XPI Draft Specification

Luke Dalessandro, Matthew Anderson, Maciej Brodowicz, Andrew Lumsdaine, and Thomas Sterling

Center for Research in
Extreme Scale Technologies (CREST),
Indiana University

October 14, 2013

Version: 1.0 (svn r237)

Acknowledgement: This material is based upon work supported by the Department of Energy under Award Number(s) [DE-SC0008809]

Contents

Chapter 1

Introduction

XPI (eXtreme Parallex Interface) is a programming interface for parallel applications and systems based on the ParalleX execution model. XPI provides a simple abstraction layer to the family of ParalleX implementation HPX runtime system software. As HPX evolves, XPI insulates application codes from such changes, ensuring stability of experimental application codes. XPI serves both as a target for source-to-source compilers of high-level languages and as a readible low-level programming interface syntax. XPI is experimental and supports current on-going sponsored research projects. Its long term future is entirely dependent on its resulting value; an unknown at this time. But it is motivated by a short-term need to advance key project goals.

XPI is superficially similar to MPI while providing dramatically differing semantics in important areas. Its look and feel suggests MPI usage with library supported bindings to mainstream sequential languages (initially C bindings) through a familiar style of directives. Where MPI commands are prefaced with `MPI_`, XPI commands prefaced with `XPI_`. Like MPI-1, an initial basic set of commands are employed by XPI for early (and rapid) implementation, experimental programming, and experimentation. Through experience with use cases, a more mature XPI will evolve that facilitate usability, generality, and interoperability.

XPI provides a simple model of execution where *Threads*, invoked and managed by *Parcel* message-driven semantics and coordinated through a global network of *Local Control* synchronization *Objects* (LCOs), interact with an asynchronous, active global address space (*AGAS*). A hierarchy of XPI *Processes* provides a dynamic structure of contexts, name spacing, protection, and termination detection, where each process potentially spans multiple system hardware nodes (synchronous domains). Processes, threads, and LCOs are first class objects; parcels are not. Data structures (e.g., matrices, graphs), allocated in the global address space, may span an entire allocated system of many discrete subsystems (nodes). Processes and threads are ephemeral: they are created and eventually terminated dynamically. Though potentially expensive, both are free to migrate—or to be migrated—in space. Similarly, LCOs are first class objects, exist ephemerally, and may migrate as well. XPI permits optional static control and prespecified resource allocations but does not require them.

Chapter 2 of this report describes some terms and conventions used throughout, as well as presenting a brief overview of the principal semantic components of XPI based on, but not identical to, the ParalleX execution model of parallel processing. This introduces the ideas of computing threads, global address space, parcel message-driven computing, and parallel control state and continuations supported in part by local control objects. Chapter 3 provides some of the necessary syntax for setting up an XPI program, running it, and

ultimately terminating it. These miscellaneous commands are as ubiquitous in all XPI programs as they are tedious to compose and are reminiscent of many of their counterparts in MPI. Chapter 4 begins the substantive content of this report with an in-depth specification of Parcels, the unified constructs for invoking threads (and other actions) anywhere in the space (logically and physically) of the user application execution. Chapter 5 describes the execution environment in terms of the global address space and specifies commands for its management and use including asynchronous access, global loads and stores, address mapping, and interfaces to parcels and threads. Chapter 6 defines the thread commands and specifies their syntax for defining program actions. Chapter 7 presents the high-level parallel control space and the basis for continuations through Local Control Objects that perform synchronization and manage asynchronous system operation. Finally, Chapter 8 specifies the constructs for XPI Processes that serve as encapsulation of context, protection, and naming, ownership of physical resources, and termination detection. A Glossary of commands is included at the end of the report. This report is a work in progress. At any point in time it represents the best thinking on the evolving XPI syntax and functional semantics with the understanding that changes are anticipated in response to experience in its use and guidance from collaborators and friendly users.

Chapter 2

Overview and Conventions

2.1 Overview of XPI Execution

In the following detailed specification of the XPI syntax, and description of the related semantic constructs, a simple parallel programming model emerges. Actions (Section 6.1) are performed by instances of ephemeral threads which expose and exploit medium-grained parallelism. Each thread (Chapter 6) is the "active" result of the delivery of XPI's messaging construct, the parcel (Chapter 4), and is executed on a single locality within the contexts of its parent process (Chapter 8). Processes (Chapter 8) embody coarse-grained parallelism, providing return values and optional termination detection. Distributed sequential computation chains are expressed using the parcel's active continuation capabilities to chain together thread execution across localities. Thread and chain synchronization is established by the sequential ordering of a chain's threads, builtin compound atomic access to the global address space (Section 5.2), and user-defined local control object (LCO) operations (Chapter 7).

An XPI application is performed within the context of an asynchronous, single global virtual address space (Chapter 5). The mapping to physical memory resources is not partitioned or static. The phrase *active global address space* or AGAS is used to describe this model and differentiate it from alternatives such as distributed shared memory, partitioned global address space (PGAS), and cache-coherent global virtual memory. First class objects, including first-class XPI model objects like threads, processes, and LCOs, have global addresses and may be manipulated through them.

Processes are named objects. The process `main` encompasses the entire application, provides access to I/O objects, integrates lower level processes, its own threads, the fully global data, program code objects, and available physical resources provided by the operating system. All child processes (and their child processes, etc.) make up a hierarchical process tree with main at the root node. Each process has a *prime thread* which is instantiated when its host process is created.

2.2 Common Terms

undefined XPI defines a set of behaviors, i.e., actions an their results, along with a set of API usage errors. At the same time, there are API usage errors that exist but are unchecked. Such errors are considered to produce globally undefined behavior.

implementation-defined In some cases, the result of a particular operation results in implementation-defined behavior. XPI applications that depend on implementation-defined behavior sacrifice portability and maintainability, and may produce unexpected results as implementations change. It is valid, given a specific implementation, for the implementation-defined result to be *undefined* in the sense specified above.

handle A handle is an opaque reference to an implementation managed object. Handles need to be created and released using interface routines. Failing to destroy a handle may lead to leaked resources in an implementation. Handles do not have global addresses and handle values should not be shared between threads.

value semantics Implementation types that are designed to be allocated and released by the application programmer, and that are passed by copying their data (i.e., through a register, set of registers, or `memcpy`) rather than by a reference have value semantics. An example of a type with value semantics is the XPI_ADDR.

> *Developer's note.* It may be a bad idea to expose value types to programmers, as this binds the compiled XPI binary to the specific version of the XPI headers that it was compiled with. At the same time, using handles to objects of these types is cumbersome, and using value types may expose static optimization opportunities to the compiler.

action An action is a thread entry point, (i.e., the target of a XPI_PARCEL_SEND). There are a number of different types of actions, along with language-specific rules for user-defined actions. See Section 6.1 for more information.

parcel continuation A parcel continuation is a parcel that will be automatically generated when an action completes. This is sometimes referred to as a *continuation action*, as all parcels encode actions. The parcel continuation is modeled as representing an entire continuation chain.

sender A thread that generates a parcel is considered the parcel's sender. In the basic model of XPI execution, parcel instances do not have receivers, XPI's active message semantics mean that a delivered parcel is in fact the thread that it spawns. This active-message abstraction can be broken through the use of the low-level parcel interface described in Chapter 4.

2.3 Interface Specification

2.3.1 XPI Interface Definitions

The XPI interface is defined in three ways.

The first is a high-level interface definition describing the name of the interface routine, the parameters to the routine, and any errors that the routine may generate.

The second is a C API specification that deals with C-specific details, like assigning C types to parameters. The C API follows the high level interface definition as closely as

possible, but may require slightly different parameters to implement some of the high-level ideas.

The third is a FORTRAN API specification.

XPI Interface (XPI_TYPE_NAME_OPERATION_NAME) . . . . . an example function

XPI_TYPE_NAME_OPERATION_NAME(p1, p2, p3)

IN	p1	an input parameter
IN/OUT	p2	a parameter that is both read and updated
OUT	p3 (optional)	an *optional* output-only parameter

[C] `XPI_Err XPI_Type_Name_operation_name(XPI_Type1_Name p1, Type2* p2, Type3* p3);`

Errors

XPI_ERR_E1	an error condition
XPI_ERR_E2	a second error condition

XPI_TYPE_NAME_OPERATION_NAME shows an example of an interface specification. All interface names will be prefixed with `XPI_` followed by the name of the implementation type that this routine operates on (e.g., `THREAD_`) and an underscore-separated list of words describing the operation (e.g., `GET_PROCESS`).

C Conventions

The C API more-or-less implements the XPI specification directly.

Most C routines will return an XPI_ERROR type that can be inspected for errors (Section 3.1 lists the entire set of XPI errors), the few that don't are routines that both (1) do not produce errors and (2) are much more convenient to use when they return their output, rather than `XPI_SUCCESS`.

C function naming will use the uppercase `XPI_`, combined with a capitalized `Type` and lower-cased, underscore-separated operation name. In rare circumstances where a `Type` is not appropriate, `XPI_operation_name` will be used (e.g., XPI_INIT).

Parameters marked IN/OUT and OUT in the XPI API will be passed by address, except in the case of XPI_ADDR parameters, which are passed by-value with the understanding that the `OUT` value is the pointed-to value in the global address space. Routines that return new XPI_ADDR values will take the XPI_ADDR by reference.

The XPI interface may be specified using data type concepts that can't be represented as a single value in C, for instance a "list" of XPI_ADDR-esses. In this case, the C API may expand the parameter list as necessary to describe the concept, e.g., adding a size-of-list parameter in the case of a list or array.

Fortran Conventions

Developer's note. Fortran conventions, and the Fortran interface, have not yet been implemented.

2.3.2 Action Specifications

XPI defines a number of builtin actions (see Section 6.1.1 for details) that an implementation must provide. Actions are XPI thread entry points similar to active message handler code, and are specified as parcel targets. An example of an action specification follows.

XPI Interface (XPI_TYPE_NAME_OPERATION_NAME) . . . . . an example action

XPI_TYPE_NAME_OPERATION_NAME_ACTION(addr, in) CONTINUE(val)

IN	addr	the target's global address
IN	in	an input parameter
CONT	val	the future forwarded to the parcel continuation

[C]
```
XPI_Err XPI_TYPE_NAME_OPERATION_NAME_ACTION(XPI_Addr addr, Type2 in);
   /* CONTINUE(Type3 val) */
```

The name of an action will contain the string suffix `_ACTION` to distinguish it from regular runtime functions. Unlike function-based interfaces, actions may not have IN/OUT or OUT parameter, instead, they forward their "return" value to their continuation (see Section 6.3 for more details about continuations). The action definition describes the value passed to the parcel continuation using the CONTINUE and CONT keywords. The C identifier for the action will be in all caps to distinguish it from function-based alternatives. The C-API for an action will specify the continued type as a comment.

XPI_TYPE_NAME_OPERATION_NAME(addr, in, val)

IN	addr	the target's global address
IN	in	an input parameter
IN	val (optional)	a future representing the value forwarded to the parcel continuation

[C]
```
XPI_Err XPI_Type_Name_operation_name(XPI_Addr addr, Type2 in, XPI_Addr val);
```

Actions will often have an *asynchronous* function-based interface, that simplifies their use when application developers prefer a request-response style or programming, rather than the continuation-passing style provided with parcel continuations. These asynchronous functions will take a future (Section 7.3.1) that represents the completed response. These functions will have the same name as the action definition, without the _ACTION suffix.

XPI_TYPE_NAME_OPERATION_NAME_SYNC(addr, in)

IN	addr	the target's global address
IN	in	an input parameter

[C]
```
XPI_Err XPI_Type_Name_operation_name_sync(XPI_Addr addr, Type2 in);
```

In addition, some actions will have a *synchronous* function-based interface that presents a remote-procedure-call interface to the application developer. These actions will have the same name as the asynchronous function, with the string `_SYNC` added to the operation name. These synchronous functions contain an implicit XPI_THREAD_WAIT semantics, though the implementation may not use one.

Errors

XPI_ERR_E1	an error condition
XPI_ERR_E2	a second error condition

2.3.3 Handle Type Specification

Often, XPI exposes handles for local internal resources. Each resource type will have its own handle type, defined using the following format.

XPI Interface (XPI_TYPE_NAME) ..... a handle type

XPI_TYPE_NAME_HANDLE

[C]
```
typedef int XPI_Type_Name;
```

The XPI specification uses the identifier suffix `_HANDLE` to make it clear that this is a specification for a type handle. The C declaration will be that of a `typedef` of the form `XPI_Type` without the handle suffix, as the declaration makes it clear that this is a handle.

Developer's note. Fortran conventions are not yet complete.

Chapter 3

Miscellaneous

3.1 Error Handling

XPI Interface (XPI_ERROR) ..... error type

XPI_ERROR

[C]
```
typedef int XPI_Err;
```

XPI errors are represented as integer error codes. Success will always be indicated with `XPI_SUCCESS`. Chapter B contains a list of the error codes currently in use, along with a reference to the context in which they are used. The detailed description of the code will be found in the referenced location.

Most XPI API routines will return an error code that can be evaluated.

3.2 Initialization and Shutdown

The following interface routines handle initialization and shutdown of the runtime system. They are native routines, and should not be used in the context of an XPI action.

XPI Interface (XPI_INIT) ..... initialize the XPI runtime

XPI_INIT(args, env)

IN/OUT	args (optional)	a list of arguments
IN/OUT	env (optional)	a list of environment variables

[C]
```
XPI_Err XPI_init(int *nargs, char ***args, char ***env);
```

This initializes the XPI runtime, using the passed arguments and the environment. C applications usually simply forward pointers to `argc`, `argv`, and `envp` (if available). XPI-specific arguments are removed from `argc` and `argv`, and the environment is updated as necessary.

XPI Interface (XPI_RUN) ..... run the XPI application

XPI_RUN(args, result)

IN	args (optional)	a list of arguments
OUT	result (optional)	the result value of the main action (59)

[C] `XPI_Err XPI_run(int argc, char* argv[], int *result);`

XPI_RUN encapsulates the creation of the main process along with its associated initial thread, the XPI_MAIN thread. XPI_RUN also manages the future required for XPI_MAIN to return a value. This routine is synchronous, and will not return until the main process terminates.

