Cray XD1™ Programming

Private

S–2433–131

New Features

Cray XD1™ Programming S–2433–131

This manual contains changes to address the FPGA driver enhancement that enables the FPGA to access a larger region of Opteron memory (up to 1 GB). The manual describes two new C functions in the FPGA API that support this enhancement:

- `fpga_register_ftrmem`(3)
- `fpga_dereg_ftrmem`(3)

Record of Revision

Version	*Description*
1.3.1	October 2005 Converted to new document format. Minor editorial changes.
1.3	July 2005 Updates to support the new and changed features in Cray XD1 release 1.3 (limited availability).
1.2.1	May 2005 Minor corrections.
1.2	April 2005 Updates to support the new and changed features in Cray XD1 release 1.2 (limited availability).
1.1	October 2004 Updates to support the new and changed features in Cray XD1 release 1.1.
1.0	August 2004 Initial release; supports Cray XD1 release 1.0.

Contents

Page

Preface **ix**

Accessing Product Documentation ix

Conventions x

Reader Comments xi

Cray XD1 Support xi

Introduction [1] **1**

Who Should Read this Manual 1

Scope of this Manual 1

How this Manual is Organized 1

Related Publications 2

Programming Environment [2] **5**

The Cray XD1 Environment 5

System Description 5

Operating System 5

Linux Synchronized Scheduler (LSS) 6

Driver for the RapidArray Interconnect 6

Driver for the FPGA Application Acceleration Processor 6

Sockets Direct Protocol 7

User Environment 7

Job Environment 8

Accessing the System 8

Development Tools 9

Standard Tools 9

Optional Tools 10

Local and Other Third-party Tools 10

Page

Libraries 10

Using Tools and Libraries [3] 15

General Compiling and Linking Considerations 15

Required Compiler Options 15

Dynamic Linking Versus Static Linking 15

Using the MPI Libraries 16

MPICH Libraries 16

Using Compiler Scripts to Build MPI Applications 17

Available Compiler Scripts 17

Setting Your PATH Variable 18

Example 1: Adding an instance of MPICH to your PATH variable 18

Invoking a Compiler Script 18

Manually Compiling and Linking MPI Applications 19

Include File Path 20

Example 2: Specifying the MPICH header file location 20

Linking the Main MPICH Library 20

Linking Other Required Libraries 21

Combined Examples 21

Example 3: Manually building an MPI application with the GNU C compiler with static linking 21

Example 4: Manually building an MPI application with the PGI Fortran 90 compiler with dynamic linking 22

Building In the Path to the Shared Library 22

Example 5: Manually building an MPI application with the GNU FORTRAN 77 compiler with dynamic linking 22

Using Other Libraries and Tools 22

ACML 23

Apprentice^2 24

ARMCI 24

CrayPAT 25

FPGA Application Acceleration Processor API 26

Page

Global Arrays 26
GPSHMEM 26
PAPI 27
ScaLAPACK 28
Using the Modules Package to Configure Your Environment 29
Overview of the Modules Package 29
Introduction to the `module` Command 29
Predefined Modulefiles 30
Developing Other Modulefiles from Templates 30
Compiler Modulefile Template 30
Example 6: Template for a PGI compiler modulefile 31
MPICH Library Modulefile Template 32
Example 7: Template for an MPICH library modulefile 32
Building an MPICH Library Instance 33
Obtaining the MPICH Source Code 33
Procedure 1: To obtain the MPICH source code 33
Example 8: Accessing the source disc image 33
Example 9: Copying the MPICH source package to the Cray XD1 system 34
Compiling the MPICH Library 34
Procedure 2: To compile the MPICH library 34
Deploying the MPICH Library Instance 36
Procedure 3: To deploy the MPICH library instance 36

Using the FPGA Application Acceleration Processor [4] 39

Overview 39
Preparing an FPGA Logic File 42
Developing a Raw FPGA Logic File 42
Converting a Raw Logic File to Loadable Form 43
Procedure 4: To convert a raw logic file to loadable form 43
Managing FPGA Logic from the Command Line 44
Loading FPGA Logic into the Device 44

Page

Resetting an FPGA 44
Releasing an FPGA from Reset State 45
Querying the Status of an FPGA 45
Erasing an FPGA 45
Managing FPGA Logic in an Application Program 45
Using an FPGA in Application Programs 46
Typical Application Workflow 46
Understanding Address Spaces on a Node 46
Data Transfer Methods 48
Using an FPGA in a C Program 48
Typographic Conventions 48
Library Files 49
Opening an FPGA 49
Loading FPGA Logic into the Device 50
Resetting an FPGA 51
Releasing an FPGA from Reset State 52
Mapping FPGA Locations to the Application Address Space 53
Synchronizing Accesses to FPGA Locations 55
Writing and Reading Individual FPGA Locations 57
Accessing Application Memory from an FPGA 60
Checking the Status of an FPGA 62
Checking the Programming State of an FPGA 63
Erasing an FPGA 64
Closing an FPGA 65
Sample Application: Using the Mersenne Twister Accelerator 66
Algorithm 66
High-level Design of Application and FPGA Logic 67
Some Design Details 67
Walkthrough 69
Getting Started with the FPGA 70

Page

Procedure 5: To get started with the FPGA 70

Appendix A Program Listing: `mta_test.c` **75**

Glossary **87**

Index **93**

Tables

Table 1. Related publications 2
Table 2. Software development tools in the Cray XD1 software distribution 9
Table 3. Key software libraries in the Cray XD1 software distribution 11
Table 4. Required compiler options 15
Table 5. MPICH libraries in the Cray XD1 software distribution 16
Table 6. MPICH subdirectories 17
Table 7. MPI compiler scripts 18
Table 8. Using MPICH 19
Table 9. Using ROMIO 20
Table 10. Resolving MPI references 21
Table 11. Using ACML 23
Table 12. Using Apprentice[2] 24
Table 13. Using ARMCI 25
Table 14. Using CrayPAT 25
Table 15. Using the FPGA application acceleration processor API 26
Table 16. Using Global Arrays 26
Table 17. Using GPSHMEM 27
Table 18. Using PAPI 27
Table 19. Using ScaLAPACK 28
Table 20. Common `module` subcommands 29
Table 21. `fpga_open`(3) arguments and return value 50
Table 22. `fpga_load`(3) arguments and return value 51
Table 23. `fpga_reset`(3) arguments and return value 52

Page

Table 24. `fpga_start`(3) arguments and return value 53
Table 25. `fpga_memmap`(3) arguments and return value 54
Table 26. `fpga_mem_sync`(3) arguments and return value 56
Table 27. `fpga_wrt_appif_val`(3) arguments and return value 58
Table 28. `fpga_rd_appif_val`(3) arguments and return value 60
Table 29. `fpga_register_ftrmem`(3) and `fpga_dereg_ftrmem`(3) arguments and return value 62
Table 30. `fpga_status`(3) arguments and return value 63
Table 31. `fpga_is_loaded`(3) arguments and return value 64
Table 32. `fpga_unload`(3) arguments and return value 65
Table 33. `fpga_close`(3) arguments and return value 66
Table 34. MTA registers 68

Figures

Figure 1. Applications and communication libraries 13
Figure 2. Development workflow for FPGA applications 41
Figure 3. Physical components of a node and related address spaces 47

Preface

The information in this preface is common to Cray documentation provided with this software release.

Accessing Product Documentation

With each software release, Cray provides books and man pages, and in some cases, third-party documentation. These documents are provided in the following ways:

CrayDoc

The Cray documentation delivery system that allows you to quickly access and search Cray books, man pages, and in some cases, third-party documentation. Access this HTML and PDF documentation via CrayDoc at the following locations:

- The local network location defined by your system administrator
- The CrayDoc public website: `docs.cray.com`

Man pages

Access man pages by entering the `man` command followed by the name of the man page. For more information about man pages, see the `man`(1) man page by entering:

```
% man man
```

Third-party documentation

Access third-party documentation not provided through CrayDoc according to the information provided with the product.

Conventions

These conventions are used throughout Cray documentation:

Convention	Meaning
`command`	This fixed-space font denotes literal items, such as file names, pathnames, man page names, command names, and programming language elements.
variable	Italic typeface indicates an element that you will replace with a specific value. For instance, you may replace *filename* with the name `datafile` in your program. It also denotes a word or concept being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
[ ]	Brackets enclose optional portions of a syntax representation for a command, library routine, system call, and so on.
. . .	Ellipses indicate that a preceding element can be repeated.
`name`(N)	Denotes man pages that provide system and programming reference information. Each man page is referred to by its name followed by a section number in parentheses. Enter: `% man man` to see the meaning of each section number for your particular system.

Reader Comments

Contact us with any comments that will help us to improve the accuracy and usability of this document. Be sure to include the title and number of the document with your comments. We value your comments and will respond to them promptly. Contact us in any of the following ways:

E-mail:
`docs@cray.com`

Telephone (inside U.S., Canada):
1–800–950–2729 (Cray Customer Support Center)

Telephone (outside U.S., Canada):
+1–715–726–4993 (Cray Customer Support Center)

Mail:
Software Publications
Cray Inc.
1340 Mendota Heights Road
Mendota Heights, MN 55120–1128
USA

Cray XD1 Support

Obtain support for the Cray XD1 product in either of the following ways:

Telephone:
1–888–279–2729 (Cray XD1 Customer Support Center)

Through the CRInform website:
`http://crinform.cray.com/xd/`

Note: Use the contact information provided here if you have a support agreement with Cray. If, however, you have a support agreement with a third-party organization that is a Cray channel partner, contact that organization instead: do not contact Cray directly.

Introduction [1]

This chapter identifies the intended audience, describes the scope and organization of the manual, and lists related publications.

1.1 Who Should Read this Manual

This manual is intended for experienced programmers who want to develop application software to run on the Cray XD1 system. The specific prerequisites for understanding this manual include working knowledge of the following items:

- The Linux command-line environment
- The C programming language
- The standard tools for developing C programs (editors, compilers, debuggers, and so on)
- The architecture and operational concepts of the Cray XD1 system

1.2 Scope of this Manual

This manual is a guide to the programming environment of the Cray XD1 system and a primer on how to use the optional application acceleration processors in an application program. The application acceleration processors are field-programmable gate arrays (FPGAs).

This manual is not a guide to programming in general or to high performance computing (HPC) in particular. It does not provide information about how to design parallel programs, and is not a general guide to the common libraries that are used in HPC applications. It does not describe how to design FPGA logic; for an overview of that process, see *Cray XD1 FPGA Development* (S–6400).

1.3 How this Manual is Organized

This manual consists of the following chapters and appendix:

- Chapter 2: Programming Environment

  Summarizes the programming environment of the Cray XD1 system.

- Chapter 3: Using Tools and Libraries

  Provides information for compiling and linking applications with the libraries in the Cray XD1 software distribution and describes the Modules package.

- Chapter 4: Using the FPGA Application Acceleration Processor

  Describes the utility program and the application programming interface (API) library that enable you to use the optional FPGA application acceleration processors. A sample program illustrates the API.

- Appendix A: Program Listing: mta_test.c

  A full source-code listing of the sample program that is discussed in Chapter 4.

1.4 Related Publications

Refer to the publications in Table 1, page 2 for related information about the Cray XD1 system.

Table 1. Related publications

Publication title	Brief description
Cray XD1 Release Description (S–2453)	Identifies the main new features and enhancements in a particular release of the product. Includes information about the hardware, embedded software, and Linux-based software of the system.
Cray XD1 System Overview (S–2429)	Overview of the Cray XD1 computer and a description of its hardware and software components.
Cray XD1 System Administration (S–2430)	System administration and monitoring. Also includes all end-user topics such as submitting jobs.
Cray XD1 FPGA Development (S–6400)	Overview of the process and tools for developing FPGA logic files.

Publication title	Brief description
Design of Cray XD1 RapidArray Transport Core (S–6411)	Companion document to *Cray XD1 FPGA Development* (S–6400). Provides in-depth design details.
Design of Cray XD1 QDR II SRAM Core (S–6412)	Companion document to *Cray XD1 FPGA Development* (S–6400). Provides in-depth design details.
Cray XD1 Release Notes (S–2455)	Information about resolved issues and known issues for a particular release of the product.

Programming Environment [2]

This chapter is a summary of the programming environment of the Cray XD1 system. It identifies the tools and libraries that are available to programmers who develop applications to run on the system. It also provides an overview of how you access the system.

2.1 The Cray XD1 Environment

This section briefly describes the Cray XD1 system, the operating system details that are relevant to programmers, and the system's user environment and job execution environment.

2.1.1 System Description

A node in the Cray XD1 system is an instance of the Linux operating system and the hardware components that it controls, including an Opteron symmetric multiprocessor (SMP), one or two RapidArray processors which interface to the RapidArray interconnect, and an optional field-programmable gate array (FPGA) application acceleration processor. Each Cray XD1 chassis has six nodes.

For a full description of the hardware components of the system, see *Cray XD1 System Overview* (S–2429).

2.1.2 Operating System

The Linux system that runs on the SMP of each node is based on a 64-bit SuSE Linux Enterprise Server (SLES) distribution. For information on the specific versions of the SuSE distribution and the Linux kernel in the Cray XD1 software distribution, see *Cray XD1 Release Description* (S–2453).

Cray has enhanced the Linux distribution to support high performance computing (HPC) applications that run on the Cray XD1 system. The enhancements are as follows:

- Linux synchronized scheduler
- Driver for the RapidArray interconnect
- Driver for the FPGA application acceleration processor
- Sockets Direct Protocol

The following subsections describe these features.

2.1.2.1 Linux Synchronized Scheduler (LSS)

In the Cray XD1 system, administrators can specify synchronized scheduling for the processes of batch jobs. This enhanced process scheduling method provides two advantages that improve the performance of parallel applications:

- Time slots on all nodes in a partition synchronize to a common time source with microsecond accuracy. This reduces the time that job processes lose while they wait to perform a collective operation.
- The scheduler applies a batch scheduling algorithm to job processes, which gives them long uninterrupted time slots and restricts overhead processes to relatively short segments of a scheduling cycle. This gives applications more processor time overall and reduces time lost to context switches.

You do not need to change application programs to take advantage of this feature. Administrators can enable it for selected partitions; for details, see *Cray XD1 System Administration*.

2.1.2.2 Driver for the RapidArray Interconnect

The device driver for the RapidArray interconnect, which is the principal mechanism for interprocessor communication in a Cray XD1 system, provides transparent access to this high-bandwidth, low-latency internal network. This driver is statically linked into the kernel. Cray customized the Message Passing Interface (MPI) library and the Aggregate Remote Memory Copy Interface (ARMCI) library to use the RapidArray interconnect directly. Computing jobs that use these libraries benefit automatically from the use of the interconnect. In addition, the system supports IP communication over the RapidArray interconnect, so any system or application process can use it.

2.1.2.3 Driver for the FPGA Application Acceleration Processor

The system includes a device driver that supports communication between an application and the optional FPGA application acceleration processor. This driver is dynamically linked into the kernel and is present only if the node that is running the Linux instance has an expansion module that includes the FPGA application acceleration processor. For more information on these and other components of the Cray XD1 system, see *Cray XD1 System Overview* (S–2429).

For information about using the FPGA application acceleration processor in an application program, see Chapter 4, page 39.

2.1.2.4 Sockets Direct Protocol

Cray XD1 Linux provides an implementation of the Sockets Direct Protocol (SDP) to accelerate applications that rely heavily on interprocessor communication via TCP/IP. Applications that are specified in a system configuration file transparently use SDP over the RapidArray interconnect instead of TCP/IP over RapidArray. You do not need to change source code or even relink applications to take advantage of this feature.

For information about using SDP, see *Cray XD1 System Administration* (S–2430).

2.1.3 User Environment

Each node in the Cray XD1 system runs Linux and potentially provides a working environment for the programmer. The nodes that are actually available for your use depend on how the administrator configured the system through the Active Manager software.

The Cray XD1 system implements a flexible concept of partitions—collections of nodes that can function as single logical computers. For partitions that the Active Manager software manages[1], the administrator must configure an attribute of a partition so that users can log in for interactive work. Normally, you log in to a partition (rather than directly to a particular node) and the system assigns your login session to a node in the partition. This distributes the load.

Before you can log in, you must have a user ID that the system recognizes. In addition, you must belong to a Linux group that is configured to access the partition you want to use. Ask your administrator to set up these items for you.

An administrator can restrict access to a partition at any time. Each partition has an access control setting that determines whether the partition is open (accepts logins or jobs, depending on how it is configured) or closed (does not accept logins or jobs). Each node also has a similar setting. Before you can log in to a node, both the node and the partition it is in must be open.

When you log in to Linux on a Cray XD1 node, you have a normal Linux environment. You can use the command line or an X Window environment such as Gnome or KDE according to your own preferences. For programming purposes, you must log in to a partition that is configured to use the full set of software packages available in the distribution. Consult your administrator to determine which partition is suitable.

[1] The system also allows custom partitions that the Active Manager software creates but does not manage.

