FUJITSU

FUJITSU Software
Technical Computing Suite V4.0L20

Development Studio
uTofu User's Guide

J2UL-2567-01ENZ0(03)
July 2021

Preface

Purpose of this Manual

This manual describes how to use uTofu, which is a programming interface for communication using the Tofu interconnect.

In this manual, a computing system with Fujitsu A64FX is called "this computing system".

Intended Readers

The intended readers of this manual are those who develop programs which communicate using the Tofu interconnect. In addition to knowledge of the C language, readers need to have a basic knowledge of Linux commands, file manipulation, and shell programming.

Structure of This Manual

The structure of this manual is as follows:

Chapter 1 Tofu Interconnect and uTofu Overview

An overview of the Tofu interconnect and uTofu

Chapter 2 uTofu Communication Models

The communication models of uTofu

Chapter 3 uTofu Interface Specifications

The interface specifications of uTofu

Chapter 4 How to Use uTofu

How to use uTofu

Chapter 5 Use Examples of uTofu

Some examples of uTofu use

Chapter 6 System Information

System-specific information

Chapter 7 Error Messages

The error messages of uTofu implementation

Glossary

Terminology

Related Manuals

The following manuals are related to this manual. Refer to these manuals in conjunction with this manual.

- *MPI User's Guide*
- *C User's Guide*
- *C++ User's Guide*
- *Fortran User's Guide*

Also, refer to the manuals provided with the related software Job Operation Software.

This manual mentions MPI in some places. To learn the MPI standard, refer to the following document:

MPI: A Message-Passing Interface Standard Version 3.1 Message Passing Interface Forum June 4, 2015

Information concerning MPI is available from https://www.mpi-forum.org/.

Expression of Units

In this manual, the following prefixes are used to express units:

Prefix	Value	Prefix	Value
k (kilo)	10^3	Ki (kibi)	2^{10}
M (mega)	10^6	Mi (mebi)	2^{20}
G (giga)	10^9	Gi (gibi)	2^{30}

Conventions Used in this Manual

The typographic conventions below are symbols used with pre-determined special meanings that express syntax.

Symbol name	Symbol	Explanation
Selection symbols	{}	Indicates to select any one of the enclosed items
	|	Used as a delimiter in a list of items
Omission permitted symbol	[]	Indicates that the enclosed item can be omitted. This symbol includes the meaning of the selection symbol "{}".

Export Controls

Exportation/release of this document may require necessary procedures in accordance with the regulations of your resident country and/or US export control laws.

Trademarks

- Linux(R) is the registered trademark of Linus Torvalds in the U.S. and other countries.
- All other trademarks are the property of their respective owners.

Date of Publication and Version

Version	Manual code
July 2021, Version 1.3	J2UL-2567-01ENZ0(03)
September 2020, Version 1.2	J2UL-2567-01ENZ0(02)
June 2020, Version 1.1	J2UL-2567-01ENZ0(01)
February 2020, 1st Version	J2UL-2567-01ENZ0(00)

Copyright

Update History

Changes	Location	Version
Added the following environment variable. - `UTOFU_SWAP_PROTECT`	4.3.2.5	Version 1.3
Modified the explanation about concentrated communications.	4.1.4	Version 1.2
Changed the number of available CQs per TNI.	6.1.2	

Changes	Location	Version
Added the explanation about virtual resource quantities.		
Added "Error Messages".	Chapter 7	
Added "The Behavior When Configured to Change the Communication Path If a Tofu Interconnect Link is Down".	6.1.4.2	Version 1.1
Improved the explanation.	-	

Contents

Chapter 1 Tofu Interconnect and uTofu Overview

This chapter provides an overview of the Tofu interconnect and uTofu.

uTofu is a programming interface for writing programs to communicate over the Tofu interconnect.

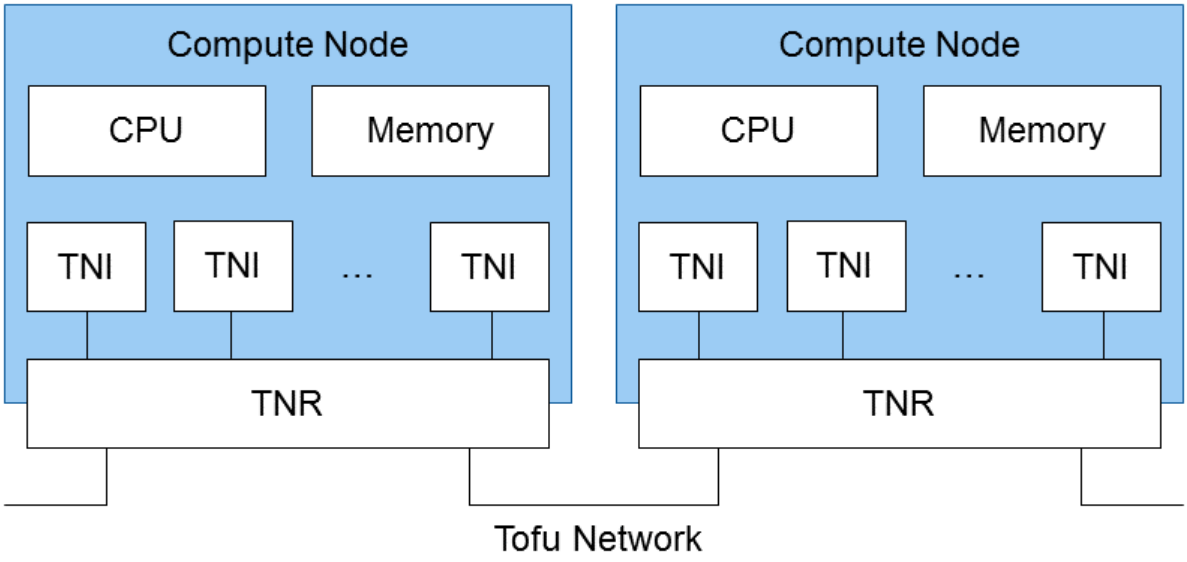
1.1 Tofu Interconnect Overview

1.1.1 Architecture

The Tofu interconnect is the interconnect for a parallel computing system consisting of a Tofu network, Tofu network interfaces (TNIs), and Tofu network routers (TNRs). In this manual, Tofu interconnect is a general name. The name collectively refers to the Tofu interconnect used with the High-end technical computing server MP10 system and Supercomputer PRIMEHPC FX10 system, the Tofu interconnect 2 used with the Supercomputer PRIMEHPC FX100 system, and the Tofu interconnect D used with this computing system.

Each of these parallel computing systems consists of multiple compute nodes, and each compute node has one TNR and at least one TNI. TNRs are interconnected to make up a Tofu network. A TNI sends and receives packets according to instructions from software. A TNR and a Tofu network transfer packets from a source TNI to a destination TNI. Simultaneous processing by two or more TNIs in one compute node enables high-throughput communication. The following figure shows a compute node configuration.

Figure 1.1 Compute Node Configuration

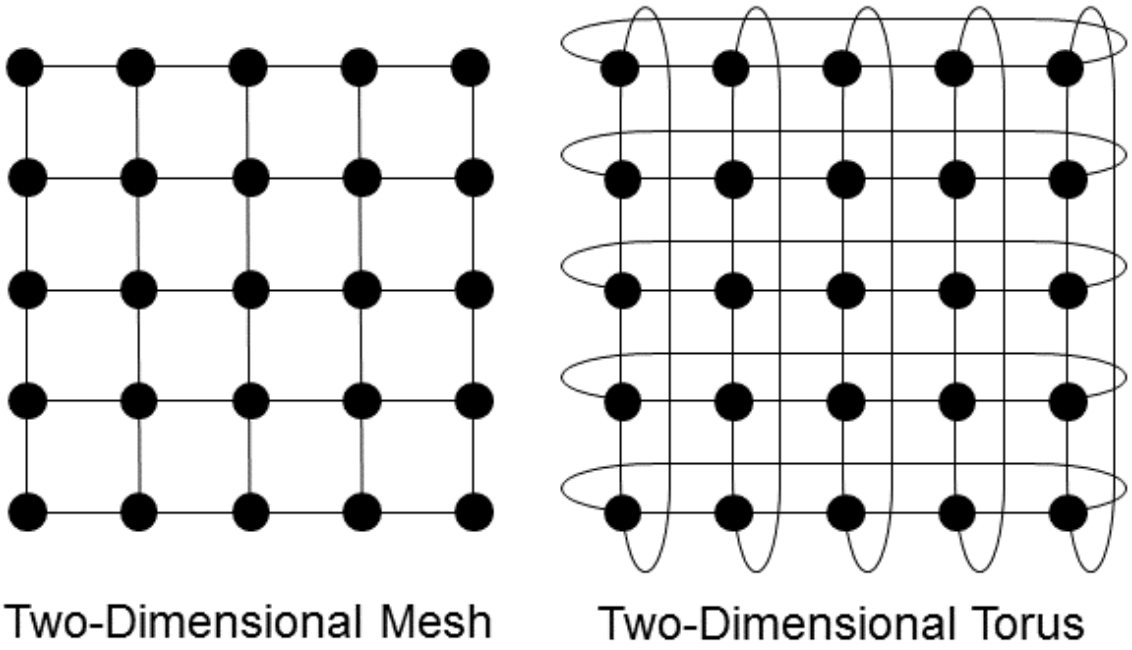


1.1.2 Network

The Tofu interconnect connects multiple compute nodes with a Tofu network.

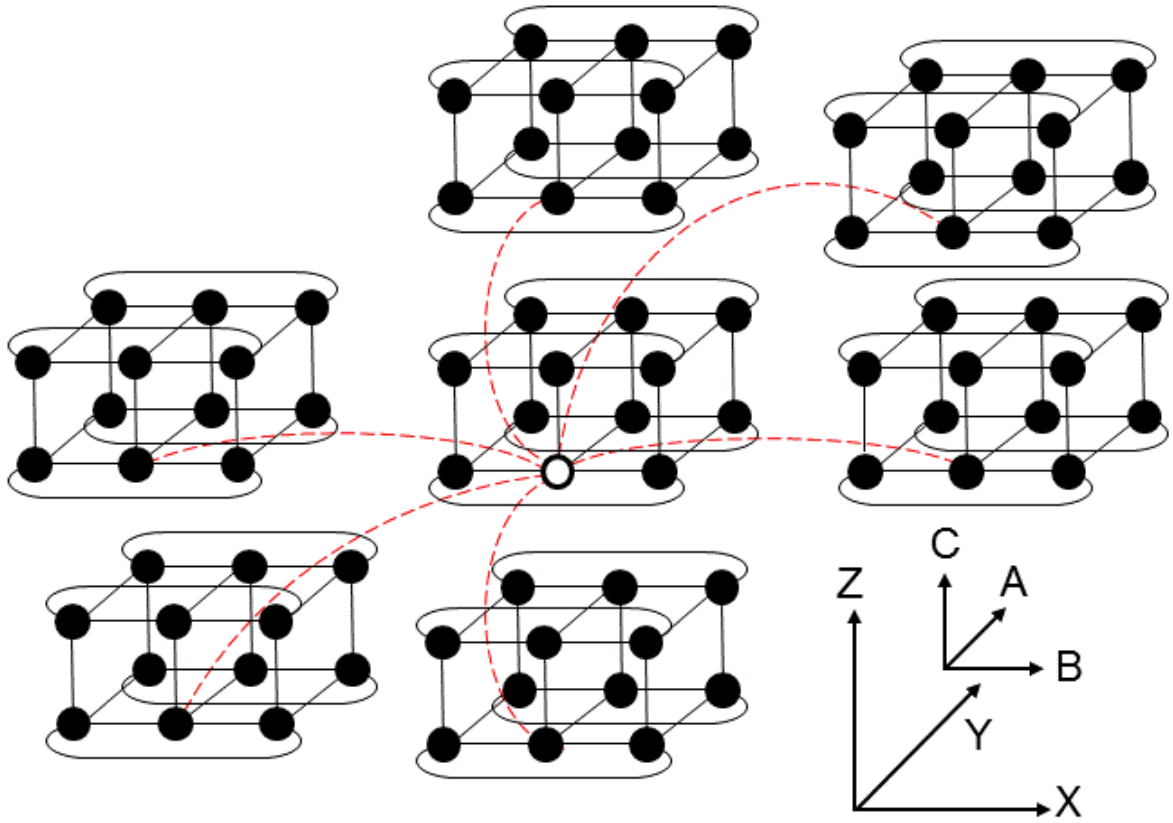
The Tofu network has a six-dimensional mesh/torus topology. The word *mesh* refers to a topology without interconnected endpoints on a coordinate axis. The word *torus* refers to a topology with interconnected endpoints on a coordinate axis. The following figure shows an example of a two-dimensional mesh and torus.

Figure 1.2 Example of a Two-Dimensional Mesh and Torus



The six-dimensional coordinate axes of the Tofu network are respectively assigned the letters X, Y, Z, A, B, and C. The three dimensions of A, B, and C have fixed lengths of 2, 3, and 2, respectively, and the B axis connects them in a torus. The three dimensions of X, Y, and Z have varying lengths depending on the system, and whether they are connected in a mesh or torus also varies with the system. For this reason, the TNR of a single compute node has up to 10 Tofu links in total: 1 Tofu link along each of A and C; 2 Tofu links along B; and either 2 Tofu links along each of X, Y, and Z on compute nodes which are not located on mesh endpoints of the axis or 1 Tofu link along each of X, Y, and Z on compute nodes which are located on mesh endpoints of the axis. The following figure shows these six-dimensional connections. In this figure, all the connections on the A, B, and C axes have been drawn with solid lines, represented in a rectangular parallelepiped. The focus for the X, Y, and Z axes is the single compute node at the center, so only the links from this compute node have been drawn with broken lines.

Figure 1.3 Six-Dimensional Connections of a Tofu Network



Each compute node can be identified by X,Y,Z,A,B,C coordinates on these six dimensions. The coordinates are called compute node coordinates.

Communication between non-neighboring compute nodes (not directly connected by a Tofu link) in the six-dimensional mesh/torus is relayed by the TNRs of their intermediate nodes.

1.1.3 Communication Methods

The Tofu interconnect can use two communication methods to communicate from the user space: one-sided communication, and barrier communication.

1.1.3.1 One-Sided Communication

One-sided communication accesses the main memory of a compute node and updates the main memory of another compute node without going through the CPU. This communication method has three types of communication function: Put, Get, and ARMW (Atomic Read Modify Write). The method is generally called RDMA (remote direct memory access).

Put is a communication function that reads contiguous data from the main memory of the local compute node and writes the data to the main memory of a remote compute node. Get is a communication function that reads contiguous data from the main memory of a remote compute node and writes the data to the main memory of the local compute node. ARMW is a communication function that reads, calculates, and writes four-byte or eight-byte data atomically on the main memory of a remote compute node.

ARMW operators, ARMW operands, and the memory regions targeted for reading and writing on the local compute node and remote compute node are all specified by software on the local compute node.

One-sided communication is executed from a process in the user space without going through the kernel.

1.1.3.2 Barrier Communication

Barrier communication is a communication function that performs barrier synchronization between multiple compute nodes. Barrier communication is configured beforehand for the processes that will be participating in barrier communication. Each process starts barrier

communication and then waits for it to complete. Once all the processes participating in the barrier communication have started the barrier communication, the barrier communication with each process is completed. The interconnect can also execute the reduction operation along with barrier synchronization.

Barrier communication can be used to implement the `MPI_BARRIER` routine, `MPI_ALLREDUCE` routine, etc. of MPI.

Barrier communication is executed from a process in the user space without going through the kernel.

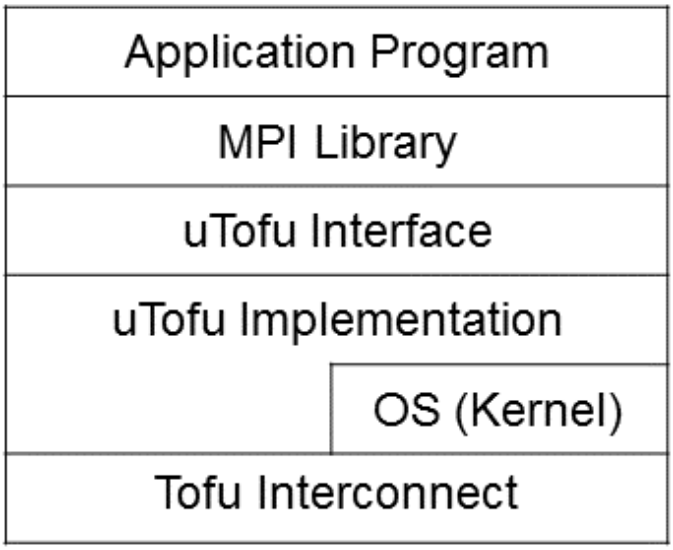
1.2 uTofu Overview

uTofu is a low-level application programming interface (API) used by software in the user space to communicate using the Tofu interconnect. uTofu supports the one-sided communication and barrier communication of the Tofu interconnect. The interface of uTofu is provided as functions of C. uTofu implementations are provided in the form of a library. In this manual, this implementation is called uTofu implementation.

The design of uTofu assumes that it is used from libraries (for example, MPI library) or the like for communication between processes by application programs.

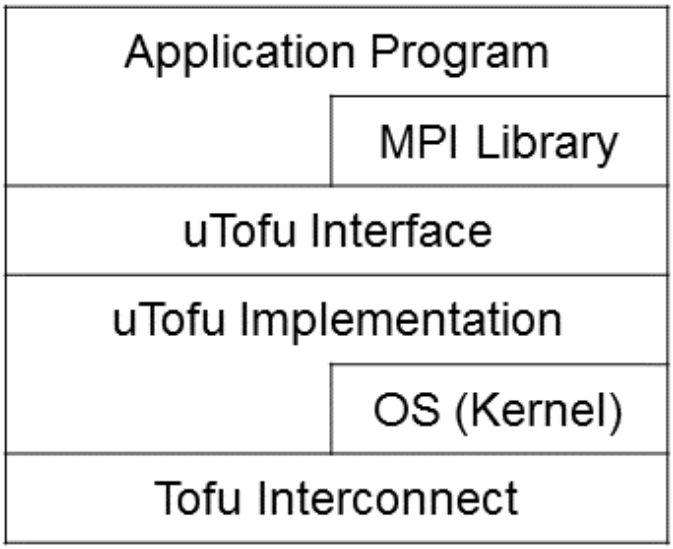
An example when uTofu is used from an MPI library is shown in "Figure 1.4 Example of a Software Stack when uTofu is Used from an MPI Library".

Figure 1.4 Example of a Software Stack when uTofu is Used from an MPI Library



Nonetheless, application programs can use uTofu directly for performance tuning of communication using MPI as shown in "Figure 1.5 Example of a Software Stack when uTofu is Used Directly from Application Programs

Figure 1.5 Example of a Software Stack when uTofu is Used Directly from Application Programs



The interface can also be used for communication (memory access) within a single thread or between threads in a single process. In this manual, the libraries and application programs that use uTofu are called uTofu programs. Similarly, the processes that execute uTofu programs are called uTofu processes.

uTofu does not have any functions for identifying and managing multiple processes like ranks and groups of MPI. If you require such functions, you will need to use uTofu in combination with MPI-like functions or create original functions.

The functions to start communication and confirm communication completion are separated in uTofu. If the function to start communication cannot start the communications immediately, that function returns a return value indicating that status. The function to confirm communication completion returns information on whether or not communication has completed. Because of that, all functions return within a finite length of time regardless of the process and thread status at the other end of the communication. And, it is necessary to confirm a return value of functions for making sure to communicate.

Chapter 2 uTofu Communication Models

This chapter describes the communication models of uTofu.

2.1 Execution Unit

The execution unit of uTofu is a process in the user space of the OS. Each execution unit has a communication context. This means, for example, that:

- a process cannot directly use the communication resources created or allocated by another process, and
- a thread can confirm the completion of communication started by another thread in the same process.

uTofu has functions (VCQ and VBG) to separate the communication context within one process. Using these functions, multiple software components (MPI library, parallel language runtime environment, etc.) or multiple threads can communicate in a logically independent manner within one process.

2.2 One-Sided Communication

In one-sided communication, the TNI communicates between two compute nodes by directly accessing and updating the main memory of the compute nodes and by sending and receiving packets.

2.2.1 VCQ (Virtual Control Queue)

Software can manipulate the TNI through a CQ (control queue). The number of CQs per TNI is finite and varies depending on the type of Tofu interconnect. For this reason, uTofu virtualizes and abstracts CQs, which are hardware resources, as VCQs (virtual control queues). uTofu performs all one-sided communication via VCQs.

Communication via a VCQ does not affect the logical behavior of communication via another VCQ. For example, after communication starts via a VCQ, only this VCQ will be notified of its completion.

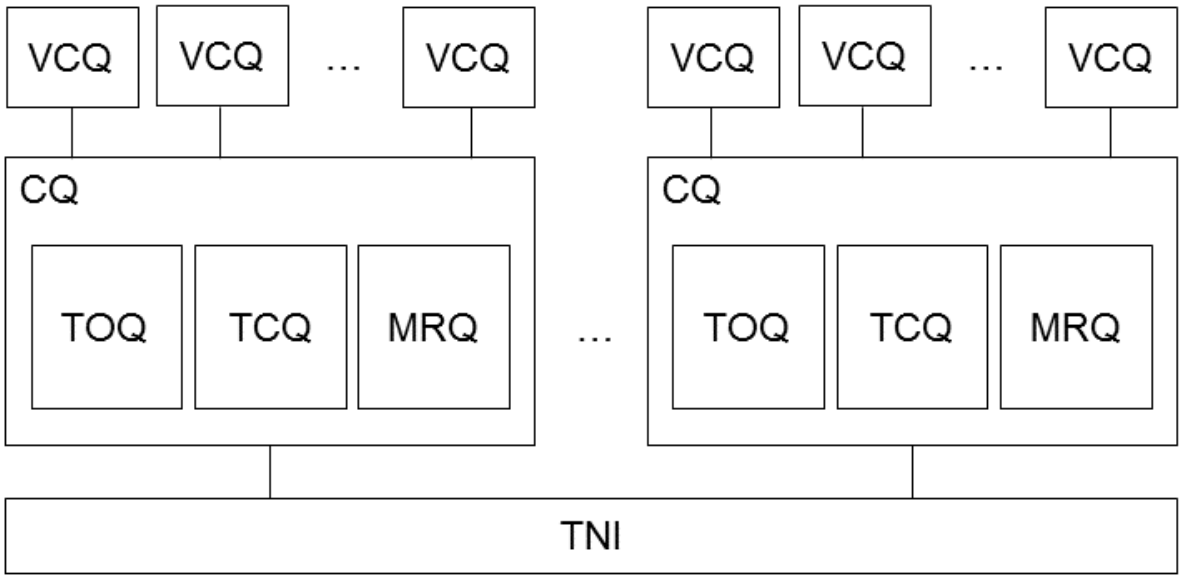
If a process creates a VCQ, it obtains a VCQ handle that is valid only within the process. All the functions of one-sided communication of uTofu have a VCQ handle as a parameter. Also, each VCQ is assigned a VCQ ID. The VCQ ID is a 64-bit unsigned integer identifying the VCQ, and it is a unique value among all the VCQs within the Tofu network.

One-sided communication is performed between two CQs in the Tofu interconnect. One of the two CQs is the CQ given a communication instruction by software (local CQ), and the other is the communication target CQ (remote CQ). To perform one-sided communication with uTofu, software gives the local VCQ handle and a remote VCQ ID to uTofu function parameters. Therefore, the software needs to know the remote VCQ ID before starting communication.

Each CQ and VCQ consists of the three queues TOQ, TCQ, and MRQ, which have different roles. Software writes and reads data called descriptors to/from these three queues to issue a communication instruction and confirm the completion of communication.

The following figure shows the TNI, CQ, and VCQ configuration.

Figure 2.1 TNI, CQ, and VCQ Configuration



2.2.1.1 TOQ (Transmit Order Queue)

The TOQ is a transmit order queue. Software uses the queue to issue a communication instruction to the TNI. Software writes a descriptor, and the TNI reads it. This descriptor is called the TOQ descriptor.

There are multiple types of communication in one-sided communication. TOQ descriptors are subdivided according to the type of communication. The following table lists the correspondence between the types of communication and the descriptor names. The subdivided TOQ descriptors are called Put descriptor, etc. Although NOP is not communication, it is included in this table for convenience because a descriptor exists.

Table 2.1 Correspondence Between the Types of Communication and Descriptor

Types of Communication	Descriptor Name
Put	Put
	Put Piggyback
Get	Get
ARMW	ARMW
NOP	NOP

2.2.1.2 TCQ (Transmit Complete Queue)

The TCQ is a transmit complete queue. The TNI uses the queue to notify software of data transmission completion. The TNI writes a descriptor, and software reads it. This descriptor is called the TCQ descriptor.

2.2.1.3 MRQ (Message Receive Queue)

The MRQ is a message receive queue. The TNI uses the queue to notify software of communication completion. The TNI writes a descriptor, and software reads it. This descriptor is called the MRQ descriptor. There are two types in MRQ descriptors: local notification and remote notification.

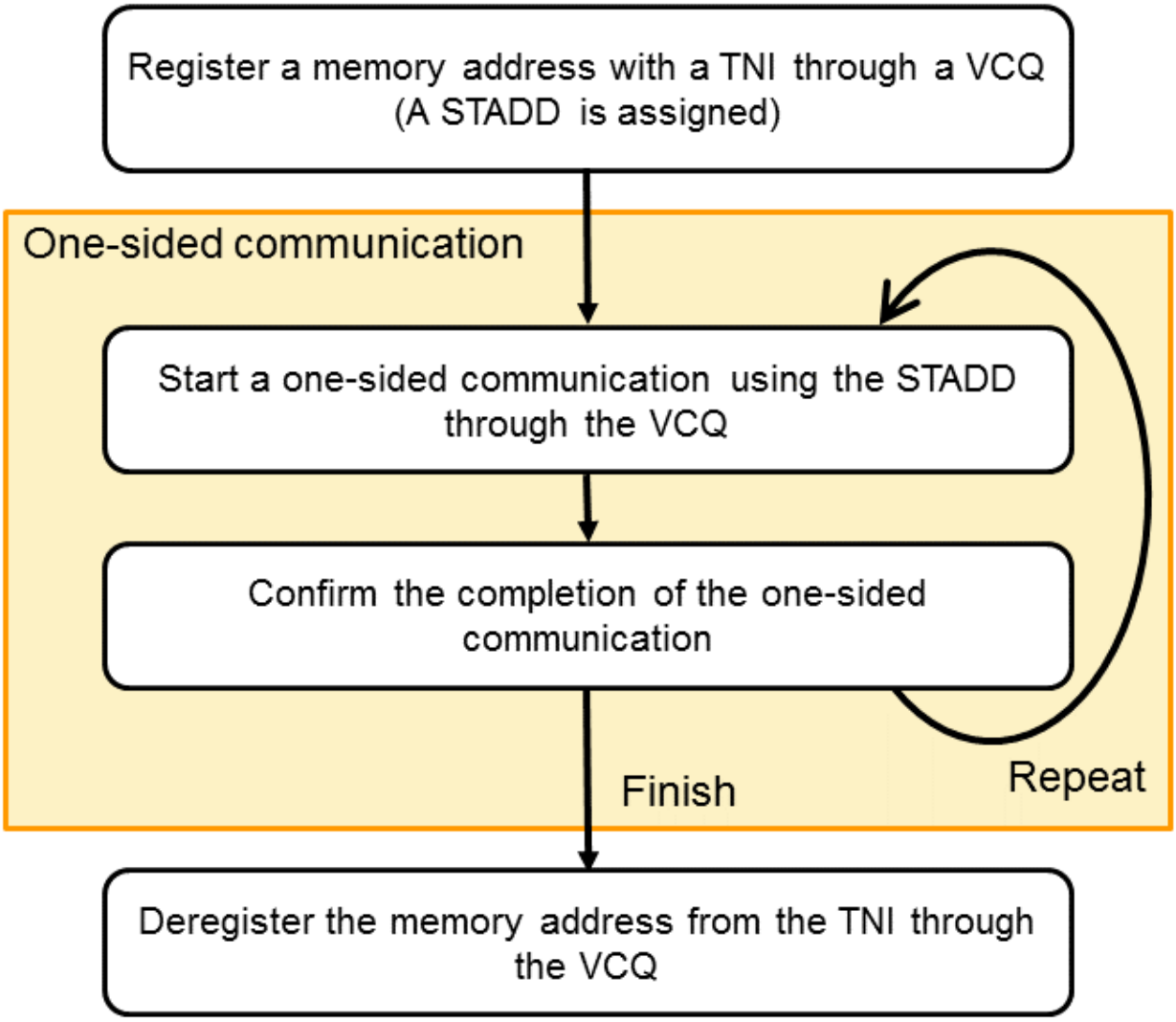
2.2.2 STADD (Steering Address)

Normally, processes in the user space identify memory regions with virtual memory addresses, but the Tofu interconnect does not recognize virtual addresses because it accesses memory without going through the OS. In the Tofu interconnect, a value called STag (steering tag) identifies the memory region accessed or updated in one-sided communication.

To make it easy to use the STag, uTofu adopts a value called STADD (steering address). The STADD is a 64-bit unsigned integer, and each is a unique value within a CQ. One memory region may have different STADD values for different CQs.

STag values (and STADD values) are not automatically assigned to the virtual addresses available to a process. Therefore, before one-sided communication starts, it is necessary to register memory addresses with a TNI to obtain the corresponding STADD values. Once a memory address is registered, the corresponding STADD can be used for one-sided communication multiple times. When that use for one-sided communication ends, that memory address registration with the TNI must be deregistered. If the registration is not deregistered and many memory addresses remain registered, STag values may be exhausted or the performance of STADD management functions of uTofu may degrade.

Figure 2.2 Flow from Registration to Deregister of STADD



To perform one-sided communication with uTofu, software gives local and remote STADD values to uTofu function parameters. Therefore, the software needs to know the remote STADD value before starting communication.

STADD values are managed for each VCQ.

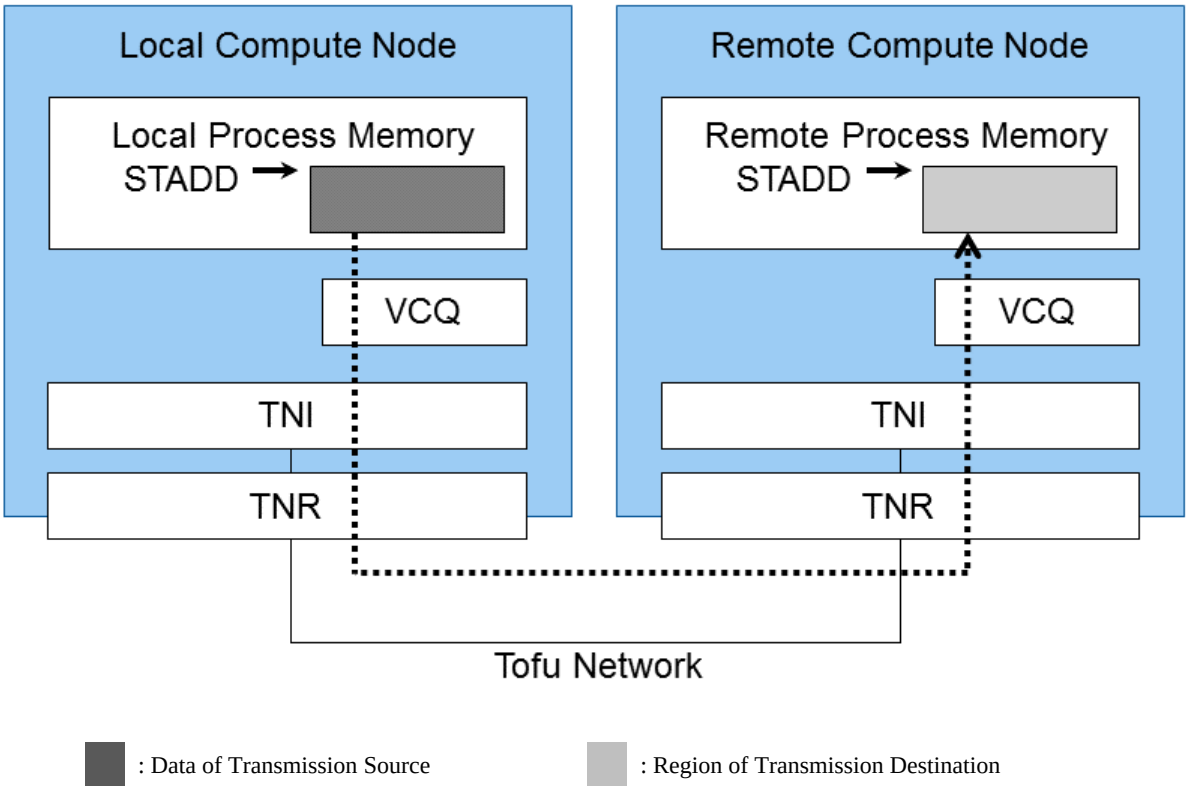
2.2.3 Put Communication

Put is a communication function that writes contiguous data from the memory of the local process (transmission source) to the memory of a remote process (transmission destination).

A VCQ handle specifies the TNI used by the local compute node. A remote VCQ ID specifies the TNI used by a remote compute node. The local STADD specifies the start address of the memory of the transmission source. A remote STADD specifies the start address of the memory of the transmission destination.

The following figure shows the Put communication model executed by the Tofu interconnect. The dotted arrow in the figure represents the data flow.

Figure 2.3 Put Communication Model



Put has the following steps.

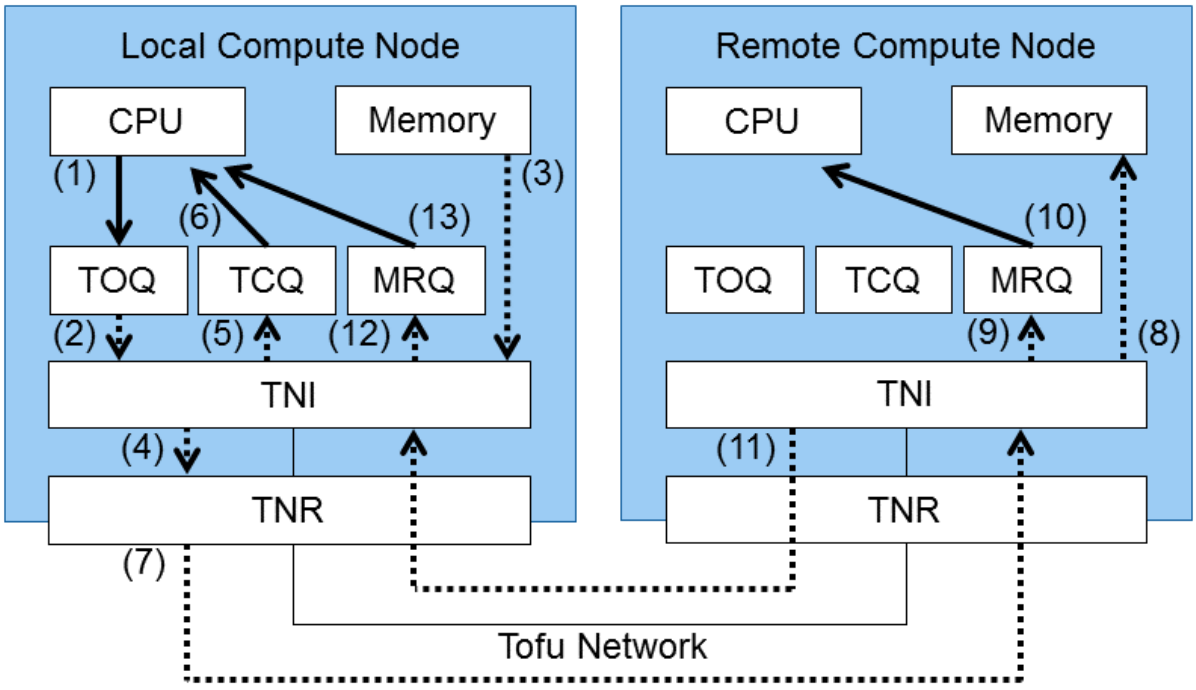
The steps 1, 6, 10 and 13 in this procedure need to be executed by calling the uTofu functions.