XPI Interface (XPI_FINALIZE) ..... finalize execution

XPI_FINALIZE

[C] `XPI_Err XPI_finalize();`

This terminates the execution of the XPI runtime, releasing resources acquired in XPI_INIT and XPI_RUN.

3.3 High-Level Interface Routines

The following routines are called from XPI code, but are not associated with any application objects and cannot be targeted as actions.

XPI Interface (XPI_ABORT) ..... abort execution

XPI_ABORT(code, message)

IN	code	application-specific error code
IN	message (optional)	a message to print

[C] `void XPI_abort(int code, char *message) __attribute__((noreturn));`

This unconditionally aborts the execution of an XPI application, returning control to the XPI_RUN site with the `result` set to the passed `code`, and optionally printing the message. This should clean up resources associated with the running application.

This may be an expensive operation.

XPI Interface (XPI_VERSION) ..... query library version

XPI_VERSION(major, minor, release)

OUT	major	the major version number
OUT	minor	the minor version number
OUT	release	the release number

[C] `void XPI_version(size_t *major, size_t *minor, size_t *release);`

This queries the specification version number that the XPI implementation conforms to.

3.4 Action Management

Processes manage the user actions that are targeted by parcels

The following sections describe the language-specific details for writing user actions and registering them with the runtime.

3.4.1 C Actions

User actions represented by statically compiled C functions must meet the following restricted programming interface.

1. C actions must have a return type of XPI_ERROR. This error code can be used to communicate unexpected conditions to the XPI runtime.
2. A C action's parameter must be of `void*` type, which will be bound to the parcel's argument data by the runtime.
3. A C action must terminate by returning an XPI_ERROR, or `XPI_SUCCESS`.
4. A C action must be registered using XPI_REGISTER_ACTION_WITH_KEY before being used in a parcel.

XPI Interface (XPI_REGISTER_ACTION_WITH_KEY) ..... register an action

XPI_ACTION

[C]
```
typedef XPI_Err (*XPI_Action)(void *args) /* CONT(...) */;
```

XPI_REGISTER_ACTION_WITH_KEY(function, key)

IN	function	language-specific address of the action
IN	key	application-specific unique key for this action

[C]
```
XPI_Err XPI_register_action_with_key(XPI_Action action, char *key);

#define XPI_register_action(act) XPI_register_action_with_key(act, #act)
```

This registers an action with the runtime. It must be performed in order to send parcels to the action. It must be performed in a native C thread run on each locality, and may be performed as part of a static constructor. Builtin actions do not need to be registered before use, and have keys equivalent to their symbol names in C. The C interface provides a registration macro that uses the symbol name as a key. Registering two actions with the same key results in undefined behavior.

3.4.2 Fortran Actions

Developer's note. Fortran bindings are not yet specified.

Chapter 4

Parcels

XPI models distributed computation using an active message and continuation model of execution, exposed via the parcel interface. A parcel encodes a chain of distributed execution, each link of which is an XPI thread (Chapter 6) resulting from an active message delivery. Parcels model five components.

- A target *action* (Section 6.1). This identifies the operation that should be performed on parcel delivery.
- A target *address* (Chapter 5). This global address establishes the location at which the target action *should* execute. With the exception of parcels targeting LCO actions (Section 6.1.4), and some primitive actions, the runtime may ignore this hint and execute the action at an arbitrary location.
- An *environment*. This is an untyped data block establishing the execution environment for the thread resulting from parcel delivery.
- The *argument* data. This is an untyped data block establishing the argument data passed to the action for the thread resulting from parcel delivery.
- A continuation stack. This is a stack of action, address, environment triples that encodes the set of continuation parcels that represents the *rest* of the computation for this chain of execution.

Programming with parcels is a low-level analog to programming in a continuation passing style, or with a continuation-based language. Semantics for such languages often model `thread_spawn` and `call/cc` using a large step operational semantics, either in shared memory or eliding the messaging inherent in distributed execution. Parcels make this communication explicit, modeling `thread_spawn` as XPI_PARCEL_SEND and `call/cc` as XPI_CONTINUE, and allowing programmers—or languages targeting XPI—to manipulate their own continuation stack directly.

The parcel model does not restrict the programming language or style used to define actions—with the constraint that at least a native C interface must be available from the source language. In particular, actions defined using C execute using the standard C-ABI including stacks (though of course it is possible to write C-based actions in a continuation passing style locally).

4.1 Parcel Generation

XPI Interface (XPI_PARCEL) ..... a handle to an XPI parcel

XPI_PARCEL_HANDLE

[C]
```
typedef int XPI_Parcel;
```

Parcels are considered XPI implementation resources and are exposed to application developers through parcel handles. Attempted use of an invalid parcel handle will generally result in an XPI_ERR_INV_PARCEL error.

XPI Interface (XPI_PARCEL_CREATE) ..... create a new empty parcel

XPI_PARCEL_CREATE(parcel)

OUT	parcel	a handle to the new parcel

[C]
```
XPI_Err XPI_Parcel_create(XPI_Parcel *parcel);
```

Errors

XPI_ERR_NO_MEM	not enough local resources to perform this operation

XPI_PARCEL_CREATE allocates a new parcel, and returns a handle for it. Allocated handles must be freed with XPI_PARCEL_FREE before a thread terminates, or the runtime resources associated with the parcel will leak. Handles are a scarce resource and XPI_-PARCEL_CREATE will produce an XPI_ERR_NO_MEM error if the request cannot be satisfied.

XPI Interface (XPI_PARCEL_FREE) ..... free a parcel

XPI_PARCEL_FREE(parcel)

IN	parcel	a handle to the parcel to free

[C]
```
XPI_Err XPI_Parcel_free(XPI_Parcel parcel);
```

Parcel handles are scarce resources and must be freed after use to avoid leaking the runtime resources allocated with them. It is not an error to attempt to free an invalid parcel handle. Parcel handles may be reissued, so freeing a handle twice may have unexpected results.

4.2 Target Field Accessors

XPI Interface (XPI_PARCEL_SET_ADDR) ..... set a parcel's target address

XPI_PARCEL_SET_ADDR(parcel, address)

IN	parcel	the parcel handle
IN	address	the target action

[C] `XPI_Err XPI_Parcel_set_addr(XPI_Parcel parcel, XPI_Addr addr);`

Errors

XPI_ERR_INV_PARCEL	the parcel handle is invalid

The target address is a global address (Chapter 5) and must have a valid mapping at the time that XPI_PARCEL_SEND or XPI_CONTINUE is performed. XPI_NULL is always a valid address, and indicates that there is no preferred processing location for this parcel.

XPI Interface (XPI_PARCEL_SET_ACTION) ..... set a parcel's target action

XPI_PARCEL_SET_ACTION(parcel, action)

IN	parcel	the parcel handle
IN	action	the target action

[C] `XPI_Err XPI_Parcel_set_action(XPI_Parcel parcel, XPI_Action action);`

Errors

XPI_ERR_INV_PARCEL	the parcel handle is invalid

The action must correspond to an action (Section 6.1) that was registered with the XPI runtime using XPI_REGISTER_ACTION_WITH_KEY prior to XPI_PARCEL_SEND being called. An ACTION_NULL action is always a valid action, and indicates that no further continuations should be processed.

XPI Interface (XPI_PARCEL_SET_ENV) ..... set a parcel's target environment data

XPI_PARCEL_SET_ENV(parcel, data)

IN	parcel	the parcel handle
IN	data	the target environment data

[C] `XPI_Err XPI_Parcel_set_env(XPI_Parcel parcel, size_t bytes, void *data);`

Errors

XPI_ERR_INV_PARCEL	the parcel handle is invalid

The parcel environment data is an untyped buffer available to the XPI thread that results from delivery of the parcel, through the XPI_THREAD_GET_ENVIRONMENT routine. This data is preserved by the parcel XPI_PARCEL_PUSH and XPI_PARCEL_POP operations.

Rationale. The environment data block is intended to support a continuation closure's environment bindings for the associate action's variables.

XPI Interface (XPI_PARCEL_SET_DATA) ..... set a parcel's argument data

XPI_PARCEL_SET_DATA(parcel, data)

IN	parcel	the parcel handle

IN	data	the argument data

[C] `XPI_Err XPI_Parcel_set_data(XPI_Parcel parcel, size_t bytes, void *data);`

Errors

XPI_ERR_INV_PARCEL	the parcel handle is invalid

The argument data is an untyped buffer available to the XPI thread that results from delivery of the parcel, and is bound to the thread's action argument parameter. This data is not modified by either the XPI_PARCEL_PUSH or XPI_PARCEL_POP operations.

Rationale. The argument data is intended to provide a continuation closure's argument binding. This buffer communicates the result of a thread to its continuation.

4.3 Continuation Stack Management

A parcel provides a stack push, and asynchronous pop, interface in order to manage the continuation stack.

XPI Interface (XPI_PARCEL_PUSH) ..... push the continuation stack

XPI_PARCEL_PUSH(parcel)

IN	parcel	a handle to the parcel

[C] `XPI_Err XPI_Parcel_push(XPI_Parcel parcel);`

Errors

XPI_ERR_INV_PARCEL	the parcel handle is invalid
XPI_ERR_NO_MEM	not enough local resources to perform this operation

XPI_PARCEL_PUSH allocates a new record, copies the current target action, address, and environment into the record, and pushes it onto the top of the stack.

XPI_PARCEL_PUSH does not modify the target fields, thus a XPI_PARCEL_PUSH followed immediately by a XPI_PARCEL_SEND will push the same action twice. This technique may be used to implement distributed loops or recursion.

XPI Interface (XPI_PARCEL_POP) ..... pop the closure off the top of a parcel's stack

XPI_PARCEL_POP(parcel, complete)

IN	parcel	a handle to the parcel
IN	complete (optional)	the address of a future representing the completion of the pop

[C] `XPI_Err XPI_Parcel_pop(XPI_Parcel parcel, XPI_Addr complete);`

Errors

XPI_ERR_INV_PARCEL	the parcel handle is invalid

XPI_PARCEL_POP copies the data fields from the top of the continuation stack into the target action, address, and environment, freeing the previous environment buffer if necessary, and then pops the record off of the stack.

XPI_PARCEL_POP is a locally asynchronous operation. Its completion is signaled through the use of the complete future, which the calling thread should wait on. This future should be allocated with a size of `0`.

Repeatedly popping the continuation stack without waiting for the previous XPI_PARCEL_POP operations to complete is not an error. The complete future is only necessary for inspecting the state of the parcel's fields.

. The state of a parcel's fields are unstable between the time of a XPI_PARCEL_POP and its completion. Overlapped XPI_PARCEL_POP operations may be pipelined, so only the most recent XPI_PARCEL_POP operation can be properly waited on.

Popping an empty stack is not an error, and has the side effect of resetting the target action, address, and environment fields to their default, freeing the previous environment buffer if necessary.

Rationale. While the continuation stack is modeled as a local block, the XPI_PARCEL_POP operation is asynchronous so as to allow an XPI implementation to implement a continuation stack through a set of distributed "frames," which can optimize the bandwidth required for distributed operation.

4.4 Sending a Parcel

XPI Interface (XPI_PARCEL_SEND) ..... send a parcel

XPI_PARCEL_SEND(parcel, complete, future)

IN	parcel	the parcel handle
IN	complete (optional)	the address of a future representing the local completion of the send
IN	future (optional)	the address of a future representing the spawned thread's global address

[C]
```
XPI_Err XPI_Parcel_send(XPI_Parcel parcel, XPI_Addr future);
```

Errors

XPI_ERR_INV_PARCEL	the parcel handle is invalid
XPI_ERR_INV_ADDR	the target address is invalid

XPI_PARCEL_SEND sends a parcel, initiating the distributed chain of operation encoded by it.

Sending a parcel is a locally asynchronous operation. The parcel structure is unsafe to inspect or update until the local operation is complete. An application that wants to reuse parcel structures must provide a future as the complete parameter to XPI_PARCEL_SEND, and may wait for local completion using it.

In addition to waiting for the local send operation to complete, the sender is able to wait for remote completion of the send—along with its active message instantiation, by providing a future as the future parameter to XPI_PARCEL_SEND and waiting on it. In

addition to signaling remote completion, this future will provide the global address of the thread that is spawned to execute the remote action. Such information can be used to query the status of the thread.

. Due to the active message nature of parcels, XPI_PARCEL_SEND is effectively equivalent to a `thread_spawn` operation, in a distributed setting.

Advice to implementors. XPI_PARCEL_SEND permits optimized library implementations that may bypass the networking layer entirely, if possible. In particular, parcels targeting local addresses may merely be instantiated as local threads, or even run as part of a loop. The only restriction for a XPI_PARCEL_SEND implementation is that, if a program does not execute XPI_PARCEL_SELECT, then no thread should be able to observe semantics incompatible with the thread specification given in Chapter 6.

Developer's note. It's not clear that there is utility in this mechanism for getting the address of the spawned thread, however this functionality is consistent with initial XPI specification attempts and can be easily optimized at runtime, so we include it.

4.5 Apply

XPI Interface (XPI_APPLY) ..... high-level asynchronous function call interface

XPI_APPLY(target, action, data, future)

IN	target	target address
IN	action	target action
IN	data	the parameter data for the action
IN	future (optional)	a future to receive the continuation data

[C]
```
XPI_Err XPI_Parcel_apply(XPI_Addr target, void *action,
                         size_t bytes, void *data, XPI_Addr future);
```

XPI_APPLY_SYNC(target, action, data)

IN	target	target address
IN	action	target action
IN	data	the parameter data for the action

[C]
```
XPI_Err XPI_Parcel_apply_sync(XPI_Addr target, void *action,
                              size_t bytes, void *data);
```

Errors

XPI_ERR_INV_ADDR	the target global address is invalid

XPI_APPLY encapsulates the steps required to assemble and send a parcel and continuation to effect a remote-procedure call. If the action does not continue a value, or if the continued value is to be ignored, then future should be set to XPI_NULL.

4.6 Advanced

Some advanced programmers may need to inspect parcels manually, before they result in active threads. The interface for such inspection is given here. This functionality breaks the active message abstraction of parcels, and should be used carefully.

In particular, XPI_PARCEL_SEND is not obliged to interact with the network layer, if the XPI_PARCEL_SEND can be satisfied otherwise without effecting the semantics of the program, then the sent parcel may not be receivable with XPI_PARCEL_SELECT.