2.1.4 Job Environment

If you develop compute-intensive applications, end users will typically run them as batch jobs. Normally, the administrator defines at least one partition with the job execution and synchronized scheduling attributes enabled (and the login attribute disabled). Such a partition is reserved for executing batch jobs.

An end user logs in to Linux in a login partition or logs in to the Active Manager graphical user interface (GUI) and submits a job to a specified partition. As part of job submission, the user specifies the resources the job needs, such as the number of processes and whether the job uses FPGA application acceleration processors.

The workload management (WLM) system for which the system is configured queues the job and launches it when the requested resources are available. The WLM system determines which nodes in the partition run the job—the user is not concerned with this. The user can see the status of the job in the Active Manager GUI or in the user interfaces of the WLM system.

2.2 Accessing the System

See *Cray XD1 System Administration* (S–2430) for details on how to access the Cray XD1 system either through the browser-based Active Manager GUI or through a Linux shell. Although you will log in to Linux for most of your development activities, you can also submit jobs through the GUI.

The following points summarize the prerequisites for accessing the Linux environment on the Cray XD1 system:

- Ensure that the Cray XD1 system is accessible from your LAN. Normally, the administrator sets up this access when he or she commissions and sets up the system. You need to know the fully qualified domain name (FQDN) of the system.
- Ensure that you have a user ID on the Cray XD1 system and that you know your password.
- Ensure that your home directory and other resources on your LAN will be accessible (if you need them) when you log in. Consult the administrator.
- Ensure that you have client software installed on your workstation for a remote login method that the system supports. The system supports only `ssh` by default.

- Identify the names of the partitions to use for your development activities and for executing jobs. Ensure that the administrator grants you access to these partitions.

When you satisfy these prerequisites, log in to Linux on a node in the Cray XD1 system by using the following command:

ssh *user@partition.system-fqdn*

For example, if your user name is `jsmith`, the site domain name is `mycompany.com`, the Cray XD1 system domain is `crayxd1`, and the partition for development work is `dev`, you log in as follows:

```
ssh jsmith@dev.crayxd1.mycompany.com
```

2.3 Development Tools

The Cray XD1 software distribution includes the standard set of Linux software development tools. In addition, Cray offers some optional third-party tools. The administrator can also install other local or third-party tools.

2.3.1 Standard Tools

Table 2, page 9 summarizes the standard software development tools that the Cray XD1 software distribution includes. The table lists only the major tools of interest in each category; in some cases, other minor tools are also in the distribution.

Table 2. Software development tools in the Cray XD1 software distribution

Category	Tool	Comments
Text editors	`vim`	`Vi` IMproved
	`emacs`	
Compilers	`gcc`	GNU C compiler
	`g++`	GNU C++ compiler
	`g77`	GNU FORTRAN 77 compiler
Debuggers	`gdb`	GNU debugger for programs in various languages

Category	Tool	Comments
Scripting	`perl`	
	`python`	
Documentation	`groff`	GNU implementation of the `troff` document formatting system. Includes a macro package for coding man pages
	`info`, `makeinfo`	Texinfo system of online documentation

2.3.2 Optional Tools

Other third-party development tools may be available for the Cray XD1 system. For the current list of third-party tools that Cray sells and supports, consult your Cray representative.

2.3.3 Local and Other Third-party Tools

One advantage of the AMD Opteron processors in the Cray XD1 system is that they support both 32–bit and 64-bit applications. Many existing local and third-party software products can run on the Cray XD1 system.

If you have your own development tools that you want to use on the system, ask your administrator to install them. The procedure to install such software varies depending on when the installation occurs and where you want the software to be available; for a more detailed discussion, see the chapter on setting up the system in *Cray XD1 System Administration* (S–2430). Ideally, the same software is available on all nodes in a partition. Ensure that you have the proper licenses for the scope of your proposed installation.

2.4 Libraries

HPC applications typically depend heavily on subprogram libraries in two main areas: mathematical and scientific routines, and communication routines. Table 3, page 11 summarizes the major libraries in the Cray XD1 software distribution or optionally available from Cray that are of general interest for HPC applications. For information on the specific version of each library, see *Cray XD1 Release Description* (S–2453).

Table 3. Key software libraries in the Cray XD1 software distribution

Category	Library	Comments
General	GNU C Library (`glibc`)	Standard Linux C library; designed for portability and high performance. Conforms to ISO C, POSIX, and other standards.
Mathematical	AMD Core Math Library (ACML)	Three standard libraries which are optimized for the AMD Opteron processor: • Basic Linear Algebra Subprograms (BLAS) • Fast Fourier Transform (FFT) • Linear Algebra Package (LAPACK) Includes both C and Fortran interfaces.
	Scalable Linear Algebra Package (ScaLAPACK)	High-performance linear algebra routines similar to LAPACK; designed specifically for distributed-memory message-passing computers.
Communication	Message Passing Interface (MPI)	The major synchronous (2-sided) communication standard for HPC. Cray modified this implementation to work with the Cray XD1 RapidArray interconnect; the API is unchanged.
	ROMIO	The Argonne National Laboratory implementation of MPI-IO.

Category	Library	Comments
	Aggregate Remote Memory Copy Interface (ARMCI)	Foundation of asynchronous (1-sided) communication functions. Not used directly by application programs. Originally part of the GA distribution. Cray separated it and modified it to work with the Cray XD1 RapidArray interconnect. Uses some routines in MPI.
	Global Arrays (GA)	Provides a shared-memory interface for distributed-memory computers. Uses ARMCI.
	Generalized Portable SHMEM (GPSHMEM)	Portable implementation of the original Cray Shared Memory (SHMEM) interface. Uses both ARMCI and MPI. Cray modified the implementation to work with its implementation of ARMCI.
FPGA application acceleration processor	Cray XD1 FPGA application acceleration processor interface library (`libufp.a`)	Functions to control FPGA application acceleration processors. See Chapter 4, page 39
Performance analysis	Performance Application Programming Interface (PAPI)	A standard API for accessing the hardware performance counters that are available on most modern microprocessors.
	Cray Performance Analysis Tool (CrayPAT)	Optional set of Cray tools for performance instrumentation and measurement. Supports use of a run-time library for collecting performance data during execution without modifications to source code. Apprentice^2 is a related optional tool that displays graphical analyses of the CrayPAT data.

Table 3, page 11 indicates the dependencies among the Cray XD1 implementations of the libraries. These dependencies are as follows:

- GPSHMEM uses both ARMCI and MPI.
- Global Arrays uses ARMCI.
- ARMCI uses MPI.

Figure 1, page 13 illustrates these relationships.

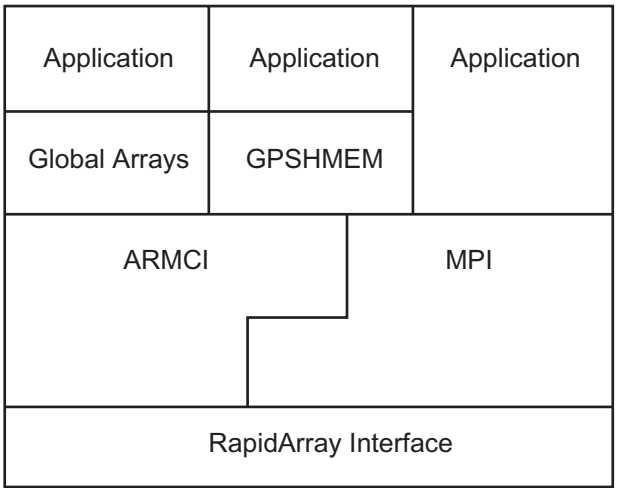

Figure 1. Applications and communication libraries

In addition to the major libraries in the table, which are applicable to all application domains, the Cray XD1 software distribution provides many other libraries that may be useful in particular application domains. These include support for graphics, image processing, cryptography, audio processing, networking, and so on. These libraries are well known in the Linux community and are outside the scope of this manual.

Using Tools and Libraries [3]

This chapter provides specific information for compiling and linking applications that use the main mathematical and communications libraries in the Cray XD1 software distribution. It focuses primarily on using the Message Passing Interface (MPI) library. It also recommends an approach to configuring your Cray XD1 environment for application development and execution.

3.1 General Compiling and Linking Considerations

If you do not have predefined environment modules to configure your shell environment (see Section 3.4, page 29), you can use the general information in the following subsections and the specific information about various libraries in the rest of this chapter to set appropriate options and paths.

3.1.1 Required Compiler Options

Table 4, page 15 lists the options that Cray recommends for various compilers to build applications for the Cray XD1 system.

Table 4. Required compiler options

Compiler	Version	Required options
GNU C	3.3.3	`-mcpu=opteron -m64`
GNU Fortran 77	3.3.3	`-mcpu=opteron -m64`
PathScale	all	`-m64` (this is the default)
PGI	all	`-tp k8-64`

3.1.2 Dynamic Linking Versus Static Linking

Cray recommends that you use shared libraries, that is, dynamically linked libraries, whenever possible. An application executable file that uses shared libraries is much more likely to run without change across successive releases of the Cray XD1 software distribution.

For example, if the Cray XD1 implementation of the MPICH library changes to accommodate a change in the operating system support for the RapidArray

interconnect, the change is completely transparent to an application that uses dynamic linking. In the same circumstances, however, a statically linked application may fail after the software upgrade.

When your application uses dynamic linking, all the nodes in a partition that runs the application must have a consistent set of dynamic libraries. Similarly, the environment variable for the dynamic library path (`LD_LIBRARY_PATH`) must also be consistent in those nodes (unless you build the path in to the executable; see Section 3.2.3.5, page 22).

3.2 Using the MPI Libraries

The Cray XD1 system has features that make it an excellent platform for running MPI-based parallel applications. This section focuses on the information that you need to build executable MPI applications from source files. It does not cover the design of MPI applications.

3.2.1 MPICH Libraries

Cray provides several instances of the MPICH object library for the Cray XD1 system. Each instance of the library is compiled from the same source code with a different compiler, including some compilers that are not in the Cray XD1 software distribution. Each Cray XD1 release includes the MPICH libraries that are available at the time of release. Cray may generate additional instances of the library between releases, and these are available on the CRInform website; for the URL, see "Cray XD1 Support," page xi.

Table 5, page 16 lists the MPICH libraries in the present release.

Table 5. MPICH libraries in the Cray XD1 software distribution

Compiler Vendor	Compiler Version	Path to MPICH resources
GNU	3.3.3	`/usr/mpich/mpich-1.2.6`
PGI	5.2-4	`/usr/mpich/mpich-1.2.6-pgi524`
PGI	6.0.1	`/usr/mpich/mpich-1.2.6-pgi601`
PathScale	2.0	`/usr/mpich/mpich-1.2.6-path20`

In the following sections, a particular MPICH directory is represented generically by the *mpich-dir* variable notation.

Each of these directories has the same subdirectory structure. Table 6, page 17 lists the subdirectories.

Table 6. MPICH subdirectories

MPICH subdirectory	Contents
`lib`	Static MPICH object libraries
`lib/shared`	Dynamic MPICH object libraries
`include`	MPICH include files
`bin`	MPICH tools and utilities

3.2.2 Using Compiler Scripts to Build MPI Applications

Each instance of the MPICH library on the Cray XD1 system includes a `bin` directory which contains executable tools and utilities that are customized for that instance of the library. In particular, the `bin` directory includes scripts that compile and link an application with the same compiler that was used to compile the library.

These scripts specify all the required parameters, including the following items:

- Directory path for MPICH include files
- Directory paths for MPICH libraries (static and shared)
- All libraries on which the Cray XD1 MPICH libraries depend

Cray recommends the use of these compiler scripts when you build your MPI applications.

3.2.2.1 Available Compiler Scripts

Table 7, page 18 lists the MPI compiler scripts that are in the Cray XD1 software distribution.

Table 7. MPI compiler scripts

Compiler vendor	MPI compiler scripts
GNU	`mpicc`, `mpiCC`, `mpif77`
PGI	`mpicc`, `mpiCC`, `mpif77`, `mpif90`
PathScale	`mpicc`, `mpiCC`, `mpif77`, `mpif90`

3.2.2.2 Setting Your PATH Variable

To use the compiler script that matches the MPICH library that you want, ensure that your PATH environment variable specifies both the path to the correct library `bin` directory and the path to the corresponding compiler. For a library other than the GNU MPICH library, you must prepend its `bin` directory to your default PATH because the latter always includes the path to the GNU MPICH `bin` directory.

Example 1: Adding an instance of MPICH to your PATH variable

If you use the `bash` shell, set your PATH variable as follows to access the scripts that use the PGI 5.2-4 compilers:

```
> export PATH=/usr/mpich/mpich-1.2.6-pgi524/bin:$PATH
```

In this case, you also have to add the path for the compiler itself to your PATH variable (not shown in the example).

3.2.2.3 Invoking a Compiler Script

Once your PATH variable is fully defined, you can build your application. To do so, use the compiler script name in place of the native compiler name. For example, use `mpicc` instead of `pgcc`. Do this whether you issue the command from the command line or in a makefile.

The best way to establish the right environment for using an MPI compiler script is to use a modulefile as described in Section 3.4, page 29. If you prepare a suitable modulefile and you use an MPI compiler script, you can very easily build your application with different compilers. For example, the commands to compile and link the MPI application `my_app` with the PGI 5.2-4 C compiler could be as simple as the following example:

```
module add pgi/5.2-4
mpicc my_app
```

3.2.3 Manually Compiling and Linking MPI Applications

If you do not use an MPI compiler script to build an application, you must manually specify all the required options and arguments when you run the compiler. Table 8, page 19 summarizes the paths for include files and libraries. This information is repeated in the remainder of this section in the context of the required compiler command arguments. The *mpich-dir* variable notation represents the path to the particular instance of the MPICH library. For the various values of *mpich-dir*, see Section 3.2.1, page 16.

Table 8. Using MPICH

	Detail	**Comment**
Include file path	*mpich-dir*`/include/mpi.h`	C header file.
	mpich-dir`/include/mpif.h`	FORTRAN 77 include file
Object library path	*mpich-dir*`/lib/libmpich.a`	Static main library.
	mpich-dir`/lib/shared/libmpich.so`	Shared main library.
	mpich-dir`/lib/libfmpich.a`	Static additional library for Fortran programs.
	mpich-dir`/lib/shared/libfmpich.a`	Shared additional library for Fortran programs.
Documentation	*MPI—The Complete Reference; Volume 1, The MPI Core*, M. Snir et al., The MIT Press	
	`> man MPI`	
	and man pages for individual routines	
	`http://www-unix.mcsl.gov/mpi/mpich`	Argonne National Laboratory website for MPICH

Table 9, page 20 provides supplementary information for the related ROMIO library. This implementation of the MPI-IO library is part of the MPICH library; you do not have to specify a separate object library path.

Table 9. Using ROMIO

	Detail	Comment
Include file path	*mpich-dir*/include/mpio.h	C header file.
	mpich-dir/include/mpiof.h	Fortran include file
Documentation	Chapter 3 of *Using MPI-2: Advanced Features of the Message-Passing Interface*, W. Gropp et al., The MIT Press	
	Man pages for individual routines; for example, > **man MPI_File_open**	
	http://www-unix.mcs.anl.gov/romio/	

Note: The remainder of this section uses the common command-line option names. You may need to translate the option names into the equivalent ones for your specific compiler.

3.2.3.1 Include File Path

Specify the path at which the compiler can find source include files for MPICH as follows:

-I*mpich-dir***/include**

Example 2: Specifying the MPICH header file location

For the MPICH 1.2.6 library that was compiled with the GNU compilers, use the following command:

```
> gcc -I/usr/mpich/mpich-1.2.6/include
  other-options files
```

3.2.3.2 Linking the Main MPICH Library

Table 10, page 21 lists the options and arguments that you include to resolve direct references to MPI routines in various languages. The table shows both the statically linked and dynamically linked cases. Include the options and arguments in your command line in the same order that they appear in the table.

Table 10. Resolving MPI references

Language	Static linking	Dynamic linking
C	**-L***mpich-dir***/lib -lmpich**	**-shared -L***mpich-dir***/lib/shared -L***mpich-dir***/lib -lmpich**
C++	**-L***mpich-dir***/lib -lpmpich++ –lmpich**	Not available
FORTRAN 77	**-L***mpich-dir***/lib -lmpich**	**-shared -L***mpich-dir***/lib/shared -L***mpich-dir***/lib -lmpichfarg –lmpich**
Fortran 90	**-L***mpich-dir***/lib -lmpich**	**-shared -L***mpich-dir***/lib/shared -L***mpich-dir***/lib -lmpichf90 –lmpichfarg –lmpich**

3.2.3.3 Linking Other Required Libraries

The Cray XD1 MPICH library source code has been customized to perform interprocessor communication directly over the RapidArray network. Consequently, when you link an MPI application, you must specify two additional libraries, as follows:

```
-L/usr/local/lib64 -lrapl -lpthread
```

Place the two `-l` options at the end of the list of libraries in the command.

3.2.3.4 Combined Examples

Example 3, page 21 shows a command to compile and statically link an MPI application with the GNU C compiler.