1. The local process writes a descriptor to the TOQ to instruct the TNI to start communication.
2. The local TNI reads the descriptor from the TOQ.
3. The local TNI reads data from memory.
4. The local TNI sends data to the TNR.
5. The local TNI writes a descriptor to the TCQ.
6. The local process reads the descriptor from the TCQ. By checking this TCQ descriptor, the local process can judge that the data written to destination memory will not be affected even if the local process update the memory region of the transmission source from this time.
7. The TNR and the Tofu network transfer the data to the remote TNI.
8. The remote TNI writes the data to memory.
9. The remote TNI writes a remote notification descriptor to the MRQ.
10. The remote process reads the descriptor from the MRQ. By checking this remote notification MRQ descriptor, the remote process can find out that the memory region of the transmission destination has been updated.
11. The remote TNI transfers a packet for memory write completion notification to the local TNI via the TNR and Tofu network.
12. The local TNI writes a local notification descriptor to the MRQ.
13. The local process reads the descriptor from the MRQ. By checking this local notification MRQ descriptor, the local process can find out that the memory region of the transmission destination has been updated.

The choice to notify or not notify can be made for each of the TCQ descriptor, remote notification MRQ descriptor, and local notification MRQ descriptor when the local process writes the descriptor to the TOQ.

The following figure shows these steps. The numbers in parentheses in this figure correspond to the above steps. The solid arrows in the figure represent steps executed by calling uTofu functions.

Figure 2.4 Put Steps



For Put, the local TNI reads data from memory after it reads the TOQ descriptor. For the Tofu interconnect, the local process can embed data in the TOQ descriptor if the data size is small. This is called piggyback. Using piggyback eliminates the local TNI processing that reads data from memory, thereby enabling low-latency communication. It also eliminates the need for the local process to register a memory region with the TNI. uTofu can inquire about which data sizes allow piggyback.

For details of the example that performs Put communication, see "5.1.1 Example of Ping-Pong Communication Using Put".

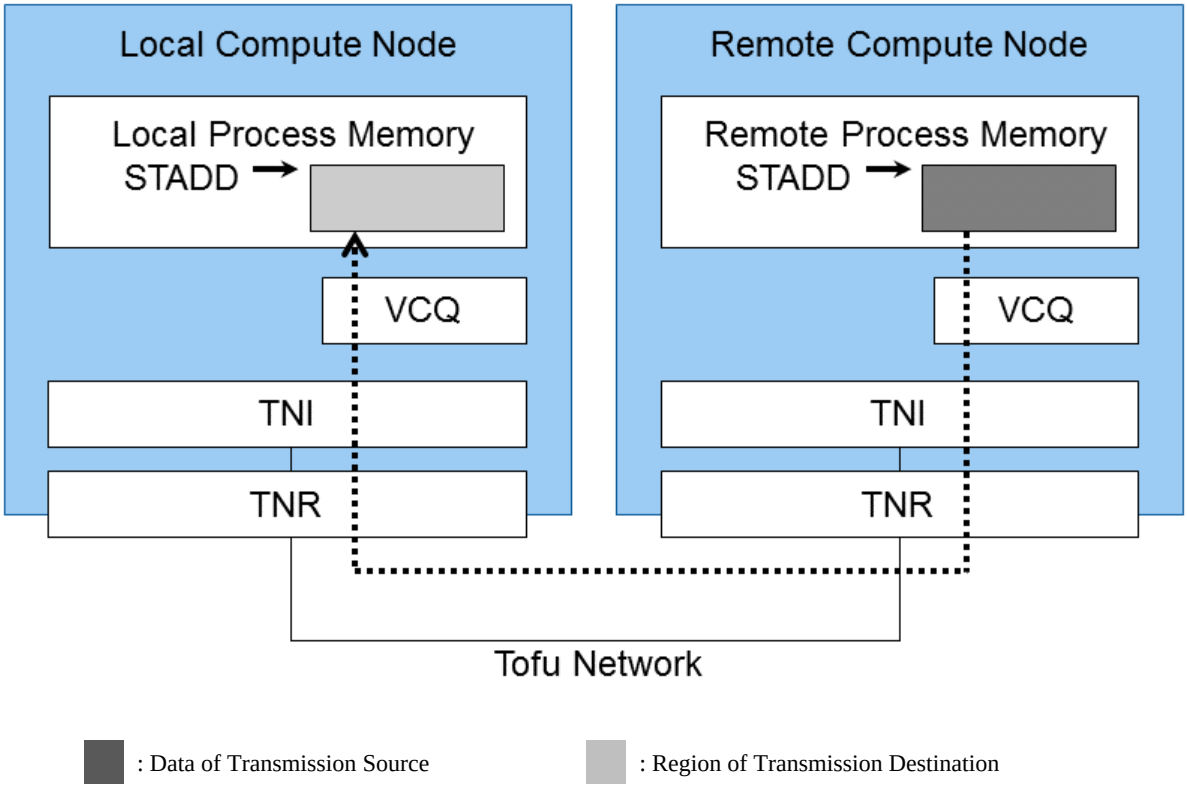
2.2.4 Get Communication

Get is a communication function that writes contiguous data from the memory of a remote process (transmission source) to the memory of the local process (transmission destination).

A VCQ handle specifies the TNI used by the local compute node. A remote VCQ ID specifies the TNI used by a remote compute node. A remote STADD specifies the start address of the memory of the transmission source. The local STADD specifies the start address of the memory of the transmission destination.

The following figure shows the Get communication model executed by the Tofu interconnect. The dotted arrow in the figure represents the data flow.

Figure 2.5 Get Communication Model



Get has the following steps.

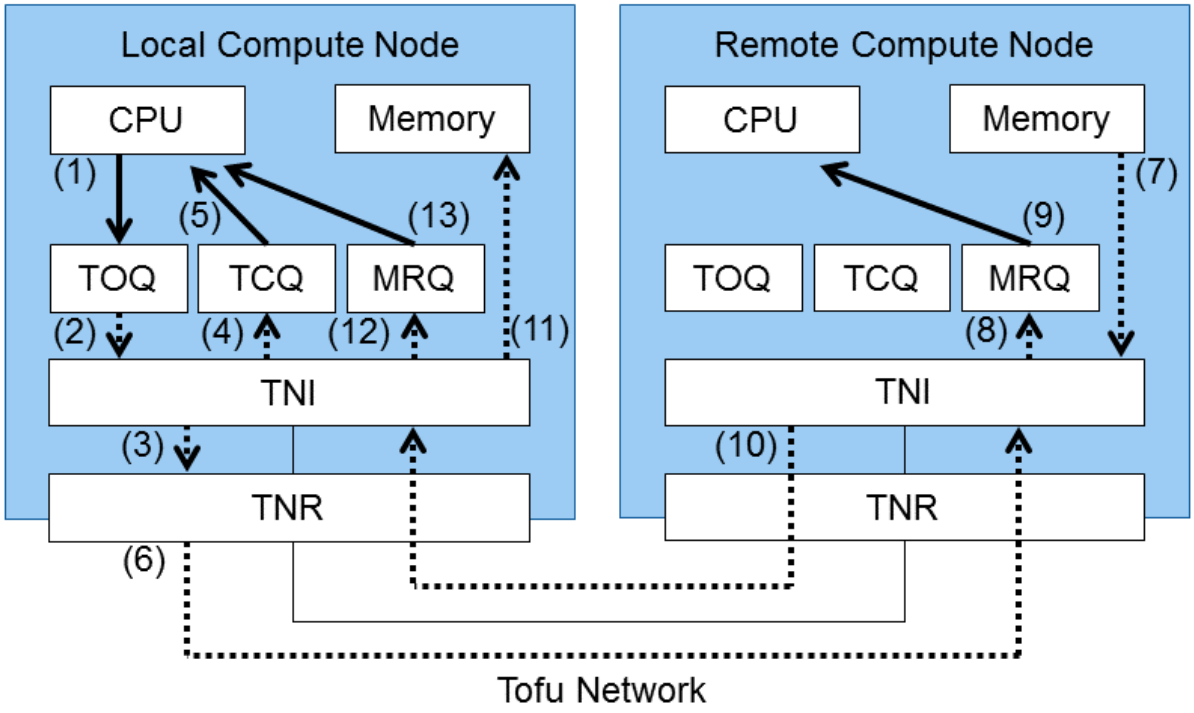
The steps 1, 5, 9 and 13 in this procedure need to be executed by calling the uTofu functions.

1. The local process writes a descriptor to the TOQ to instruct the TNI to start communication.
2. The local TNI reads the descriptor from the TOQ.
3. The local TNI sends a packet to issue a data transmission instruction to the TNR.
4. The local TNI writes a descriptor to the TCQ.
5. The local process reads the descriptor from the TCQ. By checking this TCQ descriptor, the local process can find out that the communication has started.
6. The TNR and the Tofu network transfer the instruction packet to the remote TNI.
7. The remote TNI reads data from memory.
8. The remote TNI writes a remote notification descriptor to the MRQ.
9. The remote process reads the descriptor from the MRQ. By checking this remote notification MRQ descriptor, the remote process can judge that the data written to destination memory will not be affected even if the remote process update the memory region of the transmission source from this time.
10. The remote TNI transfers the data to the local TNI via the TNR and Tofu network.
11. The local TNI writes the data to memory.
12. The local TNI writes a local notification descriptor to the MRQ.
13. The local process reads the descriptor from the MRQ. By checking this local notification MRQ descriptor, the local process can find out that the memory region of the transmission destination has been updated.

The choice to notify or not notify can be made for each of the TCQ descriptor, remote notification MRQ descriptor, and local notification MRQ descriptor when the local process writes the descriptor to the TOQ.

The following figure shows these steps. The numbers in parentheses in this figure correspond to the above steps. The solid arrows in the figure represent steps executed by calling uTofu functions.

Figure 2.6 Get Steps



For details of the example that performs Get communication, see "5.1.2 Example of Status Check Using Get".

2.2.5 ARMW (Atomic Read Modify Write) Communication

ARMW is a communication function that reads, calculates, and writes four-byte or eight-byte data atomically on the main memory of a remote compute node.

The following table lists the operators available for uTofu.

Table 2.2 ARMW Operators

Name	Operation
CSWAP	Compare and Swap
SWAP	Swap
ADD	Unsigned integer addition
XOR	Bitwise XOR
AND	Bitwise AND
OR	Bitwise OR

CSWAP is processed as follows.

1. The local process specifies two operands to a TOQ descriptor. They are swap operand and compare operand.
2. The TNI of the remote compute node writes the swap operand value to the target memory only if the pre-operation value is equal to the compare operand value.

SWAP is processed as follows.

1. The local process specifies one swap operand to a TOQ descriptor.
2. The TNI of the remote compute node writes the swap operand value to the target memory regardless of the pre-operation value on the target memory.

Other operators are processed as follows.

1. The local process specifies one operand to a TOQ descriptor.
2. The TNI of the remote compute node executes the relevant operation with the pre-operation value and operand, and then the resultant value is written.

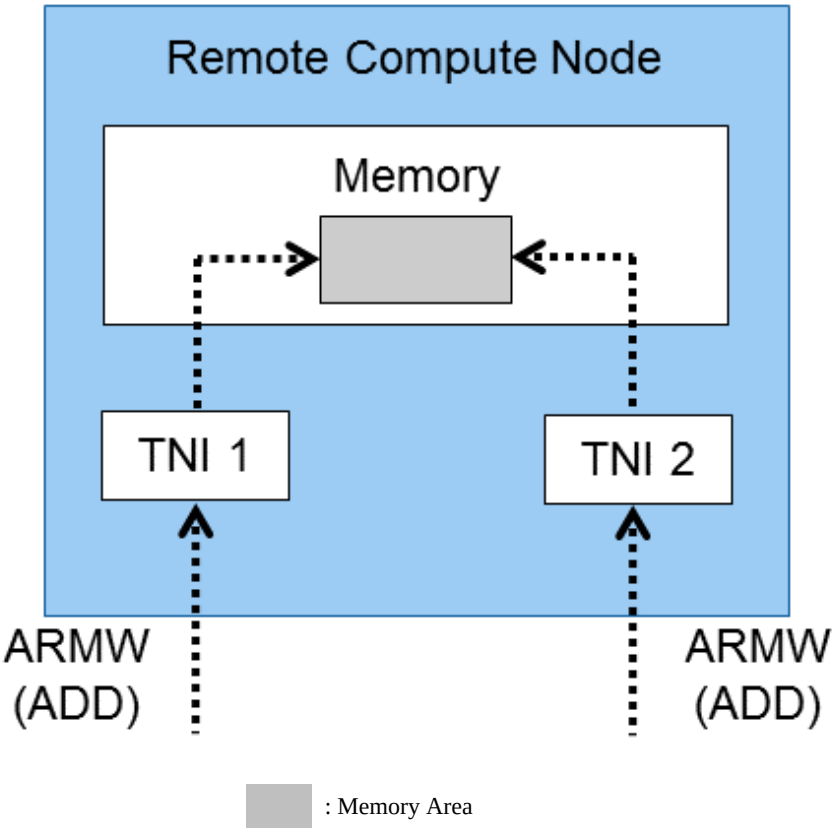
For any operation, the pre-operation value is reported using a local notification MRQ descriptor. For example, for ADD, if the local process specifies a value 11 as the operand and the pre-operation value on the target memory of the remote compute node is 22, a value 33 is written to the target memory and the value 22 is notified the local process of by a local notification MRQ descriptor.

In ARMW, the memory access/update operation on a remote compute node is performed atomically. The word *atomic* means the following: if more than one memory update operations are performed concurrently, reading, calculating, or writing of no other memory update operation are performed while reading, calculating, and writing of one memory update operation are being performed. This operation is guaranteed atomic not only with ARMW execution by the same remote VCQ but also with ARMW execution by another remote VCQ or another remote TNI, and with execution of an atomic memory access/update instruction by a remote CPU.

For example, assume that ARMW communication of ADD operation is executed simultaneously from two other compute nodes to the same memory area of one compute node.

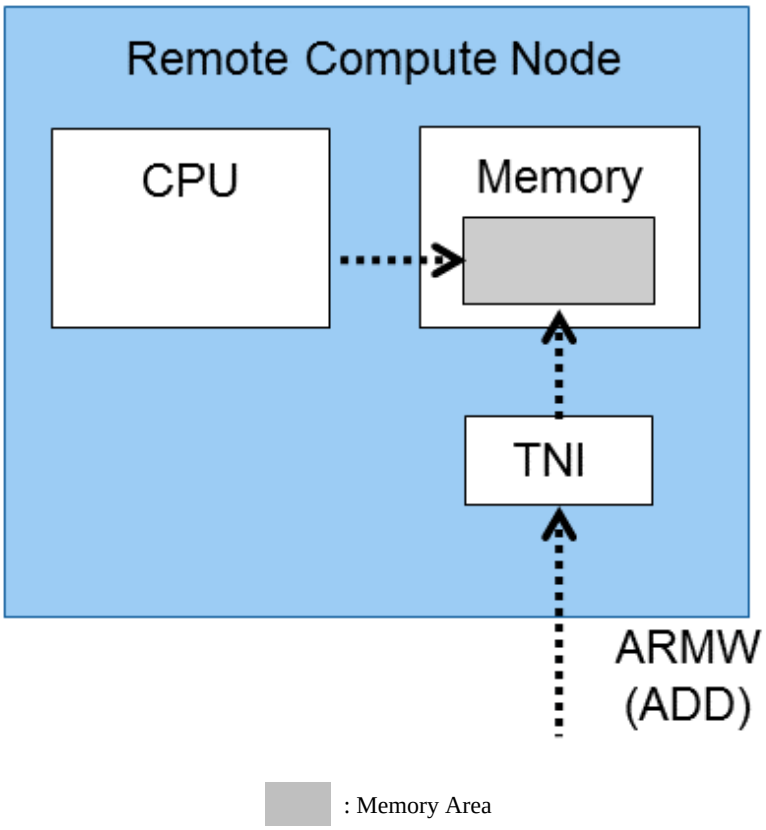
In this case, processing is performed regardless of the combination of the VCQ used. First, processing (reading, calculating, and writing) from one of the compute nodes is performed. Next, processing (reading, calculating, and writing) from the other compute node is performed. Therefore, the final result written to that memory area is the sum of the two operand values and the original value.

Figure 2.7 When ARMW Communication of the ADD Operation is Executed Simultaneously to the Same Memory Area



In the following example, ARMW communication of ADD operation is executed from one compute node to other compute nodes. At the same time, the CPU of that remote compute node atomically executes addition instructions to the same memory area. Therefore, the final result written to that memory area is the sum of the three values.

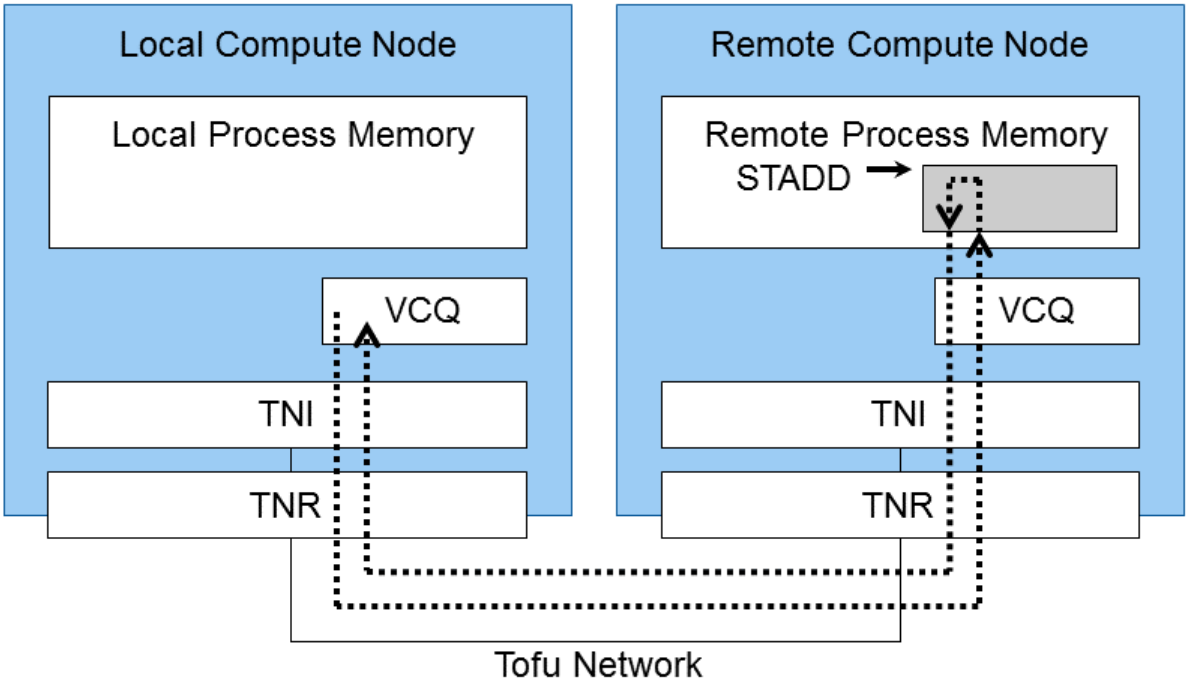
Figure 2.8 When an ARMW Communication of the ADD Operation and an Atomic Add Instruction are Executed Simultaneously on the Same Memory Area



A VCQ handle specifies the TNI used by the local compute node. A remote VCQ ID specifies the TNI used by a remote compute node. A remote STADD specifies the start address of the target memory.

The following figure shows the ARMW communication model executed by the Tofu interconnect. The dotted arrows in the figure represent the data flows.

Figure 2.9 ARMW Communication Model



: Target Region for the Operation

ARMW has the following steps.

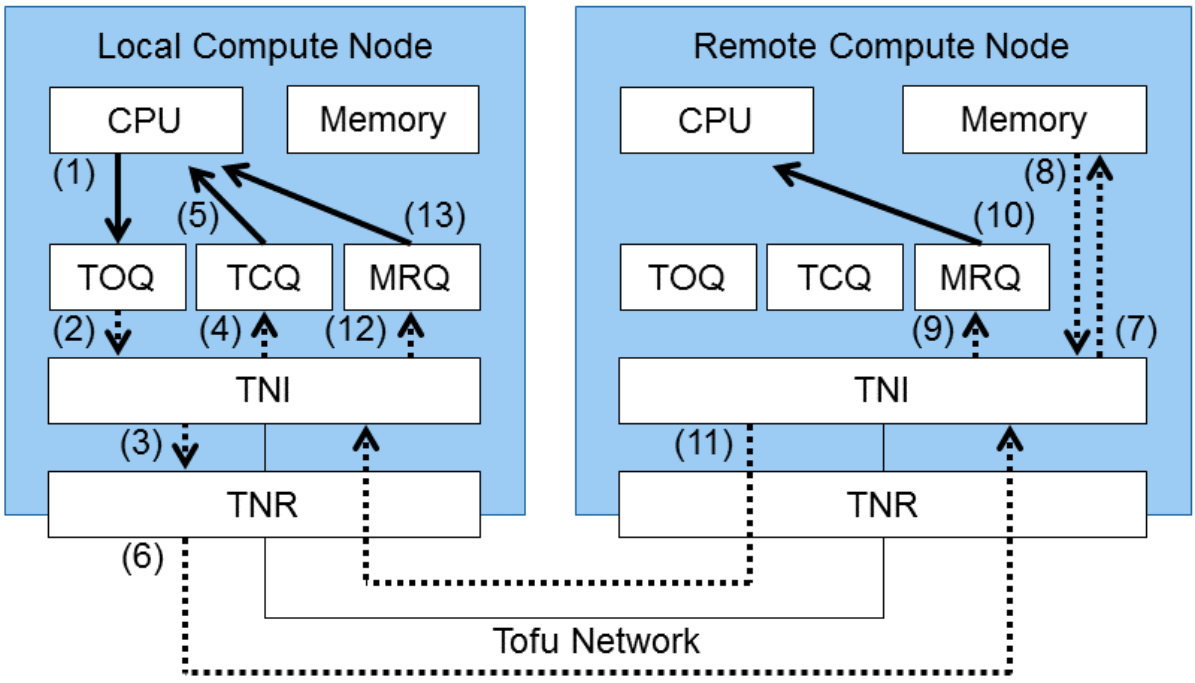
The steps 1, 5, 10 and 13 in this procedure need to be executed by calling the uTofu functions.

1. The local process writes a descriptor to the TOQ to instruct the TNI to start communication.
2. The local TNI reads the descriptor from the TOQ.
3. The local TNI sends an operator and operand to the TNR.
4. The local TNI writes a descriptor to the TCQ.
5. The local process reads the descriptor from the TCQ. By checking this TCQ descriptor, the local process can find out that the communication has started.
6. The TNR and the Tofu network transfer the operator and operand to the remote TNI.
7. The remote TNI and memory controller read the data from memory, execute the operation, and write the post-operation data to memory.
8. The remote TNI obtains the pre-operation data.
9. The remote TNI writes a remote notification descriptor to the MRQ.
10. The remote process reads the descriptor from the MRQ. By checking this remote notification MRQ descriptor, the remote process can find out that the memory region targeted for the operation has been updated.
11. The remote TNI transfers the data to the local TNI via the TNR and Tofu network.
12. The local TNI writes a local notification descriptor to the MRQ.
13. The local process reads the descriptor from the MRQ. By checking this local notification MRQ descriptor, the local process can find out that the target memory region has been updated.

The choice to notify or not notify can be made for each of the TCQ descriptor, remote notification MRQ descriptor, and local notification MRQ descriptor when the local process writes the descriptor to the TOQ.

The following figure shows these steps. The numbers in parentheses in this figure correspond to the above steps. The solid arrows in the figure represent steps executed by calling uTofu functions.

Figure 2.10 ARMW Steps



For details of the example that performs ARMW (Atomic Read Modify Write) communication, see "5.1.3 Example of a Game Using ARMW".

2.2.6 NOP

NOP does not communicate at all. It is usually used to adjust the number of TOQ descriptors in the session mode described in "2.2.8 Session Mode". It can also be used in the free mode described in "2.2.7 Free Mode" but it has no effect other than wasting space in a TOQ.

NOP has the following steps. A corresponding MRQ descriptor is not written.

1. The local process writes a descriptor to the TOQ to instruct the TNI to start processing.
2. The local TNI reads the descriptor from the TOQ.
3. The local TNI writes a descriptor to the TCQ.
4. The local process reads the descriptor from the TCQ.

The choice to notify or not notify of the TCQ descriptor can be made when the local process writes the descriptor to the TOQ.

2.2.7 Free Mode

A CQ has the following two modes. One of them is assigned to each CQ when a uTofu program starts. You can select which mode of CQ to use when you create a VCQ. Both modes cannot be used simultaneously in a VCQ.

- free mode
- session mode

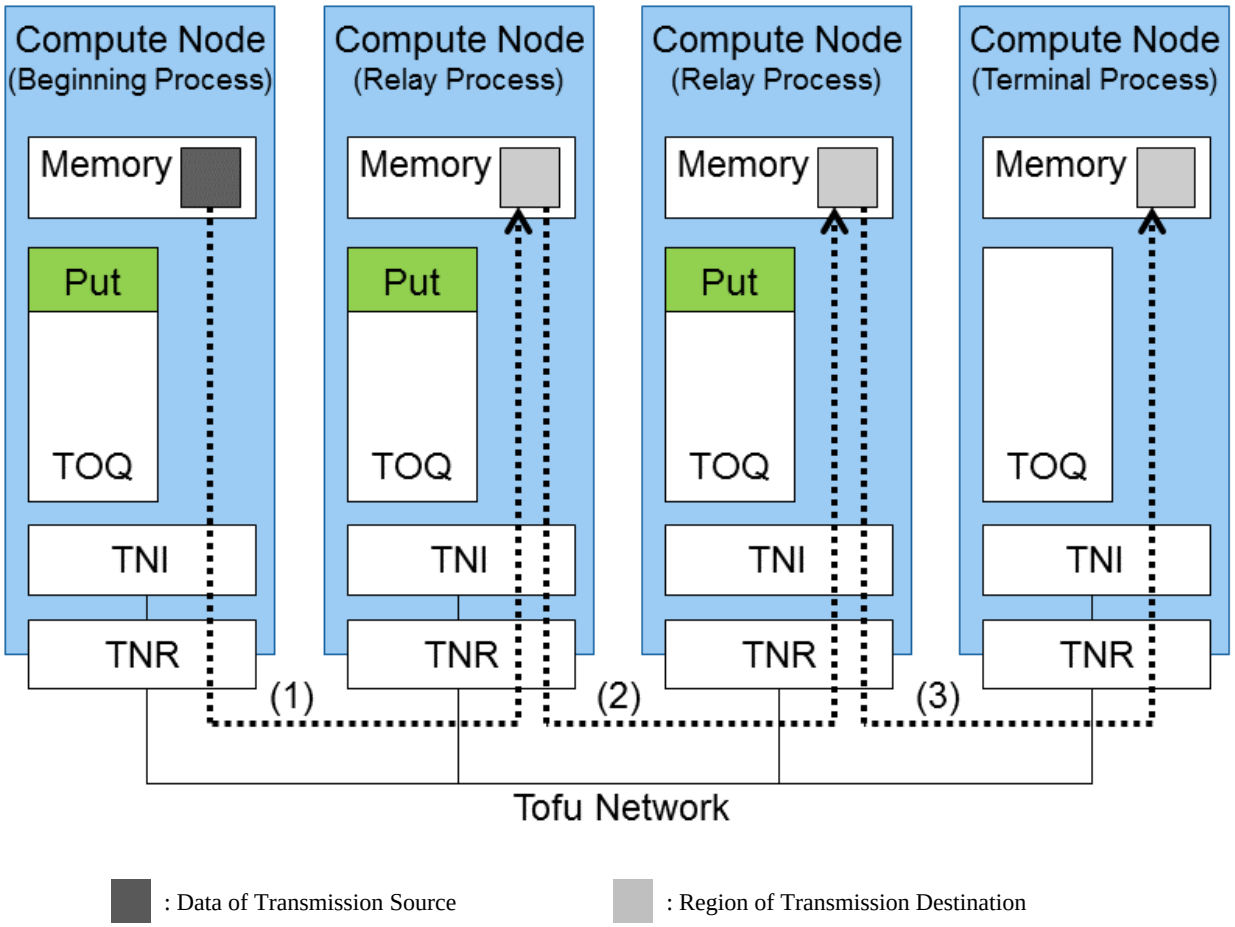
The free mode is a usual mode. Behaviors described from "2.2.3 Put Communication" to "2.2.6 NOP" are those of the free mode. In the free mode, one-sided communication start functions and one-sided communication batch start functions write a TOQ descriptor and a TNI starts the communication immediately. Therefore, when two or more processes want to perform a sequence of communications in cooperation with each other, processes must call a function with proper timing.

A VCQ created on a CQ of the free mode is called a free mode VCQ. A free mode VCQ is created if a mode is not specified when you create a VCQ.

2.2.8 Session Mode

The session mode is a mode which automatically starts instructed communications when a TNI receives another communication. In the session mode, one-sided communication start functions and one-sided communication batch start functions write a TOQ descriptor but a TNI does not start the communication immediately. By receiving another Put communication at the VCQ, the TNI starts the instructed communication automatically. Therefore, when two or more processes want to perform a sequence of communications in cooperation with each other, communications except the first communication can be started without involvement of processes. For example, if a process sends same data to two or more processes, the data can be relayed automatically like the sequence shown in the figure below.

Figure 2.11 Relaying Put Communications by Four Processes



A VCQ created on a CQ of the session mode is called a session mode VCQ. A session mode VCQ is created if the session mode is specified when you create a VCQ.

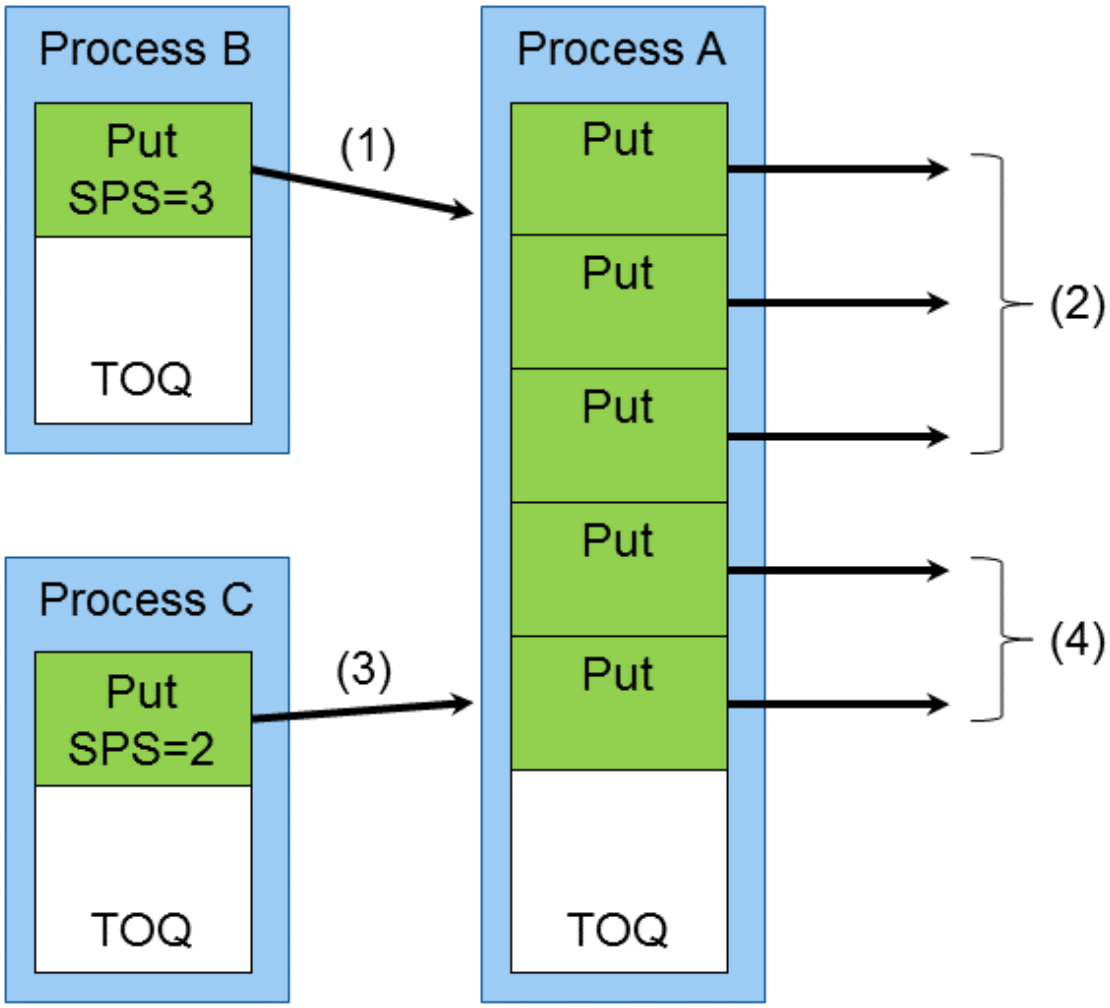
In this subsection, the following terms are used for the purpose of explanation.

Term	Explanation
beginning process	Process which starts a communication using a session mode VCQ as a remote VCQ first in a sequence of communications
relay process	Process which sends and receives automatically using a session mode VCQ
terminal process	Process which receives from a session mode VCQ in a relay process but does not send

2.2.8.1 Behavioral Details of the Session Mode

In the session mode, a local process (communication origin) specifies the number of communications which are started automatically at a TNI of a remote process (communication target). This number is called a session progress step (SPS). For example, the process A writes five descriptors to its session mode VCQ in advance and the process B instructs its VCQ to start a Put communication with SPS 3 to the session mode VCQ of the process A. When the session mode VCQ of the process A receives the Put communication, the first three TOQ descriptors are started automatically. Subsequently the process C instructs its VCQ to start a Put communication with SPS 2 to the session mode VCQ of the process A. When the session mode VCQ of the process A receives the Put communication, the remaining two TOQ descriptors are started automatically. The following figure shows the sequence.

Figure 2.12 Specifying an SPS



At a local process, an SPS can be specified only in a Put descriptor. If an SPS is specified in a Put Piggyback, Get, ARMW, or NOP descriptor, the behavior is undefined. Only Put and NOP descriptors can be started automatically at a remote process. If a Put Piggyback, Get, or ARMW descriptor is written in a session mode VCQ, the behavior is undefined.

A descriptor with an SPS can be used for both a free mode VCQ and a session mode VCQ. A beginning process usually uses a free mode VCQ. A relay process uses a session mode VCQ. A terminal process can use either a free mode VCQ or a session mode VCQ. If you use a session mode VCQ at a terminal process, you must specify 0 as an SPS in a Put descriptor targeted to the VCQ.

At a relay process, it is guaranteed that the Put communication to a next relay process or a terminal process starts after data from a beginning process or a previous relay process is written to memory of the relay process. Therefore, when the value of the local STADD of the Put communication to send is equal to the value of the remote STADD of the received Put communication, it is guaranteed that exactly the received data is sent. However, if the memory region is updated by another communication or a CPU almost at the same time, this behavior is not guaranteed exceptionally.

If a received Put communication has an error because of an incorrect remote STADD value or other reasons, communication instructed in the session mode VCQ is not started by the Put communication.