> *Rationale.* It is expected that only low level system code will need to receive parcels explicitly. Such code is likely simply forwarding parcels, or relocating data for work balancing.

XPI Interface (XPI_PARCEL_SELECT) ..... explicitly receive a parcel

XPI_PARCEL_SELECT(match, parcels, matched)

IN	match	a pattern to match
IN/OUT	parcels	an n-element buffer to store matched parcel handles
OUT	matched	the number of parcels matched

[C]
```
XPI_Err XPI_Parcel_select(char *match, size_t n, XPI_Parcel parcels[],
                          size_t *matched);
```

This can be used to explicitly receive parcels from the network layer.

> *Developer's note.* Currently, match is ignored. This routine simply receives the first parcel available at the locality at which it is called, and returns 1 for the matched value, assuming that there is at least one slot available in parcels.

XPI Interface (XPI_PARCEL_GET_ADDR) ..... get a parcel's target address

XPI_PARCEL_GET_ADDR(parcel, parcel)

IN	parcel	the parcel handle
OUT	parcel	the target address

[C]
```
XPI_Err XPI_Parcel_get_addr(XPI_Parcel parcel, XPI_Addr *addr);
```

Errors

XPI_ERR_INV_PARCEL	the parcel handle is invalid

XPI Interface (XPI_PARCEL_GET_ACTION) ..... get a parcel's target action

XPI_PARCEL_GET_ACTION(parcel, action)

IN	parcel	the parcel handle
OUT	action	the target action

[C]
```
XPI_Err XPI_Parcel_get_action(XPI_Parcel parcel, XPI_Action *action);
```

Errors

XPI_ERR_INV_PARCEL	the parcel handle is invalid

XPI Interface (XPI_PARCEL_GET_ENV) ..... get a parcel's target environment data

XPI_PARCEL_GET_ENV(parcel, data)

IN	parcel	the parcel handle
OUT	data	the target environment data

[C]
```
XPI_Err XPI_Parcel_get_env(XPI_Parcel parcel, size_t *bytes, void **data);
```

Errors

XPI_ERR_INV_PARCEL	the parcel handle is invalid

XPI Interface (XPI_PARCEL_GET_DATA) ..... get a parcel's argument data

XPI_PARCEL_GET_DATA(parcel, data)

IN	parcel	the parcel handle
OUT	data	the parcel's argument data

[C]
```
XPI_Err XPI_Parcel_get_data(XPI_Parcel parcel, size_t *bytes, void **data);
```

Errors

XPI_ERR_INV_PARCEL	the parcel handle is invalid

Chapter 5

Global Address Space

The global address space is at the heart of parallel programming with XPI. It is a global virtual byte-addressable address space defined in terms of a native parcel interface in Section 5.2.

Explicit parcel programming can be inconvenient and potentially inefficient when messaging patterns that both match common usages and have hardware support. For example, the XPI_AGAS_LOAD_ACTION is often used in conjunction with a future to perform a traditional, two-message, load operation. Where such patterns are common it makes sense to add function-based interfaces that simplify development and maintenance of code, and expose optimization opportunities.

Section 5.3.1 specifies a future-based (Section 7.3.1) asynchronous function-based interface that captures common patterns. Section 5.3.2 specifies a purely synchronous function-based interface to memory. Synchronous functions can restrict throughput, as they imply a strict ordering of memory accesses, however they are included with the intention of providing low overhead operations where such an ordering is logically required.

5.1 Address Space Properties

5.1.1 Atomicity

Primitive memory actions of the same size at the same global address are guaranteed to be performed atomically, i.e., a load action will never see data values corresponding to a partially completed store or compare-and-swap operation. This constraint implies that there exists some serial history of operations for each location, however XPI requires no ordering on parcels and thus this history is only observable with appropriate synchronization, e.g., through ordered use of the asynchronous function-based interface (Section 5.3.1) or sole use of the synchronous function-based interface (Section 5.3.2).

As with strict-aliasing in C, concurrent access to the same (or overlapping) location using types of a different size may produce an arbitrary value. The exception to this is that single-byte access to any location is always safe—in cases where a single-byte access is always atomic and can safely alias any global address.

5.1.2 Alignment

XPI's global memory space only supports accesses to aligned addresses, i.e., a 4-byte operation must be performed to a 4-byte aligned address, an 8-byte operation must be performed to an 8-byte aligned address, etc. . ..

> *Rationale.* Requiring aligned accesses to global memory may permit more efficient implementations of the address space, particularly with respect to the atomicity requirement. Unaligned accesses may be performed non-atomically with XPI_PROCESS_MEMCPY.

5.1.3 Endianness

XPI's native parcel interface is defined to interact with the memory space in a little-endian manner.

> *Rationale.* The memory space is defined as untyped bytes, but the native interface defines larger accesses as a result of performance and programmability concerns. Given the possibility of reading and writing some byte address using different native sizes—one of which must be single-byte access (Section 5.1.1), we must specify an endianness. The choice of little-endianness is arbitrary.
>
> This is a consequence of defining the native memory interface in terms of a set of C-typed operations, e.g., `uint64_t`, **`double`**, etc., rather than as byte-array (`uint8_t[]`) operations. The interface is designed in such a manner to permit efficient library-based function call operations that act on machine registers, byte-arrays must have addresses and be passed by address.

5.2 Native Parcel Interface

XPI's global address space defines an asynchronous memory interface that is accessed using parcels with a set of predefined, primitive system actions (Section 6.1.1).

XPI Interface (XPI_AGAS_LOAD) . . . . . load a value from the global address space

XPI_AGAS_LOAD_ACTION(addr) CONTINUE(val)

IN	addr	the global address from which to load
CONT	val	the loaded value

[C]
```
XPI_Err XPI_AGAS_LOAD_U8_ACTION() /* CONT(uint8_t val) */;
XPI_Err XPI_AGAS_LOAD_U16_ACTION() /* CONT(uint16_t val) */;
XPI_Err XPI_AGAS_LOAD_U32_ACTION() /* CONT(uint32_t val) */;
XPI_Err XPI_AGAS_LOAD_U64_ACTION() /* CONT(uint64_t val) */;
XPI_Err XPI_AGAS_LOAD_U128_ACTION() /* CONT(__uint128_t val) */;
XPI_Err XPI_AGAS_LOAD_S8_ACTION() /* CONT(int8_t val) */;
XPI_Err XPI_AGAS_LOAD_S16_ACTION() /* CONT(int16_t val) */;
XPI_Err XPI_AGAS_LOAD_S32_ACTION() /* CONT(int32_t val) */;
XPI_Err XPI_AGAS_LOAD_S64_ACTION() /* CONT(int64_t val) */;
XPI_Err XPI_AGAS_LOAD_S128_ACTION() /* CONT(__int128_t val) */;
XPI_Err XPI_AGAS_LOAD_F_ACTION() /* CONT(float val) */;
XPI_Err XPI_AGAS_LOAD_D_ACTION() /* CONT(double val) */;
XPI_Err XPI_AGAS_LOAD_FC_ACTION() /* CONT(float _Complex val) */;
```

```
XPI_Err XPI_AGAS_LOAD_FD_ACTION() /* CONT(double _Complex val) */;
XPI_Err XPI_AGAS_LOAD_ADDR_ACTION() /* CONT(XPI_Addr val) */;
XPI_Err XPI_AGAS_LOAD_ADDRDIFF_ACTION() /* CONT(XPI_AddrDiff val) */;
```

Errors

XPI_ERR_INV_ADDR	the address, *addr*, is not valid

This load action is an atypical in that the action does not return a value. It simply reads the local address, and transmits the read value to its continuation action. This is often a future's trigger action (Section 7.3.1), though any type-correct continuation is valid. Parcels specifying the load action with a null continuation perform no useful work and may be suppressed entirely by optimization.

Load operations can also be performed using the XPI_AGAS_LOAD and XPI_AGAS_-LOAD_SYNC function-based interface.

XPI Interface (XPI_AGAS_STORE) ..... stores a value to the global address space

XPI_AGAS_STORE_ACTION(addr, val)

IN	addr	the global address targeted by the store
IN	val	the value to store

[C]
```
XPI_Err XPI_AGAS_STORE_U8_ACTION(uint8_t *args) /* CONT() */;
XPI_Err XPI_AGAS_STORE_U16_ACTION(uint16_t *args) /* CONT() */;
XPI_Err XPI_AGAS_STORE_U32_ACTION(uint32_t *args) /* CONT() */;
XPI_Err XPI_AGAS_STORE_U64_ACTION(uint64_t *args) /* CONT() */;
XPI_Err XPI_AGAS_STORE_U128_ACTION(__uint128_t *args) /* CONT() */;
XPI_Err XPI_AGAS_STORE_S8_ACTION(int8_t *args) /* CONT() */;
XPI_Err XPI_AGAS_STORE_S16_ACTION(int16_t *args) /* CONT() */;
XPI_Err XPI_AGAS_STORE_S32_ACTION(int32_t *args) /* CONT() */;
XPI_Err XPI_AGAS_STORE_S64_ACTION(int64_t *args) /* CONT() */;
XPI_Err XPI_AGAS_STORE_S128_ACTION(__int128_t *args) /* CONT() */;
XPI_Err XPI_AGAS_STORE_F_ACTION(float *args) /* CONT() */;
XPI_Err XPI_AGAS_STORE_D_ACTION(double *args) /* CONT() */;
XPI_Err XPI_AGAS_STORE_FC_ACTION(float _Complex *args) /* CONT() */;
XPI_Err XPI_AGAS_STORE_DC_ACTION(double _Complex *args) /* CONT() */;
XPI_Err XPI_AGAS_STORE_ADDR_ACTION(XPI_Addr *args) /* CONT() */;
XPI_Err XPI_AGAS_STORE_ADDRDIFF_ACTION(XPI_AddrDiff *args) /* CONT() */;
```

Errors

Store operations atomically update the target location with the passed value. The store operation does not continue a value.

Store operations can also be performed using the XPI_AGAS_STORE and XPI_AGAS_-STORE_SYNC function-based interface.

XPI Interface (XPI_AGAS_CAS) ..... atomic read-modify-write support

XPI_AGAS_CAS_ACTION(addr, from, to) CONTINUE(actual)

IN	addr	the global address targeted by the operation
IN	from	the value expected at addr
IN	to	the new value for addr
CONT	actual	the actual value that was seen

[C]
```
/* args[0] == from value, and args[1] == to value */

XPI_Err XPI_AGAS_CAS_U8_ACTION(uint8_t args[2])
/* CONT(uint8_t actual) */;
XPI_Err XPI_AGAS_CAS_U16_ACTION(uint16_t args[2])
   /* CONT(uint16_t actual) */;
XPI_Err XPI_AGAS_CAS_U32_ACTION(uint32_t args[2])
   /* CONT(uint32_t actual) */;
XPI_Err XPI_AGAS_CAS_U64_ACTION(uint64_t args[2])
   /* CONT(uint64_t actual) */;
XPI_Err XPI_AGAS_CAS_U128_ACTION(__uint128_t args[2])
   /* CONT(__uint128_t actual) */;
XPI_Err XPI_AGAS_CAS_S8_ACTION(int8_t args[2])
   /* CONT(int8_t actual) */;
XPI_Err XPI_AGAS_CAS_S16_ACTION(int16_t args[2])
   /* CONT(int16_t actual) */;
XPI_Err XPI_AGAS_CAS_S32_ACTION(int32_t args[2])
   /* CONT(int32_t actual) */;
XPI_Err XPI_AGAS_CAS_S64_ACTION(int64_t args[2])
   /* CONT(int64_t actual) */;
XPI_Err XPI_AGAS_CAS_S128_ACTION(__int128_t args[2])
   /* CONT(__int128_t actual) */;
XPI_Err XPI_AGAS_CAS_F_ACTION(float args[2])
   /* CONT(float actual) */;
XPI_Err XPI_AGAS_CAS_D_ACTION(double args[2])
   /* CONT(double actual) */;
XPI_Err XPI_AGAS_CAS_FC_ACTION(float _Complex args[2])
   /* CONT(float _Complex actual) */;
XPI_Err XPI_AGAS_CAS_DC_ACTION(double _Complex args[2])
   /* CONT(double _Complex actual) */;
XPI_Err XPI_AGAS_CAS_ADDR_ACTION(XPI_Addr args[2])
   /* CONT(XPI_Addr actual) */;
XPI_Err XPI_AGAS_CAS_ADDRDIFF_ACTION(XPI_AddrDiff args[2])
   /* CONT(XPI_AddrDiff actual) */;
```

Errors

The compare-and-swap action is used to perform a conditional atomic read-modify-write to a global address. As with a traditional shared-memory compare-and-swap operation, XPI_AGAS_CAS compares the current value of addr with from and if they are equal, updates the value of addr to the value specified as to. XPI_AGAS_CAS continues the actual value that was seen at addr so that the action's continuation can act appropriately.

Compare-and-swap operations can be performed using the XPI_AGAS_CAS and XPI_AGAS_CAS_SYNC function-based interface.

> *Advice to users.* While function-based interfaces to compare-and-swap exists as XPI_AGAS_CAS and XPI_AGAS_CAS_SYNC, the common shared-memory use of compare-and-swap in a loop to complete an atomic read-modify-write operation is not naturally suited to a function-based interface in XPI as it has the potential for excessive inter-node network traffic. In XPI, the XPI_AGAS_CAS_ACTION is typically used directly, and paired with a continuation that checks the result of compare-and-swap locally, at the same locality.
>
> The common *hazard pointer* design pattern used to avoid the $A \longrightarrow B \longrightarrow A$ problem associated with implementations of linked data structures using compare-and-swap

requires an architecture to support compare-and-swap operations larger than that of the native address size. Depending on the size of the implementation-defined size of an XPI_ADDR, this may not be possible to implement with XPI_AGAS_CAS. When algorithms can not tolerate $A \longrightarrow B \longrightarrow A$ occurrences custom local-control-objects must be used (Chapter 7).

5.3 Function-based Interface

To assist with programmability as well as to enable potential optimizations, XPI defines a higher-level, asynchronous function-based interface that relies on *future* Local Control Objects to return data where necessary, or to order execution as desired. Furthermore, synchronous accesses to global memory are occasionally necessary, and may provide an optimization opportunity, so XPI provides a synchronous interface to memory.

A program that uses only synchronous, function-based access to global memory should have sequentially consistent global address semantics.

