. %s\n",sts.status_name());
        exit(0);
      }
      sts=A.bdestroy();
      if (sts.status()==ASC_success){
        printf("bdestroy() was successful\n");
      } else {
        printf("bdestroy() FAILED.. %s\n",sts.status_name());
```

```
            exit(0);
          }
        }


    }
```

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Department LJEB/P905
2455 South Road
Poughkeepsie, NY 12601-5400
U.S.A

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States or other countries or both:

AIX
ESCON
IBM
IBMLink
LoadLeveler
Micro Channel
RS/6000
RS/6000 SP

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, BackOffice, MS-DOS , and the Windows logo are trademarks or registered trademarks of Microsoft Corporation in the United States, other countries, or both.

Pentium and Pentium II Xeon are trademarks or registered trademarks of Intel Corporation in the United States, other countries, or both.

Tivoli Enterprise Console is a trademark of Tivoli Systems Inc. in the United States, other countries, or both.

UNIX is a registered trademark in the United States, other countries, or both and is licensed exclusively through X/Open Company Limited.

Other company, product, and service names may be the trademarks or service marks of others.

Glossary

A

abstract syntax tree. A data structure that represents logic removed from a particular syntactic representation of that logic. For example, the abstract syntax tree for the expression *a + (b x c)* is identical to the abstract syntax tree for the expression *a + b x c* (where only precedence rules force the multiplication operation to be performed first). Compilers create abstract syntax trees from a program's source code as an intermediary stage before manipulating and converting the data structure into executable instructions. Similarly, in DPCL, probe expressions to be executed within target application processes are first converted into abstract syntax trees. See also *Dynamic Probe Class Library (DPCL)* and *probe expression.*

address. A value, possibly a character or group of characters that identifies a register, a device, a particular part of storage, or some other data source or destination.

AIX. Abbreviation for Advanced Interactive Executive, IBM's licensed version of the UNIX operating system. AIX is particularly suited to support technical computing applications, including high-function graphics and floating-point computations.

AIXwindows Environment/6000. A graphical user interface (GUI) for the IBM RS/6000. It has the following components:

- A graphical user interface and toolkit based on OSF/Motif
- Enhanced X-Windows, an enhanced version of the MIT X Window System
- Graphics Library (GL), a graphical interface library for the application programmer that is compatible with Silicon Graphics' GL interface.

analysis tool. See *DPCL analysis tool.*

API. Application programming interface.

application. The use to which a data processing system is put; for example, topayroll application, an airline reservation application.

application programming interface (API). A software interface that enables applications or program components to communicate with each other. An API is the set of programming language constructs or statements that can be coded in an application program to obtain the specific functions and services provided by an underlying operating system or service program.

argument. A parameter passed between a calling program and a called program or subprogram.

attribute. A named property of an entity.

B

bandwidth. The difference, expressed in hertz, between the highest and the lowest frequencies of a range of frequencies. For example, analog transmission by recognizable voice telephone requires a bandwidth of about 3000 hertz (3 kHz). The bandwidth of an optical link designates the information-carrying capacity of the link and is related to the maximum bit rate that a fiber link can support.

blocking operation. An operation that does not complete until the operation either succeeds or fails. For example, a blocking receive will not return until a message is received or until the channel is closed and no further messages can be received.

breakpoint. A place in a program, specified by a command or a condition, where the system halts execution and gives control to the workstation user or to a specified program.

broadcast operation. A communication operation in which one processor sends (or broadcasts) a message to all other processors.

buffer. A portion of storage used to hold input or output data temporarily.

C

C. A general-purpose programming language. It was formalized by Uniforum in 1983 and the ANSI standards committee for the C language in 1984.

C++. A general-purpose programming language that is based on the C language. C++ includes extensions that support an object-oriented programming paradigm. Extensions include:

- strong typing
- data abstraction and encapsulation
- polymorphism through function overloading and templates
- class inheritance.

call arc. The representation of a call between two functions within the Xprofiler function call tree. It appears as a solid line between the two functions. The arrowhead indicates the direction of the call; the function it points to is the one that receives the call. The

function making the call is known as the *caller,* while the function receiving the call is known as the *callee.*

chaotic relaxation. An iterative relaxation method which uses a combination of the Gauss-Seidel and Jacobi-Seidel methods. The array of discrete values is divided into sub-regions that can be operated on in parallel. The sub-region boundaries are calculated using Jacobi-Seidel, whereas the sub-region interiors are calculated using Gauss-Seidel. See also *Gauss-Seidel.*

client. A function that requests services from a server and makes them available to the user.

cluster. A group of processors interconnected through a high-speed network that can be used for high-performance computing. A cluster typically provides excellent price/performance.

collective operation. An operation in which every task in the communicator, file, or window must participate.