You can write TOQ descriptors to a session mode VCQ after receiving a Put communication. When a session mode VCQ receives a Put communication, if there are TOQ descriptors of which communications are not started yet and the number is less than the SPS of the Put communication, communications of all existing TOQ descriptors are started. When TOQ descriptors are written by a one-sided communication start function or one-sided communication batch start function call subsequently, communications of TOQ descriptors which were lacking are started. If the number of TOQ descriptors is still lacking, the rest of communications are started by subsequent function calls. The number of lacking TOQ descriptors must be 2000 at most. If the number of lacking TOQ descriptors exceeds 2000, communications may not be performed correctly. For example, communication may not be started even if a one-sided communication start function or one-sided communication batch start function is called.

An SPS can be a value from 0 to 15. If a value 16 or greater is specified, the behavior is undefined.

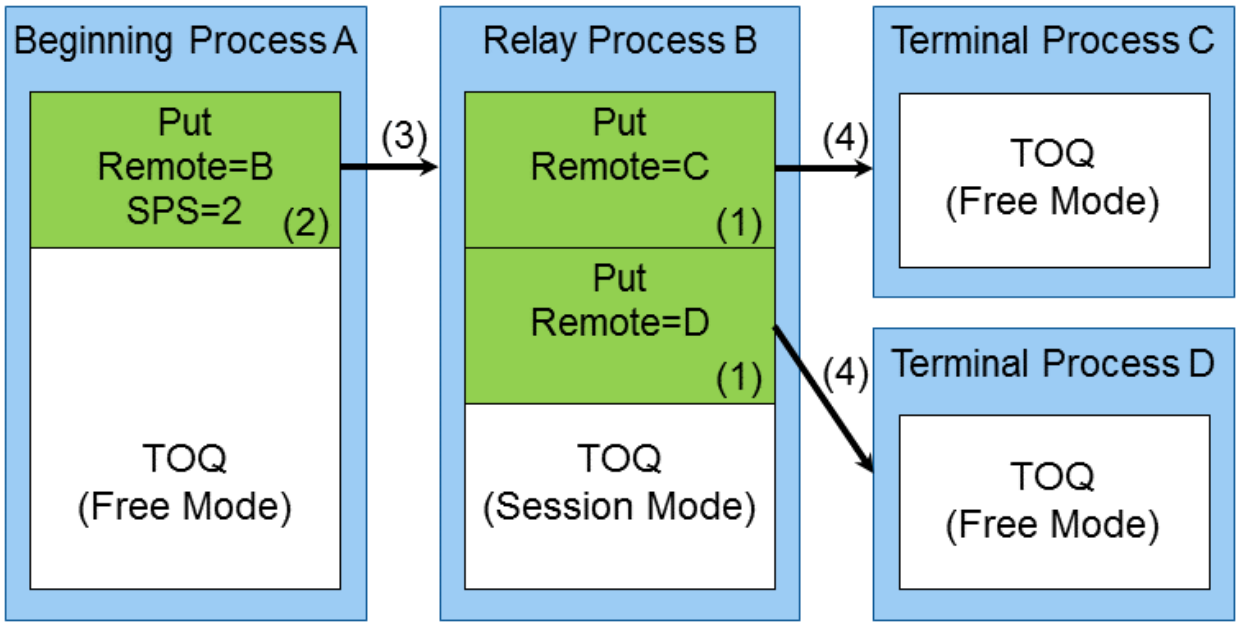
Behaviors other than explained above, for example when TCQ and MRQ descriptors are written, are same as those of the free mode.

2.2.8.2 Example of Communication Forking Using the Session Mode

A relay process can relay Put communication not only one-by-one but with forking.

The following figure shows an example of Put communication forking to two processes at a relay process.

Figure 2.13 Forking at an Relay Process



In this example, Put communication forks in the following steps. The numbers in parentheses in the figure above correspond to the steps below.

1. The relay process B writes two Put descriptors, one to the terminal process C and the other to the terminal process D.
2. The beginning process A writes a Put descriptor to the relay process B with SPS 2.
3. The Put communication of 2. is started.
4. The Put communications of 1. are started automatically when the TNI of the relay process B receives the Put communication of 3.

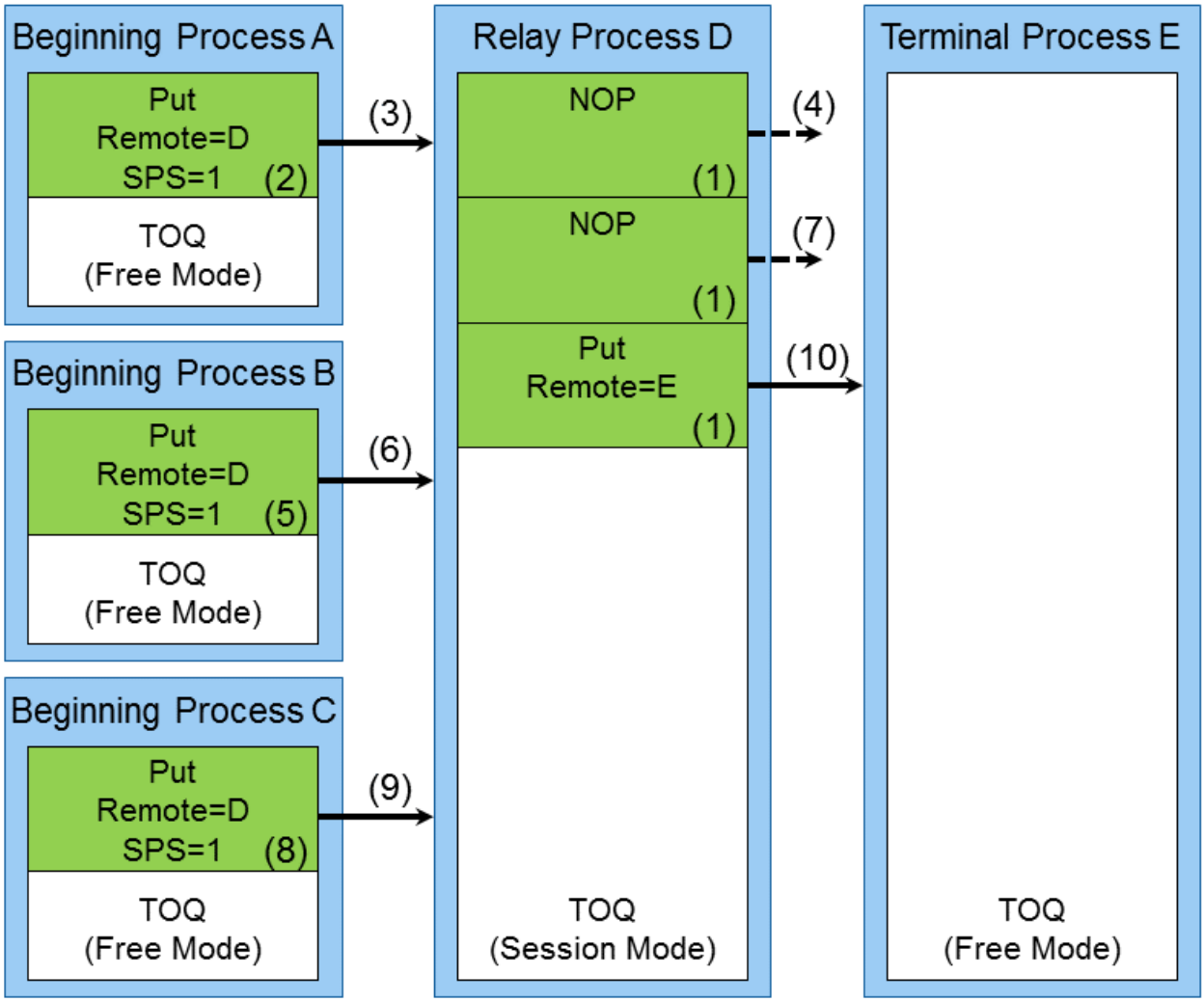
In this example, only the process B is a relay process and Put communication forks only one time. However, Put communication can fork multiple times if the processes C and D are also relay processes.

2.2.8.3 Example of Communication Joining Using the Session Mode

Furthermore, a relay process can relay Put communication with joining.

The following figure shows an example of Put communication joining from three processes at a relay process.

Figure 2.14 Joining at an Relay Process



In this example, Put communications join in the following steps. The numbers in parentheses in the figure above correspond to the steps below.

1. The relay process D writes two NOP descriptors and one Put descriptor to the terminal process E.
2. The beginning process A writes a Put descriptor to the relay process D with SPS 1.
3. The Put communication of 2. is started.
4. The TNI of the relay process D receives the Put communication of 3. but no communication is started.
5. The beginning process B writes a Put descriptor to the relay process D with SPS 1.
6. The Put communication of 5. is started.
7. The TNI of the relay process D receives the Put communication of 6. but no communication is started.
8. The beginning process C writes a Put descriptor to the relay process D with SPS 1.
9. The Put communication of 8. is started.
10. The Put communication of 1. is started automatically when the TNI of the relay process D receives the Put communication of 9.

In this example, the beginning processes A, B, and C write Put descriptors in this order and the TNI of the relay process D receives Put communication in the same order. However, the order is not significant. The Put communication from the relay process D is started once the TNI of the relay process D receives all three Put communications.

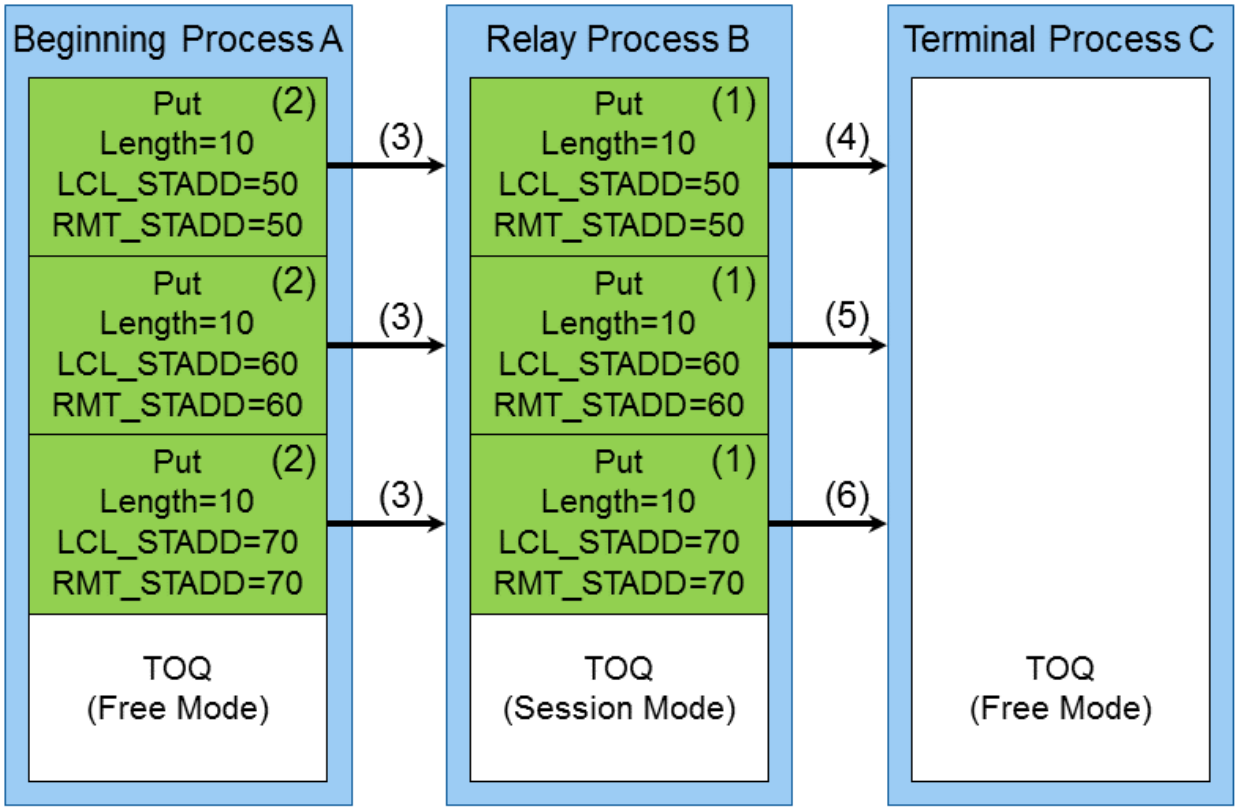
In this example, only the process D is a relay process and Put communications join only one time. However, Put communications can join multiple times if the processes A, B, and C are also relay processes.

2.2.8.4 Example of Communication Pipeline Using the Session Mode

When relaying data using one Put communication, the Put communication to the next relay process or the terminal process is started at a relay process after receiving data completely from the beginning process or the previous relay process. If the size of data is large, relaying multiple split data using multiple Put communications may shorten the overall time to deliver data to the terminal process. This type of relaying style is called a pipeline.

The following figure shows an example of pipelined Put communications when data of 30 bytes is split into thirds.

Figure 2.15 Pipelined Put Communication



In this example, Put communications are pipelined in the following steps. The numbers in parentheses in the figure above correspond to the steps below.

1. The relay process B writes three Put descriptors to the terminal process C. The first Put communication transfers first 10 bytes. The second Put communication transfers next 10 bytes. The third Put communication transfers last 10 bytes.
2. The beginning process A writes three Put descriptors to the relay process B with SPS 1 each. Each Put communication transfers data split in the same way as 1.
3. The Put communications of 2. are started in sequence.
4. The first Put communication of 1. is started automatically when the TNI of the relay process B receives the first Put communication of 2.
5. The second Put communication of 1. is started automatically when the TNI of the relay process B receives the second Put communication of 2.
6. The third Put communication of 1. is started automatically when the TNI of the relay process B receives the third Put communication of 2.

As explained in "2.2.10 Confirmation of Communication Completion and Guarantee of Ordering", local VCQs and communication path coordinates of all Put descriptors at the beginning process A should be same if pipelined Put communications are used. Otherwise, the order of receiving Put communications at the relay process B may be different from the order of sending Put communications at the beginning process A. As the result, the relay process B may send data to the terminal process C which has not been received completely from the beginning process A.

In this example, only the process B is a relay process and Put communications are pipelined in only two stages. However, Put communications can be pipelined in more stages if the process C is also a relay process.

2.2.9 Maximum Transfer Size and Transmission Gap of Packets

The Tofu interconnect splits data into units called packets and then transfers them. The maximum transfer size of a packet is called MTU (maximum transfer unit). When sending data has a size that is larger than the MTU, the Tofu interconnect splits the data into packets of the MTU and transfers the packets.

When there is contention for a communication path, the effective bandwidth of the path may drop due to congestion. If you know beforehand that such congestion will occur between multiple communications, you can mitigate that drop by suppressing the packet injection rate. For Put and Get, the TNI can suppress the injection rate with an interval between packets so as not to immediately send the next packet after sending a packet. This interval is called transmission gap.

2.2.10 Confirmation of Communication Completion and Guarantee of Ordering

In Put communication, the TNI writes data to the main memory of a remote compute node. In Get communication, the TNI writes data to the main memory of the local compute node. In these cases, if the write destination area crosses multiple CPU cache lines, the write sequence to the main memory between these CPU cache lines is not guaranteed, which may change the order. If the read source area crosses multiple pages, the write sequence to the main memory between these CPU cache lines is also not guaranteed even if the write destination area does not cross multiple CPU cache lines, which may change the order.

Consequently, in one communication operation, even if it can be confirmed that an area has been updated with a load instruction, etc. of the CPU, other areas must not be judged as also having been updated.

However, when a CPU cache line in the write destination area is focused, writing is guaranteed in the unit of granularity of CPU cache line if the read source area corresponding to the CPU cache line does not cross multiple pages.

Therefore, within the range of a single CPU cache line in the write destination area, if an area has been confirmed as updated with a load instruction, etc. of the CPU, other areas can be judged as also having been updated only when the read source area corresponding to the CPU cache line does not cross multiple pages. Thus, when writing is confirmed with a load instruction, etc. of the CPU for each area delimited by the boundary of a CPU cache line, that may confirm the completion of all writing. In ARMW communication, no area crosses multiple CPU cache lines.

It is strongly recommended to write with a load instruction, etc. of the CPU for each area delimited by the boundary of a CPU cache line. Accordingly, the TCQ and MRQ may confirm the completion of data reading or writing in one communication operation as follows.

- TCQ notification may confirm the completion of all data reading from the main memory of the local compute node in Put communication.
- Remote MRQ notification may confirm the completion of all data writing to the main memory of the remote compute node in Put communication.
- Remote MRQ notification may confirm the completion of all data reading from the main memory of the remote compute node in Get communication.
- Local MRQ notification may confirm the completion of all data writing to the main memory of the local compute node in Get communication.
- Remote MRQ notification may confirm the completion of all data writing to the main memory of the remote compute node in ARMW communication.

The following orderings are guaranteed, regardless of the combination of Put, Get, and ARMW, when multiple communication instructions are issued to the TOQ in the same local VCQ. Refer to "6.1.3 Behavioral Specifications of This Computing System" for behavioral specifications when the strong order flag is set.

- The order of reading and writing the local and remote main memory is not guaranteed by default. The strong order function exists to guarantee the order to some extent. The following two events are guaranteed when the strong order flag is set in a certain TOQ descriptor, but only if the communication path coordinates and remote VCQs are the same across the communication instructions. The first guaranteed event is the start of reading in communication after the reading in the preceding communication is completed. The second guaranteed event is the start of writing in communication after the writing in the preceding communication is completed.
- TCQ notifications are guaranteed to occur in the same order as that of communication instructions by the TOQ descriptor.

- Remote MRQ notifications are guaranteed to occur in the same order as that of communication instructions by the TOQ descriptor only if the communication path coordinates and remote VCQs are the same across the communication instructions.
- Local MRQ notifications are guaranteed to occur in the same order as that of communication instructions by the TOQ descriptor only if the communication path coordinates and remote VCQs are the same across the communication instructions.

This characteristic of the TCQ notification can be used to learn which TCQ notifications correspond to which communication instructions by TOQ descriptors. Also, the functions that start communication by writing a TOQ descriptor have a parameter specifying callback data. The function that confirms communication completion by reading a TCQ descriptor has a parameter for returning the callback data specified when the corresponding TOQ descriptor is written. This callback data can also be used to learn the correspondence between TOQ descriptors and TCQ descriptors.

Local MRQ notifications do not make it easy to learn which communication instructions by TOQ descriptors correspond to the notifications. However, the functions that start communication by writing a TOQ descriptor have a parameter specifying a value called EDATA. The function that confirms communication completion by reading an MRQ descriptor has a parameter for returning a structure object that includes the EDATA value specified when the corresponding TOQ descriptor is written. This EDATA can also be used to learn the correspondence between TOQ descriptors and MRQ descriptors.

This structure object is also returned by remote MRQ notifications. Therefore, the EDATA can be used to also learn which communication has completed in a remote process.

2.2.11 Cache Injection and Padding

The function of cache injection is to decrease the latency that the CPU reads data from the main memory after the TNI writes data to the main memory. In Put communication, the write destination is a remote compute node. In Get communication, it is the local compute node. When the cache injection flag is set in the TOQ descriptor, the function writes data to not only the main memory but also the last level cache of the CPU. This can prevent a cache miss in the last level cache when the CPU accesses the written data.

However, for cache injection to work, conditions have to be met: the write operation must overwrite a cache line entirely, and the cache line must be clean. A typical case of a clean cache line in the last level cache is a case where the CPU is polling the relevant cache line. If these conditions are not met, data is written only to the main memory, and a cache miss occurs during the subsequent data read by the CPU.

Put using piggyback has a smaller transfer size than the cache line size, so it cannot overwrite a cache line entirely. Therefore, just setting the cache injection flag in the TOQ descriptor does not enable Put to write data in the last level cache of the CPU.

Put using piggyback has the cache line padding function to suppress a cache miss. When the cache line padding flag is set in the TOQ descriptor, Put adds indefinite values to the beginning and end of data to extend up to the cache line boundaries before writing the data. Thus, with both the cache injection flag and the cache line padding flag specified at the same time, Put can write data in the last level cache of the CPU. The cache line padding flag is available only for Put using piggyback.

2.2.12 Communication Error

In one-sided communication, successful packet transfer from the source TNI to the destination TNI is guaranteed when communication is enabled and no failure has occurred in the local TNI, remote TNI, TNRs on the communication path, and Tofu network.

However, a communication error may occur due to the wrong specified STADD value or other reasons. The TCQ reports any error that occurs when the local TNI sends a packet, and the MRQ reports any error that occurs after this time. You can check error details from the return value of the one-sided communication completion confirmation functions.

2.3 Barrier Communication

In barrier communication, the TNI sends and receives packets and signals sequentially between multiple compute nodes to achieve barrier synchronization between these nodes. The reduction operation can also be executed along with barrier synchronization.

2.3.1 VBG (Virtual Barrier Gate)

Each TNI has circuits called BGs (barrier gates). Each BG sends and receives packets and signals. To use barrier communication, the BGs must be configured on all the compute nodes participating in barrier synchronization before the barrier synchronization starts. The network of the communication paths configured between the BGs prior to the barrier synchronization is called a barrier circuit.

A barrier circuit is built with the multiple BGs used per TNI. A series of circular dependencies are set among the multiple BGs of each TNI. Of these BGs, one is the start/end BG, and the others are relay BGs. Packet transfer through a series of BGs begins at a start/end BG and

ends at the BG immediately before a start/end BG. To control the transfer sequence of packets, each BG sends a signal to the next BG at the same time as it sends a packet.

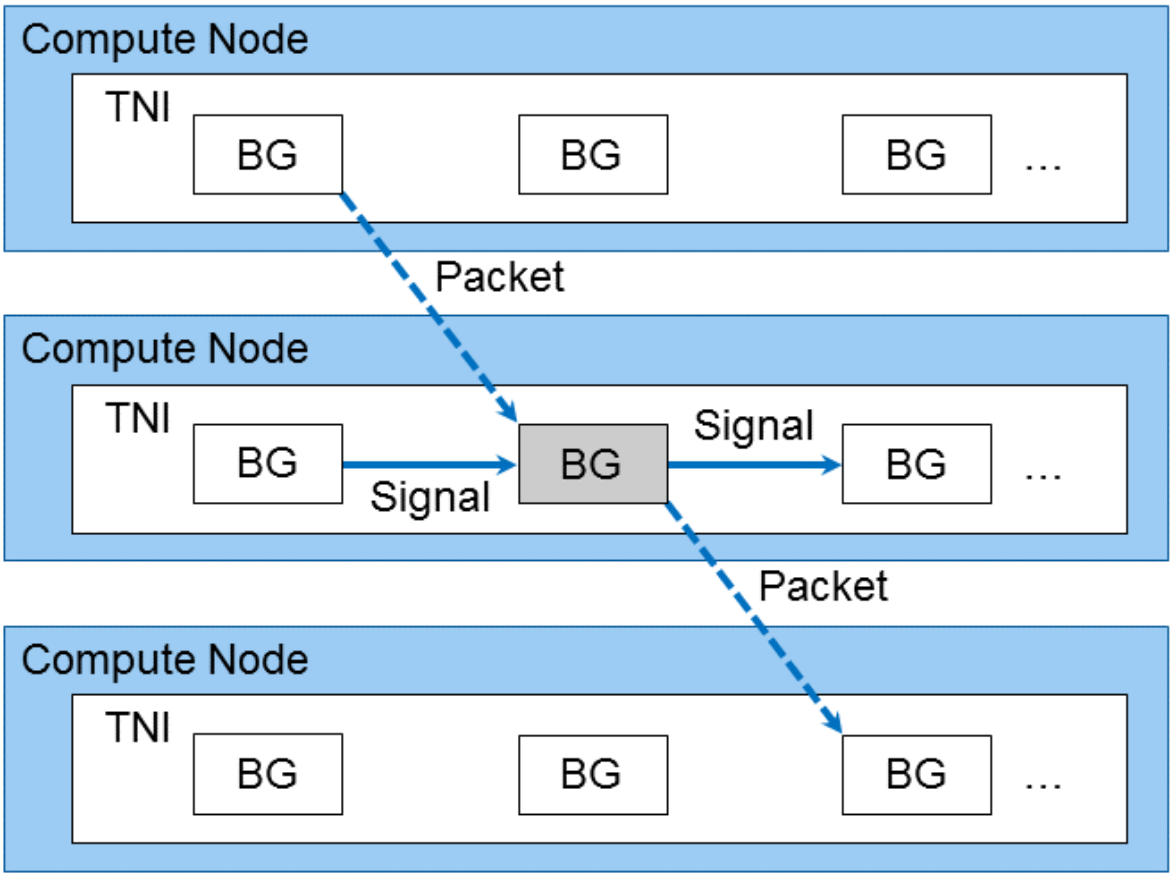
When barrier communication starts, the start/end BG sends a packet to a preconfigured BG in an arbitrary TNI and a signal to another preconfigured BG in the same TNI. After that, it waits for a packet from a preconfigured BG in an arbitrary TNI and a signal from another BG in the same TNI.

A relay BG waits for a packet from a preconfigured BG in an arbitrary TNI and a signal from another preconfigured BG in the same TNI. Upon receiving both of them, the relay BG sends the packet to a preconfigured BG in an arbitrary TNI and the signal to another preconfigured BG in the same TNI.

A packet source BG and a packet destination BG can belong to TNIs on the same compute node or TNIs on different compute nodes.

The following figure shows the sending of packets and signals. The figure focuses on one BG at the center, so only the packets and signals with this BG as their source or destination have been drawn.

Figure 2.16 BG, Packet, and Signal



uTofu abstracts a BG as a VBG (virtual barrier gate). Each VBG is assigned a VBG ID. The VBG ID is a 64-bit unsigned integer identifying the VBG, and it is a unique value among all the VBGs within the Tofu network.

2.3.2 Barrier Circuit

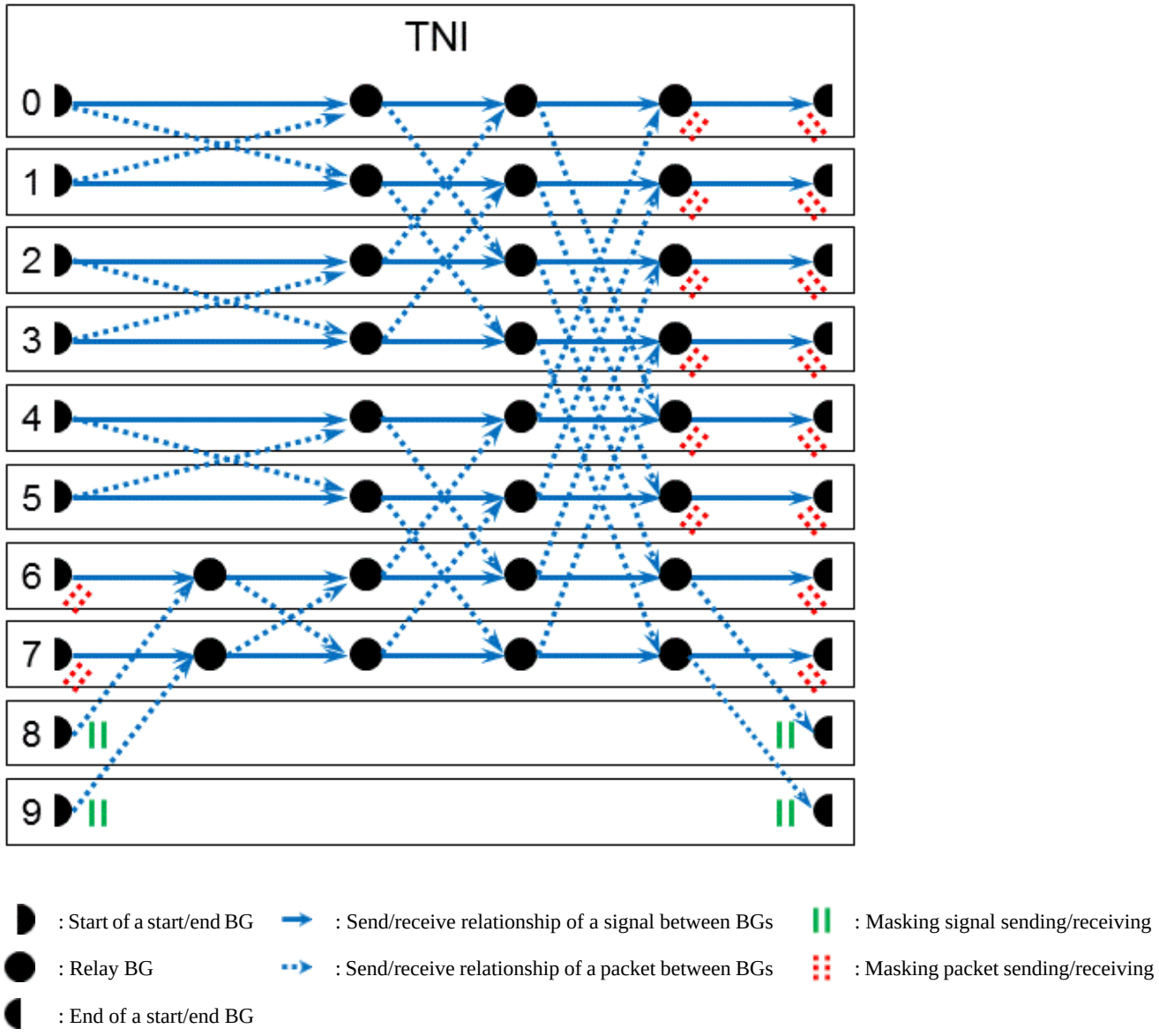
By setting the following information for each BG participating in barrier synchronization, you can build a barrier circuit. The function explained in "3.4.1 VBG Allocation/Freeing Functions" can obtain a VBG ID. For the example of setting a VBG ID, see "5.2.1 Example of Barrier Synchronization and Reduction Operation".

- VBG ID of the signal source (another VBG in the same TNI)
- VBG ID of the packet source (arbitrary VBG)
- VBG ID of the signal destination (another VBG in the same TNI)

- VBG ID of the packet destination (arbitrary VBG)

The following figure shows an example of building a barrier circuit using a 10-process butterfly exchange algorithm.

Figure 2.17 Barrier Circuit Using a 10-Process Butterfly Exchange Algorithm



The VBG obtained by the function explained in "3.4.1 VBG Allocation/Freeing Functions" is set in the function explained in "3.4.2 VBG Configuration Functions".

The settings for all the VBGs participating in barrier synchronization must be completed before barrier communication begins. For example, take the following typical approach to use barrier communication in the Tofu interconnect for barrier synchronization of MPI. First, configure the VBGs at the time when the relevant communicator is created. Then, confirm the completion of all the VBG settings by using one-sided communication or other means.

2.3.3 Barrier Synchronization

The uTofu functions for starting barrier communication will return at the time when they instruct the TNI to start barrier communication. To confirm barrier synchronization, it is necessary to call a uTofu function that confirms the completion of barrier communication and to continue polling until the function returns a return value indicating the completion.

Multiple barrier communications can be performed simultaneously when different VBGs are used.

For details of the example that performs barrier communication, see "5.2.1 Example of Barrier Synchronization and Reduction Operation".

2.3.4 Reduction Operation

When input data is given at the barrier communication start time, the reduction operation can be executed along with the barrier communication.

The following table shows the available operators for uTofu and their data types. However, the operators actually available depend on the type of Tofu interconnect.

For details on information specific to this computing system, see "Chapter 6 System Information".

Table 2.3 Reduction Operators

Name	Data Type	Operation
BAND	64-bit unsigned integer	Bitwise AND
BOR	64-bit unsigned integer	Bitwise OR
BXOR	64-bit unsigned integer	Bitwise XOR
MAX	64-bit unsigned integer	Maximum value
MAXLOC	64-bit unsigned integer	(*1)
SUM	64-bit unsigned integer	Addition
BFPSUM	Double-precision floating point	Addition

*1) For MAXLOC, each process gives a group of two 64-bit unsigned integers as input. For the first element values, it selects the maximum value like MAX. For the second element values, it selects the element that belongs to the group containing the first element, which has the maximum value. At this point, if the first element values are the same, it compares the second element values with each other and selects the smaller value.

For example, for SUM, each process gives a 64-bit unsigned integer, and their sum is calculated.

The number of possible elements of reduction operation executed in one-time barrier communication depends on the type of Tofu interconnect. If reduction operation with multiple elements can be executed, data is given in the form of an array, and the operation is executed between elements at the same location. For MAXLOC with four or more elements, the elements are grouped two pairs by two pairs, and the above operation is executed between an odd-numbered element and even-numbered element.

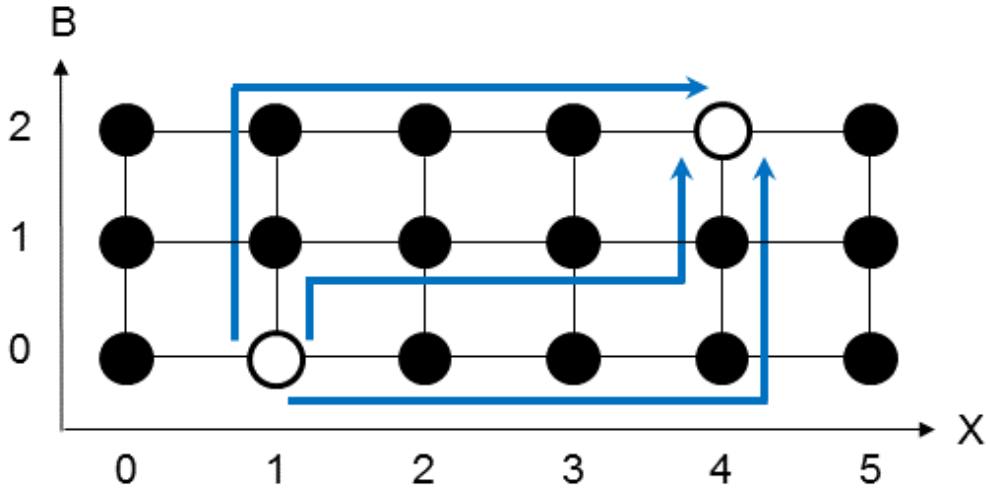
For details of the example that performs reduction operation, see "5.2.1 Example of Barrier Synchronization and Reduction Operation".

2.4 Communication Path

Since a Tofu network has a six-dimensional mesh/torus topology, there are multiple paths for a packet to move from a compute node to another compute node. The communication path is determined with a combination of communication path coordinates and destination compute node coordinates. A,B,C coordinates specify communication path coordinates, and X,Y,Z,A,B,C coordinates specify destination compute node coordinates. A packet moves to the A,B,C coordinates specified as the communication path, before moving to the X,Y,Z coordinates and then the A,B,C coordinates of the destination compute node. For example, suppose that the source compute node is at the coordinates (X_S, Y_S, Z_S, A_S, B_S, C_S), the destination compute node is at the coordinates (X_D, Y_D, Z_D, A_D, B_D, C_D), and the communication path is represented by the coordinates (A_P, B_P, C_P). A packet first moves to the coordinates (X_S, Y_S, Z_S, A_P, B_P, C_P) along the shortest path, and then moves to the coordinates (X_D, Y_D, Z_D, A_P, B_P, C_P) along the shortest path. Finally, it moves to the coordinates (X_D, Y_D, Z_D, A_D, B_D, C_D) along the shortest path. Since the A, B, and C axes have lengths of 2, 3, and 2, respectively, there is a maximum of 12 communication paths.

The following figure shows an example of communication paths on a two-dimensional surface that includes the X and B axes. This example shows three communication paths in communication from a compute node at X,B coordinates (1,0) to a compute node at X,B coordinates (4,2), where the B coordinate of the paths is (0), (1), and (2).