Example 3: Manually building an MPI application with the GNU C compiler with static linking

```
> gcc -I/usr/mpich/mpich-1.2.6/include
  –L/usr/mpich/mpich–1.2.6/lib -L/usr/local/lib64 -lmpich
  –lrapl –lpthread other-options
  files
```

Example 4, page 22 shows a command to compile and dynamically link an MPI application with the PGI Fortran 90 compiler version 5.2-4.

Example 4: Manually building an MPI application with the PGI Fortran 90 compiler with dynamic linking

```
> pgf90 -I/usr/mpich/mpich-1.2.6-pgi524/include -shared
  –L/usr/mpich/mpich–1.2.6-pgi524/lib/shared
  –L/usr/mpich/mpich–1.2.6-pgi524/lib -L/usr/local/lib64
  –lmpichf90 -lmpichfarg –lmpich –lrapl –lpthread
  other-options files
```

3.2.3.5 Building In the Path to the Shared Library

When you build an application to use the shared library, you may also want to build in the path that allows the system to resolve calls dynamically at run time. If you do not do so, users of the application must set the path in their LD_LIBRARY_PATH environment variable.

With the GNU compilers, you can build in the shared library path by including the following options in the compiler command:

-Wl,-rpath -Wl,*mpich-dir***/shared**

where *mpich-dir* is the appropriate path from Table 5, page 16.

Example 5, page 22 shows a command to compile and dynamically link an MPI application with the GNU FORTRAN 77 compiler. This example builds in the path to the shared library.

Example 5: Manually building an MPI application with the GNU FORTRAN 77 compiler with dynamic linking

```
> g77 -I/usr/mpich/mpich-1.2.6/include -shared
  –L/usr/mpich/mpich–1.2.6/lib/shared
  –L/usr/mpich/mpich–1.2.6/lib -L/usr/local/lib64 -lmpichfarg
  –lmpich –lrapl –lpthread
  -Wl,-rpath -Wl,/usr/mpich/mpich-1.2.6/lib/shared
  other–options files
```

3.3 Using Other Libraries and Tools

This section gives the specific information that you need to compile and link programs that use each of the main HPC libraries in the Cray XD1 software distribution, to access the available performance analysis tools, and to access documentation for these libraries and tools. Libraries and tools of

all types—communications, mathematical, and other—are described here in alphabetical order.

If a predefined environment module is provided with the library, it is also identified in this section. For a general description of environment modules, see Section 3.4, page 29.

3.3.1 ACML

Table 11, page 23 lists information about the AMD Core Math Library (ACML) on the Cray XD1 system.

Table 11. Using ACML

	Detail	Comment
Include file paths	`/opt/acml2.0/gnu64/include/acml.h`	C header file. Use with GNU compilers.
	`/opt/acml2.0/pgi64/include/acml.h`	C header file. Use with PGI compilers.
	`/opt/acml2.0/pgi64_mp/include/acml.h`	C header file. Use with PGI compilers and OpenMP.
Object library paths	`/opt/acml2.0/gnu64/lib/libacml.a`	Static. Use with GNU compilers.
	`/opt/acml2.0/gnu64/lib/libacml.so`	Shared. Use with GNU compilers.
	`/opt/acml2.0/pgi64/lib/libacml.a`	Static. Use with PGI compilers.
	`/opt/acml2.0/pgi64/lib/libacml.so`	Shared. Use with PGI compilers.

	Detail	Comment
	`/opt/acml2.0/pgi64_mp/lib/libacml.a`	Static. Use with PGI compilers and OpenMP.
	`/opt/acml2.0/pgi64_mp/lib/libacml.so`	Shared. Use with PGI compilers and OpenMP.
Documentation	The user guide, *AMD Core Math Library (ACML)*, is installed with the software at `/opt/acml2.5-64bit/Doc/acml.pdf`. Other formats are also present.	

3.3.2 Apprentice^2

Table 12, page 24 lists information about the optional Apprentice^2 tool on the Cray XD1 system. Use this tool in conjunction with CrayPAT to analyze performance data graphically.

Table 12. Using Apprentice^2

	Detail	Comment
Environment module	`> module use /opt/modulefiles` `> module load apprentice2`	
Documentation	`> man app2` Online help in the tool.	

3.3.3 ARMCI

Table 13, page 25 lists information about the Aggregate Remote Memory Copy Interface (ARMCI) on the Cray XD1 system.

Table 13. Using ARMCI

	Detail	Comment
Include file path	`/usr/local/include/armci.h`	C header file.
Object library path	`/usr/local/lib64/libarmci.a`	
Documentation	`http://www.emsl.pnl.gov/docs/parsoft/armci/`	

3.3.4 CrayPAT

Table 14, page 25 lists information about the optional Cray Performance Analysis Tool (CrayPAT) on the Cray XD1 system.

Table 14. Using CrayPAT

	Detail	Comment
Environment module	> **`module use /opt/modulefiles`** > **`module load craypat`**	
Include file path	`/opt/xd-pe/craytools/1.0.1/cpatx/include/hwpc.h`	C header file. The include path is not required in the compile step if you load the specified modulefile.
	`/opt/xd-pe/craytools/1.0.1/cpatx/include/hwpcf.h`	Fortran include file. The include path is not required in the compile step if you load the specified modulefile.
Object library path	`/opt/xd-pe/craytools/1.0.1/cpatx/lib/libhwpc.a`	The library path is not required in the link step if you load the specified modulefile.
Documentation	> **`man hwpc`** For information about the CrayPAT command-line tools: > **`man pat`**	

3.3.5 FPGA Application Acceleration Processor API

Table 15, page 26 lists information about the field-programmable gate array (FPGA) application acceleration processor's application programming interface (API) on the Cray XD1 system.

Table 15. Using the FPGA application acceleration processor API

	Detail	Comment
Include file path	`/usr/local/include/ufplib.h`	C header file
Object library path	`/usr/local/lib64/libufp.a`	
Documentation	`> man fpga_intro` and man pages for individual functions	
	Chapter 4, page 39	In this manual.

3.3.6 Global Arrays

Table 16, page 26 lists information about the Global Arrays (GA) library on the Cray XD1 system.

Table 16. Using Global Arrays

	Detail	Comment
Include file path	`/usr/local/include/ga.h`	C header file
Object library path	`/usr/local/lib64/libglobal.a`	
Documentation	`http://www.emsl.pnl.gov/docs/global/ga.html`	

3.3.7 GPSHMEM

Table 17, page 27 lists information about the Generalized Portable Shared Memory (GPSHMEM) library on the Cray XD1 system.

Table 17. Using GPSHMEM

	Detail	Comment
Include file path	`/usr/local/include/gpshmem-1.0/gpshmem.h`	C header file
	`/usr/local/include/gpshmem-1.0/gpshmem.Fh`	Fortran include file
Object library path	`/usr/local/lib64/libgpshmem.a`	
Documentation	`> man gpshmem` and man pages for individual routines	

3.3.8 PAPI

Table 18, page 27 lists information about the Performance Application Programming Interface (PAPI) library on the Cray XD1 system.

Table 18. Using PAPI

	Detail	Comment
Environment module	`> module use /opt/modulefiles` `> module load papi`	
Include file path	`/opt/xd-pe/papi/3.0.7/include/ papi.h`	C header file. The include path is not required in the compile step if you load the specified modulefile.
	`/opt/xd-pe/papi/3.0.7/include/ fpapi.h`	Fortran include file. The include path is not required in the compile step if you load the specified modulefile.
Object library path	`/opt/xd-pe/papi/3.0.7/lib/ libpapi.a`	The library path is not required in the link step if you load the specified modulefile.

	Detail	Comment
Documentation	> **man PAPI** and man pages for individual routines http://icl.cs.utk.edu/papi	

3.3.9 ScaLAPACK

Table 19, page 28 lists information about the Scalable Linear Algebra Package (ScaLAPACK) on the Cray XD1 system. The ScaLAPACK library uses the Basic Linear Algebra Subprograms (BLAS) library and the Basic Linear Algebra Communication Subprograms (BLACS) library. BLACS is included in the ScaLAPACK software package for the Cray XD1 system and is included in this table. A version of BLAS that is optimized for the Opteron processor is part of ACML.

Table 19. Using ScaLAPACK

	Detail	Comment
Include file path	/usr/local/include/blacs.h /usr/local/include/scalapack.h	
Object library path	/usr/local/lib/libscalapack.a /usr/local/lib/libblacs.a /usr/local/lib/libblacsCinit.a /usr/local/lib/libblacsFinit.a	32-bit libraries.
	/usr/local/lib64/libscalapack.a /usr/local/lib64/libblacs.a /usr/local/lib64/libblacsCinit.a /usr/local/lib64/libblacsFinit.a	64-bit libraries.
	To link BLAS, see Section 3.3.1, page 23.	
Documentation	http://www.netlib.org	

3.4 Using the Modules Package to Configure Your Environment

Most of the choices you make as you compile and link your application can be set by environment variables. The Cray XD1 system includes the Modules package which makes it easy to change your working environment.

3.4.1 Overview of the Modules Package

The Modules package allows you to change your environment with a single command—`module`—that can deploy a predefined environment configuration called a *modulefile*. Each modulefile is a text file that specifies the information that the command uses to configure your shell for a particular purpose; for example, for a particular version of an application. The `module` command supports all popular shells and some scripting languages.

The `module` command can access the modulefiles in the directories that are specified in your `MODULEPATH` environment variable. You can use the `module` command itself to change this variable; see the `use` subcommand in Table 20, page 29.

3.4.2 Introduction to the `module` Command

The syntax of the `module` command is as follows:

```
> module [options] subcommand [subcommand-args]
```

Table 20, page 29 describes the most common subcommands.

Table 20. Common `module` subcommands

Subcommand	Description
`help`	Lists all subcommands or displays help information for a specified modulefile.
`use`	Prepends a directory to the `MODULEPATH` variable.
`avail`	Lists all available modulefiles.
`load` or `add`	Loads a specified modulefile into the shell environment.
`list`	Lists the loaded modulefiles.
`display` or `show`	Displays the environment changes a modulefile specifies (except for conditional changes).

Subcommand	Description
`unload` or `rm`	Removes a specified modulefile from the shell environment.
`purge`	Unloads all loaded modulefiles.

For full details of all options, subcommands, and subcommand arguments, see the `module` man page.

Optionally, the system can have a global modulefile that applies to all users on a host, and individual users can have a user-specific default modulefile. When you invoke the `module` command, it automatically invokes these modulefiles (if they exist) before it performs the specified subcommand. For more details, see the man page.

3.4.3 Predefined Modulefiles

Some libraries and tools in the Cray XD1 software distribution or available as options include a predefined modulefile to provide access or to assist you in building applications. Section 3.3, page 22 identifies these modulefiles where they are available. Third-party products that you install may also include a modulefile.

Use the following command to see all the modulefiles that are available with the current value of your `MODULEPATH` environment variable:

```
> module avail
```

3.4.4 Developing Other Modulefiles from Templates

The Cray XD1 software distribution includes two templates from which you can develop your own modulefiles for the following situations:

- Using a particular compiler
- Using the MPICH library instance that was built by a particular compiler

For a description of the syntax of modulefiles, see the `modulefile` man page.

3.4.4.1 Compiler Modulefile Template

The template for a compiler modulefile—see Example 6, page 31—includes most of what you need in a modulefile for the PGI compiler. Whether you use the

PGI compiler or another compiler, you must edit this example to supply specific information for your system.

The first step is to copy the following file to another location and rename it:

```
/opt/XD1/templates/compiler_module
```

For the PGI compiler, replace the version placeholders in the file with the actual version numbers. The template file uses the following notation for placeholders to indicate text that you must replace: %*name*%; for example, %MAJOR_VERSION%.

If you use a different compiler, change the PGI-based names in the file to something else and make any other necessary adjustments. For example, you may not need to set the LM_LICENSE_FILE variable.

This template sets the MPICH environment variables that allow you to use the MPI compiler commands and have them execute the compiler of your choice.

Example 6: Template for a PGI compiler modulefile

```
#%Module
#
# pgi module
#

proc ModulesHelp { } {
   puts stderr "\tpgi - loads ...\n"
}

module-whatis "sets the environment variables for pgi compiler"

setenv PGI_MAIN_VER %MAJOR_VERSION%     #Example: 5.2
setenv COMPILER_ROOT %DIRECTORY_PATH%   #Example: /opt/pgi-5.2-4

# Search for demo license before searching flexlm servers
prepend-path LM_LICENSE_FILE $env(COMPILER_ROOT)/license.dat
set pgidir $env(COMPILER_ROOT)/linux86-64/$env(PGI_MAIN_VER)
prepend-path PATH               $pgidir/bin
prepend-path MANPATH            $env(COMPILER_ROOT)/common/man
prepend-path LD_LIBRARY_PATH    $pgidir/lib
prepend-path LD_LIBRARY_PATH    $pgidir/libso
setenv MPICH_CLINKER pgcc
```

```
setenv MPICH_CCC pgCC
setenv MPICH_CCLINKER pgcc
setenv MPICH_CC pgcc
setenv MPICH_F90 pgf90
setenv MPICH_F90LINKER pgf90
```

3.4.4.2 MPICH Library Modulefile Template

The template for an MPICH library modulefile—see Example 7, page 32—includes most of what you need in a modulefile for the instance of the MPICH library that was compiled with the PGI compiler. Whether you use the PGI compiler or another compiler, you must edit this example to supply specific information for your system.

The first step is to copy the following file to another location and rename it:

```
/opt/XD1/templates/mpich_module
```

Replace the version placeholder in the file with the actual directory name of the MPICH instance. The template file uses the following notation for the placeholder: `%MPICH_4_PGI_COMPILER%`. The comment on the same line shows an example. Table 5, page 16 shows more examples.

If you use a different compiler, change the PGI references in the heading comment and in the help string to something else.

Example 7: Template for an MPICH library modulefile

```
#%Module
#
# mpich module for PGI compiler
#
proc ModulesHelp { } {
         puts stderr "\tmpich for pgi compiler - loads ...\n"
     }

setenv MPICH_VERSION %MPICH_4_PGI_COMPILER% #mpich-1.2.6-pgi524
setenv MPICH /usr/mpich/$env(MPICH_VERSION)

setenv MPICH_HOME               $env(MPICH)
prepend-path PATH               $env(MPICH)/bin
prepend-path MANPATH            $env(MPICH)/man
prepend-path LD_LIBRARY_PATH    /lib64
prepend-path LD_LIBRARY_PATH    $env(MPICH)/lib/shared
prepend-path INCLUDE_PATH       $env(MPICH)/include
```

3.5 Building an MPICH Library Instance

If you compile your MPI applications with a compiler other than those for which an MPICH library instance already exists (either in the Cray XD1 software distribution or on the CRInform website), and you want the library to be compiled with the same compiler, the Cray XD1 administrator can build a new instance of the MPICH library for you.

3.5.1 Obtaining the MPICH Source Code

The source code for the components of the Cray XD1 software distribution that are under the GNU Public License (GPL) is available on the Cray website as an ISO 9660 disc image. This image includes an RPM package for the MPICH source code. In this procedure, you download the disc image and copy the MPICH source package to the Cray XD1 system. If you do not record the disc image onto a physical disc, and if you use the same workstation to access both the Internet and the Cray XD1 system, it must be a Linux workstation.

Procedure 1: To obtain the MPICH source code

1. From a workstation with Internet access, use your web browser to navigate to the software releases page of the CRInform website; for the URL, see "Cray XD1 Support," page xi.
2. Check the size of the ARP Linux distribution source disc image and your available disk space.
3. Download the disc image.
4. If necessary, move the disc image to a Linux workstation that has access to the Cray XD1 system.
5. Find or create a suitable directory on which to mount the disc image file, then mount the file as if it were a device:

   ```
   # mount -t iso9660 -o loop,ro path-to-disc-image mount-point
   ```

   #### Example 8: Accessing the source disc image

   This example assumes that you downloaded the disc image file to the `/tmp` directory of your workstation.

   ```
   # mkdir /tmp/xd1-src
   # mount -t iso9660 -o loop,ro /tmp/ARP_01_02_0034_src.iso \
     /tmp/xd1-src
   ```

6. Copy the `mpich-`*`version-release`*`.src.rpm` file to a temporary location on the Cray XD1 system, where *version* identifies the particular MPICH version and *release* is the release number of this package from Cray Inc.

Example 9: Copying the MPICH source package to the Cray XD1 system

This example makes the following assumptions:

- Your Cray XD1 administrator name is `xd1admin`.
- The domain name of the master node is `xd1.company.com`.
- You mounted the disc image at `/tmp/xd1-src`.
- The Cray XD1 application release package (ARP) version is 1.2-34.
- The MPICH source package version and release are 1.2.6-3.

```
# cd /tmp/xd1-src/01_02_0034/sources
# scp mpich–1.2.6-3.src.rpm xd1admin@xd1.company.com:/tmp
```

3.5.2 Compiling the MPICH Library

In this procedure, you first install the source software package, then apply Cray patches, configure certain parameters, compile the library, and finally install the library in the correct location.