command alias. When using the PE command line debugger **pdbx**, you can create abbreviations for existing commands using the **pdbx alias** command. These abbreviations are known as *command aliases.*

Communication Subsystem (CSS). A component of the Parallel System Support Programs that provides software support for the SP Switch. CSS provides two protocols: Internet Protocol (IP) for LAN-based communication and user space (US) as a message-passing interface that is optimized for performance over the switch. See also *Internet Protocol* and *User Space.*

communicator. An MPI object that describes the communication context and an associated group of processes.

compile. To translate a source program into an executable program.

condition. One of a set of specified values that a data item can assume.

control workstation. A workstation attached to the IBM RS/6000 SP SP that serves as a single point of control allowing the administrator or operator to monitor and manage the system using Parallel System Support Programs.

core dump. A process by which the current state of a program is preserved in a file. Core dumps are usually associated with programs that have encountered an unexpected, system-detected fault, such as a Segmentation Fault or a severe user error. The current program state is needed for the programmer to diagnose and correct the problem.

core file. A file that preserves the state of a program, usually just before a program is terminated for an unexpected error. See also *core dump.*

current context. When using either of the PE parallel debuggers, control of the parallel program and the display of its data can be limited to a subset of the tasks belonging to that program. This subset of tasks is called the *current context.* You can set the current context to be a single task, multiple tasks, or all the tasks in the program.

D

data decomposition. A method of breaking up (or decomposing) a program into smaller parts to exploit parallelism. One divides the program by dividing the data (usually arrays) into smaller parts and operating on each part independently.

data parallelism. Refers to situations where parallel tasks perform the same computation on different sets of data.

dbx. A symbolic command line debugger that is often provided with UNIX systems. The PE command line debugger **pdbx** is based on the **dbx** debugger.

debugger. A debugger provides an environment in which you can manually control the execution of a program. It also provides the ability to display the program' data and operation.

distributed shell (dsh). An Parallel System Support Programs command that lets you issue commands to a group of hosts in parallel. See *IBM Parallel System Support Programs for AIX: Command and Technical Reference* for details.

domain name. The hierarchical identification of a host system (in a network), consisting of human-readable labels, separated by decimals.

DPCL. See *Dynamic Probe Class Library (DPCL).*

DPCL analysis tool. A C++ application built on the Dynamic Probe Class Library (DPCL). It links in the DPCL library and uses the DPCL API calls to instrument (create probes and insert them into) one or more target application processes. Typically, an analysis tool is designed to measure the efficiency, confirm the correctness, or monitor the execution of, the target application. An analysis tool could be a complex and general-purpose tool like a debugger, or it might be a simple and specialized tool designed for only one particular program, user, or situation. See also *DPCL target application, Dynamic Probe Class Library (DPCL),* and *probe*.

DPCL target application. The executable program that is instrumented by a DPCL analysis tool. It is the process (or processes) into which the DPCL analysis tool inserts probes. A target application could be a serial or parallel program. Furthermore, if the target application is a parallel program, it could follow either the SPMD or the MPMD model, and may be designed for either a message-passing or a shared-memory system. See also *DPCL analysis tool, Dynamic Probe Class Library (DPCL),* and *probe.*

dynamic instrumentation. A form of software instrumentation in which instrumentation can be added to or removed from an application while it is running. Unlike traditional forms of software instrumentation, where instrumentation is added to the application prior to execution, dynamic instrumentation is well suited to

- examining programs, such as database servers, that do not normally terminate.
- examining long-running numerical programs.
- visualizing complex or long-running programs with a minimum of secondary storage consumption.
- enabling the user to interactively tailor, during the application's run, the type of data collected.

The Dynamic Probe Class Library (DPCL) is based on dynamic instrumentation technology. See also *Dynamic Probe Class Library (DPCL)* and *software instrumentation*.

Dynamic Probe Class Library (DPCL). A C++ class library whose application programming interface (API) enables a program to dynamically insert instrumentation code patches, or *probes,* into an executing program. The DPCL product is an asynchronous software system designed to serve as a foundation for a variety of analysis tools that need to dynamically instrument (insert probes into and remove probes from) target applications. In addition to its API, the DPCL system consists of:

- daemon processes that attach themselves to the target application process(es) to perform most of the actual work requested by the analysis tool via the API calls.
- an asynchronous callback facility for responding to messages from the daemon processes (including data messages forwarded from the installed probes).

See also *DPCL analysis tool, DPCL target application,* and *probe.*

E

environment variable. 1) A variable that describes the operating environment of the process. Common environment variables describe the home directory, command search path, and the current time zone. 2) A variable that is included in the current software environment and is therefore available to any called program that requests it.

Ethernet. A baseband local area network (LAN) that allows multiple stations to access the transmission medium at will without prior coordination, avoids contention by using carrier sense and deference, and resolves contention by using collision detection and delayed retransmission. Ethernet uses carrier sense multiple access with collision detection (CSMA/CD).

event. An occurrence of significance to a task — the completion of an asynchronous operation such as an input/output operation, for example.

executable. A program that has been link-edited and therefore can be run in a processor.