Figure 2.18 Three Communication Paths on an X-B Two-Dimensional Surface



In a single communication for one-sided communication, packets move back and forth between the local compute node and a remote compute node. Communication path coordinates specify a communication path from the local compute node to the remote compute node. This communication path is also used in reverse as the communication path from the remote compute node to the local compute node.

2.5 Thread Safety

The uTofu interface guarantees its correct behavior when multiple threads call it simultaneously without specifications regarding thread safety, except in the following cases:

- When any of the functions that are classified as follows is called simultaneously for one or more thread-unsafe VCQs that share a CQ
  - STADD management
  - One-sided communication execution
- When any of the functions that are classified as follows is called simultaneously for one thread-unsafe VBG
  - Barrier communication execution

You can specify a thread-safe VCQ or a VCQ that does not share the CQ (CQ-exclusive VCQ) for simultaneous calls from multiple threads. This also includes the calls of the above-described functions. To do so, instruct that the thread-safe VCQ or CQ-exclusive VCQ be created at the time when a VCQ is created. You can also instruct that a thread-safe VBG be created at the time when a VBG is created. However, a thread-safe VCQ or thread-safe VBG may increase the software overhead for function calls.

Chapter 3 uTofu Interface Specifications

This chapter describes the interface specifications of uTofu.

uTofu provides interfaces for C (C99 (ISO/IEC 9899:1999 standard) or later). A program that uses uTofu should include the header file `utofu.h` as follows:

```
#include <utofu.h>
```

The features of uTofu are categorized as follows. For details on corresponding functions to the features of uTofu, see references shown in the following table.

Table 3.1 Features and Functions of uTofu

Category for Feature	Feature	Details on Corresponding Functions
TNI query	TNI query	"3.2.1 TNI Query Functions"
VCQ management	VCQ creation/freeing	"3.3.1 VCQ Creation/Freeing Functions"
	VCQ ID manipulation	"3.3.2 VCQ ID Manipulation Functions"
	VCQ query	"3.3.3 VCQ Query Functions"
VBG management	VBG allocation/freeing	"3.4.1 VBG Allocation/Freeing Functions"
	VBG configuration	"3.4.2 VBG Configuration Functions"
	VBG query	"3.4.3 VBG Query Functions"
Communication path management	Communication path management	"3.5.1 Communication Path Management Functions"
STADD management	STADD management	"3.6.1 STADD Management Functions"
One-sided communication execution	One-sided communication start	"3.7.1 One-Sided Communication Start Functions"
	One-sided communication preparation	"3.7.2 One-Sided Communication Preparation Functions"
	One-sided communication batch start	"3.7.3 One-Sided Communication Batch Start Functions"
	One-sided communication completion confirmation	"3.7.4 One-Sided Communication Completion Confirmation Functions"
	One-sided communication query	"3.7.5 One-Sided Communication Query Functions"
Barrier communication execution	Barrier communication start	"3.8.1 Barrier Communication Start Functions"
	Barrier communication completion confirmation	"3.8.2 Barrier Communication Completion Confirmation Functions"
Supplemental features	Version query	"3.9.1 Version Query Functions"
	Compute node information query	"3.9.2 Compute Node Information Query Functions"

The sections below describe the interface specifications of these features.

The following table shows the notations used in explanations in this chapter.

Table 3.2 Notations Used in Explanations of uTofu Interface Specifications

Notation	Meaning
`a_*`	Any functions, enumeration constants, or macros whose identifiers start with `a_`
`a_{b|c}`	Function, enumeration constant, or macro whose identifier is `a_b` or `a_c`
`a::b`	Member `b` in structure or union `a`, or parameter `b` of function `a`

Table 3.3 Notations Used at IN/OUT Column in Tables of Function Parameters

Notation	Meaning
IN	Input to the function
OUT	Output from the function
IN,OUT	Input to the function and output from the function

3.1 Common Definitions

3.1.1 Type Definitions

3.1.1.1 `typedef`

Type definitions for IDs, etc.

Explanation

uTofu uses specially defined types for variables of IDs, etc.

Type Definitions

Typedef Name	Type	Explanation
`utofu_tni_id_t`	`uint16_t`	TNI (Tofu network interface) ID. A TNI ID is a 16-bit unsigned integer that identifies a TNI on a compute node, which is equipped with a number of TNIs. The ID ranges from 0 to "that number of TNIs `-1`". Each TNI ID on the compute node is unique.
`utofu_cq_id_t`	`uint16_t`	CQ (control queue) ID. A CQ ID is a 16-bit unsigned integer that identifies a CQ on a TNI, which is equipped with a number of CQs. The ID ranges from 0 to "that number of CQs `-1`". Each CQ ID on the TNI is unique.
`utofu_bg_id_t`	`uint16_t`	BG (barrier gate) ID. A BG ID is a 16-bit unsigned integer that identifies a BG on a TNI, which is equipped with a number of BGs. The ID ranges from 0 to "that number of BGs `-1`". Each BG ID on the TNI is unique.
`utofu_cmp_id_t`	`uint16_t`	Component ID. A component ID is a 16-bit unsigned integer that identifies a VCQ among the VCQs that share 1 CQ. Each component ID in the CQ is unique.
`utofu_vcq_hdl_t`	`uintptr_t`	VCQ (virtual control queue) handle. A VCQ handle is a pointer-sized unsigned integer that references VCQ data in a process. Each VCQ handle in the process is unique.
`utofu_vcq_id_t`	`uint64_t`	VCQ (virtual control queue) ID. A VCQ ID is a 64-bit unsigned integer that identifies a VCQ in a Tofu network. Each VCQ ID is unique among all the VCQs in the Tofu network. Default communication path coordinates can be embedded in a VCQ ID.
`utofu_vbg_id_t`	`uint64_t`	VBG (virtual barrier gate) ID. A VBG ID is a 64-bit unsigned integer that identifies a VBG in a Tofu network. Each VBG ID is unique among all the VBGs in the Tofu network.
`utofu_path_id_t`	`uint8_t`	Communication path ID. A communication path ID is an 8-bit unsigned integer that represents a communication path from a compute node to another compute node in a Tofu network. Though a

Typedef Name	Type	Explanation
		communication path is also used in barrier communication, a communication path ID is used only in one-sided communication.
utofu_stadd_t	uint64_t	STADD (steering address). A STADD is a 64-bit unsigned integer that represents a memory address that a TNI can understand. It is used in one-sided communication.

3.1.2 Return Values

3.1.2.1 enum utofu_return_code

Return values of uTofu functions

Explanation

Enumeration constants shown in the table below are used for the return values of almost all uTofu functions.

The enumeration constant UTOFU_SUCCESS means a successful operation and has the value 0. All other enumeration constants have negative values. Enumeration constants that are not UTOFU_SUCCESS do not necessarily represent errors. For example, the utofu_poll_tcq() and utofu_poll_mrq() functions return UTOFU_ERR_NOT_FOUND when no new TCQ/MRQ entry is found but this is normal.

In the list of return values in the explanation of each function in this document, only the return codes before UTOFU_ERR_INVALID_ARG in the table below are individually explained. Other return codes may also be returned depending on the arguments passed to the function or the conditions at the time.

Enumeration Constants

Enumeration Constant	Explanation
UTOFU_SUCCESS	The operation succeeded with no problems.
UTOFU_ERR_NOT_FOUND	No new entry was found.
UTOFU_ERR_NOT_COMPLETED	The operation has not completed yet.
UTOFU_ERR_NOT_PROCESSED	The operation has not been processed.
UTOFU_ERR_BUSY	The resource is busy now. Trying again later may succeed.
UTOFU_ERR_USED	The resource is already used. Trying again with another resource may succeed.
UTOFU_ERR_FULL	The resource is full. No more of the resource can be allocated.
UTOFU_ERR_NOT_AVAILABLE	The resource is not available. The resource cannot be allocated.
UTOFU_ERR_NOT_SUPPORTED	The operation is not supported.
UTOFU_ERR_TCQ_OTHER	An error that cannot be represented by other UTOFU_ERR_TCQ_* was reported by a TCQ descriptor.
UTOFU_ERR_TCQ_DESC	A TOQ descriptor error was reported by a TCQ descriptor. The wrong argument may have been passed.
UTOFU_ERR_TCQ_MEMORY	A memory access error was reported by a TCQ descriptor. The specified STADD may be invalid.
UTOFU_ERR_TCQ_STADD	A STADD error was reported by a TCQ descriptor.

Enumeration Constant	Explanation
	The STADD value may be outside the registered memory region.
UTOFU_ERR_TCQ_LENGTH	An error regarding "STADD + data length" was reported by a TCQ descriptor. The value of "the specified STADD + the specified data length" may be outside the registered memory region.
UTOFU_ERR_MRQ_OTHER	An error that cannot be represented by other UTOFU_ERR_MRQ_* was reported by an MRQ descriptor.
UTOFU_ERR_MRQ_PEER	A peer process error was reported by an MRQ descriptor. This error may have been caused by an abort of the peer process.
UTOFU_ERR_MRQ_LCL_MEMORY	A local memory access error was reported by an MRQ descriptor. The specified local STADD may be invalid.
UTOFU_ERR_MRQ_RMT_MEMORY	A remote memory access error was reported by an MRQ descriptor. The specified remote STADD may be invalid.
UTOFU_ERR_MRQ_LCL_STADD	A local STADD error was reported by an MRQ descriptor. The local STADD value may be outside the registered memory region.
UTOFU_ERR_MRQ_RMT_STADD	A remote STADD error was reported by an MRQ descriptor. The remote STADD value may be outside the registered memory region.
UTOFU_ERR_MRQ_LCL_LENGTH	An error regarding "local STADD + data length" was reported by an MRQ descriptor. The value of "the specified local STADD + the specified data length" may be outside the registered memory region.
UTOFU_ERR_MRQ_RMT_LENGTH	An error regarding "remote STADD + data length" was reported by an MRQ descriptor. The value of "the specified remote STADD + the specified data length" may be outside the registered memory region.
UTOFU_ERR_BARRIER_OTHER	A barrier communication error that cannot be represented by other UTOFU_ERR_BARRIER_* occurred.
UTOFU_ERR_BARRIER_MISMATCH	Mismatch of reduction operations is detected in barrier communication.
UTOFU_ERR_INVALID_ARG	An invalid argument was passed.
UTOFU_ERR_INVALID_POINTER	An invalid pointer was passed as an argument.
UTOFU_ERR_INVALID_FLAGS	Invalid flags were passed as an argument.
UTOFU_ERR_INVALID_COORDS	Invalid compute node coordinates were passed as an argument.
UTOFU_ERR_INVALID_PATH	Invalid communication path coordinates were passed as an argument.
UTOFU_ERR_INVALID_TNI_ID	An invalid TNI ID was passed as an argument.
UTOFU_ERR_INVALID_CQ_ID	An invalid CQ ID was passed as an argument.
UTOFU_ERR_INVALID_BG_ID	An invalid BG ID was passed as an argument.
UTOFU_ERR_INVALID_CMP_ID	An invalid component ID was passed as an argument.
UTOFU_ERR_INVALID_VCQ_HDL	An invalid VCQ handle was passed as an argument.
UTOFU_ERR_INVALID_VCQ_ID	An invalid VCQ ID was passed as an argument.
UTOFU_ERR_INVALID_VBG_ID	An invalid VBG ID was passed as an argument.
UTOFU_ERR_INVALID_PATH_ID	An invalid communication path ID was passed as an argument.
UTOFU_ERR_INVALID_STADD	An invalid STADD was passed as an argument.
UTOFU_ERR_INVALID_ADDRESS	An invalid memory address was passed as an argument.

Enumeration Constant	Explanation
UTOFU_ERR_INVALID_SIZE	An invalid size/length was passed as an argument.
UTOFU_ERR_INVALID_STAG	An invalid STag was passed as an argument.
UTOFU_ERR_INVALID_EDATA	An invalid EDATA was passed as an argument.
UTOFU_ERR_INVALID_NUMBER	An invalid entity number was passed as an argument.
UTOFU_ERR_INVALID_OP	An invalid operation was passed as an argument.
UTOFU_ERR_INVALID_DESC	An invalid descriptor was passed as an argument.
UTOFU_ERR_INVALID_DATA	Invalid structure data was passed as an argument.
UTOFU_ERR_OUT_OF_RESOURCE	A resource (except memory) is exhausted.
UTOFU_ERR_OUT_OF_MEMORY	Memory cannot be allocated.
UTOFU_ERR_FATAL	A fatal error occurred.

3.2 TNI Query

A TNI (Tofu network interface) is a physical network device supplied with each compute node that performs one-sided communication and barrier communication. More than one TNI may be supplied with each compute node.

The `utofu_get_onesided_tnis()` and `utofu_get_barrier_tnis()` functions can query available TNIs.

The `utofu_query_onesided_caps()` and `utofu_query_barrier_caps()` functions can query the capabilities of one-sided communication and barrier communication, respectively.

3.2.1 TNI Query Functions

3.2.1.1 `utofu_get_onesided_tnis`

Get an array of IDs of the available TNIs for one-sided communication.

Format

```
int utofu_get_onesided_tnis(
    utofu_tni_id_t **tni_ids,
    size_t          *num_tnis)
```

Explanation

This function returns the IDs of local TNIs that are capable of one-sided communication and available to this process. The number of available TNIs may be smaller than the number of TNIs supplied with the compute node.

The returned TNI IDs can be used as arguments of the `utofu_query_onesided_caps()` and `utofu_create_vcq()` functions.

The caller should free the returned `tni_ids` array by using the `free()` function, unless there is no available TNI.

Parameters

Parameter Name	Explanation	IN/OUT
tni_ids	Pointer to an array of allocated TNI IDs. The array length is `num_tnis`. If there is no available TNI, `NULL` is set.	OUT
num_tnis	Number of available TNIs. If there is no available TNI, 0 is set.	OUT

Return values

Value	Explanation
UTOFU_SUCCESS	Succeeded
other	Other UTOFU_ERR_* error

3.2.1.2 utofu_get_barrier_tnis

Get an array of IDs of the available TNIs for barrier communication.

Format

```
int utofu_get_barrier_tnis(
    utofu_tni_id_t **tni_ids,
    size_t          *num_tnis)
```

Explanation

This function returns the IDs of local TNIs that are capable of barrier communication and available to this process. The number of available TNIs may be smaller than the number of TNIs supplied with the compute node.

The returned TNI IDs can be used as arguments of the utofu_query_barrier_caps() and utofu_alloc_vbg() functions.

The caller should free the returned tni_ids array by using the free() function, unless there is no available TNI.

Parameters

Parameter Name	Explanation	IN/OUT
tni_ids	Pointer to an allocated array of TNI IDs. The array length is num_tnis. If there is no available TNI, NULL is set.	OUT
num_tnis	Number of available TNIs. If there is no available TNI, 0 is set.	OUT

Return values

Value	Explanation
UTOFU_SUCCESS	Succeeded
other	Other UTOFU_ERR_* error

3.2.1.3 utofu_query_onesided_caps

Query the available capabilities and limit values of one-sided communication.

Format

```
int utofu_query_onesided_caps(
    utofu_tni_id_t               tni_id,
    struct utofu_onesided_caps **tni_caps)
```

Explanation

This function returns a pointer to the memory data managed by the uTofu implementation. The caller must neither free nor overwrite the tni_caps structure.

The function can query only TNIs returned from the utofu_get_onesided_tnis() function.

Parameters

Parameter Name	Explanation	IN/OUT
tni_id	TNI ID	IN
tni_caps	Pointer to the one-sided communication capability data of the TNI	OUT

Return values

Value	Explanation
UTOFU_SUCCESS	Succeeded
other	Other UTOFU_ERR_* error

3.2.1.4 utofu_query_barrier_caps

Query the available capabilities and limit values of barrier communication.

Format

```
int utofu_query_barrier_caps(
    utofu_tni_id_t              tni_id,
    struct utofu_barrier_caps **tni_caps)
```

Explanation

This function returns a pointer to the memory data managed by the uTofu implementation. The caller must neither free nor overwrite the tni_caps structure.

The function can query only TNIs returned from the utofu_get_barrier_tnis() function.

Parameters

Parameter Name	Explanation	IN/OUT
tni_id	TNI ID	IN
tni_caps	Pointer to the barrier communication capability data of the TNI	OUT

Return values

Value	Explanation
UTOFU_SUCCESS	Succeeded
other	Other UTOFU_ERR_* error

3.2.2 Structures Indicating Available Communication Functions and Limit Values

3.2.2.1 struct utofu_onesided_caps

Capabilities of one-sided communication

Definition

```
struct utofu_onesided_caps {
    unsigned long int flags;
    unsigned long int armw_ops;
    unsigned int      num_cmp_ids;
    unsigned int      num_reserved_stags;
    size_t            cache_line_size;
    size_t            stag_address_alignment;
    size_t            max_toq_desc_size;
    size_t            max_putget_size;
    size_t            max_piggyback_size;
    size_t            max_edata_size;
    size_t            max_mtu;
    size_t            max_gap;
}
```

Explanation

The `utofu_query_onesided_caps()` function returns a pointer to this structure.

Members

Member Name	Explanation
`flags`	Flags for capabilities of one-sided communication. Bitwise OR of `UTOFU_ONESIDED_CAP_FLAG_*`.
`armw_ops`	Supported ARMW operation types. Bitwise OR of `UTOFU_ONESIDED_CAP_ARMW_OP_*`.
`num_cmp_ids`	Number of component IDs available. Used for the `utofu_create_vcq_with_cmp_id()` function.
`num_reserved_stags`	Number of reserved STags per VCQ. Used for the `utofu_reg_mem_with_stag()` function.
`cache_line_size`	Line size of the last level cache of the CPU. Used for the cache injection feature and confirmation of communication completion.
`stag_address_alignment`	Alignment of a memory address where a STag can be assigned using the `utofu_reg_mem_with_stag()` function.
`max_toq_desc_size`	Maximum byte size of a TOQ descriptor. Used for `utofu_prepare_*()` functions.
`max_putget_size`	Maximum byte size of Put and Get. The `length` parameter value specified for a one-sided communication start/preparation function must be less than or equal to this value.
`max_piggyback_size`	Maximum byte size of a piggyback. The `length` parameter value specified for the `utofu_put_piggyback()` function must be less than or equal to this value.
`max_edata_size`	Maximum byte size of a usable EDATA field
`max_mtu`	Maximum value of the MTU (maximum transfer unit) of a packet
`max_gap`	Maximum value of a transmission gap injected between packets

3.2.2.2 `struct utofu_barrier_caps`

Capabilities of barrier communication

Definition

```
struct utofu_barrier_caps {
    unsigned long int flags;
    unsigned long int reduce_ops;
    size_t            max_uint64_reduction;
    size_t            max_double_reduction;
}
```

Explanation

The `utofu_query_barrier_caps()` function returns a pointer to this structure.

Members

Member Name	Explanation
`flags`	Flags for capabilities of barrier communication.

Member Name	Explanation
	Bitwise OR of UTOFU_BARRIER_CAP_FLAG_*.
reduce_ops	Supported reduction operation types. Bitwise OR of UTOFU_BARRIER_CAP_REDUCE_OP_*.
max_uint64_reduction	Maximum number of simultaneous 64-bit integer (uint64_t) reduction operations
max_double_reduction	Maximum number of simultaneous floating-point (double) reduction operations

3.2.3 Flags Indicating Available Communication Functions

3.2.3.1 UTOFU_ONESIDED_CAP_FLAG_*

Bit flags for the flags member of the utofu_onesided_caps structure

Explanation

They can be used to see whether one-sided communication of a TNI has specific features.

Macros

Macro Name	Explanation
UTOFU_ONESIDED_CAP_FLAG_SESSION_MODE	Session mode feature
UTOFU_ONESIDED_CAP_FLAG_ARMW	Atomic Read Modify Write communication

3.2.3.2 UTOFU_BARRIER_CAP_FLAG_*

Bit flags for the flags member of the utofu_barrier_caps structure

Explanation

They can be used to see whether barrier communication of a TNI has specific features.

No flags are defined in the current version.

3.2.3.3 UTOFU_ONESIDED_CAP_ARMW_OP_*

Bit flags for the armw_ops member of the utofu_onesided_caps structure

Explanation

They can be used to see whether the ARMW (Atomic Read Modify Write) communication of one-sided communication has specific ARMW operations.

Macros

Macro Name	Explanation
UTOFU_ONESIDED_CAP_ARMW_OP_CSWAP	Compare and Swap operation
UTOFU_ONESIDED_CAP_ARMW_OP_SWAP	Swap operation
UTOFU_ONESIDED_CAP_ARMW_OP_ADD	Unsigned integer addition operation
UTOFU_ONESIDED_CAP_ARMW_OP_XOR	Bitwise XOR operation
UTOFU_ONESIDED_CAP_ARMW_OP_AND	Bitwise AND operation
UTOFU_ONESIDED_CAP_ARMW_OP_OR	Bitwise OR operation

3.2.3.4 UTOFU_BARRIER_CAP_REDUCE_OP_*

Bit flags for the reduce_ops member of the utofu_barrier_caps structure

Explanation

They can be used to see whether barrier communication has specific reduction operations.

Macros

Macro Name	Explanation
UTOFU_BARRIER_CAP_REDUCE_OP_BARRIER	Barrier synchronization (no reduction operation)
UTOFU_BARRIER_CAP_REDUCE_OP_BAND	Bitwise AND operation of uint64_t values
UTOFU_BARRIER_CAP_REDUCE_OP_BOR	Bitwise OR operation of uint64_t values
UTOFU_BARRIER_CAP_REDUCE_OP_BXOR	Bitwise XOR operation of uint64_t values
UTOFU_BARRIER_CAP_REDUCE_OP_MAX	Operation that gives maximum unsigned integer of uint64_t values
UTOFU_BARRIER_CAP_REDUCE_OP_MAXLOC	Operation that gives maximum unsigned integer of uint64_t values and its location
UTOFU_BARRIER_CAP_REDUCE_OP_SUM	Unsigned integer summation operation of uint64_t values
UTOFU_BARRIER_CAP_REDUCE_OP_BFPSUM	Floating-point (BFP) summation operation of double values

3.3 VCQ Management

A VCQ (virtual control queue) is an interface for uTofu users to control one-sided communication on a TNI.

Before starting one-sided communication, the uTofu user needs to query available local TNIs with the utofu_get_onesided_tnis() function and then create a VCQ corresponding to the local TNI with the utofu_create_vcq() function. After using one-sided communication, the uTofu user needs to release the VCQ with the utofu_free_vcq() function.

For the start of one-sided communication, the remote VCQ of the communication peer is specified by a VCQ ID. The utofu_query_vcq_id() function can obtain the local VCQ ID. To communicate with another process, the process has to notify this process of the VCQ ID. To start one-sided communication, a communication path must be specified. The utofu_set_vcq_id_path() function can embed default communication path coordinates into the VCQ ID.

The utofu_query_vcq_info() function can obtain information about a created VCQ.

More than one VCQ can be created for one TNI. Each VCQ has its independent communication context.

Once a VCQ is created in this process, the VCQ can communicate with any other VCQs in remote processes any number of times.

3.3.1 VCQ Creation/Freeing Functions

3.3.1.1 utofu_create_vcq

Create a VCQ on a given TNI on the compute node.

Format

```
int utofu_create_vcq(
    utofu_tni_id_t     tni_id,
    unsigned long int  flags,
    utofu_vcq_hdl_t   *vcq_hdl)
```

Explanation

The utofu_free_vcq() function should be used to release the created VCQ.

This function call may involve a system call.

Parameters

Parameter Name	Explanation	IN/OUT
tni_id	ID of the TNI to create a VCQ on	IN

Parameter Name	Explanation	IN/OUT
flags	Bitwise OR of UTOFU_VCQ_FLAG_*	IN
vcq_hdl	Handle of the created VCQ	OUT

Return values

Value	Explanation
UTOFU_SUCCESS	Succeeded
UTOFU_ERR_FULL	No more VCQs can be created for this TNI.
UTOFU_ERR_NOT_AVAILABLE	No VCQ of the type specified by the flags parameter is available.
UTOFU_ERR_NOT_SUPPORTED	VCQs of the type specified by the flags parameter are not supported.
other	Other UTOFU_ERR_* error

3.3.1.2 utofu_create_vcq_with_cmp_id

Create a VCQ with a component ID on a given TNI on the compute node.

Format

```
int utofu_create_vcq_with_cmp_id(
    utofu_tni_id_t     tni_id,
    utofu_cmp_id_t     cmp_id,
    unsigned long int  flags,
    utofu_vcq_hdl_t   *vcq_hdl)
```

Explanation

The difference between this function and the utofu_create_vcq() function is that a component ID for an upper layer can be specified with this function. The component ID is used to distinguish the VCQ of an upper-layer component from the VCQs of the other components sharing the same CQ. When the utofu_create_vcq() function is used to create a VCQ, the function selects a component ID not duplicated by the other VCQs and assigns it to the created VCQ. However, when this function is used, the caller must guarantee that the specified component ID is not or will not be used by another upper-layer component. Otherwise, this function returns UTOFU_ERR_USED.

The component IDs that can be used range from 0 to "utofu_onesided_caps::num_cmp_ids -1".

The utofu_free_vcq() function should be used to release the created VCQ.

This function call may involve a system call.

Other descriptions that discuss the utofu_create_vcq() function in this document also apply to this function.

Parameters

Parameter Name	Explanation	IN/OUT
tni_id	ID of the TNI to create a VCQ on	IN
cmp_id	ID of an upper-layer component. The value must be smaller than utofu_onesided_caps::num_cmp_ids.	IN
flags	Bitwise OR of UTOFU_VCQ_FLAG_*	IN
vcq_hdl	Handle of the created VCQ	OUT

Return values

Value	Explanation
UTOFU_SUCCESS	Succeeded
UTOFU_ERR_USED	Another component uses the specified component ID.

Value	Explanation
UTOFU_ERR_FULL	No more VCQs can be created for this TNI.
UTOFU_ERR_NOT_AVAILABLE	No VCQ of the type specified by the flags parameter is available.
UTOFU_ERR_NOT_SUPPORTED	VCQs of the type specified by the flags parameter are not supported.
other	Other UTOFU_ERR_* error

3.3.1.3 utofu_free_vcq

Free a VCQ.

Format

```
int utofu_free_vcq(
    utofu_vcq_hdl_t vcq_hdl)
```

Explanation

Double free is not allowed.

This function call may involve a system call.

Parameters

Parameter Name	Explanation	IN/OUT
vcq_hdl	Handle of a VCQ created by the utofu_create_vcq() function	IN

Return values

Value	Explanation
UTOFU_SUCCESS	Succeeded
other	Other UTOFU_ERR_* error

3.3.2 VCQ ID Manipulation Functions

3.3.2.1 utofu_query_vcq_id

Query the ID of the local VCQ.

Format

```
int utofu_query_vcq_id(
    utofu_vcq_hdl_t  vcq_hdl,
    utofu_vcq_id_t  *vcq_id)
```

Explanation

Though the value of a VCQ handle is valid only for this process, a VCQ ID is valid in all processes and is unique among all the VCQs in the Tofu network.

The A,B,C coordinates of the compute node are embedded in the VCQ ID as default communication path coordinates.

Parameters

Parameter Name	Explanation	IN/OUT
vcq_hdl	Handle of a VCQ created by the utofu_create_vcq() function	IN
vcq_id	ID of the VCQ	OUT

Return values

Value	Explanation
UTOFU_SUCCESS	Succeeded
other	Other UTOFU_ERR_* error

3.3.2.2 utofu_construct_vcq_id

Construct a VCQ ID.

Format

```
int utofu_construct_vcq_id(
    uint8_t         coords[],
    utofu_tni_id_t  tni_id,
    utofu_cq_id_t   cq_id,
    utofu_cmp_id_t  cmp_id,
    utofu_vcq_id_t *vcq_id)
```

Explanation

The VCQ ID of a VCQ created by the utofu_create_vcq_with_cmp_id() function can be computed from the compute node coordinates, TNI ID, CQ ID, and component ID by this function. The ID of a VCQ created on a remote compute node can also be computed.

The A,B,C coordinates of the specified compute node are embedded in the VCQ ID as default communication path coordinates.

Parameters

Parameter Name	Explanation	IN/OUT
coords	Compute node coordinates of a VCQ in the order of X, Y, Z, A, B, C. The array length must be 6.	IN
tni_id	TNI ID of a VCQ	IN
cq_id	CQ ID of a VCQ	IN
cmp_id	Component ID of a VCQ	IN
vcq_id	Constructed VCQ ID	OUT

Return values

Value	Explanation
UTOFU_SUCCESS	Succeeded
other	Other UTOFU_ERR_* error

3.3.2.3 utofu_set_vcq_id_path

Update the default communication path coordinates embedded in a VCQ ID.

Format

```
int utofu_set_vcq_id_path(
    utofu_vcq_id_t *vcq_id,
    uint8_t         path_coords[])
```

Explanation

To call a one-sided communication start/preparation function, a communication path can be specified by either a remote VCQ ID (rmt_vcq_id parameter) or bit flags (flags parameter). The utofu_query_vcq_id() and utofu_construct_vcq_id() functions embed the A,B,C coordinates of a compute node into a VCQ ID as default communication path coordinates. This function updates the communication path coordinates.

Note that this function updates the communication path coordinates locally and the information managed by uTofu implementation is not updated. For example, if there are two variables whose VCQ IDs are the same and this function is called for the one VCQ ID variable, the communication path coordinates of the other VCQ ID variable is not updated.

This function can be called for a VCQ ID variable more than once. The communication path coordinates in a communication using the VCQ ID variable is the value set by the last call of this function for the variable.

Parameters

Parameter Name	Explanation	IN/OUT
`vcq_id`	VCQ ID	IN,OUT
`path_coords`	Default communication path coordinates in the order of A, B, C to embed in the VCQ. The array length must be 3. `NULL` can be specified. In that case, appropriate communication path coordinates are automatically selected for the one-sided communications from the local compute node to the remote compute node specified by the VCQ ID.	IN

Return values

Value	Explanation
`UTOFU_SUCCESS`	Succeeded
other	Other `UTOFU_ERR_*` error

3.3.3 VCQ Query Functions

3.3.3.1 `utofu_query_vcq_info`

Query the compute node coordinates, TNI ID, and CQ ID of a given VCQ.

Format

```
int utofu_query_vcq_info(
    utofu_vcq_id_t  vcq_id,
    uint8_t         coords[],
    utofu_tni_id_t *tni_id,
    utofu_cq_id_t  *cq_id,
    uint16_t       *extra_val)
```

Explanation

The function can also query information about VCQs created on remote compute nodes.

This information may be used for performance tuning and debugging. For example, if there are multiple processes on one compute node, and you want each process to use a distinct TNI in simultaneous communications, this information can be used.

Parameters

Parameter Name	Explanation	IN/OUT
`vcq_id`	VCQ ID	IN
`coords`	Compute node coordinates of the VCQ in the order of X, Y, Z, A, B, C. The array length must be 6 or greater.	OUT
`tni_id`	TNI ID of the VCQ. The TNI ID ranges from 0 to "the number of TNIs supplied with the compute node `-1`".	OUT
`cq_id`	CQ ID of the VCQ. The CQ ID ranges from 0 to "the number of CQs supplied with the TNI `-1`".	OUT
`extra_val`	Extra value for internal use in the uTofu implementation. The value has no meaning for uTofu users.	OUT

Return values

Value	Explanation
UTOFU_SUCCESS	Succeeded
other	Other UTOFU_ERR_* error

3.3.4 VCQ Flags

3.3.4.1 UTOFU_VCQ_FLAG_*

Bit flags for the `flags` parameter of the `utofu_create_vcq()` function

Explanation

These flags can specify behaviors for creating a VCQ.

Macros

Macro Name	Explanation
UTOFU_VCQ_FLAG_THREAD_SAFE	Thread-safe VCQ. This flag requests the `utofu_create_vcq()` or `utofu_create_vcq_with_cmp_id()` function to create a new thread-safe VCQ. You can call STADD management functions and one-sided communication execution functions from multiple threads simultaneously for the single VCQ created with the flag. You can also call those functions simultaneously for the VCQ created with the flag and another VCQ.
UTOFU_VCQ_FLAG_EXCLUSIVE	CQ-exclusive VCQ. This flag requests the `utofu_create_vcq()` function to create a new CQ-exclusive VCQ, which does not share a CQ with other VCQs. You can call STADD management functions and one-sided communication execution functions from multiple threads simultaneously for this VCQ created with the flag and another VCQ. Those functions still cannot be called simultaneously for a single VCQ created with this flag but not with the UTOFU_VCQ_FLAG_THREAD_SAFE flag. The advantage of this flag over the UTOFU_VCQ_FLAG_THREAD_SAFE flag is that simultaneous calls of those functions for different CQs are not serialized in the uTofu implementation and hardware access. So communication throughput may be improved. If this flag is specified but CQs are exhausted, the `utofu_create_vcq()` function fails. The flag cannot be used for the `utofu_create_vcq_with_cmp_id()` function.
UTOFU_VCQ_FLAG_SESSION_MODE	Session mode VCQ. This flag requests the `utofu_create_vcq()` function to create a new VCQ of the session mode type. If the flag is specified but the TNI does not support session mode or the CQs for session mode are exhausted, the `utofu_create_vcq()` function fails. The flag cannot be used for the `utofu_create_vcq_with_cmp_id()` function. This flag cannot be used together with the UTOFU_VCQ_FLAG_EXCLUSIVE flag.

3.4 VBG Management

A VBG (virtual barrier gate) is a TNI component that sends and receives barrier signals and barrier packets.

Before using barrier communication, the uTofu user needs to query available TNIs with the `utofu_get_barrier_tnis()` function, allocate VBGs on a TNI with the `utofu_alloc_vbg()` function, and build a barrier circuit with the `utofu_set_vbg()` function. After using barrier communication, the uTofu user needs to free the VBGs with the `utofu_free_vbg()` function. The `utofu_query_vbg_info()` function can obtain information about an allocated VBG.

To communicate with remote compute nodes, the remote compute nodes also have to allocate their VBGs.

More than one VBG can be allocated for one TNI. Actually, multiple VBGs are usually required for building a barrier circuit.

Once a barrier circuit is built, barrier synchronization and reduction operation can be executed on the barrier circuit any number of times.

3.4.1 VBG Allocation/Freeing Functions

3.4.1.1 utofu_alloc_vbg

Allocate VBGs for a given TNI on the compute node.

Format

```
int utofu_alloc_vbg(
    utofu_tni_id_t    tni_id,
    size_t            num_vbgs,
    unsigned long int flags,
    utofu_vbg_id_t    vbg_ids[])
```

Explanation

This function writes VBG IDs to the caller-supplied `vbg_ids` array.

The first element of the array corresponds to a start/end BG and the remaining elements correspond to relay BGs.

The `utofu_free_vbg()` function should be used to release the allocated VBGs.

This function call may involve a system call.

Parameters

Parameter Name	Explanation	IN/OUT
`tni_id`	ID of the TNI to allocate VBGs on	IN
`num_vbgs`	Number of VBGs required. The value must be greater than 0.	IN
`flags`	Bitwise OR of `UTOFU_VBG_FLAG_*`	IN
`vbg_ids`	Array of IDs of the allocated VBGs. The array length must be `num_vbgs` or greater.	OUT

Return values

Value	Explanation
`UTOFU_SUCCESS`	Succeeded
`UTOFU_ERR_FULL`	The specified number of VBGs cannot be allocated on this TNI.
`UTOFU_ERR_NOT_AVAILABLE`	No VBG of the type specified by the `flags` parameter is available.
`UTOFU_ERR_NOT_SUPPORTED`	VBGs of the type specified by the `flags` parameter are not supported.
other	Other `UTOFU_ERR_*` error

3.4.1.2 utofu_free_vbg

Free VBGs.