Advice to users. Applications developers should avoid synchronous access to global memory unless it is unavoidable, as it introduces overheads due to waiting, and prevents continued execution of independent statements. However programmers can expect that the synchronous interface will outperform the equivalent asynchronous code described above.

5.3.1 Asynchronous Functions

XPI_AGAS_LOAD(addr, future)

IN	addr	the global address from which to load
IN	future	a future representing the loaded value

[C]
```
XPI_Err XPI_Agas_load_u8(XPI_Addr addr, XPI_Addr future);
XPI_Err XPI_Agas_load_u16(XPI_Addr addr, XPI_Addr future);
XPI_Err XPI_Agas_load_u32(XPI_Addr addr, XPI_Addr future);
XPI_Err XPI_Agas_load_u64(XPI_Addr addr, XPI_Addr future);
XPI_Err XPI_Agas_load_u128(XPI_Addr addr, XPI_Addr future);
XPI_Err XPI_Agas_load_s8(XPI_Addr addr, XPI_Addr future);
XPI_Err XPI_Agas_load_s16(XPI_Addr addr, XPI_Addr future);
XPI_Err XPI_Agas_load_s32(XPI_Addr addr, XPI_Addr future);
XPI_Err XPI_Agas_load_s64(XPI_Addr addr, XPI_Addr future);
XPI_Err XPI_Agas_load_s128(XPI_Addr addr, XPI_Addr future);
XPI_Err XPI_Agas_load_f(XPI_Addr addr, XPI_Addr future);
XPI_Err XPI_Agas_load_d(XPI_Addr addr, XPI_Addr future);
XPI_Err XPI_Agas_load_fc(XPI_Addr addr, XPI_Addr future);
XPI_Err XPI_Agas_load_dc(XPI_Addr addr, XPI_Addr future);
XPI_Err XPI_Agas_load_addr(XPI_Addr addr, XPI_Addr future);
XPI_Err XPI_Agas_load_addrdiff(XPI_Addr addr, XPI_Addr future);
```

[C11]
```
#define XPI_Agas_load(addr, val, future)        \
   _Generic((val),                              \
          uint8_t     :XPI_Agas_load_u8,        \
          uint16_t    :XPI_Agas_load_u16,       \
          uint32_t    :XPI_Agas_load_u32,       \
          uint64_t    :XPI_Agas_load_u64,       \
          __uint128_t :XPI_Agas_load_u128,      \
```

```
        int8_t      :XPI_Agas_load_s8,        \
        int16_t     :XPI_Agas_load_s16,       \
        int32_t     :XPI_Agas_load_s32,       \
        int64_t     :XPI_Agas_load_s64,       \
        __int128_t  :XPI_Agas_load_s128,      \
        float       :XPI_Agas_load_f,         \
        double      :XPI_Agas_load_d,         \
        float _Complex :XPI_Agas_load_fc,     \
        double _Complex :XPI_Agas_load_dc)(addr, future)
```

XPI_AGAS_STORE(addr, val, future)

IN	addr	the global address targeted by the store
IN	val	the value to store
IN	future (optional)	a future that can be used for ordering

[C]
```
XPI_Err XPI_Agas_store_u8(XPI_Addr addr, uint8_t val, XPI_Addr future);
XPI_Err XPI_Agas_store_u16(XPI_Addr addr, uint16_t val, XPI_Addr future);
XPI_Err XPI_Agas_store_u32(XPI_Addr addr, uint32_t val, XPI_Addr future);
XPI_Err XPI_Agas_store_u64(XPI_Addr addr, uint64_t val, XPI_Addr future);
XPI_Err XPI_Agas_store_u128(XPI_Addr addr, __uint128_t val, XPI_Addr future);
XPI_Err XPI_Agas_store_s8(XPI_Addr addr, int8_t val, XPI_Addr future);
XPI_Err XPI_Agas_store_s16(XPI_Addr addr, int16_t val, XPI_Addr future);
XPI_Err XPI_Agas_store_s32(XPI_Addr addr, int32_t val, XPI_Addr future);
XPI_Err XPI_Agas_store_s64(XPI_Addr addr, int64_t val, XPI_Addr future);
XPI_Err XPI_Agas_store_s128(XPI_Addr addr, __int128_t val, XPI_Addr future);
XPI_Err XPI_Agas_store_f(XPI_Addr addr, float val, XPI_Addr future);
XPI_Err XPI_Agas_store_d(XPI_Addr addr, double val, XPI_Addr future);
XPI_Err XPI_Agas_store_fc(XPI_Addr addr, float _Complex val, XPI_Addr future);
XPI_Err XPI_Agas_store_dc(XPI_Addr addr, double _Complex val, XPI_Addr future);
XPI_Err XPI_Agas_store_addr(XPI_Addr addr, XPI_Addr val, XPI_Addr future);
XPI_Err XPI_Agas_store_addrdiff(XPI_Addr addr, XPI_AddrDiff val, XPI_Addr future);
```

[C11]
```
#define XPI_Agas_store(addr, val, future)          \
   _Generic((val),                                 \
          uint8_t     :XPI_Agas_store_u8,          \
          uint16_t    :XPI_Agas_store_u16,         \
          uint32_t    :XPI_Agas_store_u32,         \
          uint64_t    :XPI_Agas_store_u64,         \
          __uint128_t :XPI_Agas_store_u128,        \
          int8_t      :XPI_Agas_store_s8,          \
          int16_t     :XPI_Agas_store_s16,         \
          int32_t     :XPI_Agas_store_s32,         \
          int64_t     :XPI_Agas_store_s64,         \
          __int128_t  :XPI_Agas_store_s128,        \
          float       :XPI_Agas_store_f,           \
          double      :XPI_Agas_store_d,           \
          float _Complex :XPI_Agas_store_fc,       \
          double _Complex :XPI_Agas_store_dc)(addr, val, future)
```

XPI_AGAS_CAS(addr, from, to, future)

IN	addr	the global address targeted by the operation
IN	from	the value expected at addr
IN	to	the new value for addr
IN	future	a future representing the actual value seen

[C]
```
XPI_Err XPI_Agas_cas_u8(XPI_Addr addr, uint8_t from, uint8_t to,
                        XPI_Addr future);
XPI_Err XPI_Agas_cas_u16(XPI_Addr addr, uint16_t from, uint16_t to,
                         XPI_Addr future);
XPI_Err XPI_Agas_cas_u32(XPI_Addr addr, uint32_t from, uint32_t to,
                         XPI_Addr future);
XPI_Err XPI_Agas_cas_u64(XPI_Addr addr, uint64_t from, uint64_t to,
                         XPI_Addr future);
XPI_Err XPI_Agas_cas_u128(XPI_Addr addr, __uint128_t from, __uint128_t to,
                          XPI_Addr future);
XPI_Err XPI_Agas_cas_s8(XPI_Addr addr, int8_t from, int8_t to,
                        XPI_Addr future);
XPI_Err XPI_Agas_cas_s16(XPI_Addr addr, int16_t from, int16_t to,
                         XPI_Addr future);
XPI_Err XPI_Agas_cas_s32(XPI_Addr addr, int32_t from, int32_t to,
                         XPI_Addr future);
XPI_Err XPI_Agas_cas_s64(XPI_Addr addr, int64_t from, int64_t to,
                         XPI_Addr future);
XPI_Err XPI_Agas_cas_s128(XPI_Addr addr, __int128_t from, __int128_t to,
                          XPI_Addr future);
XPI_Err XPI_Agas_cas_f(XPI_Addr addr, float from, float to,
                       XPI_Addr future);
XPI_Err XPI_Agas_cas_d(XPI_Addr addr, double from, double to,
                       XPI_Addr future);
XPI_Err XPI_Agas_cas_fc(XPI_Addr addr, float _Complex from,
                        float _Complex to, XPI_Addr future);
XPI_Err XPI_Agas_cas_dc(XPI_Addr addr, double _Complex from,
                        double _Complex to,
                        XPI_Addr future);
XPI_Err XPI_Agas_cas_addr(XPI_Addr addr, XPI_Addr from, XPI_Addr to,
                          XPI_Addr future);
XPI_Err XPI_Agas_cas_addrdiff(XPI_Addr addr, XPI_AddrDiff from, XPI_AddrDiff to,
                              XPI_Addr future);
```

[C11]
```
#define XPI_Agas_cas(addr, from, to, future)          \
   _Generic((from),                                   \
          uint8_t     :XPI_Agas_cas_u8,               \
          uint16_t    :XPI_Agas_cas_u16,              \
          uint32_t    :XPI_Agas_cas_u32,              \
          uint64_t    :XPI_Agas_cas_u64,              \
          __uint128_t :XPI_Agas_cas_u128,             \
          int8_t      :XPI_Agas_cas_s8,               \
          int16_t     :XPI_Agas_cas_s16,              \
          int32_t     :XPI_Agas_cas_s32,              \
          int64_t     :XPI_Agas_cas_s64,              \
          __int128_t  :XPI_Agas_cas_s128,             \
          float       :XPI_Agas_cas_f,                \
          double      :XPI_Agas_cas_d,                \
          float _Complex :XPI_Agas_cas_fc,            \
          double _Complex :XPI_Agas_cas_dc)(addr, from, to, future)
```

5.3.2 Synchronous Functions

XPI_AGAS_LOAD_SYNC(addr)

IN	addr	the global address from which to load

[C]
```
XPI_Err XPI_Agas_load_u8_sync(XPI_Addr addr, uint8_t *val);
```

```
XPI_Err XPI_Agas_load_u16_sync(XPI_Addr addr, uint16_t *val);
XPI_Err XPI_Agas_load_u32_sync(XPI_Addr addr, uint32_t *val);
XPI_Err XPI_Agas_load_u64_sync(XPI_Addr addr, uint64_t *val);
XPI_Err XPI_Agas_load_u128_sync(XPI_Addr addr, __uint128_t *val);
XPI_Err XPI_Agas_load_s8_sync(XPI_Addr addr, int8_t *val);
XPI_Err XPI_Agas_load_s16_sync(XPI_Addr addr, int16_t *val);
XPI_Err XPI_Agas_load_s32_sync(XPI_Addr addr, int32_t *val);
XPI_Err XPI_Agas_load_s64_sync(XPI_Addr addr, int64_t *val);
XPI_Err XPI_Agas_load_s128_sync(XPI_Addr addr, __int128_t *val);
XPI_Err XPI_Agas_load_f_sync(XPI_Addr addr, float *val);
XPI_Err XPI_Agas_load_d_sync(XPI_Addr addr, double *val);
XPI_Err XPI_Agas_load_fc_sync(XPI_Addr addr, float _Complex *val);
XPI_Err XPI_Agas_load_dc_sync(XPI_Addr addr, double _Complex *val);
XPI_Err XPI_Agas_load_addr_sync(XPI_Addr addr, XPI_Addr *val);
XPI_Err XPI_Agas_load_addrdiff_sync(XPI_Addr addr, XPI_AddrDiff *val);
```

[C11]
```
#define XPI_Agas_load_sync(addr, val)           \
   _Generic((val),                              \
          uint8_t     :XPI_Agas_load_u8_sync,   \
          uint16_t    :XPI_Agas_load_u16_sync,  \
          uint32_t    :XPI_Agas_load_u32_sync,  \
          uint64_t    :XPI_Agas_load_u64_sync,  \
          __uint128_t :XPI_Agas_load_u128_sync, \
          int8_t      :XPI_Agas_load_s8_sync,   \
          int16_t     :XPI_Agas_load_s16_sync,  \
          int32_t     :XPI_Agas_load_s32_sync,  \
          int64_t     :XPI_Agas_load_s64_sync,  \
          __int128_t  :XPI_Agas_load_s128_sync, \
          float       :XPI_Agas_load_f_sync,    \
          double      :XPI_Agas_load_d_sync,    \
          float _Complex :XPI_Agas_load_fc_sync, \
          double _Complex :XPI_Agas_load_dc_sync)(addr, val)
```

XPI_AGAS_STORE_SYNC(addr, val)

IN	addr	the global address targeted by the store
IN	val	the value to store

[C]
```
XPI_Err XPI_Agas_store_u8_sync(XPI_Addr addr, uint8_t val);
XPI_Err XPI_Agas_store_u16_sync(XPI_Addr addr, uint16_t val);
XPI_Err XPI_Agas_store_u32_sync(XPI_Addr addr, uint32_t val);
XPI_Err XPI_Agas_store_u64_sync(XPI_Addr addr, uint64_t val);
XPI_Err XPI_Agas_store_u128_sync(XPI_Addr addr, __uint128_t val);
XPI_Err XPI_Agas_store_s8_sync(XPI_Addr addr, int8_t val);
XPI_Err XPI_Agas_store_s16_sync(XPI_Addr addr, int16_t val);
XPI_Err XPI_Agas_store_s32_sync(XPI_Addr addr, int32_t val);
XPI_Err XPI_Agas_store_s64_sync(XPI_Addr addr, int64_t val);
XPI_Err XPI_Agas_store_s128_sync(XPI_Addr addr, __int128_t val);
XPI_Err XPI_Agas_store_f_sync(XPI_Addr addr, float val);
XPI_Err XPI_Agas_store_d_sync(XPI_Addr addr, double val);
XPI_Err XPI_Agas_store_fc_sync(XPI_Addr addr, float _Complex val);
XPI_Err XPI_Agas_store_dc_sync(XPI_Addr addr, double _Complex val);
XPI_Err XPI_Agas_store_addr_sync(XPI_Addr addr, XPI_Addr val);
XPI_Err XPI_Agas_store_addrdiff_sync(XPI_Addr addr, XPI_AddrDiff val);
```

[C11]
```
#define XPI_Agas_store_sync(addr, val)            \
   _Generic((val),                                \
          uint8_t     :XPI_Agas_store_u8_sync,     \
```

```
            uint16_t    :XPI_Agas_store_u16_sync,     \
            uint32_t    :XPI_Agas_store_u32_sync,     \
            uint64_t    :XPI_Agas_store_u64_sync,     \
            __uint128_t :XPI_Agas_store_u128_sync,    \
            int8_t      :XPI_Agas_store_s8_sync,      \
            int16_t     :XPI_Agas_store_s16_sync,     \
            int32_t     :XPI_Agas_store_s32_sync,     \
            int64_t     :XPI_Agas_store_s64_sync,     \
            __int128_t  :XPI_Agas_store_s128_sync,    \
            float       :XPI_Agas_store_f_sync,       \
            double      :XPI_Agas_store_d_sync,       \
            float _Complex :XPI_Agas_store_fc_sync,   \
            double _Complex :XPI_Agas_store_dc_sync)(addr, val)
```