execution. To perform the actions specified by a program or a portion of a program.

expression. In programming languages, a language construct for computing a value from one or more operands.

F

fairness. A policy in which tasks, threads, or processes must be allowed eventual access to a resource for which they are competing. For example, if multiple threads are simultaneously seeking a lock, no set of circumstances can cause any thread to wait indefinitely for access to the lock.

FDDI. Fiber Distributed Data Interface.

Fiber Distributed Data Interface (FDDI). An American National Standards Institute (ANSI) standard for a local area network (LAN) using optical fiber cables. An FDDI LAN can be up to 100 kilometers (62 miles) and can include up to 500 system units. There can be up to 2 kilometers (1.24 miles) between system units and concentrators.

file system. In the AIX operating system, the collection of files and file management structures on a physical or logical mass storage device, such as a diskette or minidisk.

fileset. 1) An individually-installable option or update. Options provide specific functions. Updates correct an error in, or enhance, a previously installed program. 2)

One or more separately-installable, logically-grouped units in an installation package. See also *licensed program* and *package.*

foreign host. See *remote host.*

FORTRAN. One of the oldest of the modern programming languages, and the most popular language for scientific and engineering computations. Its name is a contraction of *FORmula TRANslation.* The two most common FORTRAN versions are FORTRAN 77, originally standardized in 1978, and FORTRAN 90. FORTRAN 77 is a proper subset of FORTRAN 90.

function call tree. A graphical representation of all the functions and calls within an application, which appears in the Xprofiler main window. The functions are represented by green, solid-filled rectangles called function boxes. The size and shape of each function box indicates its CPU usage. Calls between functions are represented by blue arrows, called call arcs, drawn between the function boxes. See also *call arcs.*

function cycle. A chain of calls in which the first caller is also the last to be called. A function that calls itself recursively is not considered a function cycle.

functional decomposition. A method of dividing the work in a program to exploit parallelism. One divides the program into independent pieces of functionality, which are distributed to independent processors. This method is in contrast to data decomposition, which distributes the same work over different data to independent processors.

functional parallelism. Refers to situations where parallel tasks specialize in particular work.

G

Gauss-Seidel. An iterative relaxation method for solving Laplace's equation. It calculates the general solution by finding particular solutions to a set of discrete points distributed throughout the area in question. The values of the individual points are obtained by averaging the values of nearby points. Gauss-Seidel differs from Jacobi-Seidel in that, for the i+1st iteration, Jacobi-Seidel uses only values calculated in the ith iteration. Gauss-Seidel uses a mixture of values calculated in the ith and i+1st iterations.

global max. The maximum value across all processors for a given variable. It is global in the sense that it is global to the available processors.

global variable. A variable defined in one portion of a computer program and used in at least one other portion of the computer program.

gprof. A UNIX command that produces an execution profile of C, COBOL, FORTRAN, or Pascal programs. The execution profile is in a textual and tabular format. It is useful for identifying which routines use the most CPU time. See the man page on **gprof**.

graphical user interface (GUI). A type of computer interface consisting of a visual metaphor of a real-world scene, often of a desktop. Within that scene are icons, which represent actual objects, that the user can access and manipulate with a pointing device.

GUI. Graphical user interface.

H

SP Switch. The high-performance message-passing network of the IBM RS/6000 SP(SP) machine that connects all processor nodes.

HIPPI. High performance parallel interface.

hook. A **pdbx** command that lets you re-establish control over all tasks in the current context that were previously unhooked with this command.

home node. The node from which an application developer compiles and runs his program. The home node can be any workstation on the LAN.

host. A computer connected to a network that provides an access method to that network. A host provides end-user services.

host list file. A file that contains a list of host names, and possibly other information, that was defined by the application which reads it.

host name. The name used to uniquely identify any computer on a network.

hot spot. A memory location or synchronization resource for which multiple processors compete excessively. This competition can cause a disproportionately large performance degradation when one processor that seeks the resource blocks, preventing many other processors from having it, thereby forcing them to become idle.

I

IBM Parallel Environment (PE) for AIX. A licensed program that provides an execution and development environment for parallel C, C++, and FORTRAN programs. It also includes tools for debugging, profiling, and tuning parallel programs.

installation image. A file or collection of files that are required in order to install a software product on a IBM

RS/6000 workstation or on SP system nodes. These files are in a form that allows them to be installed or removed with the AIX **installp** command. See also *fileset*, *licensed program*, and *package.*

instrumentation point. In DPCL, a location within a target application process where an analysis tool can install point probes. Instrumentation points are locations at which the DPCL system determines it is safe to insert new code. Such locations are function entry, function exit, and function call sites. See also *DPCL analysis tool, DPCL target application,* and *Dynamic Probe Class Library (DPCL).*

Internet. The collection of worldwide networks and gateways that function as a single, cooperative virtual network.