Procedure 2: To compile the MPICH library

1. Log in to the master node of the Cray XD1 system as `root`.
2. Go to the directory that contains the Cray XD1 MPICH source software package. For example:

   ```
   # cd /tmp
   ```

3. Install the package:

   ```
   # rpm -ivv
     mpich-version-release.src.rpm
   ```

   where *version* identifies the particular MPICH version and *release* is the release number of this package from Cray Inc. For example, the package file name could be `mpich–1.2.6-3.src.rpm`.

   The command installs the contents of the package into the following directory: `/usr/src/packages/SOURCES`.

4. Apply the Cray patches to the MPICH source files:

   # **rpmbuild -bp /usr/src/packages/SPECS/mpich.spec**

   The set of source files that are ready to build is now under the following directory: /usr/src/packages/BUILD/mpich-*version*.

5. Cray recommends that you change this directory to the following name to indicate the Cray release as well as the MPICH version: mpich-*version*-*release*.

6. Cray recommends that you copy the source tree (the mpich-*version*-*release* directory and all its contents) to another location. This preserves the installed source tree for further use—that is, for compiling with other compilers.

7. Go to the working mpich-*version*-*release* directory and edit the following file: rai_configure.sh. This Cray script is a wrapper for the MPICH configure script. Apply the following changes to it:

   - Change the values of the following variables to specify which compilers to use: CC, CXX, FC, and F90. If you will not use a Fortran 90 compiler, leave the F90 variable with a null value.
   - Change the default value of the INSTALL_DIR variable which specifies the target location of this instance of the MPICH library. You should follow the convention that is described in Section 3.2.1, page 16. If you leave the default value, the new library will overwrite the GNU instance of the library.
   - If you want to use a Fortran 90 compiler, remove the --disable-f90 option from the configure command.
   - In the CFLAGS variable, if the optimization option is -O6 and you are not using GNU compilers, change the option to -O3.

8. Execute the rai_configure.sh script.

   The script checks many prerequisites and displays its progress. Watch for error messages. If the script is successful, it generates a makefile. The last line of the message output should be as follows:

   ```
   Configuration completed.
   ```

9. Compile the library:

   # **make mpi**

This process takes a minute or two and may produce many messages. Watch for error messages, but you can safely ignore the warnings.

If the compilation is successful, the final set of messages begins with the following one:

```
Completed build of MPI
```

10. Install this instance of the MPICH library:

    # **make install**

    This command installs the MPICH library (including the various object libraries, documentation, compiler scripts, and so on). It also builds some simple test applications. Watch for error and warning messages from these.

    Note: In this release, building the C++ test application generates a warning message, but you can ignore it.

    The final output from this command should be as follows:

    `installed MPICH in` *`mpich-dir`*

    where *mpich-dir* is the target directory that you configured in step 7.

3.5.3 Deploying the MPICH Library Instance

In the previous section, you compiled and installed the MPICH library instance on the master node of the Cray XD1 system. You must also make the library available on every node where it is needed. In the procedure in this section, you install the library in each relevant partition master software image and propagate the changes to the nodes that are already in the partition. For an explanation of partition master software images and a general discussion of managing custom configurations of the system, see *Cray XD1 System Administration* (S–2430).

This procedure assumes that you are still logged in to the master node as `root`. However, you can also perform this procedure as an administrator.

Procedure 3: To deploy the MPICH library instance

1. Copy the library instance into each relevant partition master software image:

   # **cp -fpr** *mpich-dir*
   /var/opt/pce/partmaster/*partition***/root/usr/mpich**

   where *mpich-dir* is the path to the directory in which you installed the library in the preceding procedure, and *partition* is the name of the target partition.

2. Propagate the changes in each relevant partition to all the nodes in the partition:

   ```
   # chpart --synchronize-partition now partition
   ```

   The command generates an update package for each node in the target partition, prints a request ID, and terminates. For a partition with the full set of software packages, this takes approximately 20 seconds. The update packages contain all of the changes to the partition master software image since you last synchronized the partition.

   Note: If you have made other changes to the partition master software image besides adding the new MPICH library instance, consider closing the partition and waiting until user activity terminates before you synchronize the partition.

   Each node processes its update package asynchronously. If a node is down when you issue this command, the update package remains in a queue; when the node boots, it detects and processes the update package.

3. You can check on the progress of the partition synchronization request at any subsequent time:

   ```
   > amrequestmon --show-request-tasks request-id
   ```

   where *request-id* is the request ID from the `chpart` command. This command displays information about the tasks that result from the asynchronous request. When a task completes, Active Manager removes it from the database and it no longer appears in the output of this command.

Using the FPGA Application Acceleration Processor [4]

This chapter describes how to use the optional field-programmable gate array (FPGA) application acceleration processors in a Cray XD1 system. Cray provides a utility program and an application programming interface that together support the use of these processors by an application program.

4.1 Overview

The FPGA application acceleration processors in a Cray XD1 system are Xilinx Virtex II Pro field-programmable gate arrays. In this chapter, we usually refer to them simply as FPGAs unless the context requires more precision (the Cray XD1 system also uses FPGAs for other purposes). An FPGA application acceleration processor is an optional component of the optional expansion module that may be added to each compute blade in a Cray XD1 chassis. (For more information on these and other components of the Cray XD1 system, see *Cray XD1 System Overview* (S–2429).) You can use the FPGAs to implement selected algorithms or parts of algorithms in hardware.

FPGAs can significantly improve the performance of some applications. Typically, they achieve this in one or both of the following ways:

- Performing the same processing step on multiple data elements in parallel
- Pipelining multiple sequential processing steps that are performed on each element of a large data set

The logic that is programmed into an FPGA exists first as a binary file that encodes the required hardware design. The FPGA utility program, `fcu`, converts a raw FPGA binary file that the FPGA development tools produce to a form that the system can use. The FPGA application programming interface (API) library provides the functions that an application needs in order to use an FPGA once the converted FPGA logic file exists.

This chapter explains the sequence of operations that you perform to use an FPGA and gives informal descriptions of the `fcu` command and the API functions. For more formal descriptions that show the full set of command options and the function prototypes, see the man pages online.

Figure 2, page 41 illustrates the process of developing an application that uses the FPGA.

For an example that applies the various commands that this chapter describes to the sample program, see Section 4.6, page 70.

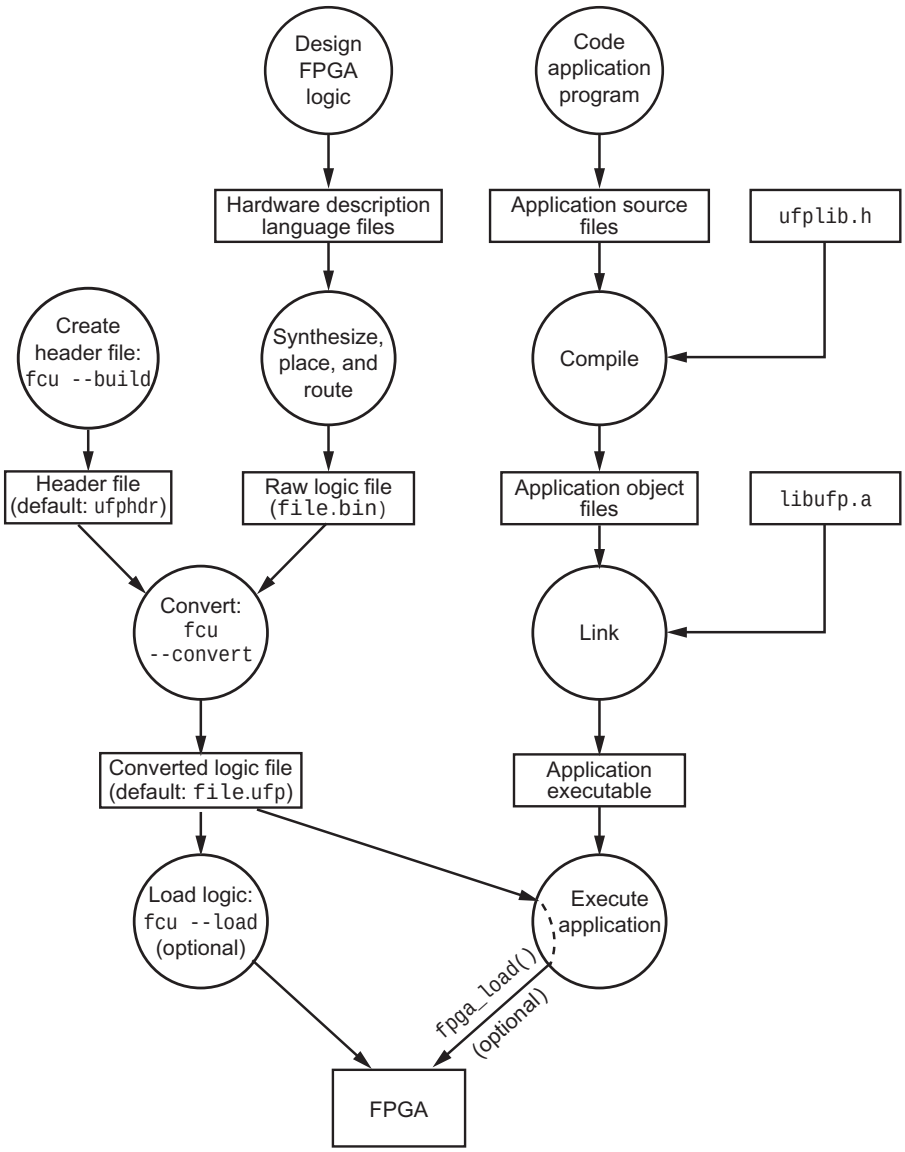

Figure 2. Development workflow for FPGA applications

4.2 Preparing an FPGA Logic File

Before you can use the FPGAs in an application program, you must have a logic file in the proper form. The two stages in the preparation of such a file are as follows (from the viewpoint of application development):

1. Develop a raw FPGA logic file.
2. Convert the raw logic file to the loadable form.

4.2.1 Developing a Raw FPGA Logic File

The details of developing a raw FPGA logic file are outside the scope of this manual. A separate manual, *Cray XD1 FPGA Development* (S–6400), provides a high-level overview of this subject. In addition, *Design of Cray XD1 RapidArray Transport Core* (S–6411) and *Design of Cray XD1 QDR II SRAM Core* (S–6412) describe two essential logic cores that are provided with the system. This section describes only the role of the application programmer in the logic development process and the output of that process.

Development of the raw FPGA logic file is a complex process that requires specialized knowledge and tools. As an application programmer, you typically collaborate with a hardware designer or other specialist in this task. Your role in the process is mainly to identify the parts of the application that are suitable for implementation in the FPGA and to communicate precise specifications to the logic developer. You also need to work closely with the logic developer to design the protocol for communication between the application and the FPGA logic.

All communication between the FPGA and an Opteron processor uses the RapidArray link between the FPGA and the RapidArray processor on the expansion module. Therefore, the logic developer must always include the RapidArray Transport Core (RT Core) in the logic design. This is a prerequisite for using the commands and functions that this chapter describes.

If the logic design uses the QDR II SRAM that is physically connected to the FPGA, the logic developer must also include the QDR II SRAM core. The developer can omit this core if the logic does not use the SRAM.

The output of the FPGA logic development process is a raw logic file (often called simply the binary file). Usually, the name of this file has a `.bin` suffix. It must have been compiled for a particular variant of the FPGA—each combination of FPGA size (number of gates) and speed grade requires a different binary file.

4.2.2 Converting a Raw Logic File to Loadable Form

The model of FPGA in the Cray XD1 system requires a slightly different file format than the development tools produce—it requires reversal of the bits in each byte. In addition, the FPGA API and the Cray XD1 Linux driver for the device assume that extra information about the expansion module is in the binary logic file. Therefore, you must transform the raw logic file to reverse the bits and embed the extra information before you can use it.

The multipurpose FPGA control utility, `fcu`(1), performs these two operations of the conversion.

Procedure 4: To convert a raw logic file to loadable form

1. Identify the Cray part number of the target FPGA. It is a string of the form `90-`*`nnnn-nn`* that specifies the particular variant of FPGA that is present. You can identify the part numbers of the FPGAs in a system by using the Active Manager `lsnode --verbose` command.

   Note: In previous releases, the FPGA part number that the command requires was of the form `87-`*`nnnn-nn`*. In the current release, the `fcu`(1) command still accepts a part number of this form, but will not in future releases.

2. Create a header file (which you will later merge with the logic file):

   > **`fcu --build`** [*`headerfile`*] [**`--partnum`** *`part-number`*] [**`--clock`** *`clock-freq`*]

   where *headerfile* is the output file name, *part-number* is the Cray part number of the FPGA, and *clock–freq* is the clock frequency at which to run the FPGA, in megahertz. If you omit *headerfile*, the output file name is `ufphdr`. The logic designer can provide the clock frequency; it must be in the range 63 through 199. If you do not specify the part number or the clock frequency in the command line, the command prompts you for the information.

   The `fcu` program creates the header file.

3. Merge the header file and the raw logic file:

   > **`fcu --convert`** *`rawfile headerfile`* [*`loadfile`*]

   where *rawfile* is the name of the input raw logic file (typically, *`design`*`.bin`), *headerfile* is the name of a header file that was previously created by the `--build` option of `fcu` (as in the preceding step), and *loadfile* is the name of the output file. If you omit *loadfile*, the output file name is *`rawfile`*`.ufp`.

The `fcu` program prepends the header file to the logic file and transforms the logic file as necessary. The output file is ready to load into the FPGA.

4.3 Managing FPGA Logic from the Command Line

You can perform some of the tasks involved in using an FPGA from the Linux command line with the FPGA control utility, `fcu`(1). Alternatively, you can perform the same tasks and more with the functions of the FPGA API; for details, see Section 4.4, page 45. Therefore, you can choose whether to perform these tasks in a job script or in the application program itself.

4.3.1 Loading FPGA Logic into the Device

After you convert the raw logic file into loadable form, you can load it into the FPGA device in preparation for running your application.

To load FPGA logic into the device from the command line, run the following command:

```
> fcu --load loadfile
```

where *loadfile* is the path name of the converted FPGA logic file.

The `fcu` program loads the specified file into the local FPGA. The application logic is reset, then released from reset.

4.3.2 Resetting an FPGA

You can explicitly reset the application logic within an FPGA from the command line when necessary.

To reset an FPGA from the command line, run the following command:

```
> fcu --reset
```

The `fcu` program resets the local FPGA by asserting the `user_reset_n` signal that the RT Core outputs. Any application logic that is connected to this signal is reset.

Note: Exercise care when you reset the FPGA application logic. In a typical design, the reset logic overrides most other logic functions in a device. If you reset the application logic while the FPGA is actively performing a task, the output may be meaningless.

4.3.3 Releasing an FPGA from Reset State

After you reset an FPGA, you must explicitly release it from reset. To do so from the command line, run the following command:

```
> fcu --exec
```

The `fcu` program takes the local FPGA out of reset by de-asserting the `user_reset_n` signal that the RT Core outputs. Any application logic that is connected to this signal is released from reset.

4.3.4 Querying the Status of an FPGA

You can display information about the status of the FPGA at the command line. To do so, run the following command:

```
> fcu --status
```

The program displays a numeric status code that depends on the state of the device. This is the value of the host latch register in the RT Core as a decimal integer. For details, see *Design of Cray XD1 RapidArray Transport Core* (S–6411). A value of 255 indicates that the FPGA is not programmed.

4.3.5 Erasing an FPGA

If security is an issue at your site, you can use the `fcu` command to completely clear the programming of the FPGA after you finish using it.

Note: You do not need to erase the FPGA explicitly before you load another logic file because the load operation also initially erases the device.

To erase an FPGA from the command line, run the following command:

```
> fcu --unload
```

4.4 Managing FPGA Logic in an Application Program

Note: In this release, the FPGA application programming interface is available only as a C library.

4.4.1 Using an FPGA in Application Programs

4.4.1.1 Typical Application Workflow

An application program uses an FPGA similarly to other devices—the program opens the device to get a file descriptor, uses the descriptor to interact with the device, and finally closes the device.

Your program begins to interact with the FPGA by loading the converted logic file into the device (unless you plan to do this from the command line prior to the start of the application; see Section 4.3.1, page 44). Then the program can set up the data and execute the logic as often as necessary.

Your program can transfer data in either direction by setting up the appropriate memory access. In addition, functions are available to read and write individual values in the FPGA address space. For example, you can read and write application-defined registers that are used for control and status operations.

Finally, a program can reset the FPGA to halt the execution of the logic and close the file descriptor.

4.4.1.2 Understanding Address Spaces on a Node

To understand the methods of communication between an application and the FPGA, you need to know about the relationships among the address spaces of the node components—the Opteron symmetric multiprocessor (SMP), the RapidArray processor (RAP) on the optional expansion module, and the FPGA application acceleration processor. Figure 3, page 47 illustrates these relationships.