XPI_AGAS_CAS_SYNC(addr, from, to)

IN	addr	the global address targeted by the operation
IN	from	the value expected at addr
IN/OUT	to	in:the new value for addr, out:actual value seen

[C]
```
XPI_Err XPI_Agas_cas_u8_sync(XPI_Addr addr, uint8_t from, uint8_t *to);
XPI_Err XPI_Agas_cas_u16_sync(XPI_Addr addr, uint16_t from, uint16_t *to);
XPI_Err XPI_Agas_cas_u32_sync(XPI_Addr addr, uint32_t from, uint32_t *to);
XPI_Err XPI_Agas_cas_u64_sync(XPI_Addr addr, uint64_t from, uint64_t *to);
XPI_Err XPI_Agas_cas_u128_sync(XPI_Addr addr, __uint128_t from,
                        __uint128_t *to);
XPI_Err XPI_Agas_cas_s8_sync(XPI_Addr addr, int8_t from, int8_t *to);
XPI_Err XPI_Agas_cas_s16_sync(XPI_Addr addr, int16_t from, int16_t *to);
XPI_Err XPI_Agas_cas_s32_sync(XPI_Addr addr, int32_t from, int32_t *to);
XPI_Err XPI_Agas_cas_s64_sync(XPI_Addr addr, int64_t from, int64_t *to);
XPI_Err XPI_Agas_cas_s128_sync(XPI_Addr addr, __int128_t from,
                        __int128_t *to);
XPI_Err XPI_Agas_cas_f_sync(XPI_Addr addr, float from, float *to);
XPI_Err XPI_Agas_cas_d_sync(XPI_Addr addr, double from, double *to);
XPI_Err XPI_Agas_cas_fc_sync(XPI_Addr addr, float _Complex from,
                       float _Complex *to);
XPI_Err XPI_Agas_cas_dc_sync(XPI_Addr addr, double _Complex from,
                       double _Complex *to);
XPI_Err XPI_Agas_cas_addr_sync(XPI_Addr addr, XPI_Addr from,
                        XPI_Addr *to);
XPI_Err XPI_Agas_cas_addrdiff_sync(XPI_Addr addr, XPI_AddrDiff from,
                           XPI_AddrDiff *to);
```

[C11]
```
#define XPI_Agas_cas_sync(addr, from, to)          \
   _Generic((from),                                \
          uint8_t     :XPI_Agas_cas_u8_sync,       \
          uint16_t    :XPI_Agas_cas_u16_sync,      \
          uint32_t    :XPI_Agas_cas_u32_sync,      \
          uint64_t    :XPI_Agas_cas_u64_sync,      \
          __uint128_t :XPI_Agas_cas_u128_sync,     \
          int8_t      :XPI_Agas_cas_s8_sync,       \
          int16_t     :XPI_Agas_cas_s16_sync,      \
          int32_t     :XPI_Agas_cas_s32_sync,      \
          int64_t     :XPI_Agas_cas_s64_sync,      \
          __int128_t  :XPI_Agas_cas_s128_sync,     \
          float       :XPI_Agas_cas_f_sync,        \
          double      :XPI_Agas_cas_d_sync,        \
```

```
float _Complex :XPI_Agas_cas_fc_sync,     \
double _Complex :XPI_Agas_cas_dc_sync)(addr, from, to)
```

5.4 Address Space Representation

All library-based interfaces to memory require an address representation that can be manipulated by the application programmer. XPI's global address space is designed as an untyped, byte-addressable virtual address space. XPI provides the most general address representation, an opaque address structure. Such structures can be used as the target of a parcel (Chapter 4), and can be manipulation using XPI library routines to create new structures. Address structures are not handles because they have value semantics.

> *Rationale.* At this time, we do not wish to formally bound the size of the global address space, or the address representation—as this may impact the network layer implementation, thus XPI does not currently use a simple integer representation. If, in the future, the size of the XPI virtual address becomes bounded this decision can be revisited.
>
> We expect addresses and address computation to be ubiquitous, however we do not wish to require that either a) users manage memory associated with the addressing structures themselves, or b) the system garbage collect addresses, so we choose not to represent addresses as handles.
>
> *Advice to users.* Users should not assume that the interface routines provided for address computations are library routines, as an implementation may choose to implement them as macros for performance reasons.

The XPI address space is designed to mimic traditional "C-style" virtual memory, and thus we define mechanisms for address structure comparison and arithmetic. To do this, XPI requires that conforming implementatations define a signed integer type large enough to represent the difference between any two addresses in the global address space.

> *Rationale.* Given that XPI does not fix bounds for the size of the global address space, we do not know how large a difference may exist between two addresses. XPI does not define address differences as **`struct`**s as this is both inconvenient for programmers and semantically unnecessary, given that the global address space is a flat, byte-addressable space in which the difference between two addresses can be represented concisely as a signed integer.

XPI Interface (XPI_ADDR) ..... a global address handle

XPI_ADDR

[C]
```
typedef struct { /* implementation defined */ } XPI_Addr;
```

The C interface for the XPI_ADDR type must be a `struct` with value semantics.

XPI Interface (XPI_ADDRDIFF) ..... a global address difference handle

XPI_ADDRDIFF

[C] `typedef /* implementation defined signed int */ XPI_AddrDiff;`

The XPI_ADDRDIFF type will be `typedef`ed to a signed integer type large enough to represent the difference between any two addresses in the global address space. As a signed integer, all standard signed integer operations are defined on XPI_ADDRDIFF values. Note that, in C, signed integer underflow/overflow is undefined, so applications must take care with respect to computations on XPI_ADDRDIFF-typed values.

XPI Interface (XPI_ADDR_INIT) ..... initialize an address structure

XPI_ADDR_INIT(index, addr)

IN	index	an integer representation of an address
OUT	addr	an XPI structure representation of the address

[C] `XPI_Err XPI_Addr_init(__uint128_t address, XPI_Addr *result);`

Errors

XPI_ERR_OUT_OF_RANGE	the integer index is out of the range supported by the implementation

This interprets the integer as an index into the byte array representation of the global virtual address space, and initializes a structure suitable for use as a global address. Arbitrarily large addresses cannot be generated this way, further address arithmetic will be required to produce very large addresses.

obj

XPI Interface (XPI_NULL) ..... the *null* global virtual address

[C]`extern XPI_Addr XPI_NULL;`

The XPI_NULL global virtual address is defined such that, when compared to an address initialized with the integer, `0`.

Advice to users. The implementation of XPI_NULL is left to the discretion of the XPI implementation, and thus portable XPI code should make no assumptions about it. In particular, XPI_NULL should not have its address taken, as it may be implemented as a macro.

5.4.1 Address Arithmetic

Advice to users. XPI's global address space is typed as bytes, thus all address arithmetic is performed in terms of bytes. This differs from traditional C-language pointer arithmetic, where, while addresses are byte-based, arithmetic is done in terms of the underlying pointer type.

Furthermore, pointer arithmetic in C is only defined in terms of pointers into the same array, or one-off-the-end. This is not currently a restriction for XPI address arithmetic, where the distance between any two addresses in the address space may be computed.

XPI Interface (XPI_ADDR_ADD) ..... adjust an address by an offset

XPI_ADDR_ADD(base, offset, result)

IN	base	the base address to adjust
IN	offset	the offset by which to adjust the base
OUT	result	and address representing base+offset

[C] `XPI_Addr XPI_Addr_add(XPI_Addr base, XPI_AddrDiff offset);`

XPI_ADDR_ADD returns the global address that is offset bytes from base. This computation is not checked for overflow, however using an invalid address as a parcel target in XPI_PARCEL_SEND will result in an error. Note that offset may be negative.

XPI Interface (XPI_ADDR_SUB) ..... compute the difference between two addresses

XPI_ADDR_SUB(lhs, rhs, diff)

IN	lhs	the left-hand side
IN	rhs	the right-hand side
OUT	diff	the difference between lhs and rhs (i.e., lhs−rhs)

[C] `XPI_AddrDiff XPI_Addr_sub(XPI_Addr lhs, XPI_Addr rhs);`

XPI_ADDR_SUB determines the distance between two addresses within the global address space. The XPI_ADDRDIFF integer type is defined to be large enough to represent the difference between any two representable addresses, thus XPI_ADDR_SUB is total.

Advice to users. XPI_ADDR_SUB is a natural comparator for addresses.

$$\text{XPI\_ADDR\_SUB(lhs, rhs, diff)} \implies \begin{cases} \text{diff} = 0 & \textit{iff} \quad \text{lhs} = \text{rhs} \\ \text{diff} < 0 & \textit{iff} \quad \text{lhs} < \text{rhs} \\ \text{diff} > 0 & \textit{iff} \quad \text{lhs} > \text{rhs} \end{cases}$$

Developer's note. XPI_ADDR_SUB-as-comparator usage depends on the fact that subtraction is a valid operation on any two representable addresses in XPI, as opposed to C where subtraction is only defined within the same array, or one-off-the-end.

XPI_ADDR_ADD and XPI_ADDR_SUB, combined with the native memory interface (Section 5.2) provide the basic mechanisms necessary to implement array and **`struct`** operations in XPI. As with any such attempt to impose structure on a fundamentally untyped, low-level address space, XPI applications must select a reasonable means for representing types and structures in memory. As of the current version, XPI does not define its own type system, thus application programmers are free to choose their own.

This document chooses to use the type system underlying the C language implementation on which the XPI application is being executed.

Rationale. It is expected that this decision will be in line with the majority of XPI applications. Such a decision allows application code to use C language types and operators (e.g., **`struct`**, **`sizeof`**, `offsetof`) to perform offset computations in many situations.

Example 5.1 *Consider an array of points, defined by the C-struct* `point_t`.

```
typedef struct {
   double x, y, z;
} point_t;
```

The following shows a simple example of iterating through this array in C, setting each **`double`** *element to* `0`.

```
point_t points[1000];

for (int i = 0; i < 1000; ++i) {
   points[i].x = 0.0;
   points[i].y = 0.0;
   points[i].z = 0.0;
}
```

If the point array is instead stored in XPI's global address space, the same results can be achieved using the same C `point_t`, **`sizeof`**`(point_t)` *to determine the current element of the array,* `offsetof(point_t, {x,y,z})` *to determine the field offset, and the XPI_ADDRDIFF type.*

```
XPI_Addr points;                /* base address of array */
XPI_Addr element;               /* address of current element */
XPI_Addr field;                 /* address of current field */

for (XPI_AddrDiff i = 0; i < 1000; ++i) {
   element = XPI_Addr_add(points, i * sizeof(point_t));
   field = XPI_Addr_add(element, offsetof(point_t, x));
   XPI_Agas_store(field, 0.0, XPI_NULL);
   field = XPI_Addr_add(element, offsetof(point_t, y));
   XPI_Agas_store(field, 0.0, XPI_NULL);
   field = XPI_Addr_add(element, offsetof(point_t, z));
   XPI_Agas_store(field, 0.0, XPI_NULL);
}
```

Advice to users. *The XPI code above depends on the extended C11 interface for synchronous memory references.*

XPI Interface (XPI_ADDR_MOD) ..... compute the integer mod of the address

XPI_ADDR_MOD(addr, denom, remainder)

IN	addr	the address
IN	denom	the denominator
OUT	remainder	the mod result (i.e., addr%denom)

[C] `unsigned int XPI_Addr_mod(XPI_Addr addr, unsigned int denom);`

XPI_ADDR_MOD performs the integer modulus operation on a global address. As with the analogous C-pointer operation, XPI_ADDR_MOD is often used by low level code to map addresses to array indexes.

XPI Interface (XPI_ADDR_DIV) ..... compute the integer div of the address

XPI_ADDR_DIV(addr, denom, quotient)

IN	addr	the address
IN	denom	the denominator
OUT	quotient	the div result (i.e., addr/denom)

[C] `unsigned int XPI_Addr_div(XPI_Addr addr, unsigned int denom);`

XPI_ADDR_DIV performs integer division on a global address. Division is an uncommon operation but is included for completeness.

Chapter 6

Threads

Threads model active computation in XPI. A thread is generated by the runtime as the result of a parcel delivery. A thread is managed and scheduled by its associated process, has synchronous access to its global address, the parcel argument data, the target address, and the data environment associated with the parcel from which it was instantiated. Furthermore, a thread may inspect and modify the continuation parcel that will be sent upon its termination. A user LCO thread has additional access to the local virtual address for the LCO's user data.

6.1 Actions

An XPI action represents the code executed as a result of delivering a parcel (Chapter 4). Note that an XPI thread is distinct from the action that is being performed; the thread being a runtime instance executing the action.

Many actions will represent statically compiled, user-defined functionality (Section 6.1.2), however parcels may carry their own code description as part of their payload. Such parcels target native process actions (Chapter 8) provided as part of the runtime that are responsible for the transformation of the action description into appropriate executable code at the target locality.

> *Developer's note.* Just-in-time compilation and self-modifying code functionality is not yet available.

In addition to user-defined actions, XPI provides builtin actions to perform specific, predefined operations that have no user-defined equivalents. These builtin actions exist where XPI defines asynchronous, parcel-based access to runtime functionality.

Ultimately, all actions serve the same role as parcel target actions in XPI.

XPI Interface (ACTION_NULL) ..... the NULL action

ACTION_NULL

[C]
```
extern XPI_Action XPI_ACTION_NULL;
```

The `NULL` action. When used as the action field in a parcel, the `NULL` action indicates that the processing of this parcel, and any continuation, should terminate.

6.1.1 Builtin Actions

This XPI specification predefines a number of builtin actions that act upon the XPI model itself, e.g., XPI_PROCESS_CREATE_CHILD_ACTION, XPI_THREAD_SET_PRIORITY_ACTION, ...

Many of these are *primitive* actions that represent asynchronous, parcel-based operations on low-level model components for which we anticipate platform-specific hardware support to be provided. These primitive actions have no equivalent, higher level implementation.

Other builtin actions, such as XPI_PROCESS_MEMCPY, can be described through the composition of primitive actions and are not strictly necessary. These higher-level actions are included both to augment the programmability of the XPI specification, as well as to convey high-level semantic information to the XPI runtime that may result in performance improvements through the use of platform specific hardware or internal runtime information.