Internet Protocol (IP). 1) The TCP/IP protocol that provides packet delivery between the hardware and user processes. 2) The SP Switch library, provided with the Parallel System Support Programs, that follows the IP protocol of TCP/IP.

IP. Internet Protocol.

J

Jacobi-Seidel. See *Gauss-Seidel.*

K

Kerberos. A publicly available security and authentication product that works with the Parallel System Support Programs software to authenticate the execution of remote commands.

kernel. The core portion of the UNIX operating system that controls the resources of the CPU and allocates them to the users. The kernel is memory-resident, is said to run in *kernel mode* (in other words, at higher execution priority level than *user mode*), and is protected from user tampering by the hardware.

L

Laplace's equation. A homogeneous partial differential equation used to describe heat transfer, electric fields, and many other applications.

The dimension-free version of Laplace's equation is:

$$\nabla^2 u = 0$$

The two-dimensional version of Laplace's equation may be written as:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

latency. The time interval between the instant at which an instruction control unit initiates a call for data transmission, and the instant at which the actual transfer of data (or receipt of data at the remote end) begins. Latency is related to the hardware characteristics of the system and to the different layers of software that are involved in initiating the task of packing and transmitting the data.

licensed program. A collection of software packages sold as a product that customers pay for to license. A licensed program can consist of packages and filesets a customer would install. These packages and filesets bear a copyright and are offered under the terms and conditions of a licensing agreement. See also *fileset* and *package.*

lightweight corefiles. An alternative to standard AIX corefiles. Corefiles produced in the *Standardized Lightweight Corefile Format* provide simple process stack traces (listings of function calls that led to the error) and consume fewer system resources than traditional corefiles.

LoadLeveler. A job management system that works with POE to let users run jobs and match processing needs with system resources, in order to make better use of the system.

local variable. A variable that is defined and used only in one specified portion of a computer program.

loop unrolling. A program transformation that makes multiple copies of the body of a loop, also placing the copies within the body of the loop. The loop trip count and index are adjusted appropriately so the new loop computes the same values as the original. This transformation makes it possible for a compiler to take additional advantage of instruction pipelining, data cache effects, and software pipelining.

See also *optimization.*

M

menu. A list of options displayed to the user by a data processing system, from which the user can select an action to be initiated.

message catalog. A file created using the AIX Message Facility from a message source file that contains application error and other messages, which can later be translated into other languages without having to recompile the application source code.

message passing. Refers to the process by which parallel tasks explicitly exchange program data.

Message Passing Interface (MPI). A standardized API for implementing the message-passing model.

MIMD. Multiple instruction stream, multiple data stream.

Multiple instruction stream, multiple data stream (MIMD). A parallel programming model in which different processors perform different instructions on different sets of data.

MPMD. Multiple program, multiple data.

Multiple program, multiple data (MPMD). A parallel programming model in which different, but related, programs are run on different sets of data.

MPI. Message Passing Interface.

N

network. An interconnected group of nodes, lines, and terminals. A network provides the ability to transmit data to and receive data from other systems and users.

Network Information Services. A set of UNIX network services (for example, a distributed service for retrieving information about the users, groups, network addresses, and gateways in a network) that resolve naming and addressing differences among computers in a network.

NIS. See *Network Information Services*.

node. (1) In a network, the point where one or more functional units interconnect transmission lines. A computer location defined in a network. (2) In terms of the IBM RS/6000 SP, a single location or workstation in a network. An SP node is a physical entity (a processor).

node ID. A string of unique characters that identifies the node on a network.

nonblocking operation. An operation, such as sending or receiving a message, that returns immediately whether or not the operation was completed. For example, a nonblocking receive will not wait until a message is sent, but a blocking receive will wait. A nonblocking receive will return a status value that indicates whether or not a message was received.

O

object code. The result of translating a computer program to a relocatable, low-level form. Object code contains machine instructions, but symbol names (such as array, scalar, and procedure names), are not yet given a location in memory. Contrast with *source code.*

one-shot probe. In DPCL, a probe that is executed by the DPCL system immediately upon request, regardless of what the application happens to be doing. See also *Dynamic Probe Class Library (DPCL), phase probe, point probe,* and *probe*.

optimization. A widely-used (though not strictly accurate) term for program performance improvement, especially for performance improvement done by a compiler or other program translation software. An optimizing compiler is one that performs extensive code transformations in order to obtain an executable that runs faster but gives the same answer as the original. Such code transformations, however, can make code debugging and performance analysis very difficult because complex code transformations obscure the correspondence between compiled and original source code.

option flag. Arguments or any other additional information that a user specifies with a program name. Also referred to as *parameters* or *command line options.*

P

package. A number of filesets that have been collected into a single installable image of licensed programs. Multiple filesets can be bundled together for installing groups of software together. See also *fileset* and *licensed program.*

parallelism. The degree to which parts of a program may be concurrently executed.

parallelize. To convert a serial program for parallel execution.