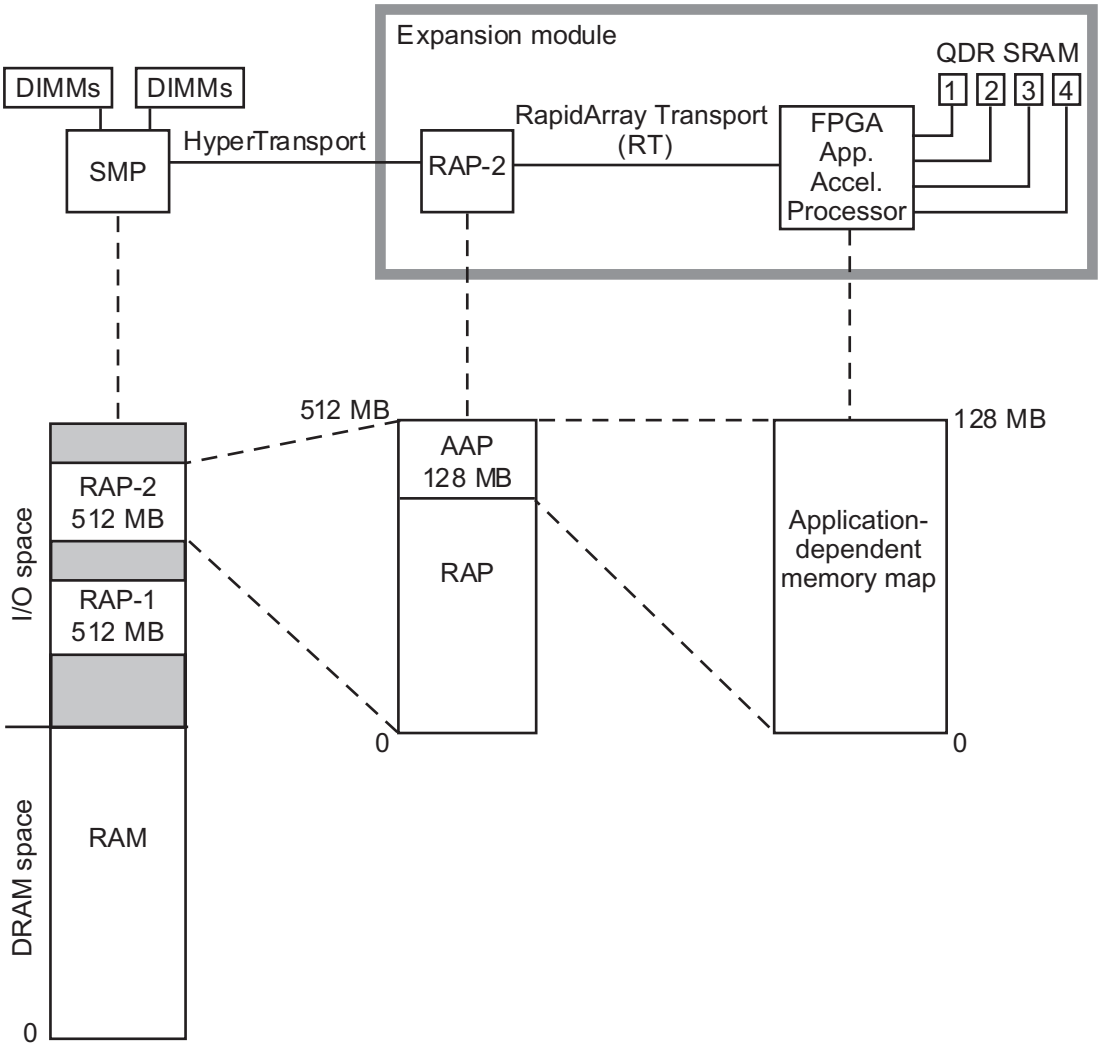

Figure 3. Physical components of a node and related address spaces

The physical interpretation of locations in the FPGA address space depends entirely on the design of the application logic in the FPGA. The logic may map the Quad Data Rate (QDR) II SRAM (which is attached to the FPGA) and the internal resources of the FPGA (such as registers and RAMs) to arbitrary addresses in its address space. For example, low addresses (starting at an offset of 0) could map to locations in the QDR II SRAM, and addresses starting at offset 0x4000000 (64 MB) could map to internal resources in the FPGA.

Note: Cray Inc. supplies an FPGA logic core—the QDR II SRAM core—that enables application logic in the FPGA to use the SRAM.

4.4.1.3 Data Transfer Methods

The protocol that you design for the interaction between your application process (which runs on an Opteron processor) and the FPGA application logic can use any or all of the following methods to transfer data:

- The application process can map a region of the FPGA address space into its own address space and access a location in the region using a normal memory reference (pointer). It can set or use the contents of any location within the mapped region. For details, see Section 4.4.2.7, page 53. If the sequence of accesses is important, the application must explicitly request synchronization of accesses at appropriate points; for details, see Section 4.4.2.8, page 55.
- The application process can allocate a block of memory within its own address space and register it for direct access by the FPGA logic. This block is called an *FPGA transfer region* (FTR). The FPGA can set or use the contents of any location in the registered region. For details, see Section 4.4.2.10, page 60.
- The application process can write and read individual 64-bit values at locations in the FPGA address space. Each such operation requires a function call. These calls guarantee the order of access.

4.4.2 Using an FPGA in a C Program

4.4.2.1 Typographic Conventions

In the code samples in this section, **bold** and *italic* text highlights only the elements that directly declare or use the function under discussion and its arguments. **Bold** text represents elements that you type exactly as shown, and *italic* text represents variables that you name. All the other code in `plain fixed-width font` is arbitrary sample code that you replace according to the needs and coding style of your application.

4.4.2.2 Library Files

The C header file, `ufplib.h`, provides the declarations of the functions and data types that are defined in the FPGA API library. Include it in any C source file that uses the library. The location of this file is specified in Section 3.3.5, page 26.

The object library name is `libufp.a`. The location of this file is specified in Section 3.3.5, page 26.

4.4.2.3 Opening an FPGA

The first operation in using an FPGA in an application program is to open the device with the `fpga_open`(3) function. This operation interacts only with the Linux kernel to prepare for the communication to follow—it does not physically affect the FPGA. The result is a file descriptor that your application uses in all other function calls that affect the device.

To open an FPGA, use statements like the following examples in your program:

```
#include <fcntl.h>
#include "ufplib.h"

int fpga_fd;
const char *fpga_path;
int flags; /* For example: O_RDWR | O_SYNC */
err_e err;

/* ... */
/* Set values of function arguments. */
/* ... */
fpga_fd = fpga_open(fpga_path, flags, &err);
if (fpga_fd < 0) {
   /* Handle error.*/
   }
```

Table 21, page 50 describes the variables in the sample code.

Table 21. `fpga_open`(3) arguments and return value

Variable	Input or output	Description
fpga_path	Input	The absolute path of the FPGA character device file. Typically: `/dev/ufp0`.
flags	Input	Controls the type of access to the device. Specify it as the bitwise OR of the appropriate masks that the `open` system call recognizes. For details, see the `open` man page.
err	Output	The error code that is set upon function return: either `NOERR` or another constant that is specified by the `err_e` enumerated type in `ufplib.h`.
fpga_fd	Output	The file descriptor of the FPGA. Used in all other function calls of the FPGA API.

4.4.2.4 Loading FPGA Logic into the Device

If you do not load the converted logic file into the FPGA before you execute your program, you must use the `fpga_load` function to load the logic file after the program opens the device. This programs the device with the specific logic for your application.

When the load operation is complete, the system resets the application logic and releases it from reset.

To load FPGA logic into the device, use statements like the following examples in your program:

```
#include "ufplib.h"

int fpga_fd;
const char*loadfile = "my_logic.ufp"; /* for example */
err_e err;
int num_bytes;

/*... */

num_bytes = fpga_load(fpga_fd, loadfile, &err);
```

```
if (num_bytes < 0) {
   /* Handle error.*/
   }
```

Table 22, page 51 describes the variables in the sample code.

Table 22. `fpga_load`(3) arguments and return value

Variable	Input or output	Description
fpga_fd	Input	The file descriptor of the FPGA that `fpga_open` returned.
loadfile	Input	The path of the converted FPGA logic file that the `fcu` command created.
err	Output	The error code that is set upon function return: either `NOERR` or another constant that is specified by the `err_e` enumerated type in `ufplib.h`.
num_bytes	Output	The value returned by `fpga_load`: either the number of bytes written to the device or `-1` on failure.

4.4.2.5 Resetting an FPGA

If the application program loads the logic file, it does not need to reset the logic initially because the load operation does so automatically. However, if the user or another application loaded the logic file before this application executes, consider resetting the logic to put it into a known initial state. The `fpga_reset`(3) function places the application logic of a previously loaded FPGA into a reset state. It does this by asserting the `user_reset_n` signal that the RT Core outputs. Any application logic that is connected to this signal is reset.

To reset an FPGA, use statements like the following examples in your program:

```
#include "ufplib.h"

int fpga_fd;
err_e err;
int status;

/* ... */
```

```
status = fpga_reset(fpga_fd, &err);
if (status < 0) {
   /* Handle error. */
   }
```

Table 23, page 52 describes the variables in the sample code.

Table 23. `fpga_reset`(3) arguments and return value

Variable	Input or output	Description
fpga_fd	Input	The file descriptor of the FPGA that `fpga_open` returned.
err	Output	The error code that is set upon function return: either `NOERR` or another constant that is specified by the `err_e` enumerated type in `ufplib.h`.
status	Output	The status value that `fpga_reset` returns: either `0` on success or `-1` on failure.

4.4.2.6 Releasing an FPGA from Reset State

If the application program loads the logic file, it does not need to release the logic from reset initially because the load operation does so automatically. However, if the application explicitly resets the application logic, it must take the logic out of the reset state to start execution of the logic. The `fpga_start`(3) function does this by de-asserting the `user_reset_n` signal that the RT Core outputs. Any application logic that is connected to this signal is released from reset.

Use the `fpga_reset`(3) and `fpga_start`(3) functions together to perform a reset cycle on the FPGA. You can also use the `fpga_start`(3) function alone to ensure that the application logic is not in the reset state.

To release an FPGA from reset state, use statements like the following examples in your program:

```
#include "ufplib.h"

int fpga_fd;
err_e err;
int status;
```

```
/* ... */
status = fpga_start(fpga_fd, &err);
if (status < 0) {
   /* Handle error. */
   }
```

Table 24, page 53 describes the variables in the sample code.

Table 24. `fpga_start`(3) arguments and return value

Variable	Input or output	Description
fpga_fd	Input	The file descriptor of the FPGA that `fpga_open` returned.
err	Output	The error code that is set upon function return: either `NOERR` or another constant that is specified by the `err_e` enumerated type in `ufplib.h`.
status	Output	The status value that `fpga_start` returns: either `0` on success or `-1` on failure.

4.4.2.7 Mapping FPGA Locations to the Application Address Space

The `fpga_memmap`(3) function maps a region of the FPGA address space to the application address space. The application can then use normal memory references (pointers) to read or write values in the mapped region. For a description of the FPGA address space, see Section 4.4.1.2, page 46.

This function uses the write-combining feature of the Opteron processor, which can lead to out-of-sequence transactions between the Opteron and the FPGA. The API lets you synchronize the transactions to ensure the proper sequence when necessary; see Section 4.4.2.8, page 55.

Instead of (or in addition to) mapping a whole region of the FPGA address space, the application can also use function calls to access individual locations in the FPGA address space; see Section 4.4.2.9, page 57.

To map FPGA locations to the application address space, use statements like the following examples in your program:

```
#include <sys/types.h>
#include "ufplib.h"
```

```
#define X_OFFSET 0x100
#define Y_OFFSET 0x200

int fpga_fd, prot, flags;
size_t len;
off_t offset;
err_e err;
void *fpga_base;
long x, y;

/* ... */
/* Set values of function arguments. */
/* ... */
fpga_base = fpga_memmap(fpga_fd, len, prot, flags,
   offset, &err);
if (fpga_base == NULL) {
   /* Handle error. */
   }

/* Use the pointer to write or read values in the FPGA. */
/* Initialize x. */
/* ... */
*(fpga_base + X_OFFSET) = x;
/* ... */
y = *(fpga_base + Y_OFFSET);
```

Table 25, page 54 describes the variables in the sample code.

Table 25. `fpga_memmap`(3) arguments and return value

Variable	Input or output	Description
fpga_fd	Input	The file descriptor of the FPGA that `fpga_open` returned.
len	Input	The number of bytes to be mapped.
prot	Input	A bit array that specifies the desired memory protection. It is either `PROT_NONE` or the bitwise OR of one or more of the other `PROT_*` flags specified by the Linux `mmap` function. These constants are defined in `sys/mman.h` which is included by `ufplib.h`.

Variable	Input or output	Description
flags	Input	A bit array that specifies mapping options like those of the Linux `mmap` function. It is the bitwise OR of either `MAP_SHARED` or `MAP_PRIVATE` and zero or more of the other flags described in the `mmap` man page. These constants are defined in `sys/mman.h` which is included by `ufplib.h`.
offset	Input	The byte offset in the FPGA address space at which the mapped region begins.
err	Output	The error code that is set upon function return: either `NOERR` or another constant that is specified by the `err_e` enumerated type in `ufplib.h`.
fpga_base	Output	A pointer to the mapped area in the application address space. `NULL` indicates failure.

4.4.2.8 Synchronizing Accesses to FPGA Locations

The `fpga_memmap`(3) function (see Section 4.4.2.7, page 53) uses the write-combining feature of the Opteron processor to communicate with an FPGA. While this can be more efficient, it can also lead to an out-of-sequence execution of accesses to locations in the FPGA address space. The `fpga_mem_sync`(3) function executes a memory barrier to flush out all transactions to the region that is mapped by the `fpga_memmap`(3) function. This function ensures that the Opteron processor completes all previous accesses before it performs any subsequent accesses. Use this function if the order of accesses in the application is important.

To synchronize accesses to FPGA locations, use statements like the following examples in your program:

```
#include <sys/types.h>
#include "ufplib.h"
#define X_OFFSET 0x100
#define Y_OFFSET 0x200

int fpga_fd;
err_e err;
```

```
int status;
int prot, flags;
size_t len;
off_t offset;
void *fpga_base;
long x, y;

/* ... */
/* Set values of fpga_memmap arguments. */
/* ... */
fpga_base = fpga_memmap(fpga_fd, len, prot, flags, offset,
                        &err);
if (fpga_base == NULL) {
   /* Handle error.*/
   }
/* Some FPGA accesses */
*(fpga_base + X_OFFSET) = x;
/* ... */
/* Synchronize */
status = fpga_mem_sync(fpga_fd, &err);
if (status < 0) {
   /* Handle error. */
   }
/* Some more FPGA accesses */
y = *(fpga_base + Y_OFFSET);
/* ... */
```

Table 26, page 56 describes the variables in the sample code.

Table 26. `fpga_mem_sync`(3) arguments and return value

Variable	Input or output	Description
fpga_fd	Input	The file descriptor of the FPGA that `fpga_open` returned.
err	Output	The error code that is set upon function return: either `NOERR` or another constant that is specified by the `err_e` enumerated type in `ufplib.h`.
status	Output	The status value that `fpga_mem_sync` returns: either `0` on success or `-1` on failure.

4.4.2.9 Writing and Reading Individual FPGA Locations

In addition to mapping a region of the FPGA address space and accessing it with ordinary memory references, an application can also write and read any individual 64-bit value in the FPGA address space by using the `fpga_wrt_appif_val`(3) and `fpga_rd_appif_val`(3) functions. These functions guarantee that the order of access is the order in which the application calls these functions. You do not need to use `fpga_memmap`(3) or `fpga_mem_sync`(3) in conjunction with these functions.

For a description of the FPGA address space, see Section 4.4.1.2, page 46.

Note: The meaning of the `offset` parameter of `fpga_rd_appif_val` and `fpga_wrt_appif_val` changed in release 1.2 of the Cray XD1 system. Previously, these functions assumed a particular mapping of FPGA resources in the FPGA address space such that a zero value of the offset parameter accessed the first FPGA register at a fixed location of 0x4000000 in the FPGA address space. Now, the parameter is generalized to allow access to both the internal resources of the FPGA and the attached QDR II SRAM, and the offset value can access any location in the whole FPGA address space. If you have an application program that calls these functions with the old meaning of `offset`, you need to change your source code to work with release 1.2 and later. Set the value of this parameter to be the address in the FPGA address space of the resource you want to access. This value depends entirely on the design of the FPGA application logic.

4.4.2.9.1 Writing to an FPGA Location

Use the `fpga_wrt_appif_val`(3) function to write a 64-bit value into a specified location in the FPGA's address space.

This function can provide special treatment for a data value that is a virtual address in an FPGA transfer region—a memory region in the application's address space that was registered by the `fpga_register_ftrmem`(3) function for direct access by the FPGA (see Section 4.4.2.10, page 60). For such a data value, the function first transforms it to a physical address that the FPGA logic can use.

To write to an FPGA location, use statements like the following examples in your program:

```
#include "ufplib.h"

int fpga_fd;
unsigned long val;
```

```
unsigned long offset;
unsigned long type;
err_e err;
int status;

/* ... */
/* Set values of function arguments. */
/* ... */
status = fpga_wrt_appif_val(fpga_fd, val, offset,
   type, &err);
if (status < 0) {
   /* Handle error. */
   }
```

Table 27, page 58 describes the variables in the sample code.