Format

```
int utofu_free_vbg(
    utofu_vbg_id_t vbg_ids[],
    size_t         num_vbgs)
```

Explanation

Though each VBG ID is unique among all the VBGs in the Tofu network, this function can simply free the VBGs allocated by this process.

Double free is not allowed.

This function call may involve a system call.

Parameters

Parameter Name	Explanation	IN/OUT
`vbg_ids`	Array of IDs of VBGs allocated by the `utofu_alloc_vbg()` function. The array contents must be the same as the array returned by the `utofu_alloc_vbg()` function. The array length must be `num_vbgs`.	IN
`num_vbgs`	Number of VBGs. The value must be same as the value passed to the corresponding `utofu_alloc_vbg()` function call.	IN

Return values

Value	Explanation
`UTOFU_SUCCESS`	Succeeded
other	Other `UTOFU_ERR_*` error

3.4.2 VBG Configuration Functions

3.4.2.1 `utofu_set_vbg`

Configure VBGs to build a barrier circuit.

Format

```
int utofu_set_vbg(
    struct utofu_vbg_setting vbg_settings[],
    size_t                   num_vbg_settings)
```

Explanation

Each VBG receives an input signal and input packet and then sends an output signal and output packet.

- The input signal is received from another VBG on the same TNI (local source).
- The input packet is received from a VBG on a TNI on the same or a remote compute node (remote source).
- The output signal is sent to another VBG on the same TNI (local destination).
- The output packet is sent to a VBG on a TNI on the same or a remote compute node (remote destination).

A barrier circuit is built by combining all participating VBGs. To build one barrier circuit, this function must be called for the local VBG set on every compute node participating in the barrier circuit.

This function configures multiple local VBGs allocated by a single `utofu_alloc_vbg()` function call at once. The set of values of `utofu_vbg_setting::vbg_id` in the `vbg_settings` array must be the same as or a subset of the VBG IDs returned by the `utofu_alloc_vbg()` function. The first VBG ID in the `vbg_settings` array corresponds to a start/end BG and must be the first VBG ID returned by the `utofu_alloc_vbg()` function. However, the remaining VBG IDs do not need to be in the same order as the VBG IDs returned by the `utofu_alloc_vbg()` function.

If this function is called for a VBG set more than once, the last call takes effect.

This function call may involve a system call.

Parameters

Parameter Name	Explanation	IN/OUT
vbg_settings	Array of VBG settings. The array length must be num_vbg_settings.	IN
num_vbg_settings	Number of VBGs to configure. The value must be greater than 0 and less than or equal to the value passed to the corresponding utofu_alloc_vbg() function call.	IN

Return values

Value	Explanation
UTOFU_SUCCESS	Succeeded
other	Other UTOFU_ERR_* error

3.4.3 VBG Query Functions

3.4.3.1 utofu_query_vbg_info

Query the compute node coordinates, TNI ID, and BG ID of a given VBG.

Format

```
int utofu_query_vbg_info(
    utofu_vbg_id_t  vbg_id,
    uint8_t         coords[],
    utofu_tni_id_t *tni_id,
    utofu_bg_id_t  *bg_id,
    uint16_t       *extra_val)
```

Explanation

The function can also query information about a VBG allocated on a remote compute node.

This information may be used for debugging.

Parameters

Parameter Name	Explanation	IN/OUT
vbg_id	VBG ID	IN
coords	Compute node coordinates of the VBG in the order of X, Y, Z, A, B, C. The array length must be 6 or greater.	OUT
tni_id	TNI ID of the VBG. The TNI ID ranges from 0 to "the number of TNIs supplied with the compute node -1".	OUT
bg_id	BG ID of the VBG. The BG ID ranges from 0 to "the number of BGs supplied with the TNI -1".	OUT
extra_val	Extra value for internal use in the uTofu implementation. The value has no meaning for uTofu users.	OUT

Return values

Value	Explanation
UTOFU_SUCCESS	Succeeded
other	Other UTOFU_ERR_* error

3.4.4 VBG Configuration Structures

3.4.4.1 `struct utofu_vbg_setting`

VBG settings

Definition

```
struct utofu_vbg_setting {
    utofu_vbg_id_t vbg_id;
    utofu_vbg_id_t src_lcl_vbg_id;
    utofu_vbg_id_t src_rmt_vbg_id;
    utofu_vbg_id_t dst_lcl_vbg_id;
    utofu_vbg_id_t dst_rmt_vbg_id;
    uint8_t        dst_path_coords[3];
}
```

Explanation

The `utofu_set_vbg()` function requires an array of this structure as the `vbg_settings` parameter.

For a VBG that does not receive/send one of the signals/packets, `UTOFU_VBG_ID_NULL` should be specified as the VBG ID for the corresponding local/remote source/destination VBG.

Members

Member Name	Explanation
`vbg_id`	ID of the local VBG to configure
`src_lcl_vbg_id`	ID of the local source VBG of an input signal to the VBG
`src_rmt_vbg_id`	ID of the remote source VBG of an input packet to the VBG
`dst_lcl_vbg_id`	ID of the local destination VBG of an output signal from the VBG
`dst_rmt_vbg_id`	ID of the remote destination VBG of an output packet from the VBG
`dst_path_coords`	Communication path coordinates in the order of A, B, C to the remote destination VBG. If `UTOFU_PATH_COORD_NULL` is specified as the coordinate A (`dst_path_coords[0]`), appropriate communication path coordinates are automatically selected. In this case, the coordinates B and C (`dst_path_coords[1]` and `dst_path_coords[2]`) are ignored.

3.4.5 VBG Flags

3.4.5.1 UTOFU_VBG_FLAG_*

Bit flags for the `flags` parameter of the `utofu_alloc_vbg()` function

Explanation

These flags can specify behaviors for allocating a VBG.

Macros

Macro Name	Explanation
`UTOFU_VBG_FLAG_THREAD_SAFE`	Thread-safe VBG. This flag requests the `utofu_alloc_vbg()` function to allocate new thread-safe VBGs. Barrier communication execution functions can be called from multiple threads simultaneously for the single VBG allocated with the flag.

3.4.6 Special Values for VBG Configuration

3.4.6.1 UTOFU_VBG_ID_NULL

Macros

Macro Name	Explanation
UTOFU_VBG_ID_NULL	Null VBG ID. This special VBG ID is used to represent no input signal/packet or no outgoing signal/packet for the utofu_set_vbg() function.

3.5 Communication Path Management

To start each one-sided communication or build each barrier circuit, a communication path on a Tofu network must be specified.

In the case of one-sided communication, a communication path can be specified with either a remote VCQ ID (rmt_vcq_id parameter) or bit flags (flags parameter) to call a one-sided communication start/preparation function. For a remote VCQ ID, the utofu_query_vcq_id() and utofu_construct_vcq_id() functions embed default communication path coordinates into the VCQ ID, and the utofu_set_vcq_id_path() function updates the coordinates. For bit flags, the UTOFU_ONESIDED_FLAG_PATH function-like macro specifies the communication path. The UTOFU_ONESIDED_FLAG_PATH function-like macro takes a parameter of a communication path ID.

The utofu_get_path_id() function creates a communication path ID from communication path coordinates. The utofu_get_path_coords() function can query the communication path coordinates from the communication path ID.

In the case of barrier communication, communication path coordinates can be specified when calling a VBG configuration function.

3.5.1 Communication Path Management Functions

3.5.1.1 utofu_get_path_id

Create a communication path ID from communication path coordinates.

Format

```
int utofu_get_path_id(
    utofu_vcq_id_t   vcq_id,
    uint8_t          path_coords[],
    utofu_path_id_t *path_id)
```

Explanation

A communication path ID is defined for each remote VCQ ID. So a communication path ID cannot be used for a different remote VCQ ID than the remote VCQ ID specified in this function argument, even if the communication path coordinates are the same.

A VCQ ID with default communication path coordinates embedded by the utofu_set_vcq_id_path() function can also be specified as the vcq_id parameter. However, the returned communication path ID is valid only for the specified VCQ ID. When new default communication path coordinates are embedded in the VCQ ID, a new communication path ID should be created with the new VCQ ID.

Parameters

Parameter Name	Explanation	IN/OUT
vcq_id	Remote VCQ ID. Given by the utofu_query_vcq_id() function in a remote process.	IN
path_coords	Communication path coordinates in the order of A, B, C.	IN

Parameter Name	Explanation	IN/OUT
	The array length must be 3. `NULL` can be specified. In that case, appropriate coordinates are automatically selected for the communication path from the local compute node to the remote compute node specified by the VCQ ID.	
`path_id`	Corresponding communication path ID	OUT

Return values

Value	Explanation
`UTOFU_SUCCESS`	Succeeded
other	Other `UTOFU_ERR_*` error

3.5.1.2 `utofu_get_path_coords`

Query the communication path coordinates of a communication path ID.

Format

```
int utofu_get_path_coords(
    utofu_vcq_id_t  vcq_id,
    utofu_path_id_t path_id,
    uint8_t         path_coords[])
```

Explanation

A communication path ID is defined for each remote VCQ ID. So the specified remote VCQ ID must match that specified for the `utofu_get_path_id()` function.

Parameters

Parameter Name	Explanation	IN/OUT
`vcq_id`	Remote VCQ ID. Given by the `utofu_query_vcq_id()` function in a remote process.	IN
`path_id`	Communication path ID created by the `utofu_get_path_id()` function.	IN
`path_coords`	Corresponding communication path coordinates in the order of A, B, C. The array length must be 3 or greater.	OUT

Return values

Value	Explanation
`UTOFU_SUCCESS`	Succeeded
other	Other `UTOFU_ERR_*` error

3.5.2 Special Values for Setting of Communication Path

3.5.2.1 `UTOFU_PATH_COORD_NULL`

Macros

Macro Name	Explanation
`UTOFU_PATH_COORD_NULL`	Null communication path coordinate. This special path coordinate is used to represent automatic selection of communication path coordinates for the `utofu_set_vbg()` function.

3.6 STADD Management

A STADD (steering address) is a memory address that a TNI can understand. Because a memory address in a user-space process is a virtual address assigned by the operating system, a TNI cannot understand such an address. A STADD is used to steer a TNI toward the right memory address in one-sided communication.

The `utofu_reg_mem()` function assigns a STADD for a VCQ by registering a memory region to the VCQ. The assigned STADD can be passed to a one-sided communication start/preparation function. After use, the assigned STADD must be freed, and the memory region must be deregistered using the `utofu_dereg_mem()` function.

The `utofu_reg_mem()` function can register a memory region to a VCQ more than once. In this case, the same STADD value is returned every time. Also, an overlapping memory region may be registered. The `utofu_dereg_mem()` function must be called as many times as `utofu_reg_mem()` function calls.

To register a memory region, the region must be accessible from the calling process. Without the `UTOFU_REG_MEM_FLAG_READ_ONLY` flag, the memory region must be readable and writable. With the `UTOFU_REG_MEM_FLAG_READ_ONLY` flag, the memory region must at least be readable.

To use a memory region for one-sided communications via two or more VCQs, the memory region must be registered to each VCQ. However, the returned STADD values may not be same.

Most one-sided communication start/preparation functions have `lcl_stadd` and `rmt_stadd` parameters. The `lcl_stadd` parameter is the STADD assigned on the compute node (communication origin). The `rmt_stadd` parameter is the STADD assigned on a remote compute node (communication target). So before one-sided communication starts, the local process must be informed of the STADD on the remote compute node by the remote process. Once a STADD is assigned for a VCQ, the STADD can be used for multiple one-sided communications on the VCQ, even for concurrent communications.

A registered memory region must be deregistered before the memory region is freed by the `free()` function, etc. With the Put communication, the local memory region can be deregistered after the corresponding TCQ descriptor or local notification MRQ descriptor is written. With the Get communication, the local memory region can be deregistered after the corresponding local notification MRQ descriptor is written. With all communications, the remote memory region can be deregistered after the corresponding remote notification MRQ descriptor is written on the remote compute node or the corresponding local notification MRQ descriptor is written on this node.

Usually, a device driver (the Tofu driver) assigns a STADD when the `utofu_reg_mem()` function is called. The value is not predictable to uTofu users. As an alternative, the `utofu_reg_mem_with_stag()` function can assign a predictable STADD. A STag is an integer that is a part of a 64-bit STADD value. A specific range of STags is reserved for uTofu users. If the local process knows that a remote process has registered its memory region with a predictable STag, the `utofu_query_stadd()` function in the local process can obtain its STADD. A memory region registered by the `utofu_reg_mem_with_stag()` function must likewise be deregistered by the `utofu_dereg_mem()` function. The `utofu_reg_mem_with_stag()` function cannot be called for the same STag more than once unless the STag has not been deregistered by the `utofu_dereg_mem()` function. Once the STag is deregistered, the `utofu_reg_mem_with_stag()` function can reuse the same STag for the same or a different memory region.

A STADD is a 64-bit unsigned integer. One-sided communication can be posted for a sub-region in a registered memory region by specifying a STADD calculated based on an offset. For example, if you want to transfer random segments in a large memory region, you can first register the whole region and get the STADD `head_stadd`. Then, you can call the `utofu_put()` function with the value of "`head_stadd` + `random_offset`" as the `lcl_stadd` parameter. `rmt_stadd` can also be calculated in the same way.

3.6.1 STADD Management Functions

3.6.1.1 utofu_reg_mem

Register a memory region to a VCQ.

Format

```
int utofu_reg_mem(
    utofu_vcq_hdl_t    vcq_hdl,
    void              *addr,
    size_t             size,
    unsigned long int  flags,
    utofu_stadd_t     *stadd)
```

Explanation

The `utofu_dereg_mem()` function should be used to deregister the registered memory.

This function call may involve a system call.

Parameters

Parameter Name	Explanation	IN/OUT
`vcq_hdl`	VCQ handle	IN
`addr`	Pointer to the beginning of a memory region	IN
`size`	Byte size of a memory region. The value must be greater than 0.	IN
`flags`	Bitwise OR of `UTOFU_REG_MEM_FLAG_*`	IN
`stadd`	STADD of the beginning of the memory region	OUT

Return values

Value	Explanation
`UTOFU_SUCCESS`	Succeeded
`UTOFU_ERR_FULL`	No more STADDs can be assigned for this VCQ. Trying again after calling the `utofu_dereg_mem()` function may succeed.
`UTOFU_ERR_NOT_AVAILABLE`	No STADD of the type specified by the `flags` parameter is available.
`UTOFU_ERR_NOT_SUPPORTED`	STADDs of the type specified by the `flags` parameter are not supported.
other	Other `UTOFU_ERR_*` error

3.6.1.2 `utofu_reg_mem_with_stag`

Register a memory region to a VCQ with a STag.

Format

```
int utofu_reg_mem_with_stag(
    utofu_vcq_hdl_t    vcq_hdl,
    void              *addr,
    size_t             size,
    unsigned int       stag,
    unsigned long int  flags,
    utofu_stadd_t     *stadd)
```

Explanation

The STags that can be assigned range from 0 to "`utofu_onesided_caps::num_reserved_stags -1`".

The `utofu_dereg_mem()` function should be used to deregister the registered memory.

Until the STag is freed by the `utofu_dereg_mem()` function, the same STag cannot be reused.

This function call may involve a system call.

Parameters

Parameter Name	Explanation	IN/OUT
`vcq_hdl`	VCQ handle	IN
`addr`	Pointer to the beginning of a memory region. The memory address must be a multiple of `utofu_onesided_caps::stag_address_alignment`.	IN

Parameter Name	Explanation	IN/OUT
size	Byte size of a memory region. The size must be a multiple of utofu_onesided_caps::stag_address_alignment.	IN
stag	STag. The value must be smaller than utofu_onesided_caps::num_reserved_stags.	IN
flags	Bitwise OR of UTOFU_REG_MEM_FLAG_*	IN
stadd	STADD of the beginning of the memory region	OUT

Return values

Value	Explanation
UTOFU_SUCCESS	Succeeded
UTOFU_ERR_USED	The specified STag is already used on this VCQ.
UTOFU_ERR_NOT_AVAILABLE	No STag of the type specified by the flags parameter is available.
UTOFU_ERR_NOT_SUPPORTED	STags of the type specified by the flags parameter are not supported.
other	Other UTOFU_ERR_* error

3.6.1.3 utofu_query_stadd

Query the STADD of the memory region where a STag is assigned.

Format

```
int utofu_query_stadd(
    utofu_vcq_id_t  vcq_id,
    unsigned int    stag,
    utofu_stadd_t  *stadd)
```

Explanation

This function can query the STADD of a memory region in both the local process and a remote process. The STADD returned by the function is the same as the STADD returned by the utofu_reg_mem_with_stag() function.

This function can be called more than once for each STag.

Parameters

Parameter Name	Explanation	IN/OUT
vcq_id	VCQ ID corresponding to the VCQ handle used in a utofu_reg_mem_with_stag() function call. To query a STADD in a remote process, this VCQ ID must be one in the remote process.	IN
stag	STag used in a utofu_reg_mem_with_stag() function call	IN
stadd	STADD of the beginning of the memory region	OUT

Return values

Value	Explanation
UTOFU_SUCCESS	Succeeded
other	Other UTOFU_ERR_* error

3.6.1.4 utofu_dereg_mem

Deregister a memory region from a VCQ.

Format

```
int utofu_dereg_mem(
    utofu_vcq_hdl_t   vcq_hdl,
    utofu_stadd_t     stadd,
    unsigned long int flags)
```

Explanation

Double free (deregistration) is not allowed.

This function call may involve a system call.

Parameters

Parameter Name	Explanation	IN/OUT
vcq_hdl	VCQ handle	IN
stadd	STADD of the beginning of a memory region. The STADD must be one that was returned by the utofu_reg_mem() or utofu_reg_mem_with_stag() function.	IN
flags	Bitwise OR of UTOFU_DEREG_MEM_*	IN

Return values

Value	Explanation
UTOFU_SUCCESS	Succeeded
UTOFU_ERR_NOT_AVAILABLE	No STADD of the type specified by the flags parameter is available.
UTOFU_ERR_NOT_SUPPORTED	STADDs of the type specified by the flags parameter are not supported.
other	Other UTOFU_ERR_* error

3.6.2 STADD Flags

3.6.2.1 UTOFU_REG_MEM_FLAG_*

Bit flags for the flags parameter of the utofu_reg_mem() and utofu_reg_mem_with_stag() functions

Explanation

These flags can specify behaviors of memory registration.

Macros

Macro Name	Explanation
UTOFU_REG_MEM_FLAG_READ_ONLY	Read-only. This flag indicates the memory region will not be updated in one-sided communication, neither by the local process nor by a remote process. If the STADD of the memory region registered by the flag is used as the remote STADD of Put or ARMW, or the local STADD of Get, a communication error occurs and the error is reported by the TCQ or MRQ. The flag must be specified for any memory region that cannot be written from the registering process. One memory region can be in a state where it is registered both with and without the flag.

3.6.2.2 UTOFU_DEREG_MEM_FLAG_*

Bit flags for the flags parameter of the utofu_dereg_mem() function

Explanation

These flags can specify behaviors of memory deregistration.

No flags are defined in the current version.

3.7 One-Sided Communication Execution

A TOQ (transmit order queue), TCQ (transmit complete queue), and MRQ (message receive queue) are VCQ components. They are queues that a TNI reads and writes.

A TOQ descriptor is posted to a TOQ via the `utofu_put()` function, etc. Once the TOQ descriptor is posted, the TNI corresponding to the VCQ reads the descriptor, and one-sided communication starts. The different kinds of TOQ descriptor are Put, Put Piggyback, Get, and ARMW (Atomic Read Modify Write).

The completion of one-sided communication can be confirmed in three ways.

- If the `UTOFU_ONESIDED_FLAG_TCQ_NOTICE` flag is specified in the one-sided communication start function call, a TCQ descriptor is written to the TCQ corresponding to the local VCQ when packets are sent out from the local compute node. For a Put TOQ descriptor, the TCQ descriptor indicates that the local memory can be overwritten with other data.
- If the `UTOFU_ONESIDED_FLAG_LOCAL_MRQ_NOTICE` flag is specified in the one-sided communication start function call, an MRQ descriptor is written to the MRQ corresponding to the local VCQ when the communication completes on both the local compute node and remote compute node. For a Put TOQ descriptor and an ARMW TOQ descriptor, the MRQ descriptor indicates that the remote memory has been updated. For a Get TOQ descriptor, the MRQ descriptor indicates that the local memory has been updated and the updated data in the memory can be read. For an ARMW TOQ descriptor, the MRQ descriptor includes the value in the remote memory before the update.
- If the `UTOFU_ONESIDED_FLAG_REMOTE_MRQ_NOTICE` flag is specified in the one-sided communication start function call, an MRQ descriptor is written to the MRQ on the remote compute node corresponding to the remote VCQ when the communication completes on the remote compute node. For a Put TOQ descriptor and an ARMW TOQ descriptor, the MRQ descriptor indicates that the remote memory has been updated. For a Get TOQ descriptor, the MRQ descriptor indicates that the remote memory can be overwritten with other data.

If the source memory is overwritten before the packet sending or communication completion, the contents of the destination memory are not guaranteed to have the value from before or after the overwriting. If the destination memory is read before the communication completion, the contents of the data are not guaranteed to have the value from before or after the communication.

The `utofu_poll_tcq()` function can poll a TCQ. For one-sided communication start functions except the `utofu_{put|get}_stride()` and `utofu_{put|get}_stride_gap()` functions, if the `UTOFU_ONESIDED_FLAG_TCQ_NOTICE` flag is specified, one function call results in one TCQ descriptor, and the `utofu_poll_tcq()` function returns `UTOFU_SUCCESS` for it. For the `utofu_{put|get}_stride()` and `utofu_{put|get}_stride_gap()` functions, if the `UTOFU_ONESIDED_FLAG_TCQ_NOTICE` flag is specified, one function call results in the same number of TCQ descriptors as the number of TOQ descriptors given by the `num_blocks` parameter. However, regardless of the `UTOFU_ONESIDED_FLAG_TCQ_NOTICE` flag, if an error occurs when the TNI sends out a packet from the local compute node, the `utofu_poll_tcq()` function returns `UTOFU_ERR_TCQ_*`. All one-sided communication start functions and the `utofu_poll_tcq()` function have a `cbdata` parameter. The `cbdata` value passed to one-sided communication start functions is set in `cbdata` in the `utofu_poll_tcq()` function so that the uTofu user can identify the completed communication when the `utofu_poll_tcq()` function returns a code other than `UTOFU_ERR_NOT_FOUND`.

The `utofu_poll_mrq()` function can poll an MRQ. The number of reported MRQ descriptors per one-sided communication start function call is determined in the same manner as the `utofu_poll_tcq()` function. The `utofu_poll_mrq()` function returns `UTOFU_SUCCESS` for communication completed successfully and returns `UTOFU_ERR_MRQ_*` for communication with an error. The completed communication can be identified using the `notice` parameter of the `utofu_poll_mrq()` function.

Even if no `UTOFU_ONESIDED_FLAG_*_NOTICE` flags are specified, the `utofu_poll_tcq()` and `utofu_poll_mrq()` functions must be called in order to confirm the occurrence of any errors and to make space in queues. When the TCQ is full because of insufficient calls of the `utofu_poll_tcq()` function, one-sided communication start functions and batch start functions return `UTOFU_ERR_BUSY`. When the MRQ is full because of insufficient calls of the `utofu_poll_mrq()` function, an MRQ overflow error occurs. In order to make enough space in the TCQ, repeat calling the `utofu_poll_tcq()` function until the function returns `UTOFU_ERR_TCQ_*` or `UTOFU_ERR_NOT_FOUND`. Similarly, in order to make enough space in the MRQ, repeat calling the `utofu_poll_mrq()` function until the function returns `UTOFU_ERR_MRQ_*` or `UTOFU_ERR_NOT_FOUND`.

All one-sided communication start functions have the `rmt_vcq_id` and `rmt_stadd` parameters. Their values should be obtained in any way (e.g., out-of-band communication other than uTofu) from the remote compute nodes unless the `utofu_reg_mem_with_stag()` function is used for the `rmt_stadd` parameter.

To post a TOQ descriptor, a communication path can be specified by either a remote VCQ ID (`rmt_vcq_id` parameter) or bit flags (`flags` parameter). If both specify communication paths, the communication path specified by bit flags is used.

An MTU and a transmission gap can be explicitly specified with `utofu_*_gap()` functions. Other functions use `utofu_onesided_caps::max_mtu` for an MTU and 0 for a transmission gap. To call `utofu_*_gap()` functions, a value from 0 up to `utofu_onesided_caps::max_gap` can be specified as the `gap` parameter. If 0 is specified, no gap is injected between packets. If a value other than 0 is specified, a gap is injected between packets and the injection rate is throttled to "`8/(gap+8)`" compared to the case of 0. For example, the injection rate would be 50% for `gap` 8 and 25% for `gap` 24.

To transfer multiple data of the same size with the same interval in a memory region using Put or Get, the `utofu_{put|get}_stride()` and `utofu_{put|get}_stride_gap()` functions can be used. These functions post multiple Put/Get TOQ descriptors at once in one function call. The distance between the start addresses of neighboring data is called a stride. A local STADD is `lcl_stadd`, a remote STADD is `rmt_stadd`, the data length is `length`, and the stride length is `stride`. The first Put transfers the region between `lcl_stadd` and "`lcl_stadd + length`" to the region between `rmt_stadd` and "`rmt_stadd + length`". The second Put transfers the region between "`lcl_stadd + stride`" and "`lcl_stadd + stride + length`" to the region between "`rmt_stadd + stride`" and "`rmt_stadd + stride + length`". The third Put transfers the region between "`lcl_stadd + (stride * 2)`" and "`lcl_stadd + (stride * 2) + length`" to the region between "`rmt_stadd + (stride * 2)`" and "`rmt_stadd + (stride * 2) + length`".

All the one-sided communication start functions `utofu_*()` have the corresponding one-sided communication preparation functions `utofu_prepare_*()`. These functions do not post a TOQ descriptor to a TOQ but write it to caller-supplied memory. The one-sided communication batch start function `utofu_post_toq()` can post the written descriptor to the TOQ. If the uTofu users repeatedly executes the same communication, these functions can be used to reduce the overhead in creating TOQ descriptors.

3.7.1 One-Sided Communication Start Functions

3.7.1.1 utofu_put

Post a Put descriptor to a TOQ.

Format

```
int utofu_put(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    utofu_stadd_t      lcl_stadd,
    utofu_stadd_t      rmt_stadd,
    size_t             length,
    uint64_t           edata,
    unsigned long int  flags,
    void              *cbdata)
```

Parameters

Parameter Name	Explanation	IN/OUT
`vcq_hdl`	Local VCQ handle. Given by the `utofu_create_vcq()` function in this process.	IN
`rmt_vcq_id`	Remote VCQ ID. Given by the `utofu_query_vcq_id()` function in a remote process.	IN
`lcl_stadd`	STADD of the start address of the local (data source) memory region. Given by the `utofu_reg_mem()` function in this process.	IN
`rmt_stadd`	STADD of the start address of a remote (data destination) memory region. Given by the `utofu_reg_mem()` function in a remote process.	IN

Parameter Name	Explanation	IN/OUT
length	Data length in bytes. A value up to utofu_onesided_caps::max_putget_size can be specified.	IN
edata	EDATA that is passed by the utofu_poll_mrq() function. A value within utofu_onesided_caps::max_edata_size bytes can be specified.	IN
flags	Bitwise OR of UTOFU_ONESIDED_FLAG_*	IN
cbdata	Callback data that is passed by the utofu_poll_tcq() function	IN

Return values

Value	Explanation
UTOFU_SUCCESS	Succeeded
UTOFU_ERR_BUSY	The TOQ is full. Call the utofu_poll_tcq() function before trying again.
other	Other UTOFU_ERR_* error

3.7.1.2 utofu_put_gap

Post a Put descriptor with a transmission gap to a TOQ.

Format

```
int utofu_put_gap(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    utofu_stadd_t      lcl_stadd,
    utofu_stadd_t      rmt_stadd,
    size_t             length,
    uint64_t           edata,
    unsigned long int  flags,
    size_t             mtu,
    size_t             gap,
    void              *cbdata)
```

Explanation

The parameters other than mtu and gap are the same as for the utofu_put() function.

Parameters

Parameter Name	Explanation	IN/OUT
vcq_hdl	Local VCQ handle. Given by the utofu_create_vcq() function in this process.	IN
rmt_vcq_id	Remote VCQ ID. Given by the utofu_query_vcq_id() function in a remote process.	IN
lcl_stadd	STADD of the start address of the local (data source) memory region. Given by the utofu_reg_mem() function in this process.	IN
rmt_stadd	STADD of the start address of a remote (data destination) memory region. Given by the utofu_reg_mem() function in a remote process.	IN
length	Data length in bytes. A value up to utofu_onesided_caps::max_putget_size can be specified.	IN

Parameter Name	Explanation	IN/OUT
edata	EDATA that is passed by the utofu_poll_mrq() function. A value within utofu_onesided_caps::max_edata_size bytes can be specified.	IN
flags	Bitwise OR of UTOFU_ONESIDED_FLAG_*	IN
mtu	MTU (maximum transfer unit) of a packet. A value up to utofu_onesided_caps::max_mtu can be specified.	IN
gap	Transmission gap injected between packets. A value up to utofu_onesided_caps::max_gap can be specified.	IN
cbdata	Callback data that is passed by the utofu_poll_tcq() function	IN

Return values

Value	Explanation
UTOFU_SUCCESS	Succeeded
UTOFU_ERR_BUSY	The TOQ is full. Call the utofu_poll_tcq() function before trying again.
other	Other UTOFU_ERR_* error

3.7.1.3 utofu_put_stride

Post multiple Put descriptors to a TOQ.

Format

```
int utofu_put_stride(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    utofu_stadd_t      lcl_stadd,
    utofu_stadd_t      rmt_stadd,
    size_t             length,
    size_t             stride,
    size_t             num_blocks,
    uint64_t           edata,
    unsigned long int  flags,
    void              *cbdata)
```

Explanation

The parameters other than stride and num_blocks are the same as for the utofu_put() function.

Parameters

Parameter Name	Explanation	IN/OUT
vcq_hdl	Local VCQ handle. Given by the utofu_create_vcq() function in this process.	IN
rmt_vcq_id	Remote VCQ ID. Given by the utofu_query_vcq_id() function in a remote process.	IN
lcl_stadd	STADD of the start address of the local (data source) memory region. Given by the utofu_reg_mem() function in this process.	IN
rmt_stadd	STADD of the start address of a remote (data destination) memory region. Given by the utofu_reg_mem() function in a remote process.	IN

Parameter Name	Explanation	IN/OUT
length	Data length in bytes. A value up to utofu_onesided_caps::max_putget_size can be specified.	IN
stride	Stride length in bytes	IN
num_blocks	Number of Put descriptors	IN
edata	EDATA that is passed by the utofu_poll_mrq() function. A value within utofu_onesided_caps::max_edata_size bytes can be specified.	IN
flags	Bitwise OR of UTOFU_ONESIDED_FLAG_*	IN
cbdata	Callback data that is passed by the utofu_poll_tcq() function	IN

Return values

Value	Explanation
UTOFU_SUCCESS	Succeeded
UTOFU_ERR_BUSY	The TOQ is full. Call the utofu_poll_tcq() function before trying again.
other	Other UTOFU_ERR_* error

3.7.1.4 utofu_put_stride_gap

Post multiple Put descriptors with a transmission gap to a TOQ.

Format

```
int utofu_put_stride_gap(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    utofu_stadd_t      lcl_stadd,
    utofu_stadd_t      rmt_stadd,
    size_t             length,
    size_t             stride,
    size_t             num_blocks,
    uint64_t           edata,
    unsigned long int  flags,
    size_t             mtu,
    size_t             gap,
    void              *cbdata)
```

Explanation

The parameters other than stride, num_blocks, mtu, and gap are the same as for the utofu_put() function.

Parameters

Parameter Name	Explanation	IN/OUT
vcq_hdl	Local VCQ handle. Given by the utofu_create_vcq() function in this process.	IN
rmt_vcq_id	Remote VCQ ID. Given by the utofu_query_vcq_id() function in a remote process.	IN
lcl_stadd	STADD of the start address of the local (data source) memory region. Given by the utofu_reg_mem() function in this process.	IN
rmt_stadd	STADD of the start address of a remote (data destination) memory region.	IN

Parameter Name	Explanation	IN/OUT
	Given by the `utofu_reg_mem()` function in a remote process.	
`length`	Data length in bytes. A value up to `utofu_onesided_caps::max_putget_size` can be specified.	IN
`stride`	Stride length in bytes	IN
`num_blocks`	Number of Put descriptors	IN
`edata`	EDATA that is passed by the `utofu_poll_mrq()` function. A value within `utofu_onesided_caps::max_edata_size` bytes can be specified.	IN
`flags`	Bitwise OR of `UTOFU_ONESIDED_FLAG_*`	IN
`mtu`	MTU (maximum transfer unit) of a packet. A value up to `utofu_onesided_caps::max_mtu` can be specified.	IN
`gap`	Transmission gap injected between packets. A value up to `utofu_onesided_caps::max_gap` can be specified.	IN
`cbdata`	Callback data that is passed by the `utofu_poll_tcq()` function	IN

Return values

Value	Explanation
`UTOFU_SUCCESS`	Succeeded
`UTOFU_ERR_BUSY`	The TOQ is full. Call the `utofu_poll_tcq()` function before trying again.
other	Other `UTOFU_ERR_*` error

3.7.1.5 `utofu_put_piggyback`

Post a Put Piggyback descriptor to a TOQ.

Format

```
int utofu_put_piggyback(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    void              *lcl_data,
    utofu_stadd_t      rmt_stadd,
    size_t             length,
    uint64_t           edata,
    unsigned long int  flags,
    void              *cbdata)
```

Explanation

Data up to `utofu_onesided_caps::max_piggyback_size` bytes can be transferred.

The parameters other than `lcl_data` are the same as for the `utofu_put()` function.

Parameters

Parameter Name	Explanation	IN/OUT
`vcq_hdl`	Local VCQ handle. Given by the `utofu_create_vcq()` function in this process.	IN
`rmt_vcq_id`	Remote VCQ ID. Given by the `utofu_query_vcq_id()` function in a remote process.	IN

Parameter Name	Explanation	IN/OUT
lcl_data	Pointer to the local data. The memory can be reused safely after this function returns.	IN
rmt_stadd	STADD of the start address of a remote (data destination) memory region. Given by the utofu_reg_mem() function in a remote process.	IN
length	Data length in bytes. A value up to utofu_onesided_caps::max_piggyback_size can be specified.	IN
edata	EDATA that is passed by the utofu_poll_mrq() function. A value within utofu_onesided_caps::max_edata_size bytes can be specified.	IN
flags	Bitwise OR of UTOFU_ONESIDED_FLAG_*	IN
cbdata	Callback data that is passed by the utofu_poll_tcq() function	IN

Return values

Value	Explanation
UTOFU_SUCCESS	Succeeded
UTOFU_ERR_BUSY	The TOQ is full. Call the utofu_poll_tcq() function before trying again.
other	Other UTOFU_ERR_* error

3.7.1.6 utofu_put_piggyback8

Post a Put Piggyback descriptor to a TOQ (up to 8 bytes).

Format

```
int utofu_put_piggyback8(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    uint64_t           lcl_data,
    utofu_stadd_t      rmt_stadd,
    size_t             length,
    uint64_t           edata,
    unsigned long int  flags,
    void              *cbdata)
```

Explanation

Data of up to 8 bytes can be transferred. If the length of the data is less than 8 bytes, the less significant bits of lcl_data are transferred.

The parameters other than lcl_data are the same as for the utofu_put() function.