Parallel Operating Environment (POE). An execution environment that smooths the differences between serial and parallel execution. It lets you submit and manage parallel jobs. It is abbreviated and commonly known as POE.

parameter. (1) In FORTRAN, a symbol that is given a constant value for a specified application. (2) An item in a menu for which the operator specifies a value or for which the system provides a value when the menu is interpreted. (3) A name in a procedure that is used to refer to an argument that is passed to the procedure.

(4) A particular piece of information that a system or application program needs to process a request.

partition. (1) A fixed-size division of storage. (2) In terms of the IBM RS/6000 SP, a logical definition of nodes to be viewed as one system or domain. System partitioning is a method of organizing the SP into groups of nodes for testing or running different levels of software of product environments.

partition manager. The component of the Parallel Operating Environment (POE) that allocates nodes, sets up the execution environment for remote tasks, and manages distribution or collection of standard input (STDIN), standard output (STDOUT), and standard error (STDERR).

pdbx. The parallel, symbolic command-line debugging facility of PE. **pdbx** is based on the **dbx** debugger and has a similar interface.

PE. The IBM Parallel Environment for AIX licensed program.

performance monitor. A utility that displays how effectively a system is being used by programs.

phase probe. In DPCL, a probe that is executed periodically, upon expiration of a timer, regardless of what part of the target application's code is executing. As opposed to one-shot probes and point probes, a phase probe must refer to a probe module function. A phase probe cannot be a simple probe expression that does not refer to a probe module function. The control mechanism for invoking these time-initiated phase probes is called a *phase.* See also *Dynamic Probe Class Library (DPCL), one-shot probe, point probe,* and *probe.*

PID. Process identifier.

POE. Parallel Operating Environment.

point probe. In DPCL, a probe that the analysis tool places at particular locations within one or more target application processes. When placed in an activated state by the analysis tool, a point probe will run as part of a target application process whenever execution reaches its installed location in the code. The fact that point probes are associated with particular locations within the target application code makes them markedly different from the other two types of probes (phase probes and one-shot probes), which are executed at a particular time regardless of what code the target application is executing. See also *Dynamic Probe Class Library (DPCL), one-shot probe, phase probe,* and *probe.*

pool. Groups of nodes on an SP that are known to LoadLeveler, and are identified by a pool name or number.

point-to-point communication. A communication operation which involves exactly two processes or tasks. One process initiates the communication through a *send* operation. The partner process issues a *receive* operation to accept the data being sent.

probe. In DPCL, the software instrumentation code patch that a DPCL analysis tool can insert into one or more processes of the DPCL target application. Probes are created by the analysis tool code (using a combination of probe expressions and probe modules), and therefore are able to perform any work required by the tool. For example, depending on the needs of the analysis tool, probes could be inserted into the target application to collect and report performance information (such as execution time), keep track of pass counts for test coverage tools, or report or modify the contents of variables for debuggers. There are three types of probes: one-shot probes, phase probes, and point probes. These three types of probes are differentiated by the manner in which their execution is triggered. See also *DPCL analysis tool, DPCL target application, Dynamic Probe Class Library (DPCL), one-shot probe, phase probe, point probe, probe expression,* and *probe module.*

probe expression. In DPCL, an abstract syntax tree that represents a simple instruction or sequence of instructions to be executed as a probe within one or more target application processes. See also *abstract syntax tree, Dynamic Probe Class Library (DPCL), probe,* and *probe module.*

probe module. In DPCL, a compiled object file containing one or more functions written in C. Once an analysis tool loads a particular probe module into a target application, a probe is able to call any of the functions contained in the module. See also *Dynamic Probe Class Library (DPCL)* and *probe*.

procedure. (1) In a programming language, a block, with or without formal parameters, whose execution is invoked by means of a procedure call. (2) A set of related control statements that cause one or more programs to be performed.

process. A program or command that is actually running the computer. It consists of a loaded version of the executable file, its data, its stack, and its kernel data structures that represent the process's state within a multitasking environment. The executable file contains the machine instructions (and any calls to shared objects) that will be executed by the hardware. A process can contain multiple threads of execution.

The process is created via a **fork()** system call and ends using an **exit()** system call. Between **fork** and **exit**, the process is known to the system by a unique process identifier (PID).

Each process has its own virtual memory space and cannot access another process's memory directly. Communication methods across processes include pipes, sockets, shared memory, and message passing.

prof. A utility that produces an execution profile of an application or program. It is useful to identifying which routines use the most CPU time. See the man page for **prof**.

profiling. The act of determining how much CPU time is used by each function or subroutine in a program. The histogram or table produced is called the execution profile.

Program Marker Array. An X-Windows run time monitor tool provided with Parallel Operating Environment, used to provide immediate visual feedback on a program's execution.

pthread. A thread that conforms to the POSIX Threads Programming Model.

R

reduced instruction-set computer. A computer that uses a small, simplified set of frequently-used instructions for rapid execution.