> *Advice to users.* Often XPI defines equivalent, higher-level, synchronous and asynchronous, function-based interfaces that provide similar behavior to builtin actions when paired with common continuations. The canonical example of this is the duality between the parcel interface to global memory and the asynchronous function-based interface to global memory (Section 5.3.1).

6.1.2 Static Actions

Static actions are user-defined actions that are written in a supported language, precompiled, and registered with the runtime through the XPI_REGISTER_ACTION_WITH_KEY routine. As XPI thread entry points, static actions must respect a constrained, language-dependent interface and will execute with XPI thread semantics.

XPI threads are asynchronous and can not be *joined* in the sense of a traditional POSIX thread and do not return values in a traditional sense. Instead, action results are transmitted to continuation parcels. Secondly, all parameters to actions are passed by-value. This reflects the distributed nature of XPI systems. Global addresses may be passed to actions to simulate traditional reference semantics, however updates to such locations must be correctly synchronized. Finally, XPI actions may signal errors to the runtime, however these errors are not delivered to the parcel's sender.

Many of a program's actions will be written by the application developer. Within the XPI framework, such user actions are managed by an XPI process. Details for writing and registering actions can be found in Section 3.4.

6.1.3 Dynamic Actions

Most generic actions represent statically compiled code, however parcels may carry their own code description as part of their payload. Such parcels target process actions provided as part of the runtime that are responsible for the transformation of the code description into appropriate executable code at the target locality.

> *Developer's note.* Dynamic user actions require complex, code generation support from the XPI runtime, and are not currently supported.

6.1.4 LCO Actions

Chapter 7 describes the special semantics associated with local control object (LCO) actions. In particular, all threads executing LCO actions are strictly serializable on a per-LCO basis.

6.2 Instantiation

A parcel specifies (1) a (potentially XPI_NULL) target address, (2) a target action, (3) the argument data block, and (4) the environment data block. In addition, a parcel contains a continuation block that models a stack of continuation parcels.

The runtime spawns a thread in response to a parcel arrival. This thread will be spawned executing the target action, with its `data` argument set to the argument data block (or a copy of it). The thread may query its global address, the target address, the environment data, or the continuation parcel using the synchronous functions in this section. The runtime sets the thread's continuation parcel to the result of XPI_PARCEL_POP on the generating parcel.

XPI Interface (XPI_THREAD_GET_SELF) ..... get the calling thread's global address

XPI_THREAD_GET_SELF(addr)

OUT	addr	the global address of the calling thread

[C]
```
XPI_Addr XPI_Thread_get_self();
```

XPI_THREAD_GET_SELF can be used to get the global address corresponding to the local thread. All threads are guaranteed to have an address, thus this call will neither fail nor produce XPI_NULL.

XPI Interface (XPI_THREAD_GET_ADDRESS) ..... get the target address

XPI_THREAD_GET_ADDRESS(addr)

OUT	addr	the target address

[C]
```
XPI_Addr XPI_Thread_get_addr();
```

XPI_THREAD_GET_ADDRESS this gets the target address that was set in the instantiating parcel. This address may be XPI_NULL if that is how the parcel was created, but this call will always return successfully.

XPI Interface (XPI_THREAD_GET_ENVIRONMENT) ..... get the environment data

XPI_THREAD_GET_ENVIRONMENT(data)

OUT	data	the environment data

[C]
```
void* XPI_Thread_get_env();
```

XPI_THREAD_GET_ENVIRONMENT returns a pointer to the environment block that was set in the instantiating parcel. This address may be `NULL` if there was no environment set, but the call will always be successful.

XPI Interface (XPI_THREAD_GET_CONTINUATION) ..... get continuation parcel

XPI_THREAD_GET_CONTINUATION(parcel)

OUT	parcel	a handle for the continuation parcel

[C]
```
XPI_Parcel XPI_Thread_get_cont();
```

XPI_THREAD_GET_CONTINUATION gets a handle for the thread's continuation parcel. Updates using this handle will change the thread's continuation action. For example, XPI_PARCEL_PUSH will add to the front of the continuation chain for the thread.

The handle to the continuation parcel must be freed using XPI_PARCEL_FREE like any other handle, however in the case of the continuation parcel, the underlying parcel will not be freed.

Repeated calls to XPI_THREAD_GET_CONTINUATION within a thread must return equivalent handles.

6.3 Continuing

Every thread has a continuation dynamically specified as part of its instantiating parcel, and will dynamically generate a continuation parcel as it terminates. Threads "continue" data by setting the argument data field for their continuation parcel. XPI contains some helper functions that allow this data to be set directly, without explicit use of the continuation parcel.

XPI Interface (XPI_CONTINUE) ..... the continue primitive

XPI_CONTINUE(sizes, vals)

IN	sizes	an n-element list of sizes describing *vals*
IN	vals	an n-element list of values to pass to the continuation

[C]
```
void XPI_continue(size_t n, size_t sizes[], void* vals[]);
void XPI_continue1(size_t size, void *val);
```

The C implementation provides two options for continuing, XPI_CONTINUE behaves as specified in the XPI declaration. Many actions will only continue simple scalar values and can use continue1 directly for this purpose.

Only the data from the last use of XPI_CONTINUE will be used as the continuation parcel's argument data.

6.4 Thread Scheduling

XPI provides a thread-scheduling interface for lower-level use. This interface is defined through a native action interface. A standard, asynchronous function interface is also provided that allows senders to order their operations with respect to the target thread's changes.

XPI Interface (XPI_THREAD_SET_PRIORITY) ..... set the priority for the target thread

XPI_THREAD_SET_PRIORITY_ACTION(priority)

	IN	priority	new priority

[C]
```
XPI_Err XPI_THREAD_SET_PRIORITY_ACTION(size_t *priority) /* CONT() */;
```

XPI_THREAD_SET_PRIORITY_SYNC(priority, address)

IN	priority	new priority
IN	address	global address of the target thread

[C]
```
XPI_Err XPI_Thread_set_priority_sync(XPI_Addr address, size_t priority);
```

XPI Interface (XPI_THREAD_SET_STATE) ..... change the state of the target thread

XPI_THREAD_SET_STATE_ACTION(state)

IN	state	requested state

[C]
```
typedef enum {
  XPI_THREAD_STATE_ACTIVE,
  XPI_THREAD_STATE_SUSPENDED,
  XPI_THREAD_STATE_DEPLETED,
  XPI_THREAD_STATE_TERMINATED
} XPI_Thread_State;

XPI_Err XPI_THREAD_SET_STATE_ACTION(XPI_Thread_State *state); /* CONT() */
```

XPI_THREAD_SET_STATE_SYNC(state, address)

IN	state	requested state
IN	address	global address of the target thread

[C]
```
XPI_Err XPI_Thread_set_state_sync(XPI_Addr address, XPI_Thread_State state);
```

6.5 Thread Suspension

Threads often need to wait for the completion of concurrent activities. Waiting is managed using local control objects (LCOs), which act as a mechanism to delay processing until certain conditions are met. More information about LCOs and their use can be found in Chapter 7.

XPI provides threads with a *wait*-based synchronization interface that allows them to suspend execution until a set of LCOs fires. When these routines return, the designated (or one of the designated) LCO's trigger event is guaranteed to have occurred, but no guarantees are made about the state of the LCO's predicate when this parcel arrives. This is a natural consequence of the design choice that LCOs may fire more than once.

These routines are synchronous, and act on the local thread of execution.

XPI Interface (XPI_THREAD_WAIT) ..... wait until an LCO fires

XPI_THREAD_WAIT(lco, value)

IN	lco	global address of an LCO for which to wait
OUT	value	the value produced by the LCO

[C] `XPI_Err XPI_Thread_wait(XPI_Addr lco, void *value);`

This blocks execution until the LCO fires. From an operational perspective, the runtime implicitly captures the thread's current live state and suspends it. When the designated thread fires, the thread is resumed and provided the local address of the LCO's value.

XPI Interface (XPI_THREAD_WAIT_ALL) . . . . . wait until all of a set of LCOs are fired

XPI_THREAD_WAIT_ALL(lcos, values)

IN	lcos	list of global addresses corresponding to LCOs on which to wait
OUT	values	list of the values produced by the LCOs

[C] `XPI_Err XPI_Thread_wait_all(size_t n, XPI_Addr lco[], void* values[]);`

This blocks until all of the LCOs in `lcos` have fired. Operationally, this behaves in the same manner as XPI_THREAD_WAIT, except that the thread is not resumed until all of the designated LCOs have fired, and is provided with the values of all of the LCOs in the values array.

6.6 Thread Resources

XPI Interface (XPI_THREAD_GET_PROCESS) . . get the global address corresponding to a thread's process

XPI_THREAD_GET_PROCESS_ACTION CONTINUE(process)

CONT	process	global address of the thread's process

[C] `XPI_Err XPI_THREAD_GET_PROCESS_ACTION() /* CONT(XPI_Addr process) */;`

XPI_THREAD_GET_PROCESS(address, future)

IN	address	global address of the target thread
IN	future	a future representing the global address of the thread's process

[C] `XPI_Err XPI_Thread_get_process(XPI_Addr address, XPI_Addr future);`

XPI_THREAD_GET_PROCESS_SYNC(address, process)

IN	address	global address of the target thread
OUT	process	the global address of the thread's process

[C] `XPI_Err XPI_Thread_get_process_sync(XPI_Addr address, XPI_Addr *process);`

Chapter 7

Local Control Objects

A Local Control Object (LCO) is a synchronization object that allows XPI applications to suspend continuation execution until arbitrary conditions are met. While LCOs are allocated as part of the global virtual address space, their physical memory allocations are guaranteed to exist in one synchronous domain. This invariant will be maintained even if the LCO needs to be relocated.

7.1 Properties

7.1.1 Strictly Serializable

One of the major characteristics of an LCO is that the system guarantees that threads executing LCO actions do so in a *strictly serializable* manner, with respect to the target LCO.

> *Rationale.* Strict serializability ensures that real-time ordering of LCO action execution is maintained. This is slightly more restrictive a constraint relative to pure serializability and can prevent numerous programmer errors due to false assumptions, as threads may communicate through synchronized memory accesses or multiple LCOs concurrently.
>
> *Advice to implementors.* We anticipate the continued adoption of transactional memory hardware within synchronous domains. This hardware is ideal for providing the synchronization guarantees laid out here.
>
> Without such hardware, XPI library implementations will be restricted to LCO boundary-style synchronization that can execute inside the runtime before and after LCO action execution. We do not at this time expect to provide an XPI software interface (e.g., `XPI_LOCAL_READ` and `XPI_LOCAL_WRITE` macros, etc.) for fine-grained, library-based synchronization of LCO actions.

7.1.2 Wait Free

LCO actions are prohibited from using the thread suspension interface in Section 6.5.

> *Rationale.* This wait-free property limits the possibility for deadlock. The strict serializability (Section 7.1.1) interferes with asynchronous request-response programming, as it introduces dependencies that can be expensive to track and recover from, and

is neither supported in the current generation of hardware transactional memory systems nor likely in future systems.

Advice to implementors. XPI implementations are encouraged to provide a debugging option that detects erroneous uses of the wait interface (Section 6.5), or XPI-defined synchronous or asynchronous function-based operations, from within LCO actions, and reports such events during execution. This behavior is not required.

Advice to users. LCO actions should not use any asynchronous or synchronous function-based interfaces defined in this specification (e.g., the global memory interfaces provided in Section 5.3.1 or Section 5.3.2) as they imply a wait operation. This restriction does not affect the LCO's ability to initiate asynchronous work with XPI_PARCEL_SEND, however it does not allow LCO actions to reuse parcel handles across multiple waits.

7.1.3 Local Synchronous Memory

LCOs are defined to be entirely resident in one synchronous domain, thus LCO actions can be, and are required to be in all implementations, provided with the local virtual address of the LCO data. LCO actions cannot read from the global address space, due to the limitation that they are not allowed to wait.

The consequence of this property is that an LCO must not have its global address mapping modified during the execution of an LCO action, as the executing thread is accessing it's physical (i.e., synchronous) addresses.

Advice to implementors. An implementation must merely ensure that LCO address remapping conforms to the LCO's strict serialization. This naturally satisfies the previous constraint, and can permit concurrent relocation in some circumstances.

7.1.4 Polymorphic Actions

LCOs are special in that every LCO type, either system-specified or user-defined, supports a specific superset of LCO actions, but provides its own concrete implementations for these actions.

Rationale. As with most object-oriented systems, it is beneficial to separate interface from implementation, and to provide polymorphic behavior. This permits user-implemented LCOs that interact in the same manner as builtin LCOs.

7.1.5 Predicates

LCOs send pending continuations when their predicate is met. An LCO's predicate is automatically evaluated by the XPI runtime after its XPI_LCO_TRIGGER is processed, and during XPI_LCO_GET_VALUE processing.

7.2 Common Interface

The following are valid for all LCOs, but have polymorphic behavior. This behavior is defined by the XPI specification for builtin LCO types, e.g., futures (Section 7.3.1), and have user-defined behavior for user-defined LCO types (Section 7.4).

XPI Interface (XPI_LCO_GET_VALUE) .....

XPI_LCO_GET_VALUE_ACTION CONTINUE(value)

CONT	value	the LCO value

[C] `XPI_Err XPI_LCO_GET_VALUE_ACTION() /* CONT(void *data) */;`

XPI_LCO_GET_VALUE, along with XPI_LCO_TRIGGER are the two primary LCO actions. The XPI_LCO_GET_VALUE action either continues the value of the LCO, or suspends its continuation until the value is available. The continued data type is dependent on the concrete class of the LCO.

XPI_LCO_GET_VALUE has neither an asynchronous nor synchronous function-based version, as it's functionality is based directly on the parcel continuation. The thread's wait interface (Section 6.5) can be used to wrap XPI_LCO_GET_VALUE in a synchronous interface.

> *Advice to users.* Application developers can use XPI_PARCEL_SEND's the optional future parameter to determine that the XPI_LCO_GET_VALUE has occurred. The strict serializable and wait-free properties of LCO threads makes this an adequate signal.