Table 27. `fpga_wrt_appif_val`(3) arguments and return value

Variable	Input or output	Description
fpga_fd	Input	The file descriptor of the FPGA that `fpga_open` returned.
val	Input	The value to write.
offset	Input	The byte offset of the location in the FPGA address space.
type	Input	One of: • `0`—The value is written as is. • `1`—The value is a user-space virtual memory address in an FPGA transfer region (see Section 4.4.2.10, page 60). The `fpga_wrt_appif_val` function transforms this virtual address to a physical address before writing it.

Variable	Input or output	Description
err	Output	The error code that is set upon function return: either NOERR or another constant that is specified by the err_e enumerated type in ufplib.h.
status	Output	The status value that fpga_wrt_appif_val returns: either 0 on success or -1 on failure.

4.4.2.9.2 Reading from an FPGA Location

Use the fpga_rd_appif_val(3) function to read a 64-bit value from a specified location in the FPGA's address space.

To read from an FPGA location, use statements like the following examples in your program:

```
#include "ufplib.h"

int fpga_fd;
unsigned long val;
unsigned long offset;
err_e err;
int status;

/* ... */
/* Set values of function arguments. */
/* ... */
status = fpga_rd_appif_val(fpga_fd, &val,
   offset, &err);
if (status <0) {
   /* Handle error. */
   }
```

Table 28, page 60 describes the variables in the sample code.

Table 28. `fpga_rd_appif_val`(3) arguments and return value

Variable	Input or output	Description
fpga_fd	Input	The file descriptor of the FPGA that `fpga_open` returned.
val	Output	The value that was read.
offset	Input	The byte offset of the location in the FPGA address space.
err	Output	The error code that is set upon function return: either `NOERR` or another constant that is specified by the `err_e` enumerated type in `ufplib.h`.
status	Output	The status value that `fpga_rd_appif_val returns:` either `0` on success or `-1` on failure.

4.4.2.10 Accessing Application Memory from an FPGA

The API also supports access by the FPGA logic to a region of application memory—the FPGA transfer region that is described in Section 4.4.1.3, page 48. Your program allocates a memory block, then uses the `fpga_register_ftrmem`(3) function to register the block as an FTR. This sets up the memory block for the FPGA to access it directly.

Note: The `fpga_set_ftrmem`(3) function that you used in previous releases to allocate and register an FTR is now deprecated. Use the `fpga_register_ftrmem`(3) function instead.

The memory block that you register as an FTR must be aligned on a memory page boundary, and its size must be a multiple of the memory page size.[1] The minimum size that you can register is the equivalent of one memory page, and the maximum size is 1 GB. (The older `fpga_set_ftrmem`(3) function can allocate a maximum of only 2 MB.)

The `fpga_register_ftrmem`(3) function does not automatically provide the address of this region to the FPGA application logic. The way that the application communicates this information depends on the protocol that you establish between the application and the FPGA logic. One way to communicate

[1] You can use the Linux `getpagesize` function to discover the memory page size and the Linux `posix_memalign` function to allocate a suitably aligned block of memory.

the address is to establish an FPGA register for that purpose and use the fpga_wrt_appif_val(3) function to write the value to the register. For more details, see Section 4.4.2.9, page 57.

When the application program finishes using the FTR, it should deregister the FTR by calling the fpga_dereg_ftrmem(3) function. This frees the memory mapping capability of the processor so that other processes can use the capability. Deregister the FTR regardless of how you registered it: the fpga_register_ftrmem(3) function or the older (and now deprecated) fpga_set_ftrmem(3) function.

To access application memory from an FPGA, use statements like the following examples in your program:

```
#define _XOPEN_SOURCE 600
#include <stdlib.h>
#include <unistd.h>
#include "ufplib.h"

int fpga_fd;
unsigned int size = getpagesize()*10000;
err_e err;
void *ftr_mem;
int status;

status = posix_memalign(&ftr_mem, getpagesize(), size);
if (status != 0) {
   /* Handle error. */
   }
/* ... */
/* Set other values of function arguments. */
/* ... */
status = fpga_register_ftrmem(fpga_fd, ftr_mem, size, &err);
if (status < 0) {
   /* Handle error. */
   }
/* Communicate the FTR base address to the FPGA */
/* ... */
/* Perform the main work of the application */
/* ... */
status = fpga_dereg_ftrmem(fpga_fd, ftr_mem, &err);
```

Table 29, page 62 describes the variables in the sample code.

Table 29. `fpga_register_ftrmem`(3) and `fpga_dereg_ftrmem`(3) arguments and return value

Variable	Input or output	Description
fpga_fd	Input	The file descriptor of the FPGA that `fpga_open` returned.
ftr_mem	Input	A pointer to a memory block in the application address space to use as the FTR.
size	Input	The size of the FTR in bytes. It must be a multiple of the memory page size. The maximum size is 1 GB.
err	Output	The error code that is set upon function return: either `NOERR` or another constant that is specified by the `err_e` enumerated type in `ufplib.h`.
status	Output	The status value that `fpga_register_ftrmem` or `fpga_dereg_ftrmem` returns: either `0` on success or `-1` on failure.

4.4.2.11 Checking the Status of an FPGA

The `fpga_status`(3) function returns a status value from a previously opened FPGA. This value is the value of the host latch register in the RapidArray Transport Core. It is an integer in the range 0 to 255. For information about the meanings of this status value, see *Design of Cray XD1 RapidArray Transport Core* (S–6411).

To check the status of an FPGA, use statements like the following examples in your program:

```
#include "ufplib.h"

int fpga_fd;
err_e err;
int dev_status;

/* ... */
dev_status = fpga_status(fpga_fd, &err);
/* Check the returned value against expected values */
```

Table 30, page 63 describes the variables in the sample code.

Table 30. `fpga_status`(3) arguments and return value

Variable	Input or output	Description
fpga_fd	Input	The file descriptor of the FPGA that `fpga_open` returned.
err	Output	The error code that is set upon function return: either `NOERR` or another constant that is specified by the `err_e` enumerated type in `ufplib.h`.
dev_status	Output	The FPGA status value that `fpga_status` returns: an integer from 0 to 255 or `-1` on failure.

4.4.2.12 Checking the Programming State of an FPGA

The `fpga_is_loaded`(3) function queries the programming state of a previously opened FPGA. The return value indicates whether a logic file is currently loaded into the FPGA.

To check the programming state of an FPGA, use statements like the following examples in your program:

```
#include "ufplib.h"

int fpga_fd;
err_e err;
int loaded;

/* ... */
loaded = fpga_is_loaded(fpga_fd, &err);
if (loaded) {
   /* do something */
   }
else {
   /* do something else */
   }
```

Table 31, page 64 describes the variables in the sample code.

Table 31. `fpga_is_loaded`(3) arguments and return value

Variable	Input or output	Description
fpga_fd	Input	The file descriptor of the FPGA that `fpga_open` returned.
err	Output	The error code that is set upon function return: either `NOERR` or another constant that is specified by the `err_e` enumerated type in `ufplib.h`.
loaded	Output	The result that `fpga_is_loaded` returns. A nonzero value means that the FPGA is loaded and zero means it is not.

4.4.2.13 Erasing an FPGA

The `fpga_unload`(3) function clears the programming of an FPGA. The function erases the logic that was previously loaded with the `fpga_load`(3) function or the `fcu`(1) command.

This function is provided for extra security only. You do not need to call it before you load your logic file because the `fpga_load`(3) function has the same effect. You need this function only if you want to completely clear the FPGA after you finish using it. In this case, call it just before you call the `fpga_close`(3) function.

To erase an FPGA, use statements like the following examples in your program:

```
#include "ufplib.h"

int fpga_fd;
err_e err;
int status;

/* ... */
status = fpga_unload(fpga_fd, &err);
if (status < 0) {
   /* Handle error. */
   }
```

Table 32, page 65 describes the variables in the sample code.

Table 32. `fpga_unload`(3) arguments and return value

Variable	Input or output	Description
fpga_fd	Input	The file descriptor of the FPGA that `fpga_open` returned.
err	Output	The error code that is set upon function return: either `NOERR` or another constant that is specified by the `err_e` enumerated type in `ufplib.h`.
status	Output	The status value that `fpga_unload` returns: either `0` on success or `-1` on failure.

4.4.2.14 Closing an FPGA

Use the `fpga_close`(3) function to close an FPGA that was opened previously. This clears any associations of the application process to the FPGA memory that the `fpga_memmap`(3) function established (see Section 4.4.2.7, page 53).

To close an FPGA, use statements like the following examples in your program:

```
#include "ufplib.h"

int fpga_fd;
err_e err;
int status;

/* ... */
status = fpga_close(fpga_fd, &err);
if (status < 0) {
   /* Handle error. */
   }
```

Table 33, page 66 describes the variables in the sample code.

Table 33. `fpga_close`(3) arguments and return value

Variable	Input or output	Description
fpga_fd	Input	The file descriptor of the FPGA that `fpga_open` returned.
err	Output	The error code that is set upon function return: either `NOERR` or another constant that is specified by the `err_e` enumerated type in `ufplib.h`.
status	Output	The status value that `fpga_close` returns: either `0` on success or `-1` on failure.

4.5 Sample Application: Using the Mersenne Twister Accelerator

Our sample application is a program that generates pseudorandom numbers by using the Mersenne Twister algorithm. Cray has implemented the twisting function of the algorithm in FPGA logic, which we refer to here as the Mersenne Twister Accelerator (MTA). The Cray XD1 software distribution includes the MTA logic file and a sample C program (`mta_test.c`) that uses it.

This section provides some background information and commentary to help you understand the sample program. It also provides a full example of how to execute this program on the Cray XD1 system.

A full listing of the program is in Appendix A, page 75. In the listing, each line of code is numbered. In this section, references such as "(line *n*)" refer to the numbered source code line in the listing.

4.5.1 Algorithm

The Mersenne Twister algorithm is an efficient algorithm for generating pseudorandom numbers with excellent statistical properties. It has become popular as a source of pseudorandom numbers in Monte-Carlo simulations. The algorithm is a form of linear feedback shift register with an extremely long period of 2^{19937}–1. A paper by Matsumoto and Nishimura[2] describes the algorithm in detail.

[2] Makoto Matsumoto and Takuji Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator," ACM Transactions on Modeling and Computer Simulation, vol 8, no. 1, pp. 3-30.

4.5.2 High-level Design of Application and FPGA Logic

The MTA logic starts with an initial seed value of 19,938 bits that is generated by the application program and implemented as a state array of 624 pseudorandom 32-bit numbers. The MTA progressively transforms the array into new pseudorandom numbers according to the algorithm. It exploits opportunities for both parallel and pipelined computations. For a description of the MTA logic design, see the following document, which is also in the Cray XD1 software distribution:

```
/usr/local/ufpapps/mta/doc/PNR-DD-0022-MtaFPGA.pdf
```

The application writes the initial state array into the FPGA registers. It also writes the addresses of the output buffers and an output format selector into the registers. The application sets up the output buffers as FPGA transfer regions, and the MTA logic outputs the results to the FTRs using a double-buffering scheme. The application extracts results from a full buffer while the MTA logic fills the empty buffer.

4.5.3 Some Design Details

The user specifies the type (format) and number of numbers to generate as command-line arguments. For details on the use of the `mta_test` command, refer to an appendix of the MTA design document.

The sample program assumes that the user previously loaded the MTA logic into the FPGA from the command line. For details on this procedure, see Section 4.3.1, page 44. This design choice is efficient for running the application multiple times (for example, for measuring performance). In a real application, you may prefer to have your program load the logic via the `fpga_load`(3) function.

The MTA design illustrates one way of establishing a protocol for communication between the application and the FPGA logic. The MTA initializes itself when it starts, then holds itself in that state. The MTA design includes a register that allows the application to start and stop the computation of pseudorandom numbers.

The MTA also defines registers to specify the output format, to configure the output buffers, to specify the output buffer addresses, and to hold the Mersenne Twister state array. The MTA registers begin at an offset of 64 MB (0x4000000) in the FPGA address space. Table 34, page 68 shows the additional offsets of the registers of interest from that base address. Application programs must use the total offset value (for example 0x4000008) in calls to the `fpga_wrt_appif_val`(3) function. (In this design, the application does not read any registers.)

Table 34. MTA registers

Offset (hexadecimal) 0x4000000 +	Offset (decimal) 64MB +	Description
0x8	8	Application configuration register. Bit 0 is the flag that holds the MTA in its initial state or releases it to start computation.
0x18	24	Buffer configuration register. Holds the following items: • Bits 0 to 8 specify the buffer size in units of 4 KB pages. • Bits 32 to 47 specify the buffer status polling interval in units of user clock cycles.
0x20	32	Buffer 0 base pointer register. Holds the base address of the first output buffer.
0x28	40	Buffer 1 base pointer register. Holds the base address of the second output buffer.
0x1000 and up	4096 and up	Mersenne Twister state array.

The FPGA typically generates results faster than an Opteron processor can fetch them. The double-buffering scheme that is used in this sample application for FPGA output uses a handshake mechanism to make the operation efficient. After filling one buffer, the MTA continues to generate results in the next buffer while the application extracts those in the previous buffer. The block size (eight 64-bit numbers) that is used in the sample program to extract the results provides optimum performance.

The first 64 bytes of each output buffer are reserved for status information, of which the first 8 bytes are a buffer status flag (0—empty or 1—full) that triggers buffer switching. After the sample program creates the output buffers, it zeroes them, which also marks them as empty initially.

When the MTA logic is computing, it polls the status of the next output buffer until the status flag shows that the buffer is empty. It then computes numbers and transfers them to the output buffer until the buffer is full. It updates the buffer status to full, then polls the status of the next buffer.

The sample program only illustrates how to use the MTA logic; it does not use the pseudorandom numbers that are generated. However, it does use some microsecond-precision timing functions (also in the software distribution) that enable you to measure the rate of production of the numbers.

4.5.4 Walkthrough

The `mta_test` program structure includes four major functions and three wrapper functions that cast the output as different data types:

- `parse_opt`—Parses the command-line options and arguments (line 129 to line 157).
- `mt_1999_set`—Initializes the state array, sets up and configures the output buffers, and starts the computation (line 162 to line 260).
- `mt_get`—Fetches blocks of pseudorandom numbers from the output buffers for use in the main program (line 271 to line 311). In addition, see the functions `mt_get_int`, `mt_get_float`, and `mt_get_double` at line 317 to line 330.
- `main`—The main program (line 336 to line 461).

The main program first parses the command-line arguments (line 357). It uses the `argp_parse` function from the GNU C Library, which in turn calls the `parse_opt` function in this program.

Then it opens the FPGA device (line 368) and resets and starts the logic to ensure that it is in a known state (line 377 and line 378).

The main program calls `mt_1999_set` (line 381), which performs the rest of the initialization:

- Computes the initial contents of the Mersenne Twister state array—the seed value of the algorithm (line 169 to line 179).
- Allocates the output buffers (line 184 and line 190), registers them as FTRs (line 197 and line 205), and zeroes them (line 214 and line 215).
- Writes the buffer addresses (line 220 to line 223), buffer configuration (line 229 to line 232), and initial state array (line 235 to line 240) to the registers.
- Sets the application configuration register to start the computation (line 244 to line 263).

The main program is now ready to receive the pseudorandom numbers that the FPGA generates. It calls the appropriate `mt_get` wrapper function for every set of eight 64-bit numbers that it wants (line 384 to line 430). If the current buffer is empty (ready to write), the `mt_get` function polls the buffer flag until it changes to full (ready to read).

On each call, `mt_get` returns the next block of eight 64-bit numbers (line 295, line 296, and line 310). After it takes the last eight numbers in the buffer, `mt_get` marks the buffer as empty (line 300 or line 306) and switches to the next buffer.

Finally, the program sets the MTA application configuration register to stop the computation (line 439), deregisters the FTRs (line 442 and line 447), frees the buffers (line 454 and line 455), and closes the device (line 458).

4.6 Getting Started with the FPGA

This chapter described both the command-line tools and the API that enable you to use an FPGA on the Cray XD1 system. The source code of the sample Mersenne Twister application uses many of the API functions. This section illustrates the commands to compile the sample application, convert the appropriate logic file, load the FPGA, and run the sample application. This procedure is one way to verify that an FPGA and the communication path to it work correctly.

This procedure makes the following assumptions:

- You have an account on the Cray XD1 system.
- The nodes of the partition that allows you to log in do not have FPGAs. Therefore, you must submit the job to another partition.
- You have access to a job execution partition that has at least one node with an FPGA.

Procedure 5: To get started with the FPGA

1. Log in to Linux on the Cray XD1 system.
2. Copy the sample program and related files to your home directory:

   ```
   > cp -r /opt/ufpapps
   ```

3. Examine the contents of `~/ufpapps`.

   Although both the source and executable `mta_test` programs are present, you will rebuild the executable in this example; a suitable `Makefile` is

already present. Similarly, although both the raw and loadable MTA logic files are present, you will rebuild the loadable file in this example.