Parameters

Parameter Name	Explanation	IN/OUT
vcq_hdl	Local VCQ handle. Given by the utofu_create_vcq() function in this process.	IN
rmt_vcq_id	Remote VCQ ID. Given by the utofu_query_vcq_id() function in a remote process.	IN
lcl_data	Value of the local data	IN
rmt_stadd	STADD of the start address of a remote (data destination) memory region. Given by the utofu_reg_mem() function in a remote process	IN

Parameter Name	Explanation	IN/OUT
length	Data length in bytes. A value up to 8 can be specified.	IN
edata	EDATA that is passed by the utofu_poll_mrq() function. A value within utofu_onesided_caps::max_edata_size bytes can be specified.	IN
flags	Bitwise OR of UTOFU_ONESIDED_FLAG_*	IN
cbdata	Callback data that is passed by the utofu_poll_tcq() function	IN

Return values

Value	Explanation
UTOFU_SUCCESS	Succeeded
UTOFU_ERR_BUSY	The TOQ is full. Call the utofu_poll_tcq() function before trying again.
other	Other UTOFU_ERR_* error

3.7.1.7 utofu_get

Post a Get descriptor to a TOQ.

Format

```
int utofu_get(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    utofu_stadd_t      lcl_stadd,
    utofu_stadd_t      rmt_stadd,
    size_t             length,
    uint64_t           edata,
    unsigned long int  flags,
    void              *cbdata)
```

Parameters

Parameter Name	Explanation	IN/OUT
vcq_hdl	Local VCQ handle. Given by the utofu_create_vcq() function in this process.	IN
rmt_vcq_id	Remote VCQ ID. Given by the utofu_query_vcq_id() function in a remote process.	IN
lcl_stadd	STADD of the start address of the local (data destination) memory region. Given by the utofu_reg_mem() function in this process.	IN
rmt_stadd	STADD of the start address of a remote (data source) memory region. Given by the utofu_reg_mem() function in a remote process.	IN
length	Data length in bytes. A value up to utofu_onesided_caps::max_putget_size can be specified.	IN
edata	EDATA that is passed by the utofu_poll_mrq() function. A value within utofu_onesided_caps::max_edata_size bytes can be specified.	IN
flags	Bitwise OR of UTOFU_ONESIDED_FLAG_*	IN
cbdata	Callback data that is passed by the utofu_poll_tcq() function	IN

Return values

Value	Explanation
UTOFU_SUCCESS	Succeeded
UTOFU_ERR_BUSY	The TOQ is full. Call the utofu_poll_tcq() function before trying again.
other	Other UTOFU_ERR_* error

3.7.1.8 utofu_get_gap

Post a Get descriptor with a transmission gap to a TOQ.

Format

```
int utofu_get_gap(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    utofu_stadd_t      lcl_stadd,
    utofu_stadd_t      rmt_stadd,
    size_t             length,
    uint64_t           edata,
    unsigned long int  flags,
    size_t             mtu,
    size_t             gap,
    void              *cbdata)
```

Explanation

The parameters other than mtu and gap are the same as for the utofu_get() function.

Parameters

Parameter Name	Explanation	IN/OUT
vcq_hdl	Local VCQ handle. Given by the utofu_create_vcq() function in this process.	IN
rmt_vcq_id	Remote VCQ ID. Given by the utofu_query_vcq_id() function in a remote process.	IN
lcl_stadd	STADD of the start address of the local (data destination) memory region. Given by the utofu_reg_mem() function in this process.	IN
rmt_stadd	STADD of the start address of a remote (data source) memory region. Given by the utofu_reg_mem() function in a remote process.	IN
length	Data length in bytes. A value up to utofu_onesided_caps::max_putget_size can be specified.	IN
edata	EDATA that is passed by the utofu_poll_mrq() function. A value within utofu_onesided_caps::max_edata_size bytes can be specified.	IN
flags	Bitwise OR of UTOFU_ONESIDED_FLAG_*	IN
mtu	MTU (maximum transfer unit) of a packet. A value up to utofu_onesided_caps::max_mtu can be specified.	IN
gap	Transmission gap injected between packets. A value up to utofu_onesided_caps::max_gap can be specified.	IN
cbdata	Callback data that is passed by the utofu_poll_tcq() function	IN

Return values

Value	Explanation
UTOFU_SUCCESS	Succeeded
UTOFU_ERR_BUSY	The TOQ is full. Call the utofu_poll_tcq() function before trying again.
other	Other UTOFU_ERR_* error

3.7.1.9 utofu_get_stride

Post multiple Get descriptors to a TOQ.

Format

```
int utofu_get_stride(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    utofu_stadd_t      lcl_stadd,
    utofu_stadd_t      rmt_stadd,
    size_t             length,
    size_t             stride,
    size_t             num_blocks,
    uint64_t           edata,
    unsigned long int  flags,
    void              *cbdata)
```

Explanation

The parameters other than stride and num_blocks are the same as for the utofu_get() function.

Parameters

Parameter Name	Explanation	IN/OUT
vcq_hdl	Local VCQ handle. Given by the utofu_create_vcq() function in this process.	IN
rmt_vcq_id	Remote VCQ ID. Given by the utofu_query_vcq_id() function in a remote process.	IN
lcl_stadd	STADD of the start address of the local (data destination) memory region. Given by the utofu_reg_mem() function in this process.	IN
rmt_stadd	STADD of the start address of a remote (data source) memory region. Given by the utofu_reg_mem() function in a remote process.	IN
length	Data length in bytes. A value up to utofu_onesided_caps::max_putget_size can be specified.	IN
stride	Stride length in bytes	IN
num_blocks	Number of Get descriptors	IN
edata	EDATA that is passed by the utofu_poll_mrq() function. A value within utofu_onesided_caps::max_edata_size bytes can be specified.	IN
flags	Bitwise OR of UTOFU_ONESIDED_FLAG_*	IN
cbdata	Callback data that is passed by the utofu_poll_tcq() function	IN

Return values

Value	Explanation
UTOFU_SUCCESS	Succeeded
UTOFU_ERR_BUSY	The TOQ is full. Call the utofu_poll_tcq() function before trying again.
other	Other UTOFU_ERR_* error

3.7.1.10 utofu_get_stride_gap

Post multiple Get descriptors with a transmission gap to a TOQ.

Format

```
int utofu_get_stride_gap(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    utofu_stadd_t      lcl_stadd,
    utofu_stadd_t      rmt_stadd,
    size_t             length,
    size_t             stride,
    size_t             num_blocks,
    uint64_t           edata,
    unsigned long int  flags,
    size_t             mtu,
    size_t             gap,
    void              *cbdata)
```

Explanation

The parameters other than stride, num_blocks, mtu, and gap are the same as for the utofu_get() function.

Parameters

Parameter Name	Explanation	IN/OUT
vcq_hdl	Local VCQ handle. Given by the utofu_create_vcq() function in this process.	IN
rmt_vcq_id	Remote VCQ ID. Given by the utofu_query_vcq_id() function in a remote process.	IN
lcl_stadd	STADD of the start address of the local (data destination) memory region. Given by the utofu_reg_mem() function in this process.	IN
rmt_stadd	STADD of the start address of a remote (data source) memory region. Given by the utofu_reg_mem() function in a remote process.	IN
length	Data length in bytes. A value up to utofu_onesided_caps::max_putget_size can be specified.	IN
stride	Stride length in bytes	IN
num_blocks	Number of Get descriptors	IN
edata	EDATA that is passed by the utofu_poll_mrq() function. A value within utofu_onesided_caps::max_edata_size bytes can be specified.	IN
flags	Bitwise OR of UTOFU_ONESIDED_FLAG_*	IN
mtu	MTU (maximum transfer unit) of a packet. A value up to utofu_onesided_caps::max_mtu can be specified.	IN
gap	Transmission gap injected between packets.	IN

Parameter Name	Explanation	IN/OUT
	A value up to `utofu_onesided_caps::max_gap` can be specified.	
`cbdata`	Callback data that is passed by the `utofu_poll_tcq()` function	IN

Return values

Value	Explanation
`UTOFU_SUCCESS`	Succeeded
`UTOFU_ERR_BUSY`	The TOQ is full. Call the `utofu_poll_tcq()` function before trying again.
other	Other `UTOFU_ERR_*` error

3.7.1.11 `utofu_armw4`

Post a 32-bit ARMW descriptor to a TOQ (except Compare and Swap).

Format

```
int utofu_armw4(
    utofu_vcq_hdl_t     vcq_hdl,
    utofu_vcq_id_t      rmt_vcq_id,
    enum utofu_armw_op  armw_op,
    uint32_t            op_value,
    utofu_stadd_t       rmt_stadd,
    uint64_t            edata,
    unsigned long int   flags,
    void               *cbdata)
```

Parameters

Parameter Name	Explanation	IN/OUT
`vcq_hdl`	Local VCQ handle. Given by the `utofu_create_vcq()` function in this process.	IN
`rmt_vcq_id`	Remote VCQ ID. Given by the `utofu_query_vcq_id()` function in a remote process.	IN
`armw_op`	ARMW operation type	IN
`op_value`	Local operand	IN
`rmt_stadd`	STADD of the start address of a remote (data read/modify/write) memory region. Given by the `utofu_reg_mem()` function in a remote process. The remote memory address must be aligned to 4 bytes.	IN
`edata`	EDATA that is passed by the `utofu_poll_mrq()` function. A value within `utofu_onesided_caps::max_edata_size` bytes can be specified.	IN
`flags`	Bitwise OR of `UTOFU_ONESIDED_FLAG_*`	IN
`cbdata`	Callback data that is passed by the `utofu_poll_tcq()` function	IN

Return values

Value	Explanation
`UTOFU_SUCCESS`	Succeeded
`UTOFU_ERR_BUSY`	The TOQ is full. Call the `utofu_poll_tcq()` function before trying again.

Value	Explanation
other	Other UTOFU_ERR_* error

3.7.1.12 utofu_armw8

Post a 64-bit ARMW descriptor to a TOQ (except Compare and Swap).

Format

```
int utofu_armw8(
    utofu_vcq_hdl_t     vcq_hdl,
    utofu_vcq_id_t      rmt_vcq_id,
    enum utofu_armw_op  armw_op,
    uint64_t            op_value,
    utofu_stadd_t       rmt_stadd,
    uint64_t            edata,
    unsigned long int   flags,
    void               *cbdata)
```

Parameters

Parameter Name	Explanation	IN/OUT
vcq_hdl	Local VCQ handle. Given by the utofu_create_vcq() function in this process.	IN
rmt_vcq_id	Remote VCQ ID. Given by the utofu_query_vcq_id() function in a remote process.	IN
armw_op	ARMW operation type	IN
op_value	Local operand	IN
rmt_stadd	STADD of the start address of a remote (data read/modify/write) memory region. Given by the utofu_reg_mem() function in a remote process. The remote memory address must be aligned to 8 bytes.	IN
edata	EDATA that is passed by the utofu_poll_mrq() function. A value within utofu_onesided_caps::max_edata_size bytes can be specified.	IN
flags	Bitwise OR of UTOFU_ONESIDED_FLAG_*	IN
cbdata	Callback data that is passed by the utofu_poll_tcq() function	IN

Return values

Value	Explanation
UTOFU_SUCCESS	Succeeded
UTOFU_ERR_BUSY	The TOQ is full. Call the utofu_poll_tcq() function before trying again.
other	Other UTOFU_ERR_* error

3.7.1.13 utofu_cswap4

Post a 32-bit Compare and Swap ARMW descriptor to a TOQ.

Format

```
int utofu_cswap4(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
```

```
    uint32_t           old_value,
    uint32_t           new_value,
    utofu_stadd_t      rmt_stadd,
    uint64_t           edata,
    unsigned long int  flags,
    void              *cbdata)
```

Parameters

Parameter Name	Explanation	IN/OUT
vcq_hdl	Local VCQ handle. Given by the utofu_create_vcq() function in this process.	IN
rmt_vcq_id	Remote VCQ ID. Given by the utofu_query_vcq_id() function in a remote process.	IN
old_value	Old value (compare operand)	IN
new_value	New value (swap operand)	IN
rmt_stadd	STADD of the start address of a remote (data read/modify/write) memory region. Given by the utofu_reg_mem() function in a remote process. The remote memory address must be aligned to 4 bytes.	IN
edata	EDATA that is passed by the utofu_poll_mrq() function. A value within utofu_onesided_caps::max_edata_size bytes can be specified.	IN
flags	Bitwise OR of UTOFU_ONESIDED_FLAG_*	IN
cbdata	Callback data that is passed by the utofu_poll_tcq() function	IN

Return values

Value	Explanation
UTOFU_SUCCESS	Succeeded
UTOFU_ERR_BUSY	The TOQ is full. Call the utofu_poll_tcq() function before trying again.
other	Other UTOFU_ERR_* error

3.7.1.14 utofu_cswap8

Post a 64-bit Compare and Swap ARMW descriptor to a TOQ.

Format

```
int utofu_cswap8(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    uint64_t           old_value,
    uint64_t           new_value,
    utofu_stadd_t      rmt_stadd,
    uint64_t           edata,
    unsigned long int  flags,
    void              *cbdata)
```

Parameters

Parameter Name	Explanation	IN/OUT
vcq_hdl	Local VCQ handle. Given by the utofu_create_vcq() function in this process.	IN

Parameter Name	Explanation	IN/OUT
rmt_vcq_id	Remote VCQ ID. Given by the utofu_query_vcq_id() function in a remote process.	IN
old_value	Old value (compare operand)	IN
new_value	New value (swap operand)	IN
rmt_stadd	STADD of the start address of a remote (data read/modify/write) memory region. Given by the utofu_reg_mem() function in a remote process. The remote memory address must be aligned to 8 bytes.	IN
edata	EDATA that is passed by the utofu_poll_mrq() function. A value within utofu_onesided_caps::max_edata_size bytes can be specified.	IN
flags	Bitwise OR of UTOFU_ONESIDED_FLAG_*	IN
cbdata	Callback data that is passed by the utofu_poll_tcq() function	IN

Return values

Value	Explanation
UTOFU_SUCCESS	Succeeded
UTOFU_ERR_BUSY	The TOQ is full. Call the utofu_poll_tcq() function before trying again.
other	Other UTOFU_ERR_* error

3.7.1.15 utofu_nop

Post a NOP descriptor to a TOQ.

Format

```
int utofu_nop(
    utofu_vcq_hdl_t    vcq_hdl,
    unsigned long int  flags,
    void              *cbdata)
```

Parameters

Parameter Name	Explanation	IN/OUT
vcq_hdl	Local VCQ handle. Given by the utofu_create_vcq() function in this process.	IN
flags	Bitwise OR of UTOFU_ONESIDED_FLAG_*	IN
cbdata	Callback data that is passed by the utofu_poll_tcq() function	IN

Return values

Value	Explanation
UTOFU_SUCCESS	Succeeded
UTOFU_ERR_BUSY	The TOQ is full. Call the utofu_poll_tcq() function before trying again.
other	Other UTOFU_ERR_* error

3.7.2 One-Sided Communication Preparation Functions

A one-sided communication preparation function writes a TOQ descriptor to the memory specified as an argument.

All the one-sided communication preparation functions `utofu_prepare_*()` have the corresponding one-sided communication start functions `utofu_*()`. In comparison to the corresponding one-sided communication start function, the `cbdata` parameter is removed and the `desc` and `desc_size` parameters are added.

Described below, the `desc` and `desc_size` parameters and return values are the same among all one-sided communication preparation functions. Therefore, explanations of parameters and return values are omitted from the explanation of each one-sided communication preparation function.

Parameters

Parameter Name	Explanation	IN/OUT
`desc`	Written descriptor. For `utofu_prepare_*_stride()` functions, the byte size of memory must be "`utofu_onesided_caps::max_toq_desc_size * num_blocks`" or greater. For other functions, the byte size of memory must be `utofu_onesided_caps::max_toq_desc_size` or greater. The memory address must be aligned to 8 bytes.	OUT
`desc_size`	Byte size of the written descriptor. The value may be smaller than "`utofu_onesided_caps::max_toq_desc_size * num_blocks`" or `utofu_onesided_caps::max_toq_desc_size`.	OUT

Return values

Value	Explanation
`UTOFU_SUCCESS`	Succeeded
other	Other `UTOFU_ERR_*` error

3.7.2.1 `utofu_prepare_put`

Prepare a Put descriptor.

Format

```
int utofu_prepare_put(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    utofu_stadd_t      lcl_stadd,
    utofu_stadd_t      rmt_stadd,
    size_t             length,
    uint64_t           edata,
    unsigned long int  flags,
    void              *desc,
    size_t            *desc_size)
```

3.7.2.2 `utofu_prepare_put_gap`

Prepare a Put descriptor with a transmission gap.

Format

```
int utofu_prepare_put_gap(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    utofu_stadd_t      lcl_stadd,
    utofu_stadd_t      rmt_stadd,
    size_t             length,
```

```
    uint64_t           edata,
    unsigned long int  flags,
    size_t             mtu,
    size_t             gap,
    void              *desc,
    size_t            *desc_size)
```

3.7.2.3 utofu_prepare_put_stride

Prepare multiple Put descriptors.

Format

```
int utofu_prepare_put_stride(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    utofu_stadd_t      lcl_stadd,
    utofu_stadd_t      rmt_stadd,
    size_t             length,
    size_t             stride,
    size_t             num_blocks,
    uint64_t           edata,
    unsigned long int  flags,
    void              *desc,
    size_t            *desc_size)
```

3.7.2.4 utofu_prepare_put_stride_gap

Prepare multiple Put descriptors with a transmission gap.

Format

```
int utofu_prepare_put_stride_gap(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    utofu_stadd_t      lcl_stadd,
    utofu_stadd_t      rmt_stadd,
    size_t             length,
    size_t             stride,
    size_t             num_blocks,
    uint64_t           edata,
    unsigned long int  flags,
    size_t             mtu,
    size_t             gap,
    void              *desc,
    size_t            *desc_size)
```

3.7.2.5 utofu_prepare_put_piggyback

Prepare a Put Piggyback descriptor.

Format

```
int utofu_prepare_put_piggyback(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    void              *lcl_data,
    utofu_stadd_t      rmt_stadd,
    size_t             length,
    uint64_t           edata,
    unsigned long int  flags,
    void              *desc,
    size_t            *desc_size)
```

3.7.2.6 utofu_prepare_put_piggyback8

Prepare a Put Piggyback descriptor (up to 8 bytes).

Format

```
int utofu_prepare_put_piggyback8(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    uint64_t           lcl_data,
    utofu_stadd_t      rmt_stadd,
    size_t             length,
    uint64_t           edata,
    unsigned long int  flags,
    void              *desc,
    size_t            *desc_size)
```

3.7.2.7 utofu_prepare_get

Prepare a Get descriptor.

Format

```
int utofu_prepare_get(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    utofu_stadd_t      lcl_stadd,
    utofu_stadd_t      rmt_stadd,
    size_t             length,
    uint64_t           edata,
    unsigned long int  flags,
    void              *desc,
    size_t            *desc_size)
```

3.7.2.8 utofu_prepare_get_gap

Prepare a Get descriptor with a transmission gap.

Format

```
int utofu_prepare_get_gap(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    utofu_stadd_t      lcl_stadd,
    utofu_stadd_t      rmt_stadd,
    size_t             length,
    uint64_t           edata,
    unsigned long int  flags,
    size_t             mtu,
    size_t             gap,
    void              *desc,
    size_t            *desc_size)
```

3.7.2.9 utofu_prepare_get_stride

Prepare multiple Get descriptors.

Format

```
int utofu_prepare_get_stride(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    utofu_stadd_t      lcl_stadd,
    utofu_stadd_t      rmt_stadd,
```

```
    size_t              length,
    size_t              stride,
    size_t              num_blocks,
    uint64_t            edata,
    unsigned long int   flags,
    void               *desc,
    size_t             *desc_size)
```

3.7.2.10 utofu_prepare_get_stride_gap

Prepare multiple Get descriptors with a transmission gap.

Format

```
int utofu_prepare_get_stride_gap(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    utofu_stadd_t      lcl_stadd,
    utofu_stadd_t      rmt_stadd,
    size_t             length,
    size_t             stride,
    size_t             num_blocks,
    uint64_t           edata,
    unsigned long int  flags,
    size_t             mtu,
    size_t             gap,
    void              *desc,
    size_t            *desc_size)
```

3.7.2.11 utofu_prepare_armw4

Prepare a 32-bit ARMW descriptor.

Format

```
int utofu_prepare_armw4(
    utofu_vcq_hdl_t     vcq_hdl,
    utofu_vcq_id_t      rmt_vcq_id,
    enum utofu_armw_op  armw_op,
    uint32_t            op_value,
    utofu_stadd_t       rmt_stadd,
    uint64_t            edata,
    unsigned long int   flags,
    void               *desc,
    size_t             *desc_size)
```

3.7.2.12 utofu_prepare_armw8

Prepare a 64-bit ARMW descriptor.

Format

```
int utofu_prepare_armw8(
    utofu_vcq_hdl_t     vcq_hdl,
    utofu_vcq_id_t      rmt_vcq_id,
    enum utofu_armw_op  armw_op,
    uint64_t            op_value,
    utofu_stadd_t       rmt_stadd,
    uint64_t            edata,
    unsigned long int   flags,
    void               *desc,
    size_t             *desc_size)
```

3.7.2.13 utofu_prepare_cswap4

Prepare a 32-bit Compare and Swap ARMW descriptor.

Format

```
int utofu_prepare_cswap4(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    uint32_t           old_value,
    uint32_t           new_value,
    utofu_stadd_t      rmt_stadd,
    uint64_t           edata,
    unsigned long int  flags,
    void              *desc,
    size_t            *desc_size)
```

3.7.2.14 utofu_prepare_cswap8

Prepare a 64-bit Compare and Swap ARMW descriptor.

Format

```
int utofu_prepare_cswap8(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    uint64_t           old_value,
    uint64_t           new_value,
    utofu_stadd_t      rmt_stadd,
    uint64_t           edata,
    unsigned long int  flags,
    void              *desc,
    size_t            *desc_size)
```

3.7.2.15 utofu_prepare_nop

Prepare a NOP descriptor.

Format

```
int utofu_prepare_nop(
    utofu_vcq_hdl_t    vcq_hdl,
    unsigned long int  flags,
    void              *desc,
    size_t            *desc_size)
```

3.7.3 One-Sided Communication Batch Start Functions

3.7.3.1 utofu_post_toq

Post descriptors to a TOQ to start the communication at once.

Format

```
int utofu_post_toq(
    utofu_vcq_hdl_t  vcq_hdl,
    void            *desc,
    size_t           desc_size,
    void            *cbdata)
```

Explanation

This function posts a TOQ descriptor to a TOQ to start one-sided communication. A `utofu_prepare_*()` function can create the TOQ descriptor for posting.

If a `utofu_prepare_*()` function is called multiple times and the TOQ descriptors are written to contiguous memory regions, passing the memory region to this function can start those multiple communications at once. Also, communications of different types can be started.

This function also starts one-sided communications that have not yet started due to the `UTOFU_ONESIDED_FLAG_DELAY_START` flag.

Parameters

Parameter Name	Explanation	IN/OUT
`vcq_hdl`	Local VCQ handle. Given by the `utofu_create_vcq()` function in this process.	IN
`desc`	Descriptors created by `utofu_prepare_*()` functions. All local VCQ handles specified for those `utofu_prepare_*()` functions must be same as the VCQ handle specified for this function. This parameter is ignored if `desc_size` is 0.	IN
`desc_size`	Total byte size of descriptors. If 0, this function is starts only the one-sided communications that have not yet started due to the `UTOFU_ONESIDED_FLAG_DELAY_START` flag.	IN
`cbdata`	Callback data that is passed by the `utofu_poll_tcq()` function. This parameter is ignored if `desc_size` is 0. The same callback data is used for all communications if multiple TOQ descriptors are posted.	IN

Return values

Value	Explanation
`UTOFU_SUCCESS`	Succeeded
`UTOFU_ERR_BUSY`	The TOQ is full. Call the `utofu_poll_tcq()` function before trying again.
other	Other `UTOFU_ERR_*` error

3.7.4 One-Sided Communication Completion Confirmation Functions

3.7.4.1 `utofu_poll_tcq`

Poll a TCQ.

Format

```
int utofu_poll_tcq(
    utofu_vcq_hdl_t     vcq_hdl,
    unsigned long int   flags,
    void              **cbdata)
```

Explanation

This function returns `UTOFU_SUCCESS` and sets `cbdata` when a new TCQ descriptor for the specified VCQ is found. The function returns `UTOFU_ERR_NOT_FOUND` when no new TCQ descriptor for the specified VCQ is found.

The order of TCQ descriptors is the same as the order of TOQ descriptors posted with the `UTOFU_ONESIDED_FLAG_TCQ_NOTICE` flag.

Parameters

Parameter Name	Explanation	IN/OUT
`vcq_hdl`	Local VCQ handle	IN

Parameter Name	Explanation	IN/OUT
flags	Bitwise OR of UTOFU_POLL_FLAG_*	IN
cbdata	Pointer to the storage for storing the callback data that was passed to the utofu_put() function, etc. The callback data is stored only if this function returns a code other than UTOFU_ERR_NOT_FOUND.	OUT

Return values

Value	Explanation
UTOFU_SUCCESS	A new TCQ descriptor for successfully completed communication was found.
UTOFU_ERR_NOT_FOUND	No new TCQ descriptor was found.
UTOFU_ERR_TCQ_*	A new TCQ descriptor for communication with an error was found.
other	Other UTOFU_ERR_* error

3.7.4.2 utofu_poll_mrq

Poll an MRQ.

Format

```
int utofu_poll_mrq(
    utofu_vcq_hdl_t          vcq_hdl,
    unsigned long int        flags,
    struct utofu_mrq_notice *notice)
```

Explanation

This function returns UTOFU_SUCCESS and writes data in notice when a new MRQ descriptor for the specified VCQ is found. The function returns UTOFU_ERR_NOT_FOUND when no new MRQ descriptor for the specified VCQ is found.

This function handles both local notification MRQ descriptors and remote notification MRQ descriptors. The type of completed communication can be confirmed using notice.

Parameters

Parameter Name	Explanation	IN/OUT
vcq_hdl	Local VCQ handle	IN
flags	Bitwise OR of UTOFU_POLL_FLAG_*	IN
notice	MRQ notification. MRQ notification is set only when this function returns a code other than UTOFU_ERR_NOT_FOUND.	OUT

Return values

Value	Explanation
UTOFU_SUCCESS	A new MRQ descriptor for successfully completed communication was found.
UTOFU_ERR_NOT_FOUND	No new MRQ descriptor was found.
UTOFU_ERR_MRQ_*	A new MRQ descriptor for communication with an error was found.
other	Other UTOFU_ERR_* error

3.7.5 One-Sided Communication Query Functions

3.7.5.1 utofu_query_num_unread_tcq

Query the number of unread TCQ descriptors.

Format

```
int utofu_query_num_unread_tcq(
    utofu_vcq_hdl_t  vcq_hdl,
    size_t          *count)
```

Explanation

This function can be used to estimate how busy the TNI is now.

This function returns the number of written TOQ descriptors whose corresponding TCQ descriptors have not been read by the uTofu implementation. This number is approximately equal to the number of one-sided communications that are scheduled but not yet completed. If the function returns 0, the next one-sided communication start function call can start the communications immediately.

Though this function requires a VCQ handle as a parameter, the number of descriptors is counted per CQ, not VCQ. This is because a TNI processes TOQ descriptors in order per CQ. So if a CQ is shared among some VCQs, the returned number may be different from the number managed per VCQ.

Parameters

Parameter Name	Explanation	IN/OUT
vcq_hdl	Local VCQ handle	IN
count	Number of unread TCQ descriptors	OUT

Return values

Value	Explanation
UTOFU_SUCCESS	Succeeded
other	Other UTOFU_ERR_* error

3.7.6 Communication Completion Notification Structures

3.7.6.1 struct utofu_mrq_notice

Contents of MRQ notification

Definition

```
struct utofu_mrq_notice {
    uint8_t        notice_type;
    uint8_t        padding1[7];
    utofu_vcq_id_t vcq_id;
    uint64_t       edata;
    uint64_t       rmt_value;
    utofu_stadd_t  lcl_stadd;
    utofu_stadd_t  rmt_stadd;
    uint64_t       reserved[2];
}
```

Explanation

The utofu_poll_mrq() function writes this structure.

Members

Member Name	Explanation
notice_type	MRQ notification type. The value is one of the enumeration constants UTOFU_MRQ_TYPE_* in the utofu_mrq_notice_type enumeration.
padding1	Padding field. Do not use this field.
vcq_id	VCQ ID of the communication peer. The value is a remote (communication target) VCQ ID for local notification and the local (communication origin) VCQ ID for remote notification. Communication path coordinates embedded in this VCQ ID are the A,B,C coordinates of the compute node, which are the same as the ones obtained by the utofu_query_vcq_id() and utofu_construct_vcq_id() functions. So if other default communication path coordinates are embedded by the utofu_set_vcq_id_path() function, this value may differ from the remote VCQ ID specified in one-sided communication.
edata	Value of the EDATA specified in the corresponding one-sided communication start/preparation function call
rmt_value	Value in remote memory before the update. The more significant bits are 0 if 32-bit ARMW is used. This field is set only for UTOFU_MRQ_TYPE_LCL_ARMW.
lcl_stadd	Local (communication origin) STADD. This STADD points to the end of data ("lcl_stadd + length" when posting a TOQ descriptor) for Put/Get and the beginning of data (lcl_stadd when posting a TOQ descriptor) for ARMW. This field is set only for UTOFU_MRQ_TYPE_{LCL|RMT}_GET.
rmt_stadd	Remote (communication target) STADD. This STADD points to the end of data ("rmt_stadd + length" when posting a TOQ descriptor) for Put/Get and the beginning of data (rmt_stadd when posting a TOQ descriptor) for ARMW. This field is set only for UTOFU_MRQ_TYPE_{LCL|RMT}_PUT, UTOFU_MRQ_TYPE_RMT_GET, and UTOFU_MRQ_TYPE_{LCL|RMT}_ARMW.
reserved	Reserved field for future extensions. Do not use this field.

3.7.7 ARMW Operation Types

3.7.7.1 enum utofu_armw_op

ARMW (Atomic Read Modify Write) operation types of one-sided communication

Explanation

These enumeration constants are used for the arguments of utofu_armw4(), utofu_armw8(), and their corresponding utofu_prepare_*() functions to start one-sided communication.

Enumeration Constants

Enumeration Constant	Explanation
UTOFU_ARMW_OP_SWAP	Swap operation
UTOFU_ARMW_OP_ADD	Unsigned integer addition operation
UTOFU_ARMW_OP_XOR	Bitwise XOR operation
UTOFU_ARMW_OP_AND	Bitwise AND operation
UTOFU_ARMW_OP_OR	Bitwise OR operation

3.7.8 Communication Completion Notification Types

3.7.8.1 enum utofu_mrq_notice_type

MRQ descriptor types

Explanation

These enumeration constants are set in `utofu_mrq_notice::notice_type` when a new MRQ descriptor is found on the `utofu_poll_mrq()` function.

Enumeration Constants

Enumeration Constant	Explanation
UTOFU_MRQ_TYPE_LCL_PUT	Local MRQ notification of the Put communication
UTOFU_MRQ_TYPE_RMT_PUT	Remote MRQ notification of the Put communication
UTOFU_MRQ_TYPE_LCL_GET	Local MRQ notification of the Get communication
UTOFU_MRQ_TYPE_RMT_GET	Remote MRQ notification of the Get communication
UTOFU_MRQ_TYPE_LCL_ARMW	Local MRQ notification of the ARMW communication
UTOFU_MRQ_TYPE_RMT_ARMW	Remote MRQ notification of the ARMW communication

3.7.9 One-Sided Communication Flags

3.7.9.1 UTOFU_ONESIDED_FLAG_*

Bit flags for the `flags` parameter of one-sided communication start functions and one-sided communication preparation functions.

Explanation

They can specify behaviors of one-sided communication.

Macros

Macro Name	Explanation
UTOFU_ONESIDED_FLAG_TCQ_NOTICE	TCQ notification. This flag orders the `utofu_poll_tcq()` function to return the corresponding TCQ notification.
UTOFU_ONESIDED_FLAG_REMOTE_MRQ_NOTICE	Remote MRQ notification. This flag orders the `utofu_poll_mrq()` function to return the corresponding remote MRQ notification.
UTOFU_ONESIDED_FLAG_LOCAL_MRQ_NOTICE	Local MRQ notification. This flag orders the `utofu_poll_mrq()` function to return the corresponding local MRQ notification.
UTOFU_ONESIDED_FLAG_STRONG_ORDER	Strong order. This flag ensures that this one-sided communication reads from memory or writes to memory after prior one-sided communications with the same local VCQ, same remote VCQ, and same communication path have read from memory or written to memory.
UTOFU_ONESIDED_FLAG_CACHE_INJECTION	Cache injection. This flag orders the local/remote TNI to write data to the last level cache of the CPU in addition to memory.
UTOFU_ONESIDED_FLAG_PADDING	Cache line padding.

Macro Name	Explanation
	This flag orders the remote TNI to write indefinite values in an area that is not the Put Piggyback target area but is on the target cache line. Used only for Put Piggyback, the flag works with the `UTOFU_ONESIDED_FLAG_CACHE_INJECTION` flag to use the cache injection feature with Put Piggyback.
`UTOFU_ONESIDED_FLAG_DELAY_START`	Delay starting communication. This flag orders a one-sided communication start function not to start communication immediately and to put it in a pending state if possible. The next one-sided communication start function call for the same VCQ without the flag or a `utofu_post_toq()` function call for the same VCQ starts the pending communication. The flag may be ignored if the TOQ is full due to the pending one-sided communication or for other reasons. The flag can be used to speed up repeated one-sided communication start function calls because starting communication incurs some costs. Because a one-sided communication start function may start communication even if the flag is specified, the communication must be ready to start when the function is called. For example, a remote VCQ must have been created and the source data must have been written to memory. The flag does not have an effect on one-sided communication preparation functions because these functions by their nature do not start communications.

3.7.9.2 UTOFU_ONESIDED_FLAG_PATH

Bit flags for specifying a communication path by the `flags` parameter of a one-sided communication start/preparation function

Macro

```
UTOFU_ONESIDED_FLAG_PATH(path_id)
```

Explanation

To call a one-sided communication start/preparation function, a communication path can be specified by either a remote VCQ ID (`rmt_vcq_id` parameter) or bit flags (`flags` parameter).

This function-like macro gives a communication path value that can be set in the `flags` parameter of a one-sided communication start/preparation function. This bit flag can be combined with other `UTOFU_ONESIDED_FLAG_*` macros by bitwise OR.

Parameters

Parameter Name	Explanation	IN/OUT
`path_id`	Communication path ID created by the `utofu_get_path_id()` function	IN

3.7.9.3 UTOFU_ONESIDED_FLAG_SPS

Bit flags for specifying a session progress step (SPS) by the `flags` parameter of a one-sided communication start/preparation function

Macro

```
UTOFU_ONESIDED_FLAG_SPS(sps)
```

Explanation

To call a one-sided communication start/preparation function for a remote VCQ created as a session mode VCQ, a session progress step (SPS) must be specified by the `flags` parameter.

This function-like macro gives an SPS value that can be set in the `flags` parameter of a one-sided communication start/preparation function. This bit flag can be combined with other `UTOFU_ONESIDED_FLAG_*` macros by bitwise OR.

If the remote VCQ was not created as a session mode VCQ, this flag is ignored. If the remote VCQ was created as a session mode VCQ but no SPS is set in the `flags` parameter of the one-sided communication start/preparation function, the SPS is treated as 0.

Parameters

Parameter Name	Explanation	IN/OUT
`sps`	SPS of the session mode	IN

3.7.10 Polling Flags

3.7.10.1 UTOFU_POLL_FLAG_*

Bit flags for specifying by the `flags` parameter of one-sided communication completion confirmation functions or barrier communication completion confirmation functions

Explanation

They can specify behaviors of polling functions.