reduction operation. An operation, usually mathematical, that reduces a collection of data by one or more dimensions. For example, the arithmetic SUM operation is a reduction operation which reduces an array to a scalar value. Other reduction operations include MAXVAL and MINVAL.

remote host. Any host on a network except the one at which a particular operator is working.

remote shell (rsh). A command supplied with both AIX and the Parallel System Support Programs that lets you issue commands on a remote host.

Report. In Xprofiler, a tabular listing of performance data that is derived from the gmon.out files of an application. Xprofiler generates five types of reports. Each type of report presents different statistical information for an application.

RISC. See *reduced instruction-set computer*.

S

shell script. A sequence of commands that are to be executed by a shell interpreter such as the Bourne shell (**sh**), the C shell (**csh**), or the Korn shell (**ksh**). Script commands are stored in a file in the same form as if they were typed at a terminal.

segmentation fault. A system-detected error, usually caused by referencing an non-valid memory address.

server. A functional unit that provides shared services to workstations over a network — a file server, a print server, or a mail server, for example.

signal handling. A type of communication that is used by message passing libraries. Signal handling involves using AIX signals as an asynchronous way to move data in and out of message buffers.

Single program, multiple data (SPMD). A parallel programming model in which different processors execute the same program on different sets of data.

software instrumentation. Code that is inserted into a program to gather information regarding the program's run. As the instrumented application executes, the instrumented code then generates the desired information, which could include performance, trace, test coverage, diagnostic, or other data. See also *dynamic instrumentation.*

source code. The input to a compiler or assembler, written in a source language. Contrast with *object code.*

source line. A line of source code.

source object. In DPCL, an object that provides a coarse, source-code-level view of a target application process, and enables an analysis tool to display or navigate a hierarchical representation of a particular target application process. See also *DPCL target application* and *Dynamic Probe Class Library (DPCL).*

SP. IBM RS/6000 SP; a scalable system arranged in various physical configurations, that provides a high-powered computing environment.

SPMD. Single program, multiple data.

standard error. In the AIX operating system, the secondary destination of data produced by a command. Standard error goes to the display unless redirection or piping is used, in which case standard error can go to a file or to another command.

standard input. In the AIX operating system, the primary source of data entered into a command. Standard input comes from the keyboard unless redirection or piping is used, in which case standard input can be from a file or the output from another command.

standard output. In the AIX operating system, the primary destination of data produced by a command. Standard output goes to the display unless redirection or piping is used, in which case standard output can go to a file or to another command.

STDERR. Standard error.

STDIN. Standard input.

STDOUT. Standard output.

stencil. A pattern of memory references used for averaging. A 4-point stencil in two dimensions for a given array cell, x(i,j), uses the four adjacent cells, x(i-1,j), x(i+1,j), x(i,j-1), and x(i,j+1).

subroutine. (1) A sequence of instructions whose execution is invoked by a call. (2) A sequenced set of instructions or statements that may be used in one or more computer programs and at one or more points in a computer program. (3) A group of instructions that can be part of another routine or can be called by another program or routine.

synchronization. The action of forcing certain points in the execution sequences of two or more asynchronous procedures to coincide in time.

system administrator. (1) The person at a computer installation who designs, controls, and manages the use of the computer system. (2) The person who is responsible for setting up, modifying, and maintaining the Parallel Environment.

System Data Repository. A component of the Parallel System Support Programs software that provides configuration management for the SP system. It manages the storage and retrieval of system data across the control workstation, file servers, and nodes.

T

target application. See *DPCL target application.*

task. A unit of computation analogous to an AIX process.

thread. A single, separately dispatchable, unit of execution. There may be one or more threads in a process, and each thread is executed by the operating system concurrently.

tracing. In PE, the collection of information about the execution of the program. This information is accumulated into a trace file that can later be examined.

tracepoint. Tracepoints are places in the program that, when reached during execution, cause the debugger to print information about the state of the program.

trace record. In PE, a collection of information about a specific event that occurred during the execution of your program. For example, a trace record is created for each send and receive operation that occurs in your program (this is optional and may not be appropriate). These records are then accumulated into a trace file that can later be examined.

U

unrolling loops. See *loop unrolling.*

US. User space.

user. (1) A person who requires the services of a computing system. (2) Any person or any thing that may issue or receive commands and message to or from the information processing system.

user space (US). A version of the message passing library that is optimized for direct access to the SP Switch, that maximizes the performance capabilities of the SP hardware.

utility program. A computer program in general support of computer processes; for example, a diagnostic program, a trace program, a sort program.

utility routine. A routine in general support of the processes of a computer; for example, an input routine.

V

variable. (1) In programming languages, a named object that may take different values, one at a time. The values of a variable are usually restricted to one data type. (2) A quantity that can assume any of a given set of values. (3) A name used to represent a data item whose value can be changed while the program is running. (4) A name used to represent data whose value can be changed, while the program is running, by referring to the name of the variable.