XPI Interface (XPI_LCO_TRIGGER) ..... trigger an LCO

XPI_LCO_TRIGGER_ACTION(value)

IN	value (optional)	an optional trigger value

[C] `XPI_Err XPI_LCO_TRIGGER_ACTION(void *data) /* CONT() */;`

XPI_LCO_TRIGGER(value, lco, future)

IN	value (optional)	an optional trigger value
IN	lco	address of the lco to trigger
IN	future (optional)	future to be used for ordering

[C] `XPI_Err XPI_LCO_trigger(XPI_Addr lco, void *data, XPI_Addr future);`

XPI_LCO_TRIGGER_SYNC(value, lco)

IN	value (optional)	an optional trigger value
IN	lco	address of the lco to trigger

[C] `XPI_Err XPI_LCO_trigger_sync(XPI_Addr lco, void *data);`

With XPI_LCO_GET_VALUE, XPI_LCO_TRIGGER forms the primary interface to an LCO. This action triggers the LCO to potentially change state. The LCO's predicate is automatically tested after the trigger executes, and if it evaluates as true, all of the LCO's pending continuations are released.

Unlike XPI_LCO_GET_VALUE, XPI_LCO_TRIGGER provides both asynchronous and synchronized function-based interfaces.

XPI Interface (XPI_LCO_GET_SIZE) ..... get the size of an LCO, in bytes

XPI_LCO_GET_SIZE_ACTION CONTINUE(size)

CONT	size	the size of the user-portion of the LCO

[C] `XPI_Err XPI_LCO_GET_SIZE_ACTION() /* CONT(size_t size) */;`

XPI_LCO_GET_SIZE(lco, future)

IN	lco	global address of the LCO to query
IN	future	future representing the size of the user-portion of the LCO

[C] `XPI_Err XPI_LCO_get_size(XPI_Addr lco, XPI_Addr future);`

XPI_LCO_GET_SIZE_SYNC(lco, future)

IN	lco	global address of the LCO to query
OUT	future	the size of the user-portion of the LCO

[C] `XPI_Err XPI_LCO_get_size_sync(XPI_Addr lco, size_t *size);`

This action is used to read the size, in bytes, of the LCO structure. This will not include any additional bytes allocated by the process in order to provide support for LCO semantics for this object.

XPI Interface (XPI_LCO_HAD_GET_VALUE) .....

XPI_LCO_HAD_GET_VALUE_ACTION CONTINUE(value)

CONT	value	true if any threads have performed XPI_LCO_GET_-VALUE on the target LCO

[C] `XPI_Err XPI_LCO_HAD_GET_VALUE_ACTION() /* CONT(bool) */;`

XPI_LCO_HAD_GET_VALUE(lco, future)

IN	lco	global address of the LCO to query
IN	future	future representing the result of the query

[C] `XPI_Err XPI_LCO_had_get_value(XPI_Addr lco, XPI_Addr future);`

XPI_LCO_HAD_GET_VALUE_SYNC(lco, value)

IN	lco	global address of the LCO to query
OUT	value	the result of the query

[C] `XPI_Err XPI_LCO_had_get_value_sync(XPI_Addr lco, bool *value);`

XPI_LCO_HAD_GET_VALUE allows the application programmer to determine if any XPI_LCO_GET_VALUE actions have been performed on the target LCO.

Rationale. This action is included in order to allow an application developer to effectively use process termination detection (Section 8.2) in a recursive design, where one process provides work for the next process by attaching continuations (XPI_PROCESS_ATTACH) to LCOs in the next process.

The next process can begin by querying the state of its LCOs with XPI_LCO_HAD_GET_VALUE to determine if it should terminate the recursive algorithm.

7.3 Builtin LCOs

7.3.1 Future

Future LCOs have a special place in XPI, as they represent the results of asynchronous computation and are used throughout the XPI interface as a means of ordered synchronization.

Allocation

XPI Interface (XPI_PROCESS_FUTURE_NEW) ..... allocate an array of futures

XPI_PROCESS_FUTURE_NEW_ACTION(count, bytes) CONTINUE(address)

IN	count	the number of futures to allocate
IN	bytes	size of the buffer for the LCOs
CONT	address	the address of the array

[C]
```
struct XPI_Process_Future_New {
  size_t count;
  size_t bytes;
  XPI_Distribution distribution;
};

XPI_Err XPI_PROCESS_FUTURE_NEW_ACTION(struct XPI_Process_Future_New *arg)
   /* CONT(XPI_Addr address) */;
```

XPI_PROCESS_FUTURE_NEW(count, bytes, process, future)

IN	count	the number of futures to allocate
IN	bytes	size of the buffer for the LCOs
IN	process	address of the process
IN	future (optional)	a future representing the address of the array

[C]
```
XPI_Err XPI_Process_future_new(XPI_Addr process, size_t count, size_t bytes,
                        XPI_Distribution distribution,
                        XPI_Addr future);
```

XPI_PROCESS_FUTURE_NEW_SYNC(count, bytes, process, future)

IN	count	the number of futures to allocate
IN	bytes	size of the buffer for the LCOs
IN	process	address of the process
OUT	future (optional)	the address of the array

[C]
```
XPI_Err XPI_Process_future_new_sync(XPI_Addr process,
                                    size_t count, size_t bytes,
                                    XPI_Distribution distribution,
                                    XPI_Addr *address);
```

These three routines are used for future allocation. As with all LCOs, futures are allocated in the global namespace.

The asynchronous function-based version of XPI_PROCESS_FUTURE_NEW requires a future that has been allocated with the action or synchronous version, and thus is of limited use.

Futures should be freed using the standard XPI_PROCESS_GLOBAL_FREE action interface.

Trigger The trigger functionality for a future takes a single value, and makes the value available using the XPI_LCO_GET_VALUE action.

Get Value The get value functionality for a future simply continues the value of the future to the pending continuations once it is available.

7.3.2 Reduction

The reduction LCO suspends execution of waiting continuations until it has seen the expected number of XPI_LCO_TRIGGER events. This LCO will compute a reduction (as designated in XPI_PROCESS_REDUCTION_NEW) of the trigger values, and continue the reduced value in XPI_LCO_GET_VALUE.

Allocation

XPI Interface (XPI_PROCESS_REDUCTION_NEW) . . . . . allocate an array of reductions

XPI_PROCESS_REDUCTION_OPERATOR

[C]
```
typedef void (*XPI_reduction_operator)(void *lhs, void *rhs);
```

XPI_PROCESS_REDUCTION_NEW_ACTION(count, bytes, inputs, op) CONTINUE(address)

IN	count	the number of lcos to allocate
IN	bytes	size of the value type for the LCOs
IN	inputs	the number of inputs to the reduction
IN	op	the binary reduction operator
CONT	address	the address of the array

[C]
```
struct XPI_Process_Reduction_New_Descriptor {
  size_t count;
  size_t bytes;
  size_t inputs;
  XPI_reduction_operator op;
  XPI_Distribution distribution;
};
```

```
XPI_Err XPI_PROCESS_REDUCTION_NEW_ACTION(size_t count, size_t bytes,
                                         size_t inputs,
                                         XPI_reduction_operator op,
                                         XPI_Distribution distribution)
   /* CONT(XPI_Addr address) */;
```

XPI_PROCESS_REDUCTION_NEW(count, bytes, inputs, op, process, future)

IN	count	the number of lcos to allocate
IN	bytes	size of the value type for the LCOs
IN	inputs	the number of inputs to the reduction
IN	op	the binary reduction operator
IN	process	address of the process
IN	future (optional)	a future representing the address of the array

[C]
```
XPI_Err XPI_Process_reduction_new(XPI_Addr process,
                          size_t count, size_t bytes,
                          size_t inputs,
                          XPI_reduction_operator op,
                          XPI_Distribution distribution,
                          XPI_Addr reduction);
```

XPI_PROCESS_REDUCTION_NEW_SYNC(count, bytes, inputs, op, process, future)

IN	count	the number of lcos to allocate
IN	bytes	size of the value type for the LCOs
IN	inputs	the number of inputs to the reduction
IN	op	the binary reduction operator
IN	process	address of the process
OUT	future (optional)	the address of the array

[C]
```
XPI_Err XPI_Process_reduction_new_sync(XPI_Addr process,
                               size_t count, size_t bytes,
                               size_t inputs,
                               XPI_reduction_operator op,
                               XPI_Distribution distribution,
                               XPI_Addr *address);
```

The binary reduction operator, op, must be both commutative and associative. User LCOs (Section 7.4) can be used to define more complex reductions. A NULL operator can be used to implement a local barrier.

Trigger The trigger for the builtin reduction calls the relevant reduction operator with the address of the existing value and the address of the trigger data argument.

Get Value The get value functionality for a reduction simply continues the reduced value to the pending continuations once it has been computed.

7.4 User LCOs

Special considerations need to be taken for the development of user-defined LCOs, due in particular to their unique capability to polymorphic actions. These considerations are language specific.

User LCOs need to handle six events.

- The initialization event handler. Initialization occurs during XPI_PROCESS_LCO_-MALLOC, after local memory has been allocated but before the global address is available. As with standard object-oriented design, initialization is a chance to initialize the LCO's state.
- The finalization event handler. Finalization occurs during XPI_PROCESS_GLOBAL_-FREE for the LCO's global address.
- The trigger event handler. The trigger event handler is used inside the XPI_LCO_-TRIGGER action to update the private state of the LCO.
- The get value event handler. The get value event handler is used inside the XPI_-LCO_GET_VALUE action to return the value of the lco.
- The get size event handler. The get size event handler is used inside the XPI_LCO_-GET_SIZE action to return the size of the private state for the LCO.
- The predicate evaluation event handler. The predicate evaluation is used by the XPI runtime in both the XPI_LCO_TRIGGER action, after the trigger event handler has executed, and in XPI_LCO_GET_VALUE action, before the parcel continuation is suspended.

7.4.1 C Specification

The per-object function pointer table pattern common in many C-based object-oriented designs is used to specify the behavior of a user LCO in C. At LCO allocation-time (XPI_PROCESS_LCO_MALLOC), handlers for each of the six events listed above must be provided to the runtime.

XPI Interface (XPI_PROCESS_LCO_MALLOC) . . . . . allocate an array of LCOs

XPI_LCO_DESCRIPTOR

[C]
```
typedef struct {
   void (*init)   (void * const lco, const void * const data);
   void (*fini)   (void * const lco);
   void (*trigger) (void * const lco, const void * const data);
   void (*get_value) (const void * const lco, void * const data_out);
   size_t (*get_size) (const void * const lco);
   bool (*eval)   (const void * const lco);
} XPI_LCO_Descriptor;
```

The previous interface is used to describe the user LCO event handlers. These handlers are run inside of LCO actions, and thus can assume strictly serializable semantics with respect to the `lco` state and any `data` inputs.

XPI_PROCESS_LCO_MALLOC_ACTION(count, size, handlers, distribution) CONTINUE(address)

IN	count	the number of LCOs to allocate
IN	size	number of bytes required for LCO state
IN	handlers	user LCO event handlers
IN	distribution	the distribution for the array
CONT	address	global address of the allocated LCO

[C]
```
struct XPI_Process_LCO_Malloc_Descriptor {
  size_t count;
  size_t size;
  XPI_LCO_Descriptor handlers;
  XPI_Distribution distribution;
};

XPI_Err XPI_PROCESS_LCO_MALLOC_ACTION(struct XPI_Process_LCO_Malloc_Descriptor *arg)
   /* CONT(XPI_Addr address) */;
```

XPI_PROCESS_LCO_MALLOC(count, size, handlers, distribution, process, future)

IN	count	the number of LCOs to allocate
IN	size	number of bytes required for LCO state
IN	handlers	user LCO event handlers
IN	distribution	the distribution for the array
IN	process	global address of the allocating process
IN	future	a future representing the global address of the allocated LCO

[C]
```
XPI_Err XPI_Process_lco_malloc(XPI_Addr process,
                        size_t count, size_t size,
                        XPI_LCO_Descriptor handlers,
                        XPI_Distribution distribution,
                        XPI_Addr future);
```

XPI_PROCESS_LCO_MALLOC_SYNC(count, size, handlers, distribution, process, address)

IN	count	the number of LCOs to allocate
IN	size	number of bytes required for LCO state
IN	handlers	user LCO event handlers
IN	distribution	the distribution for the array
IN	process	global address of the allocating process
OUT	address	the address of the allocated LCO

[C]
```
XPI_Err XPI_Process_lco_malloc_sync(XPI_Addr process,
                           size_t count, size_t size,
                           XPI_LCO_Descriptor handlers,
                           XPI_Distribution distribution,
                           XPI_Addr *address);
```

This action, and its corresponding function-based interface, deal with the special allocation requirements for LCOs. In particular, LCOs must specify the set of event handlers required by the User LCO interface.

LCOs should be freed using the standard XPI_PROCESS_GLOBAL_FREE action.

7.4.2 Fortran Specification

Developer's note. Fortran bindings are not yet available, and may never be appropriate for LCO development.

Chapter 8

Processes

XPI processes manage and partition application resources, and provide a source of distributed control, serving as the largest granularity of parallelism in the system. Processes are first-class objects in the global address space.

Processes are organized into a tree structure. The XPI *main* process is always the root process in the process tree, and has some special properties, described thoroughly in Section 8.3. < Processes provide optional termination detection.

8.1 Instantiation & Destruction

Processes are always instantiated as children of parent processes.

XPI Interface (XPI_PROCESS_CREATE_CHILD) ..... create a child process

XPI_PROCESS_CREATE_CHILD_ACTION(terminate) CONTINUE(address)

IN	terminate	address of an LCO that will be triggered at termination
CONT	address	the address of the new process

[C]
```
XPI_Err XPI_PROCESS_CREATE_CHILD_ACTION(void *terminate)
   /* CONT(XPI_Addr address) */;
```

The XPI_PROCESS_CREATE_CHILD_ACTION creates a new process as the child of the target, parent process. The address of the new process is continued to the parcel continuation, which will be executed as the initial action within the context of the new process.

The termination address should either be XPI_NULL, or the address of an LCO that will be triggered when the process terminates. If the termination address is not XPI_NULL, then the process will implement automatic termination detection (Section 8.2), otherwise it will not.

> *Advice to users.* Processes often compute some sort of return value. This should be implemented using the continuation chain that XPI_PROCESS_CREATE_CHILD is instantiated with.