No flags are defined in the current version.

3.8 Barrier Communication Execution

Barrier communication is started by a barrier communication start function and its completion can be confirmed by the barrier communication completion confirmation function corresponding to the barrier communication start function used. When a reduction operation is executed with barrier synchronization, the input data is passed to a barrier communication start function, and the corresponding output data (operation results) is obtained from a barrier communication completion confirmation function.

Only one barrier communication is performed with one barrier circuit built by the `utofu_set_vbg()` function. Before starting the next barrier communication, completion of the previous barrier communication must be confirmed. Using multiple barrier circuits, multiple barrier communications can be performed at the same time.

In one barrier communication, all processes participating in the barrier communication must call the same barrier communication start function and specify the same reduction operation type. If not, the corresponding barrier communication completion confirmation function returns `UTOFU_ERR_BARRIER_MISMATCH`.

The first parameter `vbg_id` of barrier communication start functions and barrier communication completion confirmation functions determines which barrier circuit is used. The VBG ID must be the first local VBG ID in the VBG ID array returned by the `utofu_alloc_vbg()` function. It corresponds to a start/end BG.

3.8.1 Barrier Communication Start Functions

3.8.1.1 utofu_barrier

Start barrier synchronization on a barrier circuit.

Format

```
int utofu_barrier(
    utofu_vbg_id_t    vbg_id,
    unsigned long int flags)
```

Explanation

This function is nonblocking. The `utofu_poll_barrier()` function can confirm the completion of the barrier synchronization.

Parameters

Parameter Name	Explanation	IN/OUT
`vbg_id`	Local first VBG ID. Given by the `utofu_alloc_vbg()` function in this process.	IN

Parameter Name	Explanation	IN/OUT
flags	Bitwise OR of UTOFU_BARRIER_FLAG_*. All processes participating in the barrier communication must specify the same value.	IN

Return values

Value	Explanation
UTOFU_SUCCESS	Succeeded
UTOFU_ERR_BUSY	Barrier communication has been already started on the barrier circuit.
other	Other UTOFU_ERR_* error

3.8.1.2 utofu_reduce_uint64

Start reduction operation of uint64_t values on a barrier circuit.

Format

```
int utofu_reduce_uint64(
    utofu_vbg_id_t       vbg_id,
    enum utofu_reduce_op op,
    uint64_t             data[],
    size_t               num_data,
    unsigned long int    flags)
```

Explanation

This function is nonblocking. The utofu_poll_reduce_uint64() function can confirm the completion of the reduction operation.

Parameters

Parameter Name	Explanation	IN/OUT
vbg_id	Local first VBG ID. Given by the utofu_alloc_vbg() function in this process.	IN
op	Reduction operation type. Only operations for uint64_t are allowed. All processes participating in the barrier communication must specify the same value.	IN
data	Array of reduction input data. The array length must be num_data.	IN
num_data	Reduction input data count. The value must be in a range from 1 to utofu_barrier_caps::max_uint64_reduction. All processes participating in the barrier communication must specify the same value.	IN
flags	Bitwise OR of UTOFU_BARRIER_FLAG_*. All processes participating in the barrier communication must specify the same value.	IN

Return values

Value	Explanation
UTOFU_SUCCESS	Succeeded
UTOFU_ERR_BUSY	Barrier communication has been already started on the barrier circuit.
other	Other UTOFU_ERR_* error

3.8.1.3 utofu_reduce_double

Start reduction operation of `double` values on a barrier circuit.

Format

```
int utofu_reduce_double(
    utofu_vbg_id_t       vbg_id,
    enum utofu_reduce_op op,
    double               data[],
    size_t               num_data,
    unsigned long int    flags)
```

Explanation

This function is nonblocking. The `utofu_poll_double()` function can confirm the completion of the reduction operation.

Parameters

Parameter Name	Explanation	IN/OUT
`vbg_id`	Local first VBG ID. Given by the `utofu_alloc_vbg()` function in this process.	IN
`op`	Reduction operation type. Only operations for `double` are allowed. All processes participating in the barrier communication must specify the same value.	IN
`data`	Array of reduction input data. The array length must be `num_data`.	IN
`num_data`	Reduction input data count. The value must be in a range from 1 to `utofu_barrier_caps::max_double_reduction`. All processes participating in the barrier communication must specify the same value.	IN
`flags`	Bitwise OR of `UTOFU_BARRIER_FLAG_*`. All processes participating in the barrier communication must specify the same value.	IN

Return values

Value	Explanation
`UTOFU_SUCCESS`	Succeeded
`UTOFU_ERR_BUSY`	Barrier communication has been already started on the barrier circuit.
other	Other `UTOFU_ERR_*` error

3.8.2 Barrier Communication Completion Confirmation Functions

3.8.2.1 utofu_poll_barrier

Poll the completion of barrier synchronization.

Format

```
int utofu_poll_barrier(
    utofu_vbg_id_t    vbg_id,
    unsigned long int flags)
```

Explanation

Before this function is called, the `utofu_barrier()` function must have started barrier synchronization.

This function is nonblocking. The function returns `UTOFU_SUCCESS` when the barrier synchronization is completed. Otherwise, it returns `UTOFU_ERR_NOT_COMPLETED` immediately.

Parameters

Parameter Name	Explanation	IN/OUT
`vbg_id`	Local first VBG ID. Given by the `utofu_alloc_vbg()` function in this process.	IN
`flags`	Bitwise OR of `UTOFU_POLL_FLAG_*`. All processes participating in the barrier communication must specify the same value.	IN

Return values

Value	Explanation
`UTOFU_SUCCESS`	Completed
`UTOFU_ERR_NOT_COMPLETED`	Not yet completed
`UTOFU_ERR_BUSY`	Barrier communication has not been started on the barrier circuit.
`UTOFU_ERR_BARRIER_*`	A barrier communication error occurred.
other	Other `UTOFU_ERR_*` error

3.8.2.2 `utofu_poll_reduce_uint64`

Poll the completion of reduction operation of `uint64_t` values.

Format

```
int utofu_poll_reduce_uint64(
    utofu_vbg_id_t    vbg_id,
    unsigned long int flags,
    uint64_t          data[])
```

Explanation

Before this function is called, the `utofu_reduce_uint64()` function must have started reduction operation.

This function is nonblocking. The function returns `UTOFU_SUCCESS` and fills result data when the reduction operation is completed. Otherwise, it returns `UTOFU_ERR_NOT_COMPLETED` immediately.

Parameters

Parameter Name	Explanation	IN/OUT
`vbg_id`	Local first VBG ID. Given by the `utofu_alloc_vbg()` function in this process.	IN
`flags`	Bitwise OR of `UTOFU_POLL_FLAG_*`. All processes participating in the barrier communication must specify the same value.	IN
`data`	Array of `uint64_t` reduction result data. The array length must be the `num_data` specified in the `utofu_reduce_uint64()` function call or greater. Only the first `num_data` elements are updated. The array contents are updated only when this function returns `UTOFU_SUCCESS`.	OUT

Return values

Value	Explanation
`UTOFU_SUCCESS`	Completed
`UTOFU_ERR_NOT_COMPLETED`	Not yet completed

Value	Explanation
UTOFU_ERR_BUSY	Barrier communication has not been started on the barrier circuit.
UTOFU_ERR_BARRIER_*	A barrier communication error occurred.
other	Other UTOFU_ERR_* error

3.8.2.3 utofu_poll_reduce_double

Poll the completion of reduction operation of double values.

Format

```
int utofu_poll_reduce_double(
    utofu_vbg_id_t    vbg_id,
    unsigned long int flags,
    double            data[])
```

Explanation

Before this function is called, the utofu_reduce_double() function must have started reduction operation.

This function is nonblocking. The function returns UTOFU_SUCCESS and fills result data when the reduction operation is completed. Otherwise, it returns UTOFU_ERR_NOT_COMPLETED immediately.

Parameters

Parameter Name	Explanation	IN/OUT
vbg_id	Local first VBG ID. Given by the utofu_alloc_vbg() function in this process.	IN
flags	Bitwise OR of UTOFU_POLL_FLAG_*. All processes participating in the barrier communication must specify the same value.	IN
data	Array of double reduction result data. The array length must be the num_data specified in the utofu_reduce_double() function call or greater. Only the first num_data elements are updated. The array contents are updated only when this function returns UTOFU_SUCCESS.	OUT

Return values

Value	Explanation
UTOFU_SUCCESS	Completed
UTOFU_ERR_NOT_COMPLETED	Not yet completed
UTOFU_ERR_BUSY	Barrier communication has not been started on the barrier circuit.
UTOFU_ERR_BARRIER_*	A barrier communication error occurred.
other	Other UTOFU_ERR_* error

3.8.3 Reduction Operation Types

3.8.3.1 enum utofu_reduce_op

Reduction operation types of barrier communication

Explanation

These enumeration constants are used for arguments of the utofu_reduce_uint64() and utofu_reduce_double() functions to start barrier communication.

Enumeration Constants

Enumeration Constant	Explanation
UTOFU_REDUCE_OP_BARRIER	Barrier synchronization (no reduction operation)
UTOFU_REDUCE_OP_BAND	Bitwise AND operation of uint64_t values
UTOFU_REDUCE_OP_BOR	Bitwise OR operation of uint64_t values
UTOFU_REDUCE_OP_BXOR	Bitwise XOR operation of uint64_t values
UTOFU_REDUCE_OP_MAX	Operation that gives maximum unsigned integer of uint64_t values
UTOFU_REDUCE_OP_MAXLOC	Operation that gives maximum unsigned integer of uint64_t values and its location
UTOFU_REDUCE_OP_SUM	Unsigned integer summation operation of uint64_t values
UTOFU_REDUCE_OP_BFPSUM	Floating-point (BFP) summation operation of double values

3.8.4 Barrier Communication Flags

3.8.4.1 UTOFU_BARRIER_FLAG_*

Bit flags for the flags parameter of barrier communication start functions

Explanation

They can specify behaviors of barrier communication.

No flags are defined in the current version.

3.9 Supplemental Features

3.9.1 Version Query Functions

3.9.1.1 utofu_query_tofu_version

Query the version of the Tofu interconnect.

Format

```
void utofu_query_tofu_version(
    int *major_ver,
    int *minor_ver)
```

Explanation

For details on returned version values, see "Chapter 6 System Information".

Version values returned by this function are numbered by uTofu, and they differ from the version of the Tofu interconnect itself.

Parameters

Parameter Name	Explanation	IN/OUT
major_ver	Major version	OUT
minor_ver	Minor version	OUT

3.9.1.2 utofu_query_utofu_version

Query the version of the uTofu runtime API.

Format

```
void utofu_query_utofu_version(
    int *major_ver,
    int *minor_ver)
```

Parameters

Parameter Name	Explanation	IN/OUT
major_ver	Major version	OUT
minor_ver	Minor version	OUT

3.9.2 Compute Node Information Query Functions

3.9.2.1 utofu_query_my_coords

Query the coordinates of this compute node in the Tofu network.

Format

```
int utofu_query_my_coords(
    uint8_t coords[])
```

Parameters

Parameter Name	Explanation	IN/OUT
coords	Compute node coordinates in the order of X, Y, Z, A, B, C. The array length must be 6 or greater.	OUT

Return values

Value	Explanation
UTOFU_SUCCESS	Succeeded
other	Other UTOFU_ERR_* error

3.9.3 Version Information Macros

3.9.3.1 UTOFU_VERSION_*

uTofu compile-time API version

Explanation

These macros show the uTofu compile-time API version (header file).

The utofu_query_utofu_version() function can obtain the uTofu runtime API version (library file).

Macros

Macro Name	Explanation
UTOFU_VERSION_MAJOR	uTofu API major version
UTOFU_VERSION_MINOR	uTofu API minor version

Chapter 4 How to Use uTofu

This chapter describes how to use uTofu.

4.1 uTofu Program Design

This section describes necessary matters regarding uTofu program design.

4.1.1 Use From a Language Other Than C

The uTofu header file `utofu.h` for C, if processed by a C++ compiler, provides declarations of functions, etc. with the C linkage specification (`extern "C"`). Therefore, the including of the `utofu.h` header file from a C++ source program enables use of uTofu through its C interfaces in the C++ program. Also, by using mixed language programming technic, linking a Fortran program with a C object program or C++ object program that uses uTofu enables indirect use of uTofu in the Fortran program. For the meanings of linkage specifications and mixed language programming, see the C++ standard and respective compiler manuals.

4.1.2 Using uTofu Together With MPI

The descriptions here assume the use of Fujitsu MPI. They may not apply when another MPI library is used.

uTofu and MPI can be used together in one program. Since uTofu and MPI communicate logically independently over the Tofu interconnect, one of them can communicate while the other is also communicating.

The following restrictions apply when one program uses uTofu together with MPI.

- A call of uTofu function must be within the period between a call of the `MPI_INIT` routine or `MPI_INIT_THREAD` routine of MPI and a call of the `MPI_FINALIZE` routine of MPI. This is because the `MPI_INIT` or `MPI_INIT_THREAD` routine call appropriately allocates communication resources (CQs) between processes on a compute node or between uTofu and MPI within a process. This is also because the `MPI_INIT` or `MPI_INIT_THREAD` routine call enables automatic selection of appropriate communication paths. For details on MPI routines, see the MPI standard.
- The VCQ created with uTofu in the following two cases must be created as a thread-safe VCQ or CQ-exclusive VCQ. In one case, multiple threads may call uTofu functions and MPI routines simultaneously. In the other case, the MPI function for promoting asynchronous communication using an assistant core is used. For details on this function, see the MPI library manual.

The following restrictions apply when one program does not use uTofu and MPI together.

- If there is an MPI process on a compute node, the other processes in the compute node cannot use uTofu. This is due to a conflict of communication resources between MPI processes and uTofu processes. This also means that a child process created from the MPI process cannot use uTofu.

You need to note the following points when one program does not use uTofu and MPI together.

- Each process communicating with another process needs to notify the other process of the VCQ ID, VBG ID, etc. before communicating using uTofu. However, the means of this notification must be other than MPI communication.
- Duplication of communication resources (CQs) between processes must be prevented when multiple uTofu processes are executed on a single compute node.

If you want to easily write a uTofu program, we recommend calling the `MPI_INIT` routine or `MPI_INIT_THREAD` routine and sending information of the VCQ ID, VBG ID, etc. with MPI communication even if MPI communication is unnecessary.

Note that this manual collectively refers to functions in C, Fortran, and C++ and subroutines in Fortran as routines, according to the notation used in the MPI standard. All the MPI routine names are capitalized.

4.1.3 Possible Range of Communication

A uTofu program is executed as a job on a compute node, as described in "4.3.1 Spawning a uTofu Process". The possible range of communication by a uTofu process executed as a job is the range of the job. This is the same as the possible range of communication by an MPI process.

4.1.4 Preventing Concentrated Communications

If many communications (more than several thousands or several tens of thousands) concentrate on one compute node at the same time, the Tofu network has a bottleneck near this compute node. Consequently, network processing becomes extremely slow. If a device driver (the Tofu driver) detects such a phenomenon, the uTofu implementation may output an error message and force the program to exit.

As examples of measures to prevent many communications from concentrating on one compute node at the same time, take the following actions.

- Review algorithms so that communications do not concentrate on specific compute nodes.
- Insert synchronization among processes to distribute communications temporally.
- For repeated communications to one compute node, maintain an interval between the repetitions in order to distribute the communications temporally.

4.2 Compiling/Linking a uTofu Program

You can compile/link a uTofu program in a similar way to other C programs. However, for linking, the `-ltofucom` option must be handed over to the compiler or linker to link the `libtofucom.so` shared library to the executable program.

The directories where the `utofu.h` header file exists and the `libtofucom.so` shared library exists are usually included in the standard search path list of the compiler or the linker. Therefore, no directory specification is required in the `-I` option or `-L` option.

For compiling/linking a uTofu program, use the cross compiler on the login node or the native compiler on a compute node. To use uTofu together with MPI in one program, use the compile/link command of MPI. For details on the cross compiler or the native compiler, see the respective compiler manuals. For details on the compile/link command of MPI, see the MPI library manual.

Example

Example of the `mpifccpx` command compiling/linking an MPI program

1. Compile the `test.c` user program to create the `test.o` object program.

```
$ mpifccpx -c test.c
```

2. Link the `test.o` object program to create the `test` executable program.

```
$ mpifccpx -ltofucom -o test test.o
```

4.3 Executing a uTofu Program

This section describes necessary matters regarding uTofu program execution.

4.3.1 Spawning a uTofu Process

uTofu programs must be executed on compute nodes. The user does not directly execute a uTofu program on a compute node but instead requests the Job Operation Software to submit a job that executes the program. For details on how to submit the job, see the Job Operation Software manuals.

When using uTofu and MPI together in one program, use the `mpiexec` command in a job to execute the uTofu program. No special preparations are necessary except that you can set uTofu-specific environment variables, which are described in "4.3.2 Environment Variables", as needed. For details on how to use the `mpiexec` command, see the MPI library manual.

Even when not using uTofu and MPI together in one program, you can use the `mpiexec` command in a job to execute the uTofu program. The number and locations of spawned uTofu processes are the same as for an MPI program. The `--debuglib` option in the `mpiexec` command of Fujitsu MPI is ignored. Among the MCA parameters that can be specified in the `--mca` or `-am` option in the `mpiexec` command of Fujitsu MPI, the parameters relating only to MPI library behavior are ignored.

Also when not using uTofu and MPI together in one program, you can directly execute a uTofu program without using the `mpiexec` command.

If the job type is node-sharing job, uTofu cannot be used. For details on node-sharing jobs, see the Job Operation Software manuals.

4.3.2 Environment Variables

You can change parts of uTofu behavior by using environment variables. The environment variables of uTofu must be set before the first uTofu or MPI function call in a uTofu process. If an environment variable of uTofu is set, changed, or removed after even one uTofu or MPI function is called, the behavior is not guaranteed.

4.3.2.1 UTOFU_NUM_EXCLUSIVE_CQS

You can set the required number of CQs for CQ-exclusive VCQs per TNI by using the `UTOFU_NUM_EXCLUSIVE_CQS` environment variable.

You can specify 0 or higher. The default is 0.

To enable high-performance collective communication, the MPI library may use multiple CQs per TNI. Some CQs are reserved for this purpose. However, the number of the CQs is finite. Due to a shortage of CQs, you may not be able to create CQ-exclusive VCQs with the `utofu_create_vcq()` function.

When the required number of CQs for CQ-exclusive VCQs is set in this environment variable, the uTofu implementation tries to reserve as many CQs as possible up to the required number for CQ-exclusive VCQs. It may not be able to reserve as many CQs as the required number since the number of CQs is finite and CQs are distributed to all the processes on the compute node.

If uTofu and MPI are used together, setting a value of 1 or greater decreases the number of CQs for collective communication of the MPI library. This may affect its performance.

For details on the CQ-exclusive VCQ, see the description of the `UTOFU_VCQ_FLAG_EXCLUSIVE` macro in "3.3.4.1 UTOFU_VCQ_FLAG_*".

4.3.2.2 UTOFU_NUM_SESSION_MODE_CQS

You can set the required number of CQs for session mode VCQs per TNI by using the `UTOFU_NUM_SESSION_MODE_CQS` environment variable.

You can specify 0 or higher. The default is 3.

To enable high-performance collective communication, the MPI library may use multiple session mode CQs per TNI. However, the number of the CQs is finite. Due to a shortage of CQs, you may not be able to create session mode VCQs with the `utofu_create_vcq()` function.

When the required number of CQs for session mode VCQs is set in this environment variable, the uTofu implementation tries to reserve as many CQs as possible up to the required number. It may not be able to reserve as many CQs as the required number since the number of CQs is finite and CQs are distributed to all the processes on the compute node.

If both this environment variable and the `UTOFU_NUM_EXCLUSIVE_CQS` environment variable are set and the number of CQs is insufficient, the `UTOFU_NUM_EXCLUSIVE_CQS` environment variable has priority.

If uTofu and MPI are used together, setting a value of 2 or less decreases the number of CQs for collective communication of the MPI library. This may affect its performance.

For details on the session mode VCQ, see "2.2.8 Session Mode".

4.3.2.3 UTOFU_NUM_MRQ_ENTRIES

You can set the number of MRQ entries of a free mode VCQ by using the `UTOFU_NUM_MRQ_ENTRIES` environment variable.

You can specify any of the following numerical values: 2048, 8192, 32768, 131072, 524288, and 2097152. The default is 131072. Instead of these numerical values, you can also specify any of the following character strings: 2Ki, 8Ki, 32Ki, 128Ki, 512Ki, and 2Mi. The specified characters are not case-sensitive. If the specified numerical value is not one of the above, the above value closest to it is assumed specified. If two values are equally close to it, the higher value is assumed specified.

The Tofu interconnect reports the completion of communication by writing a descriptor in the MRQ. If the `utofu_poll_mrq()` function is not called and descriptors continue to accumulate in the MRQ, an error called MRQ overflow occurs. A higher number of MRQ entries can reduce the probability of an MRQ overflow event. However, the downside is increased memory consumption.

4.3.2.4 UTOFU_NUM_MRQ_ENTRIES_SESSION

You can set the number of MRQ entries of a session mode VCQ by using the `UTOFU_NUM_MRQ_ENTRIES_ENTRIES` environment variable.

Values which can be specified are same as those of `UTOFU_NUM_MRQ_ENTRIES`.

For details on the session mode VCQ, see "2.2.8 Session Mode".

4.3.2.5 UTOFU_SWAP_PROTECT

On a compute node, an allocated memory region may be swapped out due to memory shortage or other reasons. When a memory region used for communication on the Tofu interconnect is swapped out, an error may occur in communication using the memory region. In this case, the one-sided communication completion confirmation functions return any of the following return values.

Which the `utofu_poll_tcq()` function or the `utofu_poll_mrq()` function returns the return value of the error depends on when swap out occurs.

- Case of the `utofu_poll_tcq()` function
  - `UTOFU_ERR_TCQ_MEMORY`
- Case of the `utofu_poll_mrq()` function
  - `UTOFU_ERR_MRQ_LCL_MEMORY`
  - `UTOFU_ERR_MRQ_RMT_MEMORY`

By specifying the environment variable `UTOFU_SWAP_PROTECT`, you can select whether to exclude the memory region for communication from swap out.

An integer value 0 or 1 can be specified, and the default value is 0. If a value other than 0 or 1 is specified, it is assumed to be 1.

When the value 1 is specified for this environment variable, the mlock system call is called inside the uTofu implementation and the memory region used for communication on the Tofu interconnect is guaranteed not to be subject to swap out. The size of the memory region that can be excluded from the swap out target is subject to the software resource limit `RLIMIT_MEMLOCK`. See the Job Operation Software manual for details on how to check or change the `RLIMIT_MEMLOCK` at job execution.

When the value 0 is specified for this environment variable, the memory region used for communication on the Tofu interconnect may be subject to swap out.

Communication performance may decrease if a memory region for communication is excluded from swap out.

If the value of `RLIMIT_MEMLOCK` is less than the total size of the memory region for communication used by the job, the communication error due to swap out may not be avoided even if the value 1 is specified for this environment variable.

Chapter 5 Use Examples of uTofu

This chapter provides some examples of uTofu use.

The sample programs in this chapter are written in the following style.

- MPI is also used for simplified processing, as described in "4.1.2 Using uTofu Together With MPI".
- Error checks using function return values are mostly omitted in order to focus on program flow, though they are necessary in practical programs.
- The `assert()` function-like macro is used to explain values set in a program.
- `lcl` means a local process, `rmt` means a remote process, and `num` means a number in variable names.

5.1 Use Examples of One-Sided Communication

5.1.1 Example of Ping-Pong Communication Using Put

In this program, two processes alternately perform Put communications. This communication pattern is generally called ping-pong communication.

Each process performs Put communication after it receives data of Put communication from its peer process. The first half of the program confirms receiving data by remote MRQ notification. The last half of the program confirms receiving data by polling the memory region to be updated.

This program runs with two processes.

```
#include <stdlib.h>
#include <assert.h>
#include <mpi.h>
#include <utofu.h>

// send data and confirm its completion
static void send(utofu_vcq_hdl_t vcq_hdl, utofu_vcq_id_t rmt_vcq_id,
                 utofu_stadd_t lcl_send_stadd, utofu_stadd_t rmt_recv_stadd, size_t length,
                 uint64_t edata, uintptr_t cbvalue, unsigned long int post_flags)
{
    int rc;

    // instruct the TNI to perform a Put communication
    utofu_put(vcq_hdl, rmt_vcq_id, lcl_send_stadd, rmt_recv_stadd, length,
              edata, post_flags, (void *)cbvalue);

    // confirm the TCQ notification
    if (post_flags & UTOFU_ONESIDED_FLAG_TCQ_NOTICE) {
        void *cbdata;
        do {
            rc = utofu_poll_tcq(vcq_hdl, 0, &cbdata);
        } while (rc == UTOFU_ERR_NOT_FOUND);
        assert(rc == UTOFU_SUCCESS);
        assert((uintptr_t)cbdata == cbvalue);
    }

    // confirm the local MRQ notification
    if (post_flags & UTOFU_ONESIDED_FLAG_LOCAL_MRQ_NOTICE) {
        struct utofu_mrq_notice notice;
        do {
            rc = utofu_poll_mrq(vcq_hdl, 0, &notice);
        } while (rc == UTOFU_ERR_NOT_FOUND);
        assert(rc == UTOFU_SUCCESS);
        assert(notice.notice_type == UTOFU_MRQ_TYPE_LCL_PUT);
        assert(notice.edata == edata);
```

```
    }
}

// confirm receiving data
static void recv(utofu_vcq_hdl_t vcq_hdl, volatile uint64_t *recv_buffer, uint64_t expected_value,
                 uint64_t edata, unsigned long int post_flags)
{
    int rc;

    // confirm the remote MRQ notification or the memory update
    if (post_flags & UTOFU_ONESIDED_FLAG_REMOTE_MRQ_NOTICE) {
        struct utofu_mrq_notice notice;
        do {
            rc = utofu_poll_mrq(vcq_hdl, 0, &notice);
        } while (rc == UTOFU_ERR_NOT_FOUND);
        assert(rc == UTOFU_SUCCESS);
        assert(notice.notice_type == UTOFU_MRQ_TYPE_RMT_PUT);
        assert(notice.edata == edata);
        assert(*recv_buffer == expected_value);
    } else {
        while (*recv_buffer != expected_value);
    }
}

int main(int argc, char *argv[])
{
    int i, rc, iteration = 100, num_processes, lcl_rank, rmt_rank;
    unsigned long int post_flags;
    size_t num_tnis, length = sizeof(uint64_t);
    uint64_t edata, send_buffer;
    volatile uint64_t recv_buffer;
    uintptr_t cbvalue;
    utofu_tni_id_t tni_id, *tni_ids;
    utofu_vcq_hdl_t vcq_hdl;
    utofu_vcq_id_t lcl_vcq_id, rmt_vcq_id;
    utofu_stadd_t lcl_send_stadd, lcl_recv_stadd, rmt_recv_stadd;
    struct utofu_onesided_caps *onesided_caps;

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &num_processes);
    if (num_processes != 2) {
        MPI_Abort(MPI_COMM_WORLD, 1);
        return 1;
    }

    MPI_Comm_rank(MPI_COMM_WORLD, &lcl_rank);
    rmt_rank = (lcl_rank == 0) ? 1 : 0;

    // get an ID of a TNI available for one-sided communication
    rc = utofu_get_onesided_tnis(&tni_ids, &num_tnis);
    if (rc != UTOFU_SUCCESS || num_tnis == 0) {
        MPI_Abort(MPI_COMM_WORLD, 1);
        return 1;
    }
    tni_id = tni_ids[0];
    free(tni_ids);

    // query the capabilities of one-sided communication of the TNI
    utofu_query_onesided_caps(tni_id, &onesided_caps);

    // create a VCQ and get its VCQ ID
    utofu_create_vcq(tni_id, 0, &vcq_hdl);
```

```
utofu_query_vcq_id(vcq_hdl, &lcl_vcq_id);

// register memory regions and get their STADDs
utofu_reg_mem(vcq_hdl, (void *)&send_buffer, length, 0, &lcl_send_stadd);
utofu_reg_mem(vcq_hdl, (void *)&recv_buffer, length, 0, &lcl_recv_stadd);

// notify peer processes of the VCQ ID and the STADD
MPI_Sendrecv(&lcl_vcq_id, 1, MPI_UINT64_T, rmt_rank, 0,
             &rmt_vcq_id, 1, MPI_UINT64_T, rmt_rank, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Sendrecv(&lcl_recv_stadd, 1, MPI_UINT64_T, rmt_rank, 0,
             &rmt_recv_stadd, 1, MPI_UINT64_T, rmt_rank, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);

// embed the default communication path coordinates into the received VCQ ID.
utofu_set_vcq_id_path(&rmt_vcq_id, NULL);

recv_buffer = UINT64_MAX;
MPI_Barrier(MPI_COMM_WORLD);

// perform ping-pong communication using the remote MRQ notification
post_flags = UTOFU_ONESIDED_FLAG_TCQ_NOTICE |
             UTOFU_ONESIDED_FLAG_REMOTE_MRQ_NOTICE |
             UTOFU_ONESIDED_FLAG_LOCAL_MRQ_NOTICE;

// The MRQ of recv corresponding to the TOQ executed with send can be identified by
// changing the edata value of each iteration.
for (i = 0; i < iteration; i++) {
    // The edata that can be used is 8 bytes. Because of that, edata is reset to 0
    // times every 256 times.
    edata = i % (1UL << (8 * onesided_caps->max_edata_size));
    cbvalue = i;
    send_buffer = i;
    if (lcl_rank == 0) {
        send(vcq_hdl, rmt_vcq_id, lcl_send_stadd, rmt_recv_stadd, length,
             edata, cbvalue, post_flags);
        recv(vcq_hdl, &recv_buffer, send_buffer, edata, post_flags);
        recv_buffer = UINT64_MAX;
    } else {
        recv(vcq_hdl, &recv_buffer, send_buffer, edata, post_flags);
        recv_buffer = UINT64_MAX;
        send(vcq_hdl, rmt_vcq_id, lcl_send_stadd, rmt_recv_stadd, length,
             edata, cbvalue, post_flags);
    }
}

recv_buffer = UINT64_MAX;
MPI_Barrier(MPI_COMM_WORLD);

// perform ping-pong communication using the memory update
post_flags = UTOFU_ONESIDED_FLAG_TCQ_NOTICE |
             UTOFU_ONESIDED_FLAG_LOCAL_MRQ_NOTICE;

// The MRQ of recv corresponding to the TOQ executed with send can be identified by
// changing the edata value of each iteration.
for (i = 0; i < iteration; i++) {
    // The edata that can be used is 8 bytes. Because of that, edata is reset to 0
    // times every 256 times.
    edata = i % (1UL << (8 * onesided_caps->max_edata_size));
    cbvalue = i;
    send_buffer = i;
    if (lcl_rank == 0) {
        send(vcq_hdl, rmt_vcq_id, lcl_send_stadd, rmt_recv_stadd, length,
```

```
                edata, cbvalue, post_flags);
            recv(vcq_hdl, &recv_buffer, send_buffer, edata, post_flags);
            recv_buffer = UINT64_MAX;
        } else {
            recv(vcq_hdl, &recv_buffer, send_buffer, edata, post_flags);
            recv_buffer = UINT64_MAX;
            send(vcq_hdl, rmt_vcq_id, lcl_send_stadd, rmt_recv_stadd, length,
                edata, cbvalue, post_flags);
        }
    }

    // free resources
    utofu_dereg_mem(vcq_hdl, lcl_send_stadd, 0);
    utofu_dereg_mem(vcq_hdl, lcl_recv_stadd, 0);
    utofu_free_vcq(vcq_hdl);

    MPI_Finalize();

    return 0;
}
```

5.1.2 Example of Status Check Using Get

In this program, rank 0 performs computation while rank 1 checks its progress status.

Rank 0 writes the progress status of the computation to the variable `step` as needed. Rank 1 uses Get communication to read the value of this variable every five seconds.

In order that optimization by the compiler does not eliminate updating the variable `step`, you need to add the `volatile` qualifier to the declaration of the variable `step`. This program assumes that the CPU performs a store operation to the memory region of the `step` variable atomically.

This program runs with two processes.

```
#define _POSIX_C_SOURCE 200809L // for clock_gettime

#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <unistd.h>
#include <time.h>
#include <mpi.h>
#include <utofu.h>

static volatile unsigned long step;
static unsigned long sum;
static int loop_sec = 60;

// computation
static void compute(void)
{
    struct timespec start_time, current_time;

    // get the start time
    clock_gettime(CLOCK_MONOTONIC, &start_time);

    do {
        // compute
        for (int i = 0; i < 10000; i++) {
            sum += i;
        }

        // update the variable to hold the progress status
```

```
        step++;

        // get the current time
        clock_gettime(CLOCK_MONOTONIC, &current_time);
    } while(current_time.tv_sec - start_time.tv_sec <= loop_sec);
}

// status checking
static void check(utofu_vcq_hdl_t vcq_hdl, utofu_vcq_id_t rmt_vcq_id, utofu_stadd_t lcl_stadd,
utofu_stadd_t rmt_stadd)
{
    int rc;
    unsigned long int post_flags = UTOFU_ONESIDED_FLAG_LOCAL_MRQ_NOTICE;
    void *cbdata;
    struct utofu_mrq_notice notice;
    struct timespec start_time, current_time;

    // get the start time
    clock_gettime(CLOCK_MONOTONIC, &start_time);

    do {
        sleep(5);

        // instruct the TNI to perform a Get communication to copy the step variable in the remote
process to the step variable in the local process
        while (1) {
            rc = utofu_get(vcq_hdl, rmt_vcq_id, lcl_stadd, rmt_stadd, sizeof(step), 0, post_flags,
NULL);
            if (rc != UTOFU_ERR_BUSY) {
                break;
            }
            utofu_poll_tcq(vcq_hdl, 0, &cbdata);
        }
        assert(rc == UTOFU_SUCCESS);

        // confirm the local MRQ notice
        do {
            rc = utofu_poll_mrq(vcq_hdl, 0, &notice);
        } while (rc == UTOFU_ERR_NOT_FOUND);
        assert(rc == UTOFU_SUCCESS);
        assert(notice.notice_type == UTOFU_MRQ_TYPE_LCL_GET);

        // output the obtained progress status
        printf("step: %lu\n", step);
        fflush(stdout);

        // get the current time
        clock_gettime(CLOCK_MONOTONIC, &current_time);
    } while(current_time.tv_sec - start_time.tv_sec <= loop_sec);
}

int main(int argc, char *argv[])
{
    int rc, num_processes, rank;
    size_t num_tnis;
    utofu_tni_id_t tni_id, *tni_ids;
    utofu_vcq_hdl_t vcq_hdl;
    utofu_vcq_id_t lcl_vcq_id, rmt_vcq_id;
    utofu_stadd_t lcl_stadd, rmt_stadd;

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &num_processes);
```

```
    if (num_processes != 2) {
        MPI_Abort(MPI_COMM_WORLD, 1);
        return 1;
    }

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // get an ID of a TNI available for one-sided communication
    rc = utofu_get_onesided_tnis(&tni_ids, &num_tnis);
    if (rc != UTOFU_SUCCESS || num_tnis == 0) {
        MPI_Abort(MPI_COMM_WORLD, 1);
        return 1;
    }
    tni_id = tni_ids[0];
    free(tni_ids);

    // create a VCQ
    utofu_create_vcq(tni_id, 0, &vcq_hdl);

    // register a memory region and get its STADD
    utofu_reg_mem(vcq_hdl, (void *)&step, sizeof(step), 0, &lcl_stadd);

    if (rank == 0) {

        // get the VCQ ID
        utofu_query_vcq_id(vcq_hdl, &lcl_vcq_id);

        // notify rank 1 of the VCQ ID and the STADD
        MPI_Send(&lcl_vcq_id, 1, MPI_UINT64_T, 1, 0, MPI_COMM_WORLD);
        MPI_Send(&lcl_stadd,  1, MPI_UINT64_T, 1, 0, MPI_COMM_WORLD);

    } else {

        // receive the VCQ ID and the STADD from rank 0
        MPI_Recv(&rmt_vcq_id, 1, MPI_UINT64_T, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&rmt_stadd,  1, MPI_UINT64_T, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        // embed the default communication path coordinates into the received VCQ ID.
        utofu_set_vcq_id_path(&rmt_vcq_id, NULL);

    }

    MPI_Barrier(MPI_COMM_WORLD);
    if (rank == 0) {
        compute();
    } else {
        check(vcq_hdl, rmt_vcq_id, lcl_stadd, rmt_stadd);
    }
    MPI_Barrier(MPI_COMM_WORLD);

    // free resources
    utofu_dereg_mem(vcq_hdl, lcl_stadd, 0);
    utofu_free_vcq(vcq_hdl);

    MPI_Finalize();

    return 0;
}
```

5.1.3 Example of a Game Using ARMW

In this program, the ranks other than rank 0 repeat incrementing a variable number in rank 0 all together, and the rank that fortunately updates the variable just for the ten thousandth time is determined as a winner.