view. (1) To display and look at data on screen.

(2) A special display of data, created as needed. A view temporarily ties two or more files together so that the combined files can be displayed, printed, or queried. The user specifies the fields to be included. The original files are not permanently linked or altered; however, if the system allows editing, the data in the original files will be changed.

X

X Window System. The UNIX industry's graphics windowing standard that provides simultaneous views of several executing programs or processes on high resolution graphics displays.

xpdbx. The former name of the PE graphical interface debugging facility, which is now called **pedb**.

Xprofiler. An AIX tool that is used to analyze the performance of both serial and parallel applications, via a graphical user interface. Xprofiler provides quick access to the profiled data, so that the functions that are the most CPU-intensive can be easily identified.

Bibliography

This bibliography helps you find product documentation related to the RS/6000 SP hardware and software products.

You can find most of the IBM product information for RS/6000 SP products on the World Wide Web. Formats for both viewing and downloading are available.

PE documentation is shipped with the PE licensed program in a variety of formats and can be installed on your system. See “Accessing PE Documentation Online” and “Parallel Environment (PE) Publications” on page 224 for more information.

This bibliography also contains a list of non-IBM publications that discuss parallel computing and other topics related to the RS/6000 SP.

Information Formats

Documentation supporting RS/6000 SP software licensed programs is no longer available from IBM in hardcopy format. However, you can view, search, and print documentation in the following ways:

- On the World Wide Web
- Online (on the product media and via the SP Resource Center)

Finding Documentation on the World Wide Web

Most of the RS/6000 SP hardware and software books are available from the IBM RS/6000 Web site at:

http://www.rs6000.ibm.com

The serial and parallel programs that you find in the *IBM Parallel Environment for AIX: Hitchhiker's Guide* are also available from the IBM RS/6000 Web site, in the same location as the PE online library.

You can view a book, download a Portable Document Format (PDF) version of it, or download the sample programs from the *IBM Parallel Environment for AIX: Hitchhiker's Guide*.

At the time this manual was published, the Web address of the “RS/6000 SP Product Documentation Library” page was:

http://www.rs6000.ibm.com/resource/aix_resource/sp_books

However, the structure of the RS/6000 Web site may change over time.

Accessing PE Documentation Online

On the same medium as the PE product code, IBM ships PE man pages, HTML files, and PDF files. To use the PE online documentation, you must first install these filesets:

- **ppe.html**
- **ppe.man**
- **ppe.pdf**

To view the PE HTML publications, you need access to an HTML document browser such as Netscape. The HTML files and an index that links to them are installed in the

/usr/lpp/ppe.html directory. Once the HTML files are installed, you can also view them from the RS/6000 SP Resource Center.

If you have installed the SP Resource Center on your SP system, you can access it by entering this command:

```
/usr/lpp/ssp/bin/resource_center
```

If you have the SP Resource Center on CD-ROM, see the **readme.txt** file for information about how to run it.

To view the PE PDF publications, you need access to the Adobe Acrobat Reader 3.0 or later. The Acrobat Reader is shipped with the AIX Version 4.3 Bonus Pack and is also freely available for downloading from the Adobe web site at:

http://www.adobe.com

To successfully print a large PDF file (approximately 300 or more pages) from the Adobe Acrobat reader, you may need to select the “Download Fonts Once” button on the Print window.

RS/6000 SP Publications

SP Hardware and Planning Publications

The following publications are related to this book only if you run parallel programs on the IBM RS/6000 SP. These books are not related if you use an IBM RS/6000 network cluster.

- *IBM RS/6000 SP: Planning, Volume 1, Hardware and Physical Environment*, GA22-7280
- *IBM RS/6000 SP: Planning, Volume 2, Control Workstation and Software Environment*, GA22-7281

SP Software Publications

LoadLeveler Publications

- *LoadLeveler Diagnosis and Messages Guide*, GA22-7277
- *Using and Administering LoadLeveler*, SA22-7311

Parallel Environment (PE) Publications

- *IBM Parallel Environment for AIX: DPCL Class Reference*, SA22-7421
- *IBM Parallel Environment for AIX: DPCL Programming Guide*, SA22-7420
- *IBM Parallel Environment for AIX: Hitchhiker's Guide*, SA22-7424
- *IBM Parallel Environment for AIX: Installation*, GA22-7418
- *IBM Parallel Environment for AIX: Messages*, GA22-7419
- *IBM Parallel Environment for AIX: MPI Programming Guide*, SA22-7422
- *IBM Parallel Environment for AIX: MPI Subroutine Reference*, SA22-7423
- *IBM Parallel Environment for AIX: MPL Programming and Subroutine Reference*, GC23-3893
- *IBM Parallel Environment for AIX: Operation and Use, Volume 1*, SA22-7425
- *IBM Parallel Environment for AIX: Operation and Use, Volume 2*, SA22-7426

PSSP Publications

The following publications are related to this book only if you run parallel programs on the IBM RS/6000 SP. These books are not related if you use an IBM RS/6000 network cluster.