4. Build the executable application program:

```
> cd ~/ufpapps/mta/src
> make mta_test
```

The new `mta_test` executable file that results is in `~/ufpapps/mta/bin`.

5. Identify the FPGA variant that is present in the system:

```
> lsnode -v | less
```

You may want to redirect the `lsnode`(1) output to a file rather than read it on the screen.

The output includes about 34 lines for each node in the system, so it will be long, especially if you have multiple chassis in the system. The information for each node begins with lines such as the following examples:

```
Hardware ID:                  65439.4
Partition:                    compute
```

Look for a line similar to the following example in the middle of the information for each node:

```
App Accelerator:              90-0003-05
```

This is the Cray part number for an FPGA. If an FPGA is not present on the node, the part number is listed as `--`. Normally, all the FPGAs in a system have the same part number.

Take note of the part number and which partition or partitions have FPGAs.

6. Build the appropriate FPGA logic file:

```
> cd ~/ufpapps/mta/bin
> fcu --build --partnum 90-0003-05 --clock 180
creating header file ufphdr....
> fcu --convert mta50.bin ufphdr
header size 59
input fpga load size 2377668
```

The raw MTA logic file is provided for several variants of the FPGA. The file named `mta50.bin` corresponds to the part number 90-0003-05.

The output of the last command is the loadable logic file `mta50.bin.ufp`.

7. Create a job script that will load the logic file and run the application to generate 500 million pseudorandom numbers; do this in a separate directory:

```
> mkdir ~/test
> cd ~/test
> cat > mta_job
fcu --load ~/ufpapps/mta/bin/mta50.bin.ufp
~/ufpapps/mta/bin/mta_test 500000000
Ctrl+D
> chmod +x mta_job
```

8. Submit the job to a partition that has an FPGA; use the appropriate command for the configured workload management (WLM) system. For example, to submit the job to the `compute` partition and request the use of an FPGA with PBS Pro:

```
> qsub -q compute -l nodes=1:fpga mta_job
```

   Take note of the job ID that the command returns (it may be embedded in a string with other information, depending on the WLM system); for example: 1798.

9. Monitor the progress of the job by using the WLM system. Once the WLM system launches this sample job, it takes only a few seconds to execute.

10. After the job is complete, examine the output file; its name follows the conventions of the WLM system. For example, here are the contents of `mta_job.o1798`:

```
file size 2381764
setting device location /dev/ufp0
programming device 2381764 bytes
opening device /dev/ufp0
programmed FPGA 2381764 bytes
closing device file descriptor 4

Generating 500000000 32 bit pseudo-random numbers with FPGA.
Calibrating timer ... cpuHz is 2200000000

  Last number  : 0x25E3795A
  Elapsed time : 1559268 microseconds
  Rate         : 320663285.593 32 bit integers/second
```

   The first part of the output shows that the `fcu --load` command was successful. The second part is the output from the application. It shows that the program generated 500,000,000 pseudorandom numbers (32-bit) in

1,559,268 microseconds, which is an average rate of 320,663,285 numbers per second. Any rate that exceeds 250,000,000 numbers per second indicates that the FPGA and the infrastructure that supports it are working properly.

Program Listing: `mta_test.c` [A]

```
 1  /*
 2   * This program demonstrates random number generation on the Cray XD1
 3   * system with an FPGA co-processor.
 4   * Usage : mta_test <number>
 5   * Generates a pseudo-random sequence of <number> length, using the
 6   * default seed 4357
 7   * Mersenne-Twister algorithm is used to generate the numbers.
 8   * The FPGA must be programmed with the official Mersenne-Twister
 9   * logic distributed by Cray along with this program.
10   * Timing code is provided in file timer.c.
11   */
12
13  #include <stdio.h>
14  #define __USE_XOPEN
15  #define __USE_XOPEN2K
16  #define _XOPEN_SOURCE 600
17  #include <stdlib.h>
18  #include <sys/types.h>
19  #include <sys/stat.h>
20  #include <fcntl.h>
21  #include <assert.h>
22  #include <unistd.h>
23  #include <string.h>
24  #include <argp.h>
25
26  #include <einlib.h>
27  #include "timer.h"
28
29  #define N 624   /* Period parameters */
30  #define INT32_MASK 0xFFFFFFFF
31  #define DEFAULT_SEED 4357UL
32  #define MULTIPLIER 1812433253UL /* Don Knuth, Vol 2 */
33
34  /* values specific to Cray FPGA logic */
35  #define TYPE_VAL  0UL
36  #define TYPE_ADDR 1UL
37  #define READ_NUMS 8 /* every call of mt_get fetches us this many 64-bit random numbers */
38  #define BUFF_SIZE (1 * 1024 * 1024)
39  #define BUFF_RD_RDY    1UL /* buffer is ready to be read by the Opteron */
40  #define BUFF_WRT_READY 0UL /* buffer is ready to be written into by the FPGA */
```

```
41  #define QUAD_WRDS_PER_BUF (BUFF_SIZE/8)
42
43  /* Define the address offsets for the MTA FPGA Registers */
44  #define REG_OFFSET     (64 * 1024 *1024)     /* Registers start at 64M */
45  #define APP_ID_REG     (REG_OFFSET + 0x00UL)
46  #define APP_CFG_REG    (REG_OFFSET + 0x08UL)
47  #define APP_LATCH_REG  (REG_OFFSET + 0x10UL)
48  #define BUFF_CFG_REG   (REG_OFFSET + 0x18UL)
49  #define BUFF0_PTR_REG  (REG_OFFSET + 0x20UL)
50  #define BUFF1_PTR_REG  (REG_OFFSET + 0x28UL)
51  #define MTA_RAM_ARRAY  (REG_OFFSET + 0x1000UL)
52
53  /* Convenient bit masks for the MTA registers. */
54  #define MTA_INTEGER  0x0UL
55  #define MTA_FLOAT    0x1UL
56  #define MTA_DOUBLE   0x2UL
57  #define MTA_INIT_KEY 0x1UL
58  #define MTA_FORMAT   0x6UL
59
60  typedef unsigned long u_64;
61  typedef struct {
62      unsigned long mt[N];
63      int mti;
64  } mt_state_t;
65
66
67  int fp_id;
68  volatile u_64 * buf0_ptr;
69  volatile u_64 * buf1_ptr;
70
71  /*****************************************************************************/
72  /* The following code relates to the commmand line parsing and can pretty */
73  /* much be ignored.                                                          */
74  /*****************************************************************************/
75  static struct argp_option options[] = {
76    {"verbose", 'v', 0,        0, "Produce verbose output"},
77    {"format",  'f', "STRING", 0, "Output format ('int, 'float' or 'double')."},
78    {0}
79  };
80  static error_t parse_opt (int key, char *arg, struct argp_state *state);
81
82  struct arguments
83  {
```

```
 84     char *args[1];
 85     int   verbose;   /* The -v flag     */
 86     char *format;    /* Argument for -f */
 87   };
 88
 89   const  char *argp_program_version = "mta_test 1.1";
 90   const  char *argp_program_bug_address = "<http://crinform.cray.com/xd>";
 91   static char args_doc[] = "number";
 92   static char doc[] = "mta_test -- A program that generates random numbers using the MTA FPGA.";
 93   static struct argp argp = {options, parse_opt, args_doc, doc};
 94
 95   int print_err (err_e e)
 96   {
 97       switch (e) {
 98       case NOERR:
 99         printf("Success.\n");
100         break;
101       case FILEOPRERR:
102         printf("File operation system call failed.\n");
103         break;
104       case INVALIDOP:
105         printf("Invalid API operation requested.\n");
106         break;
107       case INVALIDVAL:
108         printf("Invalid value passed to the API call.\n");
109         break;
110       case INVALIDARGS:
111         printf("Invalid argument passed to the API call.\n");
112         break;
113       case INVALIDINP:
114         printf("Invalid input given to the API call.\n");
115         break;
116       case DEVOPRERR:
117         printf("FPGA device operation error.\n");
118         break;
119       case UNKNOWNERR:
120         printf("Unknown error.\n");
121         break;
122       default:
123         break;
124       }
125       return 0;
126   }
```

```
127
128  /* Provide a function to parse the parameters. */
129  static error_t parse_opt (int key, char *arg, struct argp_state *state)
130  {
131    struct arguments *arguments = state->input;
132
133    switch (key) {
134    case 'v':
135      arguments->verbose = 1;
136      break;
137    case 'f':
138      arguments->format = arg;
139      break;
140    case ARGP_KEY_ARG: // one argument accepted
141      if (state->arg_num >= 1) {
142        argp_usage(state);
143      }
144      arguments->args[state->arg_num] = arg;
145      break;
146    case ARGP_KEY_END:
147      if (state->arg_num < 1) {
148        /* Not enough arguments. */
149        argp_usage (state);
150      }
151      break;
152    default:
153      return ARGP_ERR_UNKNOWN;
154    }
155
156    return 0;
157  }
158
159  /*****************************************************************************/
160  /* Initialize the FPGA                                                       */
161  /*****************************************************************************/
162  static void
163  mt_1999_set (void *vstate, unsigned long int s, struct arguments *arguments)
164  {
165      mt_state_t *state = (mt_state_t *) vstate;
166      int i, page_size;
167      err_e e;
168      unsigned long val;
169      if (s == 0)  s = DEFAULT_SEED;   /* the default seed is 4357 */
```

```
170
171       state->mt[0]= s & INT32_MASK;
172
173       for (i = 1; i < N; i++)
174       {
175         state->mt[i] = (MULTIPLIER*(state->mt[i-1]^(state->mt[i-1] >> 30)) + i);
176         state->mt[i] &= INT32_MASK;
177       }
178
179       state->mti = i;
180
181       /* set up the FTR memory */
182       /* Allocate two buffers of 1MB each */
183       page_size = getpagesize();
184       i = posix_memalign((void *) &buf0_ptr, page_size, BUFF_SIZE);
185       if (i != 0) {
186         printf("Unable to allocate memory for buffer 0.\n");
187         exit(1);
188       }
189
190       i = posix_memalign((void *) &buf1_ptr, page_size, BUFF_SIZE);
191       if (i != 0) {
192         printf("Unable to allocate memory for buffer 1.\n");
193         exit(1);
194       }
195
196       /* Register the buffers with the FPGA device driver */
197       fpga_register_ftrmem(fp_id, (void *) buf0_ptr, BUFF_SIZE, &e);
198       if (e != NOERR) {
199         printf("Unable to register buffer 0 with the FPGA device driver.\n");
200         print_err(e);
201         free((void *) buf0_ptr);
202         free((void *) buf1_ptr);
203         exit(1);
204       }
205       fpga_register_ftrmem(fp_id, (void *) buf1_ptr, BUFF_SIZE, &e);
206       if (e != NOERR) {
207         printf("Unable to register buffer 1 with the FPGA device driver.\n");
208         print_err(e);
209         free((void *) buf0_ptr);
210         free((void *) buf1_ptr);
211         exit(1);
212       }
```

```
213
214       bzero ((void *) buf0_ptr, BUFF_SIZE);
215       bzero ((void *) buf1_ptr, BUFF_SIZE);
216
217       /* FTR memory is split into two buffers for */
218       /* data transfer between FPGA and the SMP */
219       /* Program the buffer addresses. */
220       fpga_wrt_appif_val (fp_id,
221           (u_64) buf0_ptr,
222           BUFF0_PTR_REG, TYPE_ADDR, &e);
223       fpga_wrt_appif_val (fp_id,
224           (u_64) buf1_ptr,
225           BUFF1_PTR_REG, TYPE_ADDR, &e);
226
227       /* set up the config register */
228       /* the value contains polling frequency and number of pages per buffer */
229       val = (1UL << 8) - 1;  /* polling value set at 255 */
230       val <<= 32;
231       val |= ((1UL << 8) - 1); /* each buffer has 256 pages */
232       fpga_wrt_appif_val (fp_id, val, BUFF_CFG_REG, TYPE_VAL, &e);
233
234       /* write the state array */
235       for (i = 0; i < N/2; i++) {
236         fpga_wrt_appif_val (fp_id,
237             (((0UL | state->mt [2*i+1]) << 32) |
238             state->mt [2*i]),
239             MTA_RAM_ARRAY + (unsigned long) i*8, TYPE_VAL, &e);
240       }
241
242       /* Generate the configuration register value. */
243       /* Start by turning the init key off. */
244       val = 0;
245       val |= (MTA_INIT_KEY & 0UL);
246
247       /* Set the output format bits according to the command line parameter. */
248       switch(arguments->format[0]) {
249       case 'i': /* integer */
250         val |= (MTA_FORMAT & (MTA_INTEGER<<1));
251         break;
252       case 'f': /* float */
253         val |= (MTA_FORMAT & (MTA_FLOAT<<1));
254         break;
255       case 'd': /* double */
```

```
256         val |= (MTA_FORMAT & (MTA_DOUBLE<<1));
257         break;
258       default:
259         break;
260       }
261
262       /* Write the format and start value to the FPGA */
263       fpga_wrt_appif_val (fp_id, val, APP_CFG_REG, TYPE_VAL, &e);
264   }
265
266   /*************************************************************************/
267   /* Fetches random numbers written into the buffer space by the FPGA.     */
268   /* Returns a pointer to a block_ptr of 64-bit numbers.                   */
269   /*************************************************************************/
270
271   static inline volatile u_64 * mt_get (void)
272   {
273     static volatile u_64 * block_ptr = 0 ;
274     static volatile u_64 * curr_ptr = 0;
275     static int n = 1;
276     volatile u_64 * buf_ptr_arr [2];
277
278     buf_ptr_arr [0] = buf0_ptr;
279     buf_ptr_arr [1] = buf1_ptr;
280
281     if (n) {
282       curr_ptr = buf0_ptr;
283       n = 0;
284     }
285
286     if ((curr_ptr == buf_ptr_arr [0])
287         || (curr_ptr == buf_ptr_arr [1])) {
288       while (*curr_ptr != BUFF_RD_RDY) {
289         /* Wait ... */
290       }
291       /* first 64 bytes are unused except for the read-write flags */
292       curr_ptr += 8;
293     }
294
295     block_ptr = curr_ptr;
296     curr_ptr += READ_NUMS;
297
298     if ((curr_ptr == buf_ptr_arr [0] + QUAD_WRDS_PER_BUF)) {
```

```
299       /* done reading this buffer */
300       *(curr_ptr - QUAD_WRDS_PER_BUF) = BUFF_WRT_READY;
301       curr_ptr = buf_ptr_arr[1];
302     }
303
304     if (curr_ptr == (buf_ptr_arr [1] + QUAD_WRDS_PER_BUF)) {
305       /* done reading this buffer */
306       *(curr_ptr - QUAD_WRDS_PER_BUF) = BUFF_WRT_READY;
307       curr_ptr = buf_ptr_arr [0];
308     }
309
310     return block_ptr;
311   }
312
313   /************************************************************************/
314   /* Cast the pointer type returned by mt_get().                          */
315   /************************************************************************/
316
317   unsigned int *mt_get_int(void)
318   {
319     return (unsigned int *) mt_get();
320   }
321
322   float *mt_get_float(void)
323   {
324     return (float *) mt_get();
325   }
326
327   double *mt_get_double(void)
328   {
329     return (double *) mt_get();
330   }
331
332   /************************************************************************/
333   /* Main body                                                            */
334   /************************************************************************/
335
336   int main (int argc, char *argv [])
337   {
338     struct arguments arguments;
339     u_64 i=0,j=0,n=0;
340     mt_state_t state;
341     err_e e;
```

```
342     long begin=0, end=0, diff=0;
343     double rate;
344     volatile unsigned int *int_array = NULL;
345     volatile float *float_array = NULL;
346     volatile double *double_array = NULL;
347     unsigned int k = 0;
348     float  f = 0.0;
349     double d = 0.0;
350
351     /* Set argument defaults */
352     arguments.verbose = 0;
353     arguments.format  = "int";
354
355     /* Parse any command line options and arguments. Store them in */
356     /* the arguments structure. */
357     argp_parse (&argp, argc, argv, 0, 0, &arguments);
358
359     /* Check that the length of the test is reasonable */
360     n = strtol (arguments.args[0], NULL, 10);
361     if (n < 2) {
362       printf ("The minumum number of comparisons is 2 ... exiting.\n");
363       return(1);
364     }
365
366     /* Open the FPGA device */
367     /* We install the FPGA as /dev/ufp0 */
368     fp_id = fpga_open ("/dev/ufp0", O_RDWR|O_SYNC, &e);
369
370     /* If FPGA open failed exit. */
371     if (e != NOERR) {
372       printf ("Failed to open FPGA device. Exiting.\n");
373       return(1);
374     }
375
376     /* Reset the then restart. This puts the FPGA in a known state. */
377     fpga_reset (fp_id, &e);
378     fpga_start (fp_id, &e);
379
380     /* Initialize the FPGA. */
381     mt_1999_set (&state, 0UL, &arguments);
382
383     /* Fetch the reqired set of numbers based on command line arguments. */
384     switch(arguments.format[0]) {
```

```
385     case 'i': /* integer */
386       printf ("\nGenerating %ld 32 bit integers with FPGA.\n", n);
387       printf ("Calibrating timer ... ");
388       fflush(stdout);
389       begin = ustime ();
390       for (i = 0; i < n; i += READ_NUMS*2) {
391         int_array = mt_get_int();
392         for (j = 0; j < READ_NUMS*2; j++) {
393           k = int_array[j];
394         }
395       }
396       end = ustime ();
397       printf ("\n  Last number  : 0x%08X\n", k);
398       break;
399     case 'f': /* float */
400       printf ("\nGenerating %ld 32 bit floats with FPGA.\n", n);
401       printf ("Calibrating timer ... ");
402       fflush(stdout);
403       begin = ustime ();
404       for (i = 0; i < n; i += READ_NUMS*2) {
405         float_array = mt_get_float();
406         for (j = 0; j < READ_NUMS*2; j++) {
407           f = float_array[j];
408         }
409       }
410       end = ustime ();
411       printf ("\n  Last number  : %1.10f\n", f);
412       break;
413     case 'd': /* double */
414       printf ("\nGenerating %ld 64 bit doubles with FPGA.\n", n);
415       printf ("Calibrating timer ... ");
416       fflush(stdout);
417       begin = ustime ();
418       for (i = 0; i < n; i += READ_NUMS) {
419         double_array = mt_get_double();
420         for (j = 0; j < READ_NUMS; j++) {
421           d = double_array[j];
422         }
423       }
424       end = ustime ();
425       printf ("\n  Last number  : %1.20f\n", d);
426       break;
427     default:
```

```
428       printf ("\nUnknown format requested.\n");
429       break;
430     }
431
432     /* Calculate the time taken to generate the numbers. */
433     diff = end - begin;
434     rate = (double) n /(double) diff;
435     printf ("  Elapsed time : %ld microseconds\n", diff);
436     printf ("  Rate         : %4.3lf million numbers/second\n\n", rate);
437
438     /* stop the random number generation. */
439     fpga_wrt_appif_val (fp_id, 1UL, APP_CFG_REG, TYPE_VAL, &e);
440
441     /* deregister the buffers */
442     fpga_dereg_ftrmem(fp_id, (void *)buf0_ptr, &e);
443     if (e != NOERR) {
444       printf("Unable to deregister buffer 0 with the FPGA device driver\n");
445       print_err(e);
446     }
447     fpga_dereg_ftrmem(fp_id, (void *)buf1_ptr, &e);
448     if (e != NOERR) {
449       printf("Unable to deregister buffer 1 with the FPGA device driver\n");
450       print_err(e);
451     }
452
453     /* Free the buffers */
454     free((void *) buf0_ptr);
455     free((void *) buf1_ptr);
456
457     /* close the device */
458     fpga_close (fp_id, &e);
459
460     return 0;
461   }
```