Note

If many communications concentrate on one process like the case of this program, network processing may become extremely slow. Do not execute this program with several thousands of processes or more processes. For details, see "4.1.4 Preventing Concentrated Communications".

```
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <mpi.h>
#include <utofu.h>

int main(int argc, char *argv[])
{
    int rc, num_processes, rank;
    unsigned long int post_flags = UTOFU_ONESIDED_FLAG_LOCAL_MRQ_NOTICE;
    size_t num_tnis;
    uint64_t number = 1, lucky_number = 10000;
    utofu_tni_id_t tni_id, *tni_ids;
    utofu_vcq_hdl_t vcq_hdl;
    utofu_vcq_id_t vcq_id;
    utofu_stadd_t stadd;
    void *cbdata;
    struct utofu_mrq_notice notice;

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &num_processes);
    if (num_processes < 2) {
        MPI_Abort(MPI_COMM_WORLD, 1);
        return 1;
    }

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // get an ID of a TNI available for one-sided communication
    rc = utofu_get_onesided_tnis(&tni_ids, &num_tnis);
    if (rc != UTOFU_SUCCESS || num_tnis == 0) {
        MPI_Abort(MPI_COMM_WORLD, 1);
        return 1;
    }
    tni_id = tni_ids[0];
    free(tni_ids);

    // create a VCQ and get its VCQ ID
    utofu_create_vcq(tni_id, 0, &vcq_hdl);
    utofu_query_vcq_id(vcq_hdl, &vcq_id);

    // register a memory region and get its STADD at rank 0
    if (rank == 0) {
        utofu_reg_mem(vcq_hdl, &number, sizeof(number), 0, &stadd);
    }

    // broadcast the VCQ ID and the STADD from rank 0 to the ranks other than 0
    MPI_Bcast(&vcq_id, 1, MPI_UINT64_T, 0, MPI_COMM_WORLD);
    MPI_Bcast(&stadd,  1, MPI_UINT64_T, 0, MPI_COMM_WORLD);
```

```
    MPI_Barrier(MPI_COMM_WORLD);

    if (rank != 0) {
        // embed the default communication path coordinates into the received VCQ ID.
        utofu_set_vcq_id_path(&vcq_id, NULL);

        do {
            // instruct the TNI to perform an ARMW communication to increment the number variable at
rank 0
            while (1) {
                rc = utofu_armw8(vcq_hdl, vcq_id, UTOFU_ARMW_OP_ADD, 1, stadd, 0, post_flags, NULL);
                if (rc != UTOFU_ERR_BUSY) {
                    break;
                }
                utofu_poll_tcq(vcq_hdl, 0, &cbdata);
            }
            assert(rc == UTOFU_SUCCESS);

            // confirm the local MRQ notification
            do {
                rc = utofu_poll_mrq(vcq_hdl, 0, &notice);
            } while (rc == UTOFU_ERR_NOT_FOUND);
            assert(rc == UTOFU_SUCCESS);
            assert(notice.notice_type == UTOFU_MRQ_TYPE_LCL_ARMW);

            // confirm the value before the incrementation
            if (notice.rmt_value == lucky_number) {
                printf("The winner is rank %d!\n", rank);
            }
        } while (notice.rmt_value < lucky_number);
    }

    MPI_Barrier(MPI_COMM_WORLD);

    // free resources
    if (rank == 0) {
        utofu_dereg_mem(vcq_hdl, stadd, 0);
    }
    utofu_free_vcq(vcq_hdl);

    MPI_Finalize();

    return 0;
}
```

5.1.4 Example of Stride Communication Using Put

In this program, each process exchanges data that is placed discretely on the memory mutually with the processes next to it on its both sides.

In this program, multiple processes repeat computation in parallel. On each stage of computation, each process requires a part of computation result on the previous stage of each of the preceding and next ranks. This communication pattern is a version of so-called halo exchange which is modified for code simplicity.

To reduce the costs for creating TOQ descriptors at the time of communication, the program first creates TOQ descriptors and reuses them.

This program performs one Put communication per data which is contiguous on the memory. In order that the TCQ notification and the remote MRQ notification are performed when all Put communications complete, these notifications are enabled only for the last Put communication.

This program has the `compute_center` and `compute_edge` function to show timing of computation and communication. However, they are not implemented because computation content has nothing to do with the explanation.

This program runs with three or more processes.

```
#include <stdlib.h>
#include <mpi.h>
#include <utofu.h>

#define NX 100
#define NY 100

static void compute_center(double data[NY][NX])
{
    // compute a part of data which does not require received data
}

static void compute_edge(double data[NY][NX])
{
    // compute a part of data which requires received data
}

int main(int argc, char *argv[])
{
    int i, rc, iteration = 100, num_processes, lcl_rank, rmt_ranks[2];
    unsigned long int post_flags = UTOFU_ONESIDED_FLAG_TCQ_NOTICE |
UTOFU_ONESIDED_FLAG_REMOTE_MRQ_NOTICE;
    size_t num_tnis, length, stride, toq_desc_sizes[2], toq_desc_size;
    utofu_tni_id_t tni_id, *tni_ids;
    utofu_vcq_hdl_t vcq_hdls[2];
    utofu_vcq_id_t lcl_vcq_ids[2], rmt_vcq_ids[2];
    utofu_stadd_t lcl_stadds[2], rmt_stadds[2], lcl_offset, rmt_offset;
    double data[NY][NX];
    struct utofu_onesided_caps *onesided_caps[2];
    void *toq_descs[2], *cbdata;
    struct utofu_mrq_notice notice;

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &num_processes);
    if (num_processes < 3) {
        MPI_Abort(MPI_COMM_WORLD, 1);
        return 1;
    }

    MPI_Comm_rank(MPI_COMM_WORLD, &lcl_rank);
    rmt_ranks[0] = (lcl_rank + num_processes - 1) % num_processes; // preceding rank
    rmt_ranks[1] = (lcl_rank + 1) % num_processes;                 // next rank

    // get IDs of available TNIs for one-sided communication
    rc = utofu_get_onesided_tnis(&tni_ids, &num_tnis);
    if (rc != UTOFU_SUCCESS || num_tnis == 0) {
        MPI_Abort(MPI_COMM_WORLD, 1);
        return 1;
    }

    for (i = 0; i < 2; i++) {
        // use distinct TNIs for two peer processes for load balancing if there are two or more TNIs
available
        tni_id = (i == 0 || num_tnis == 1) ? tni_ids[0] : tni_ids[1];

        // query the capabilities of one-sided communication of the TNIs
        utofu_query_onesided_caps(tni_id, &onesided_caps[i]);

        // create two VCQs and get their VCQ IDs
        utofu_create_vcq(tni_id, 0, &vcq_hdls[i]);
        utofu_query_vcq_id(vcq_hdls[i], &lcl_vcq_ids[i]);
```

```
        // register memory regions and get their STADDs
        utofu_reg_mem(vcq_hdls[i], data, sizeof(data), 0, &lcl_stadds[i]);
    }
    free(tni_ids);

    // notify the VCQ IDs and the STADDs to the preceding rank and the next rank
    MPI_Sendrecv(&lcl_vcq_ids[0],  1, MPI_UINT64_T, rmt_ranks[0], 0,
                 &rmt_vcq_ids[1],  1, MPI_UINT64_T, rmt_ranks[1], 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Sendrecv(&lcl_vcq_ids[1],  1, MPI_UINT64_T, rmt_ranks[1], 0,
                 &rmt_vcq_ids[0],  1, MPI_UINT64_T, rmt_ranks[0], 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Sendrecv(&lcl_stadds[0],   1, MPI_UINT64_T, rmt_ranks[0], 0,
                 &rmt_stadds[1],   1, MPI_UINT64_T, rmt_ranks[1], 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Sendrecv(&lcl_stadds[1],   1, MPI_UINT64_T, rmt_ranks[1], 0,
                 &rmt_stadds[0],   1, MPI_UINT64_T, rmt_ranks[0], 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    // embed the default communication path coordinates into received VCQ IDs.
    utofu_set_vcq_id_path(&rmt_vcq_ids[0], NULL);
    utofu_set_vcq_id_path(&rmt_vcq_ids[1], NULL);

    // prepare Put communications to
    // send left-edge data of a NX * NY rectangle to the preceding rank and
    // send right-edge data of a NX * NY rectangle to the next rank
    length = sizeof(double);
    stride = length * NX;
    for (i = 0; i < 2; i++) {
        toq_descs[i] = malloc(onesided_caps[i]->max_toq_desc_size * NY);
        // disable the TCQ notifications and remote MRQ notifications for NY - 1 Put communications
        lcl_offset = length * ((i == 0) ? 1 : (NX - 2));
        rmt_offset = length * ((i == 0) ? 0 : (NX - 1));
        utofu_prepare_put_stride(vcq_hdls[i], rmt_vcq_ids[i], lcl_stadds[i] + lcl_offset,
rmt_stadds[i] + rmt_offset,
                                 length, stride, NY - 1, 0, 0, toq_descs[i], &toq_desc_sizes[i]);
        // enable the TCQ notification and remote MRQ notification for the last Put communication
        lcl_offset += stride * (NY - 1);
        rmt_offset += stride * (NY - 1);
        // When not identifying the correspondence between a TOQ descriptor and MRQ descriptor,
        // 0 can be specified for an edata.
        utofu_prepare_put(vcq_hdls[i], rmt_vcq_ids[i], lcl_stadds[i] + lcl_offset, rmt_stadds[i] +
rmt_offset,
                          length, 0, post_flags, (char *)toq_descs[i] + toq_desc_sizes[i],
&toq_desc_size);
        toq_desc_sizes[i] += toq_desc_size;
    }

    for (i = 0; i < iteration; i++) {
        // instruct the TNIs to perform the prepared Put communications
        // (send data computed in the previous loop stage)
        utofu_post_toq(vcq_hdls[0], toq_descs[0], toq_desc_sizes[0], NULL);
        utofu_post_toq(vcq_hdls[1], toq_descs[1], toq_desc_sizes[1], NULL);

        // confirm the TCQ notifications (send completion) of the last Put communications
        do {
            rc = utofu_poll_tcq(vcq_hdls[0], 0, &cbdata);
        } while (rc == UTOFU_ERR_NOT_FOUND);
        do {
            rc = utofu_poll_tcq(vcq_hdls[1], 0, &cbdata);
        } while (rc == UTOFU_ERR_NOT_FOUND);

        // compute a part of data which does not require received data
```

```
        // (data in the send data region can be updated because the send operation has completed)
        compute_center(data);

        // confirm the remote MRQ notifications (receive completion) of the last Put communications
        do {
            rc = utofu_poll_mrq(vcq_hdls[0], 0, &notice);
        } while (rc == UTOFU_ERR_NOT_FOUND);
        do {
            rc = utofu_poll_mrq(vcq_hdls[1], 0, &notice);
        } while (rc == UTOFU_ERR_NOT_FOUND);

        // compute a part of data which requires received data
        // (data in the receive data region can be referenced because the receive operation has
completed)
        compute_edge(data);
    }

    // free resources
    for (i = 0; i < 2; i++) {
        utofu_dereg_mem(vcq_hdls[i], lcl_stadds[i], 0);
        utofu_free_vcq(vcq_hdls[i]);
        free(toq_descs[i]);
    }

    MPI_Finalize();

    return 0;
}
```

5.2 Use Examples of Barrier Communication

5.2.1 Example of Barrier Synchronization and Reduction Operation

This program performs barrier synchronization and reduction operation among processes. In the reduction operation, the sum of the floating-point data owned by all the processes is calculated.

When a process gets out of the loop of the `utofu_poll_barrier()` or `utofu_poll_reduce_double()` function, it is guaranteed that the `utofu_barrier()` or `utofu_reduce_double()` function is called in all the other processes.

In the reduction operation, all the processes obtain the same calculation result.

This program runs with eight processes.

```
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <mpi.h>
#include <utofu.h>

int main(int argc, char *argv[])
{
    int i, rc, iteration = 100, num_processes, rank;
    double data_in, data_out;
    size_t num_tnis;
    utofu_tni_id_t tni_id, *tni_ids;
    utofu_vbg_id_t lcl_vbg_ids[3], rmt_vbg_ids[8][3];
    struct utofu_vbg_setting vbg_settings[3];

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &num_processes);
    if (num_processes != 8) {
```

```
        MPI_Abort(MPI_COMM_WORLD, 1);
        return 1;
    }

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // get an ID of a TNI available for barrier communication
    rc = utofu_get_barrier_tnis(&tni_ids, &num_tnis);
    if (rc != UTOFU_SUCCESS || num_tnis == 0) {
        MPI_Abort(MPI_COMM_WORLD, 1);
        return 1;
    }
    tni_id = tni_ids[rank % num_tnis];
    free(tni_ids);

    // allocate VBGs
    utofu_alloc_vbg(tni_id, 3, 0, lcl_vbg_ids);

    // share VBG IDs allocated in all processes
    MPI_Allgather(lcl_vbg_ids, 3, MPI_UINT64_T, rmt_vbg_ids, 3, MPI_UINT64_T, MPI_COMM_WORLD);

    // build a barrier circuit using an 8-process butterfly exchange algorithm

    vbg_settings[0].vbg_id = lcl_vbg_ids[0];
    vbg_settings[0].src_lcl_vbg_id = lcl_vbg_ids[2];
    vbg_settings[0].src_rmt_vbg_id = rmt_vbg_ids[rank ^ 4][2];
    vbg_settings[0].dst_lcl_vbg_id = lcl_vbg_ids[1];
    vbg_settings[0].dst_rmt_vbg_id = rmt_vbg_ids[rank ^ 1][1];
    vbg_settings[0].dst_path_coords[0] = UTOFU_PATH_COORD_NULL;

    vbg_settings[1].vbg_id = lcl_vbg_ids[1];
    vbg_settings[1].src_lcl_vbg_id = lcl_vbg_ids[0];
    vbg_settings[1].src_rmt_vbg_id = rmt_vbg_ids[rank ^ 1][0];
    vbg_settings[1].dst_lcl_vbg_id = lcl_vbg_ids[2];
    vbg_settings[1].dst_rmt_vbg_id = rmt_vbg_ids[rank ^ 2][2];
    vbg_settings[1].dst_path_coords[0] = UTOFU_PATH_COORD_NULL;

    vbg_settings[2].vbg_id = lcl_vbg_ids[2];
    vbg_settings[2].src_lcl_vbg_id = lcl_vbg_ids[1];
    vbg_settings[2].src_rmt_vbg_id = rmt_vbg_ids[rank ^ 2][1];
    vbg_settings[2].dst_lcl_vbg_id = lcl_vbg_ids[0];
    vbg_settings[2].dst_rmt_vbg_id = rmt_vbg_ids[rank ^ 4][0];
    vbg_settings[2].dst_path_coords[0] = UTOFU_PATH_COORD_NULL;

    // configure VBGs
    utofu_set_vbg(vbg_settings, 3);

    // wait completion of configuring VBGs at all processes using MPI_Barrier before performing
barrier communication
    MPI_Barrier(MPI_COMM_WORLD);

    for (i = 0; i < iteration; i++) {

        // start barrier synchronization
        utofu_barrier(lcl_vbg_ids[0], 0);

        // wait completion of the barrier synchronization
        do {
            rc = utofu_poll_barrier(lcl_vbg_ids[0], 0);
        } while (rc == UTOFU_ERR_NOT_COMPLETED);
        assert(rc == UTOFU_SUCCESS);
    }
```

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

        // start reduction operation
        data_in = i + 0.1 * rank;
        utofu_reduce_double(lcl_vbg_ids[0], UTOFU_REDUCE_OP_BFPSUM, &data_in, 1, 0);

        // wait completion of the reduction operation
        do {
            rc = utofu_poll_reduce_double(lcl_vbg_ids[0], 0, &data_out);
        } while (rc == UTOFU_ERR_NOT_COMPLETED);
        assert(rc == UTOFU_SUCCESS);
    }

    // wait completion of barrier communication at all processes using MPI_Barrier before freeing
resources
    MPI_Barrier(MPI_COMM_WORLD);

    // free resources
    utofu_free_vbg(lcl_vbg_ids, 3);

    MPI_Finalize();

    return 0;
}
```

Chapter 6 System Information

This chapter describes system-specific information.

6.1 Information of This Computing System

This section describes information specific to this computing system.

6.1.1 Version Information of This Computing System

The following table lists the return values of the `utofu_query_tofu_version()` function.

Table 6.1 Tofu Interconnect Versions for This Computing System

Type	Value
major version	3
minor version	0

6.1.2 Functional Characteristics of This Computing System

The following table lists hardware resource quantities.

Table 6.2 This Computing System Hardware Resource Quantities

Resource	Value
Number of TNIs per compute node	6
Number of available CQs per TNI	9

The following table lists virtual resource quantities.

Table 6.3 This Computing System Virtual Resource Quantities

Resource	Value
Number of available VCQs per CQ	8
Number of available VBGs per TNI (Number of start/end VBGs, Number of relay VBGs)	48 (16, 32)

The `utofu_query_onesided_caps()` function sets the following table values in each member of the `utofu_onesided_caps` structure object. For the meaning of each member, see "3.2.2.1 struct utofu_onesided_caps".

Table 6.4 One-Sided Communication Functions of This Computing System

Member Name	Value
`flags`	`UTOFU_ONESIDED_CAP_FLAG_SESSION_MODE |` `UTOFU_ONESIDED_CAP_FLAG_ARMW`

`armw_ops`	`UTOFU_ONESIDED_CAP_ARMW_OP_CSWAP |`
	`UTOFU_ONESIDED_CAP_ARMW_OP_SWAP |`
	`UTOFU_ONESIDED_CAP_ARMW_OP_ADD |`
	`UTOFU_ONESIDED_CAP_ARMW_OP_XOR |`
	`UTOFU_ONESIDED_CAP_ARMW_OP_AND |`
	`UTOFU_ONESIDED_CAP_ARMW_OP_OR`

`num_cmp_ids`	8
`num_reserved_stags`	256
`cache_line_size`	256
`stag_address_alignment`	256

Member Name	Value
max_toq_desc_size	64
max_putget_size	16,777,215 (2^{24} - 1)
max_piggyback_size	32
max_edata_size	1
max_mtu	1920
max_gap	255

The utofu_query_barrier_caps() function sets the following table values in each member of the utofu_barrier_caps structure object. For the meaning of each member, see "3.2.2.2 struct utofu_barrier_caps".

Table 6.5 Barrier Communication Functions of This Computing System

Member Name	Value
flags	0
reduce_ops	UTOFU_BARRIER_CAP_REDUCE_OP_BARRIER | UTOFU_BARRIER_CAP_REDUCE_OP_BAND | UTOFU_BARRIER_CAP_REDUCE_OP_BOR | UTOFU_BARRIER_CAP_REDUCE_OP_BXOR | UTOFU_BARRIER_CAP_REDUCE_OP_MAX | UTOFU_BARRIER_CAP_REDUCE_OP_MAXLOC | UTOFU_BARRIER_CAP_REDUCE_OP_SUM | UTOFU_BARRIER_CAP_REDUCE_OP_BFPSUM
max_uint64_reduction	6
max_double_reduction	3

6.1.3 Behavioral Specifications of This Computing System

The applicable specifications include the following behavioral specifications in addition to the interface specifications.

6.1.3.1 Strong Order Flag

When the strong order flag is set for one-sided communication, the order of read and write operations on the main memory is guaranteed relative to the preceding communication in accordance with the interface specification. In addition, that also partially guarantees the order of read and write operations between CPU cache lines in one Put or Get communication. In particular, contents in the following table is also guaranteed when the read source area or write destination area in the main memory of the local compute node or a remote compute node crosses multiple CPU cache lines:

Target of Guarantee	Condition	Contents of Guarantee
The last CPU cache line in the read source area	Arbitrary	Reading from the last CPU cache line starts after reading from another CPU cache line is completed
The last CPU cache line in the write destination area	The corresponding area in the read source area does not cross multiple pages	Writing to the last CPU cache line starts after writing to another CPU cache lines is completed
	The corresponding area in the read source area crosses multiple pages	Writing to the area in the last CPU cache line corresponding to after the page boundary starts after writing to another CPU cache lines and the rest area in the last CPU cache line are completed

Accordingly, by polling the last byte in the last CPU cache line to confirm that it was written, you can confirm that the entire write operation has completed. Furthermore, an arbitrary byte in the last CPU cache line can be used for polling if the user ensured that the corresponding area in the read source buffer does not cross multiple pages.

6.1.3.2 Page Size Managed by OS

In this computing system, the page sizes managed by the OS are the following.

- 2097152 bytes or 65536 bytes if the largepage is enabled
- 65536 bytes if the largepage is disabled

As described in "2.2.10 Confirmation of Communication Completion and Guarantee of Ordering", in order to guarantee writing to a CPU cache line in the unit of granularity of CPU cache line, it is necessary that the read source area corresponding to the CPU cache line in the write destination area does not cross multiple pages. Whether the largepage is enabled or not, if the quotient of the start address of a memory region divided by 65536 is equal to the quotient of "the end address of the memory region `-1`" divided by 65536, it can be judged that the memory region does not cross multiple pages. Similarly, if the quotient of the start address of a memory region divided by 256 is equal to the quotient of "the end address of the memory region `-1`" divided by 256, it can be judged that the memory region does not cross multiple CPU cache lines because the CPU cache line size is 256 bytes in this computing system.

6.1.4 Restrictions on This Computing System

6.1.4.1 Process Creation From Inside a uTofu Program

The following restriction applies to uTofu programs if they create a child process using a system call (for example, `fork`), library function (for example, `system`), service routine provided by a Fortran system (for example, `FORK`), or the like.

- If there is a memory region registered by the `utofu_reg_mem()` function or `utofu_reg_mem_with_stag()` function but not deregistered by the corresponding `utofu_dereg_mem()` function at the time of process creation, pages containing the memory region are not inherited to child processes. Therefore, child processes cannot access pages containing the memory region and the child process cannot access the pages.

If the same memory region is registered multiple times or an overlapping memory regions are registered, the memory region is not considered to be deregistered until deregistration is done as many times as the registration times.

For example, suppose that a child process created by the `fork` system call accesses such a memory region before the process calls the `execve` system call or `_exit` system call. Then, due to this restriction, a segmentation fault error will occur in the child process.

Whether accessible or not is decided for each page managed by the OS. Therefore, if a memory region is registered, the neighboring memory regions cannot also be accessed. Note that, if the largepage is enabled, each page is treated as the large page.

6.1.4.2 The Behavior When Configured to Change the Communication Path If a Tofu Interconnect Link is Down

The behavior of uTofu programs is unpredictable when they are executed by a job configured to change the communication path if a Tofu interconnect link is down.

Refer to the Job Operation Software manual for information on specifying the operation when a Tofu interconnect link is down.

Chapter 7 Error Messages

This chapter explains the error messages output by the uTofu implementation.

utofu: asynchronous error: *description* on TNI *tniid* BG *bgid*

- Description

  An internal error was detected in barrier communication. Execution of the uTofu program ends.

- Parameters

  description : Error message corresponding to the detected internal error

  tniid : ID of the TNI which detected the error

  bgid : ID of the BG which detected the error

- Action method

  Consult System Engineer about the message that was output.

utofu: asynchronous error: *description* on TNI *tniid* CQ *cqid*

- Description

  An internal error was detected in one-sided communication. Execution of the uTofu program ends.

- Parameters

  description : Error message corresponding to the detected internal error

  tniid : ID of the TNI which detected the error

  cqid : ID of the CQ which detected the error

- Action method

  If the ***description*** is "MRQ Overflow", it means that the number of MRQ descriptors written in the MRQ exceeded the number of MRQ entries. Revise the processing of the one-sided communication completion confirmation in the uTofu program, or increase the number of MRQ entries as described in "4.3.2.3 UTOFU_NUM_MRQ_ENTRIES" and "4.3.2.4 UTOFU_NUM_MRQ_ENTRIES_SESSION".

  If the ***description*** is "CQs Cacheflush Timeout", it means that congestion was detected on the Tofu interconnect. Revise the processing of the uTofu program so that communication is not congested to specific compute nodes. For details, see "4.1.4 Preventing Concentrated Communications".

  In cases other than the above, consult System Engineer about the message that was output.

Glossary

ARMW (Atomic Read Modify Write)

One-sided communication function for reading, calculating, and writing on the main memory of another compute node atomically. ARMW is executed using an ARMW descriptor of a TOQ.

barrier communication

Communication function for barrier synchronization between compute nodes. The reduction operation can be executed along with the barrier synchronization. Protocol processing is completely offloaded, and control by the CPU is unnecessary.

barrier synchronization

Synchronization processing between BGs using barrier communication. BGs that will be participating in barrier communication are configured beforehand. Barrier communication starts at each start/end BG. Once the barrier communication has started at all the start/end BGs participating in the barrier communication, the barrier communication at each start/end BG is completed.

BG (barrier gate)

One of the TNI resources. The BG is used to build a barrier circuit. Each BG has functions to wait for signals or packets one by one, execute the reduction operation, and send signals or packets one by one. The types of BG are start/end BG and relay BG.

callback data

Value that is specified when a TOQ descriptor is written and that is returned when a TCQ descriptor is read. Callback data can be used to learn the correspondence between a TOQ descriptor and TCQ descriptor.

communication path

Path traveled by a packet on a Tofu network in communication from one compute node to another compute node

communication path coordinates

Three-dimensional coordinates of A, B, and C, showing the communication path from one compute node to another compute node communicating on a Tofu network

communication path ID

Value used to specify a communication path when one-sided communication is performed. The communication path ID is created by the `utofu_get_path_id()` function. The `UTOFU_ONESIDED_FLAG_PATH` function-like macro specifies this ID for a one-sided communication start function or one-sided communication preparation function.

component ID

Value used to distinguish the VCQ of multiple upper-layer components that share a CQ

compute node

Unit of hardware equipped with a CPU, main memory, TNIs, and TNR. Usually, the compute node is also the unit of a running OS (host OS).

compute node coordinates

Six-dimensional coordinates of X, Y, Z, A, B, and C, showing the location of a compute node on a Tofu network

CQ (control queue)

Queue for controlling a TNI from software in one-sided communication. The CQ consists of three types of queue: TOQ, TCQ, and MRQ. There are multiple CQs per TNI.

CQ-exclusive VCQ

VCQ that does not share the CQ with another VCQ. Multiple threads can simultaneously call functions of STADD management or one-sided communication execution for the CQ-exclusive VCQ and another VCQ. When these functions are called for different CQs, uTofu implementation and hardware access may be processed in parallel, and communication throughput may increase. You can create a CQ-exclusive VCQ by specifying the `UTOFU_VCQ_FLAG_EXCLUSIVE` flag for the time when a VCQ is created.

default communication path

Communication path used when no communication path ID is specified for a one-sided communication start function or one-sided communication preparation function. The default communication path is used by embedding communication path coordinates in a VCQ ID by the `utofu_set_vcq_id_path()` function.

descriptor

Control data showing a data transfer instruction or completion notification. The three types of descriptor are TOQ descriptor, TCQ descriptor, and MRQ descriptor. Software writes a TOQ descriptor in a TOQ to instruct the Tofu interconnect to transfer data, and the Tofu interconnect reads and executes the instruction. The Tofu interconnect writes a TCQ descriptor or MRQ descriptor in a TCQ or MRQ, respectively, to notify software of the completion of a data transfer. The software reads the descriptor to confirm the completion. TOQ descriptors are further subdivided into Put, Get, and other descriptors according to the contents of the instruction. And TCQ and MRQ descriptors are also subdivided according to the contents of the completion notification.

EDATA

Value that is specified when a TOQ descriptor is written and that is returned when an MRQ descriptor is read. EDATA can be used to learn the correspondence between a TOQ descriptor and MRQ descriptor.

free Mode

Usual mode of a CQ. One-sided communication start functions and one-sided communication batch start functions write a TOQ descriptor and a TNI starts the communication immediately. A VCQ created on a CQ of the free mode is called a free mode VCQ.

Fujitsu MPI

MPI library included in Technical Computing Suite

Get

One-sided communication function for reading data from the main memory of another compute node. Get is executed using a Get descriptor of a TOQ.

local

VCQ that issues the communication instruction when one-sided communication is used for communication between two VCQs. Local also refers to the compute node, process, TNI, STADD, etc. of this one of the VCQs. Those of the other one are called "remote".

main memory

Memory area that exists on a compute node and is read or written by the load, store, and other instructions of the CPU and by the one-sided communication of the Tofu interconnect. In this manual, the main memory is also simply called memory.

MRQ (message receive queue)

Queue, one per CQ, in the main memory and used for notification of data transfer completion in one-sided communication. The notification is written to an MRQ. The `UTOFU_NUM_MRQ_ENTRIES` environment variable can specify the number of entries in the MRQ.

MRQ overflow

Phenomenon where MRQ descriptors of local notifications or remote notifications are written continually to the MRQ and the MRQ overflows with these descriptors. The MRQ overflow is caused by the `utofu_poll_mrq()` function not being called even as MRQ descriptors continue to be written.

MTU (maximum transfer unit)

Maximum packet transfer size. In one-sided communication, when sending or receiving data with a larger size than the MTU, a TNI splits the data into packets with the size of the MTU and transfers the packets.

NOP

One-sided communication function which does nothing. NOP is executed using a NOP descriptor of a TOQ. It is used to adjust the number of TOQ descriptors in the session mode.

one-sided communication

Communication function for accessing and updating the main memory of another compute node. One-sided communication has the functions of Put, Get, ARMW, and NOP. This communication method is generally called RDMA (remote direct memory access). Protocol processing is completely offloaded, and control by the CPU is unnecessary.

packet

Unit of transmission of communication data in a Tofu network. In one-sided communication, when sending or receiving a large size of data, a TNI splits the data into packets and transfers them. In barrier communication, packets are sent and received between BGs in the same TNI or different TNIs to relay the data.

piggyback

Mechanism for low-latency communication using Put. Normally, for Put, software gives a transmission instruction to a TNI by means of a TOQ descriptor, and the TNI reads data from the main memory. By using piggyback, software embeds the data in the TOQ descriptor. Using piggyback eliminates the need for the TNI to read the data from the main memory, enabling low-latency communication.

Put

One-sided communication function for writing data to the main memory of another compute node. Put is executed using a Put descriptor or Put Piggyback descriptor of a TOQ.

reduction operation

Operation that derives one value from multiple values

remote

VCQ that is the communication target when one-sided communication is used for communication between two VCQs. Remote also refers to the compute node, process, TNI, STADD, etc. of this one of the VCQs. Those of the other one are called "local".

session mode

Special mode of a CQ. One-sided communication start functions and one-sided communication batch start functions write a TOQ descriptor but a TNI does not start the communication immediately. By receiving another Put communication at the VCQ, a TNI starts the instructed communication automatically. A VCQ created on a CQ of the session mode is called a session mode VCQ.

signal

Data relayed between BGs in the same TNI in barrier communication

STADD (steering address)

Memory address that a TNI can understand. In one-sided communication, the TNI reads from and writes to main memory to transfer data between compute nodes. However, the TNI cannot understand virtual addresses assigned by the OS. The virtual addresses are ordinary memory addresses that can be seen from user processes. Therefore, in one-sided communication, a virtual address of memory is converted to a STADD, which the TNI can understand, before the communication starts. The TNI is notified of this STADD to perform the communication.

STag (steering tag)

Integer value that indicates a memory region and is understandable to a TNI. Normally, uTofu uses the STag in the form of a STADD.

TCQ (transmit complete queue)

Queue, one per CQ, in the main memory and used for notification of the completion of a data transmission in one-sided communication. The TCQ has as many entries as in the TOQ in the same CQ, and the entries of the TOQ and TCQ have one-to-one correspondence.

thread-safe VBG

VBG that can be used concurrently by multiple threads. Multiple threads can simultaneously call a function of barrier communication execution for a single thread-safe VBG. You can allocate a thread-safe VBG by specifying the `UTOFU_VBG_FLAG_THREAD_SAFE` flag for the time when a VBG is allocated.

thread-safe VCQ

VCQ that can be used concurrently by multiple threads. Multiple threads can simultaneously call functions of STADD management or one-sided communication execution for a single thread-safe VCQ. Also, multiple threads can simultaneously call these functions for the thread-safe VCQ and another VCQ. You can create a thread-safe VCQ by specifying the `UTOFU_VCQ_FLAG_THREAD_SAFE` flag for the time when a VCQ is created.

TNI (Tofu network interface)

Network interface of the Tofu interconnect. The TNI sends and receives packets according to instructions from software, and performs one-sided communication and barrier communication. There is at least one TNI per compute node, and it is connected to the main memory and a TNR.

TNR (Tofu network router)

Network router of the Tofu interconnect. The TNR relays a packet sent from a TNI to deliver it to the destination TNI. There is one TNR per compute node, and it is connected to TNIs and a Tofu network.

Tofu interconnect

General name for the interconnects between compute nodes used in the High-end technical computing server MP10 system, Supercomputer PRIMEHPC FX10 system, Supercomputer PRIMEHPC FX100 system, and this computing system. Its configuration includes TNIs, TNRs, and a Tofu network. The Tofu interconnect is also simply called Tofu.

Tofu network

Network of the Tofu interconnect. The Tofu network has a six-dimensional coordinate space. It is built by interconnecting TNRs.

TOQ (transmit order queue)

Queue, one per CQ, in the main memory and used to issue a communication instruction in one-sided communication.

transmission gap

Interval between packets when they are transmitted. In a Tofu network, when there is contention for a communication path, the effective bandwidth of the path may drop due to congestion. If you know beforehand that such congestion will occur, you can mitigate that drop by suppressing the injection rate of transmitted data. In one-sided communication, you can suppress the injection rate by specifying a value other than 0 for the transmission gap so that the TNI does not send the next packet immediately after sending a packet.

uTofu

Programming interface used by processes in the user space to communicate through the Tofu interconnect.

It is named for "user-level Tofu communication interface".

VBG (virtual barrier gate)

Abstracted BG by means of software. The unique 64-bit integer value identifying the VBG in a Tofu network is called a VBG ID.

VCQ (virtual control queue)

Abstracted and virtualized CQ by means of software. The VCQ is used to virtualize a CQ since a single CQ is shared by multiple software components in uTofu. The 64-bit unsigned integer value that uniquely identifies the VCQ in a Tofu network is called a VCQ ID. uTofu uses a handle called a VCQ handle to handle the VCQ created in a process by the process.