- *IBM Parallel System Support Programs for AIX: Administration Guide*, SA22-7348
- *IBM Parallel System Support Programs for AIX: Command and Technical Reference*, GA22-7351
- *IBM Parallel System Support Programs for AIX: Diagnosis Guide*, GA22-7350
- *IBM Parallel System Support Programs for AIX: Installation and Migration Guide*, GA22-7347
- *IBM Parallel System Support Programs for AIX: Messages Reference*, GA22-7352

AIX and Related Product Publications

For the latest information on AIX and related products, including RS/6000 hardware products, see *AIX and Related Products Documentation Overview*, SC23-2456. You can order a printed copy of the book from IBM. You can also view it online from the "AIX Online Publications and Books" page of the RS/6000 Web site at:

http://www.rs6000.ibm.com/resource/aix_resource/Pubs

DCE Publications

You can view a DCE book or download a PDF version of it from the IBM DCE Web site at:

http://www.ibm.com/software/network/dce/library

Red Books

IBM's International Technical Support Organization (ITSO) has published a number of redbooks related to the RS/6000 SP. For a current list, see the ITSO Web site at:

http://www.redbooks.ibm.com

Non-IBM Publications

Here are some non-IBM publications that you may find helpful.

- Almasi, G. and A. Gottlieb. *Highly Parallel Computing*, Benjamin-Cummings Publishing Company, Inc., 1989.
- Bergmark, D., and M. Pottle. *Optimization and Parallelization of a Commodity Trade Model for the SP1*. Cornell Theory Center, Cornell University, June 1994.
- Foster, I. *Designing and Building Parallel Programs*, Addison-Wesley, 1995.
- Gropp, William, Ewing Lusk, and Anthony Skjellum. *Using MPI*, The MIT Press, 1994.

  As an alternative, you can use SR28-5757 to order this book through your IBM representative or IBM branch office serving your locality.
- Koelbel, Charles H., David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance FORTRAN Handbook*, The MIT Press, 1993.
- Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard, Version 1.1*, University of Tennessee, Knoxville, Tennessee, June 6, 1995.
- Message Passing Interface Forum, *MPI-2: Extensions to the Message-Passing Interface, Version 2.0*, University of Tennessee, Knoxville, Tennessee, July 18, 1997.

- Pfister, Gregory, F. *In Search of Clusters*, Prentice Hall, 1998.
- Snir, M., Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference* The MIT Press, 1996.
- Spiegel, Murray R. *Vector Analysis* McGraw-Hill, 1959.

For more information about the Message Passing Interface Forum and the MPI standards documents, see:

http://www.mpi-forum.org

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

Communicating Your Comments to IBM

IBM Parallel Environment for AIX
Dynamic Probe Class Library
Programming Guide
Version 3 Release 1

Publication No. SA22-7420-00

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM. Whichever method you choose, make sure you send your name, address, and telephone number if you would like a reply.

Feel free to comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. However, the comments you send should pertain to only the information in this manual and the way in which the information is presented. To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

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

If you are mailing a reader's comment form (RCF) from a country other than the United States, you can give the RCF to the local IBM branch office or IBM representative for postage-paid mailing.

- If you prefer to send comments by mail, use the RCF at the back of this book.
- If you prefer to send comments by FAX, use this number:
  - FAX: (International Access Code)+1+914+432-9405
- If you prefer to send comments electronically, use one of these network IDs:
  - IBM Mail Exchange: USIB6TC9 at IBMMAIL
  - Internet e-mail: mhvrcfs@us.ibm.com

Make sure to include the following in your note:

- Title and publication number of this book
- Page number or topic to which your comment applies

Optionally, if you include your telephone number, we will be able to respond to your comments by phone.

Reader's Comments — We'd Like to Hear from You

IBM Parallel Environment for AIX
Dynamic Probe Class Library
Programming Guide
Version 3 Release 1

Publication No. SA22-7420-00

You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

Today's date: _____

What is your occupation?

Newsletter number of latest Technical Newsletter (if any) concerning this publication:

How did you use this publication?

[ ]	As an introduction	[ ]	As a text (student)
[ ]	As a reference manual	[ ]	As a text (instructor)
[ ]	For another purpose (explain)		

_____

_____

Is there anything you especially like or dislike about the organization, presentation, or writing in this manual? Helpful comments include general usefulness of the book; possible additions, deletions, and clarifications; specific errors and omissions.

Page Number: Comment:

Name	Address
Company or Organization	
Phone No.	

Reader's Comments — We'd Like to Hear from You
SA22-7420-00

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

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

Fold and Tape | **Please do not staple** | Fold and Tape

Cut or Fold
Along Line

Program Number: 5765-D93

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SA22-7420-00