Glossary

ACML

AMD Core Math Library

Active Manager

The software that monitors and manages all aspects of the Cray XD1 system. Its user interfaces provide administrators and end users with a single point of control for the system.

administrator

A user of the Cray XD1 system with unlimited access privileges, including permission to issue all Active Manager commands. The administrator is responsible for monitoring and managing the system.

AMD Core Math Library

A software package included with Cray XD1 Linux that includes routines for BLAS, FFT, and LAPACK. Routines are available for both Fortran 77 and C interfaces.

application release package

The major unit of distribution of software for the Cray XD1 system; contains HPC-optimized Linux, the Active Manager application layer, any third-party applications bundled by Cray, and, optionally, the FPGA logic framework and sample FPGA logic. The application release package consists of RPM packages.

compute blade

One of six circuit boards in a Cray XD1 chassis; contains Opteron processors configured as an SMP, DIMMs, and a RapidArray processor. A compute blade may also have an expansion module.

Cray XD1 system

A stand-alone Cray XD1 chassis or multiple chassis that communicate over both the supervisory network and the RapidArray interconnect.

end user

A user of the Cray XD1 system who does not have administrator privileges.

expansion module

Optional Cray XD1 hardware that connects to each compute blade; if they are present, a chassis has six expansion modules. The expansion modules provide a node with a second RapidArray processor, two additional Rapid Array links, and an optional application acceleration processor.

field-programmable gate array

An integrated circuit that consists of arrays of AND and OR gates (typically thousands) that can be programmed to perform complex functions. The Cray XD1 system has optional FPGAs available for use as application acceleration processors.

FPGA

See *field-programmable gate array*.

FPGA application acceleration processor

An FPGA that users can program to accelerate computationally intensive and repetitive algorithms; acts as a co-processor to the Opteron processor. This is an optional component on the expansion module. See also *JTAG interface card*.

infrastructure release package

The major unit of distribution of software for the Cray XD1 system that contains software and firmware components for the hardware supervisory subsystem and the compute blades. See also *application release package*.

interconnect

See *RapidArray interconnect*.

job

A computing task that runs on one processor or multiple processors concurrently. The workload management (WLM) system assigns the requested resources and launches the job.

JTAG interface card

Optional hardware that tests the application acceleration processor's integrated circuits. This card connects to one of the high-speed I/O slots on the main board of a Cray XD1 chassis.

Linux, Cray XD1

The HPC-optimized Linux operating system, based on the SuSE Linux Enterprise Server (SLES) distribution, that runs on each node. Cray optimizations include the implementation of a synchronized scheduler. See also *synchronized scheduler, Linux*.

LSS

Linux synchronized scheduler. See *synchronized scheduler, Linux*.

master node

The node on which the Active Manager server runs.

Message Passing Interface (MPI)

A widely accepted standard for communication among nodes that run a parallel program on a distributed-memory system. MPI is a library of routines that can be called from Fortran, C, and C++ programs.

node

An instance of the Linux operating system and the hardware components that it controls. The hardware components in a Cray XD1 node include an SMP and its associated memory, one or two RapidArray processors (depending on configuration) and, optionally, an FPGA application acceleration processor.

node working software image

The software image that is configured for an individual node and from which the node boots; generated automatically from the partition master software image by the Active Manager software when the node is allocated to a partition. The node working software image is stored either on the local disk of the node or in the Active Manager repository and NFS-mounted.

partition

A logical group of nodes with the same operating system version and

configuration; may reside in more than one Cray XD1 chassis. Partitions enable an organization to dedicate a set of nodes to perform a particular function (run a type of job, host a system-wide service, or serve a particular user group). Users treat the set of nodes in a partition as a single, homogeneous computing resource. Administrators specify the attributes of a partition. See also *partition master software image*.

partition master software image

The software image associated with a partition; used to generate the working software images of nodes that are allocated to the partition. The partition master software image is created from a combination of an application release master, a configuration determined by the partition's attributes, any other partition-wide configuration (such as services), and any installed local or third-party software.

RAP

RapidArray processor.

RapidArray interconnect

The high-speed network that interconnects the nodes in a Cray XD1 chassis, and connects all nodes in a Cray XD1 system via cables and optional external RapidArray switches. The RapidArray interconnect consists of a main and an optional expansion fabric, each with its own set of fabric components. The configuration of the RapidArray interconnect in a multichassis system is called the physical topology.

RapidArray link

The physical communication path between two RapidArray ports. Each link can carry two gigabytes per second.

RapidArray processor

The special-purpose processor on a Cray XD1 compute blade; responsible for most communication functions within the system. The RapidArray processor interfaces an Opteron processor to the RapidArray fabric.

RapidArray Transport core

An IP core for the FPGA application acceleration processor that provides the logic necessary for an FPGA design to interface (via the RapidArray fabric) to the rest of the Cray XD1 system.

release package

A unit in which Cray delivers software and firmware upgrades for the Cray XD1 system. See also *application release package* and *infrastructure release package*.

software image

A directory tree that contains the Cray XD1 Linux operating system, application software, and configuration information that is appropriate for the use of the image. See also *partition master software image* and *node working software image*.

symmetric multiprocessor (SMP)

In a Cray XD1 system, an SMP is formed from two single- or dual-core Opteron processors and their associated memory. One compute blade holds one SMP. Each chassis contains six compute blades and therefore contains six SMPs. See also *node*.

synchronized scheduler, Linux

The Linux process scheduler customized for the Cray XD1 system. It synchronizes time slots across all nodes in the system and allocates more time slots to computing jobs to maximize application performance. Administrators can configure the allotment of time slots on a partition-by-partition basis.

ufp

User FPGA processor. A combining form that occurs in file and directory names; for example, in <literal>libufp.a</literal>. It is a synonym for the FPGA application acceleration processor.

workload management (WLM) system

Software that schedules jobs for execution in a system of networked nodes.

Index

A

access control
  nodes, 7
  partitions, 7
ACML
  BLAS, 11
  FFT, 11
  LAPACK, 11
  using, 23
Aggregate Remote Memory Copy Interface
  See ARMCI
AMD Core Math Library
  See ACML
API
  See FPGA API
application acceleration processors
  See FPGAs
application programming interface (for FPGAs)
  See FPGA API
applications, sample
  See Mersenne Twister pseudorandom numbers
Apprentice^2
  using, 24
ARMCI
  in software distribution, 12
  using, 24

B

Basic Linear Algebra Subprograms
  See BLAS
batch process scheduling, 6
BLAS, 11

C

C compilers
  GNU, 9
C++ compilers
  GNU, 9
closing FPGAs, 65
command line
  See managing FPGA logic: command line
communication libraries
  ARMCI, 12
  GA, 12
  GPSHMEM, 12
  MPI, 11
  relationships (illustration), 13
  ROMIO, 11
compiler options, required, 15
compiler scripts, MPICH, 17
compilers
  in software distribution, 9
  modulefile template, 30
compiling
  MPICH library, 33
compiling and linking
  general considerations, 15
  MPICH applications, 19
compute blades, 39
converting raw FPGA logic files
  creating header file, 43
  merging header file, 43
  procedure, 43
  purpose, 43
coprocessors
  See FPGAs
CrayPAT
  in software distribution, 12
  using, 25

D

debuggers, 9
development tools
  See tools, development
device drivers
  FPGA application acceleration processor, 6

RapidArray interconnect, 6
documentation tools, 10
domain names
partitions, 9
system, 8
double-buffering, in MTA, 68
drivers
See device drivers

E

editors, text, 9
emacs, 9
environment
Cray XD1, 5
user, 7
environment modules
See Modules package
erasing, FPGAs, 45, 64

F

Fast Fourier Transform
See FFT
fcu command
examples, 70
introduction, 39
options
build, 43
convert, 43
exec, 45
load, 44
reset, 44
status, 45
unload, 45
FFT, 11
field-programmable gate arrays
See FPGAs
file descriptors, FPGA, 49
FORTRAN 77 compiler, 9
FPGA API
compiling and linking applications, 26
data transfer methods, 48
device files, 50
file descriptors, 49
fpga_close, 65
fpga_dereg_ftrmem, 61
fpga_is_loaded, 63
fpga_load, 50
fpga_mem_sync, 55
fpga_memmap, 53
fpga_open, 49
fpga_rd_appif_val, 59
fpga_register_ftrmem, 60
fpga_reset, 51
fpga_set_ftrmem, 60
fpga_start, 52
fpga_status, 62
fpga_unload, 64
fpga_wrt_appif_val, 57
introduction, 39
overview, 46
workflow, typical, 46
FPGA logic
loadable files
creating, 43
defined, 39
loading
API, 50
command line, 44
raw files
converting, 43
defined, 42
developing, 42
role of application programmer, 42
FPGA transfer region
See FTR
fpga_close function, 65
fpga_deprogram function, 64
fpga_dereg_ftrmem function, 61
fpga_is_loaded function, 63
fpga_load function, 50
fpga_mem_sync function, 55
fpga_memmap function, 53
fpga_open function, 49
fpga_rd_appif_val function, 59

`fpga_register_ftrmem` function, 61
fpga_reset function, 51
`fpga_set_ftrmem` function, 60
`fpga_start` function, 52
`fpga_status` function, 62
`fpga_wrt_appif_val` function, 57
FPGAs
  address space, mapped, 48
  clock speed, 43
  closing, 65
  device driver, 6
  erasing
    API, 64
    command line, 45
  getting started, 70
  opening, 49
  optional component, 39
  parallel computations, 39
  part number of variant, 43
  pipelined computations, 39
  programming state, 63
  purpose, 39
  releasing from reset
    API, 52
    command line, 45
  resetting
    API, 51
    command line, 44
  resetting user logic, 44
  sample application
    described, 66
    running, 70
  status, 45, 62
  variants, 42
  verifying operation, 70
FTR
  communicating address to FPGA, 60
  deregistering, 61
  overview, 48
  registering, 60
  sample application, 67
  size, 62

G

`g++`, 9
`g77`, 9
GA, 12
`gcc`, 9
`gdb`, 9
Generalized Portable SHMEM
    See GPSHMEM
Global Arrays
    See GA
  using, 26
GNU C compiler
  in software distribution, 9
  options, required, 15
GNU C++ compiler, 9
GNU FORTRAN 77
  options, required, 15
GPSHMEM
  in software distribution, 12
  using, 26
`groff`, 10

H

halting FPGA logic
    See resetting FPGAs
header files
  converting raw FPGA logic file, 43
  FPGA API, 49

I

`info` command (Linux), 10
interconnect
    See RapidArray interconnect

J

jobs, overview, 8

L

LAPACK, 11
libraries, software development
  communication, 11
  domain-specific, 13

FPGA API, 12
general, 11
HPC, 10
mathematics, 11
performance analysis, 12
libufp.a, 12
libufp.a, FPGA API object library, 49
Linear Algebra Package
See LAPACK
linking
dynamic vs. static, 15
Linux, Cray XD1
enhancements for HPC, 5
overview, 5
synchronized scheduler, 6
loading FPGA logic
API, 50
choosing a method, 67
command line, 44
locations, FPGA
accessing from application, 53
physical interpretation, 47
logging in
command, typical, 9
prerequisites, 7–8
to partitions, 7
logic
See FPGA logic

M

makeinfo command (Linux), 10
managing, FPGA logic
API, 45
command line, 44
mathematics libraries
BLAS, 11
FFT, 11
LAPACK, 11
Matsumoto, 66
memory, application, accessing from FPGA, 60
memory, FPGA
mapping options, 55
offset, 55
pointer in application address space, 55
protection, 54
type, 47
Mersenne Twister Accelerator
See MTA
Mersenne Twister pseudorandom numbers
algorithm, 66
high-level design, application, 67
sample program, 66
source code listing, 75
structure of program, 69
walkthrough, 69
Message Passing Interface
See MPI
mmap, 54–55
module command, 29
modulefiles
described, 29
predefined, 30
templates
compiler, 30
MPICH library, 32
MODULEPATH environment variable, 29
Modules package
module command, 29
modulefiles, 29
MODULEPATH environment variable, 29
overview, 29
MPI, 11
MPI-IO
See ROMIO
MPICH
applications
compiling and linking, 19
compiler scripts
available, 17
described, 17
invoking, 18
paths, 18
compiling the library, 33
include file path, 20

instances, multiple, 16
linking applications
dependencies, 21
examples, 21
main library, 20
shared library path, 22
modulefile template, 32
subdirectories, 16
MTA
double-buffering, 68
FPGA logic, sample, 66
high-level design, 67
protocol, 67
registers, 67

N

Nishimura, 66
nodes
access control, 7
overview, 5

O

opening FPGAs, 49
operating system, Cray XD1, 5
options, compiler, 15

P

PAPI
in software distribution, 12
using, 27
partitions
access control, 7
defined, 7
domain name, 9
PathScale compiler
options, required, 15
performance analysis
CrayPAT, 12
PAPI, 12
Performance Application Programming Interface
See PAPI
`perl`, 10
PGI compilers
options, required, 15
programming state, FPGA, 63
protection, FPGA memory, 54
protocol, FPGA, sample application, 67
`python`, 10

Q

QDR II SRAM, 47

R

RapidArray interconnect, device driver, 6
registers
MTA, 67
reading, 59
transforming an address before writing, 58
writing, 57
releasing from reset, FPGAs
API, 52
command line, 45
resetting FPGAs
API, 51
command line, 44
ROMIO
compiling and linking applications, 19
in software distribution, 11

S

ScaLAPACK
using, 28
scripting tools, 10
SDP, 7
SHMEM
See GPSHMEM
SLES, 5
SMPs, 5, 46
Sockets Direct Protocol
See SDP
`ssh`, 8
status, FPGA, 45, 62
SuSE Linux Enterprise Server, version, 5
symmetric multiprocessors

See SMPs
synchronized process scheduling, 6
synchronizing, accesses to FPGA locations, 55

T

templates, modulefile
  compiler, 30
  MPICH library, 32
text editors, 9
time measurement, in sample application, 68
time slots, synchronized scheduling, 6
tools, development
  included, 9
  local and third-party, 10
  optional, 10
`troff`, 10

U

`ufplib.h`, FPGA API header file, 49
user FPGAs
  See FPGAs
utility programs
  See `fcu`

V

variant FPGAs, 42
`vim`, 9

X

Xilinx Virtex II Pro, 39