Example 8.1 *Creating a process*

In this example, we will define an action, `action_my_external`*, that spawns a new process as the child of its parent process, and enable termination detection for the new process.* `action_my_external` *will wait for the new process to terminate.*

This example, as well as Example 8.3 and Example 8.2, relies on the following action-registration code.

```
XPI_Err action_my_process(XPI_Addr *process) /* CONT(int) */;
XPI_Err action_my_get_address(XPI_Addr *process) /* CONT(XPI_Aaddr) */;

/* register the local actions we need with the runtime */
static __attribute__((constructor)) void
init() {
   XPI_register_action((XPI_Action)action_my_process);
   XPI_register_action((XPI_Action)action_my_get_address);
}
```

The process itself is described by the `action_my_process` *action, which can have arbitrarily complex behavior but generates an* **`int`** *as a result—this can be considered part of the process' type.*

```
XPI_Err
action_my_process(XPI_Addr *process) {
   int result;

    ⋮

   XPI_continue1(sizeof(result), &result);
   return XPI_SUCCESS;
}
```

The declaration for the `action_my_external` *simply conforms to the C action interface.* `action_my_external` *does not use its argument data, though a real application likely would.* `action_my_external` *starts by getting the global address of the thread, and the process.*

```
1  XPI_Err
2  action_my_external(void* args __attribute__((unused))) {
3     XPI_Addr self = XPI_Thread_get_self();
4     XPI_Addr parent; XPI_Thread_get_process_sync(self, &parent);
```

It then creates a future that will be sent as the termination *data argument to the XPI_PROCESS_CREATE_CHILD action, indicating that we would like this process to employ a termination detection algorithm.*

```
5     XPI_Addr get_terminate;
6     XPI_Process_future_new_sync(parent, /* process */
7                           1,      /* count */
8                           0,      /* bytes */
9                           XPI_LOCAL, /* distribution */
10                          &get_terminate);
```

It then creates the parcel chain that describes the behavior of the process itself. In this case, we are simply creating the action defined by `action_my_process`*, registered as* `"process"`*, and then pushing the XPI_PROCESS_CREATE_CHILD on the chain.*

```
11    XPI_Parcel p;
12    XPI_Parcel_create(&p);

13    XPI_Parcel_push(p);
14    XPI_Parcel_set_addr(p, XPI_NULL);
15    XPI_Parcel_set_action(p, (XPI_Action)action_my_process);
```

```
16     XPI_Parcel_push(p);
17     XPI_Parcel_set_addr(p, parent);
18     XPI_Parcel_set_action(p, XPI_PROCESS_CREATE_CHILD_ACTION);
19     XPI_Parcel_set_data(p, sizeof(XPI_Addr), &get_terminate);
```

Finally, `action_my_external` *sends the request out, and waits for the future to be triggered, indicating that termination of the child process has been detected.*

```
20     XPI_Parcel_send(p, XPI_NULL);
21     XPI_Parcel_free(p);

22     XPI_Thread_wait(get_terminate, NULL);

23     return XPI_SUCCESS;
24 }
```

Note that termination detection cannot retrieve a value from the process. Combining termination detection with a return value requires a separation of concerns, as described in Example 8.3.

Example 8.2 *Returning the address of a process*

Example 8.1 demonstrates the example of creating a child process that depends on termination detection, and waiting for it to complete. It can be useful for the spawning thread to get the address of the child process, before the child process completes. In such a situation we can push an additional action onto the new process' chain that returns the new process' address to an externally-allocated LCO.

The `action_my_get_address` *below does exactly this (using the same action registration code as Example 8.1).*

```
XPI_Err
action_my_get_address(XPI_Addr *process) {
   /* environment consists of an LCO address to send the process */
   void *env = XPI_Thread_get_env();
   if (!env)
      return XPI_SUCCESS;

   XPI_Addr lco = *(XPI_Addr*)env;
   if (XPI_Addr_sub(lco, XPI_NULL) != 0)
      return XPI_SUCCESS;

   /* trigger the LCO from the environment with the address */
   XPI_Parcel p;
   XPI_Parcel_create(&p);
   XPI_Parcel_set_addr(p, lco);
   XPI_Parcel_set_action(p, XPI_LCO_TRIGGER_ACTION);
   XPI_Parcel_set_data(p, sizeof(XPI_Addr), process);
   XPI_Parcel_send(p, XPI_NULL);
   XPI_Parcel_free(p);

   /* continue the address on */
   XPI_continue1(sizeof(XPI_Addr), process);
   return XPI_SUCCESS;
}
```

Given this implementation, we can modify `action_my_external` *to (1) allocate a future to receive the address once it's available, (2) push the* `action_my_get_address` *action onto parcel* `p`*'s continuation stack before* `action_my_process`*, and (3) wait for the address to be returned.*

```
XPI_Err
action_my_external(void* args __attribute__((unused))) {
   XPI_Addr self = XPI_Thread_get_self();
   XPI_Addr parent; XPI_Thread_get_process_sync(self, &parent);

    ⋮

   XPI_Addr get_child;
   XPI_Process_future_new_sync(parent, /* process */
                       1,           /* count */
                       sizeof(XPI_Addr), /* bytes */
                       XPI_LOCAL,   /* distribution */
                       &get_child);

    ⋮

   XPI_Parcel_push(p);
   XPI_Parcel_set_addr(p, parent);
   XPI_Parcel_set_action(p, (XPI_Action)action_my_get_address);
   XPI_Parcel_set_env(p, sizeof(XPI_Addr), &get_child);

    ⋮

   XPI_Parcel_send(p, XPI_NULL);
   XPI_Parcel_free(p);

    ⋮

   XPI_Addr child;
   XPI_Thread_wait(get_child, &child);

    ⋮

   return XPI_SUCCESS;
}
```

Example 8.3 *Returning a value from a process Example 8.2 shows the basic tools required to return a value from a process. The process creator allocates a future, and provides the appropriate XPI_LCO_TRIGGER parcel as part of the XPI_PROCESS_CREATE_CHILD continuation stack.*

Recall that the `action_my_process` *action continues the computed result value.*

```
XPI_Err
action_my_process(XPI_Addr *process) {
   int result;

    ⋮

   XPI_continue1(sizeof(result), &result);
   return XPI_SUCCESS;
}
```

In order to retrieve this, `action_my_external` *must (1) allocate an appropriate future to receive the result, (2) push XPI_LCO_TRIGGER onto the process' continuation stack—before* `action_my_process` *so that it is the "final operation" within the process, and (3) wait for the result.*

```
XPI_Err
action_my_external(void* args __attribute__((unused))) {
   XPI_Addr self = XPI_Thread_get_self();
   XPI_Addr parent; XPI_Thread_get_process_sync(self, &parent);
```

```
    ⋮
    XPI_Addr get_result;
    XPI_Process_future_new_sync(parent, /* process */
                        1,           /* count */
                        sizeof(int), /* bytes */
                        XPI_LOCAL,   /* distribution */
                        &get_result);

    ⋮
    XPI_Parcel p;
    XPI_Parcel_create(&p);

    XPI_Parcel_set_addr(p, get_result);
    XPI_Parcel_set_action(p, XPI_LCO_TRIGGER_ACTION);

    ⋮
    XPI_Parcel_send(p, XPI_NULL);
    XPI_Parcel_free(p);

    ⋮
    int result;
    XPI_Thread_wait(get_result, &result);

    ⋮
    return XPI_SUCCESS;
}
```

XPI Interface (XPI_PROCESS_FREE) ..... frees a process

XPI_PROCESS_FREE_ACTION

[C] `XPI_Err XPI_PROCESS_FREE_ACTION() /* CONT() */;`

XPI_PROCESS_FREE(process, future)

IN	process	the process being freed
IN	future (optional)	a future to be used for ordering

[C] `XPI_Err XPI_Process_free(XPI_Addr process, XPI_Addr future);`

XPI_PROCESS_FREE_SYNC(process)

IN	process	the process being freed

[C] `XPI_Err XPI_Process_free_sync(XPI_Addr process);`

XPI_PROCESS_FREE frees the target process. It may be performed by a thread in the context of the to-be-destroyed process. Any continuation actions associated with the XPI_-PROCESS_FREE thread will be performed in the context of the process' parent. Any active threads or parcels within the context of the process at the time of the XPI_PROCESS_FREE will result in undefined behavior, as will the use of either the synchronous or asynchronous function-based interface from within the context of the process.

Orphaned children of a destroyed process are reparented to the `main` process.

Example 8.4 *Freeing a process In most cases requiring termination detection, freeing a process should be done by sending a XPI_PROCESS_FREE action to the process once termination is detection. An optional way to do this is by suspending a XPI_PROCESS_FREE action on the LCO used to signal termination.*

In Example 8.2 we use `action_my_get_address` *to retrieve the address of the child process in* `action_my_process`*. Using the infrastructure from this example, we can modify* `action_my_process` *to correctly free the process by spawning a parcel chain that waits for the* `get_address` *LCO and then continues with XPI_PROCESS_FREE.*

```
XPI_Err
action_my_external(void* args __attribute__((unused))) {
   XPI_Addr self = XPI_Thread_get_self();
   XPI_Addr parent; XPI_Thread_get_process_sync(self, &parent);

    ⋮

   XPI_Addr child;
   XPI_Thread_wait(get_child, &child);

   XPI_Parcel q;
   XPI_Parcel_create(&q);
   XPI_Parcel_set_addr(q, child);
   XPI_Parcel_set_action(q, XPI_PROCESS_FREE_ACTION);

   XPI_Parcel_push(q);
   XPI_Parcel_set_addr(q, get_terminate);
   XPI_Parcel_set_action(q, XPI_LCO_GET_VALUE_ACTION);

   XPI_Parcel_send(q, XPI_NULL);
   XPI_Parcel_free(q);

   XPI_Thread_wait(get_terminate, NULL);

   return XPI_SUCCESS;
}
```

In a process that does not use termination detection, it often makes sense to push the XPI_-PARCEL_FREE action as the last action performed within the context of the process itself, or, as with the termination detection case, suspending a XPI_PARCEL_FREE parcel targeting the child on a return result LCO.

8.2 Termination Detection

Processes provide termination detection. A process is considered to be terminated when there are (1) no sent-but-not yet instantiated parcels within the process, (2) no active threads, and (3) no LCOs with suspended continuations.

When termination is detected, the process will craft and send an XPI_LCO_TRIGGER parcel to the address specified during XPI_PROCESS_CREATE_CHILD, if it was non-XPI_-NULL. Termination detection does not continue any data through the XPI_LCO_TRIGGER. A process that relies on termination detection is only invoked for the purposes of its side effects.

XPI Interface (XPI_PROCESS_ATTACH) . . . . . attach a continuation to a process

XPI_PROCESS_ATTACH_ACTION

[C] `XPI_Err XPI_PROCESS_ATTACH_ACTION() /* CONT() */;`

External threads may attach continuations to an active process using the XPI_PROCESS_ATTACH action. The XPI_PROCESS_ATTACH action's thread occurs in the context of the sender's process, and the continuation parcel occurs in the context of the target process.

This transition between processes is *atomic*, i.e., there does not exist a time at which XPI_PROCESS_ATTACH has completed and the continuation parcel has not yet been sent in the context of the target process. This ensures that no thread or parcel becomes orphaned as a side effect of termination detection.

Of course, if the target process is terminated before the XPI_PROCESS_ATTACH continuation is sent, a runtime error will occur. It is the responsibility of the application programmer to ensure that the target process does not terminate in this circumstance.

8.3 Main Process

The XPI *main* process is the root of the process hierarchy, and has some special properties with respect to XPI execution. The main process has a designated initial continuation, the `main action`, that must be provided by the application developer. This action cannot be targeted by application parcels, and thus has no `XPI_MAIN` definition suitable for registration, but merely has the C and Fortran specifications for XPI_MAIN.

XPI Interface (XPI_MAIN) ..... main action interface

XPI_MAIN

[C] `XPI_Err XPI_main(size_t n, void* args[]);`

This is not implemented by XPI. It merely describes the interface that XPI applications are required to provide as the initial action for the main process.

The main process initially owns the entire global address space mapping. As the process hierarchy evolves, applications can allocate parts of this space in other, child processes using the allocation and distribution interface. Ownership is exclusive, but reverts back to parent processes as children are terminated.

The main process serves as the reparenting target for orphaned process, serving a role similar to the *init* process in Unix-like systems. A reparented process maintains the structure of it's subtree in the original hierarchy, and will promote its allocated memory into that that of the main process when it terminates.

Terminating the main process terminates the entire XPI application, as any orphaned processes cannot be reparented. The main process will terminate itself if it has no child processes nor active threads.

8.4 Hierarchy Inspection

The process hierarchy can be traversed using the following tree-style traversal routines.

Advice to users. These routines are not synchronized in any manner. Applications requiring synchronized process hierarchy traversals must provide their own synchronization structures on top of this interface.

XPI Interface (XPI_PROCESS_GET_PARENT) . . . . . get a process' parent process

XPI_PROCESS_GET_PARENT_ACTION CONTINUE(parent)

CONT	parent	the parent process address

[C] `XPI_Err XPI_PROCESS_GET_PARENT_ACTION() /* CONT(XPI_Addr parent) */;`

XPI_PROCESS_GET_PARENT(process, future)

IN	process	the process being queried
IN	future	a future representing the process address

[C] `XPI_Err XPI_Process_get_parent(XPI_Addr process, XPI_Addr future);`

XPI_PROCESS_GET_PARENT_SYNC(process, parent)

IN	process	the process being queried
OUT	parent	the parent process address

[C] `XPI_Err XPI_Process_get_parent_sync(XPI_Addr process, XPI_Addr *parent);`

This gets the address of a processes' parent process. The `main` process will return XPI_NULL. An orphaned process will return the address of the `main` process—see Section 8.3 for details.

XPI Interface (XPI_PROCESS_GET_N_CHILDREN) . . . get a process' number of children

XPI_PROCESS_GET_N_CHILDREN_ACTION CONTINUE(n)

CONT	n	the number of children of the process

[C] `XPI_Err XPI_PROCESS_GET_N_CHILDREN_ACTION() /* CONT(size_t n) */;`

XPI_PROCESS_GET_N_CHILDREN(process, future)

IN	process	the process being queried
IN	future	a future representing the number of children of the process

[C] `XPI_Err XPI_Process_get_n_children(XPI_Addr process, XPI_Addr future);`

XPI_PROCESS_GET_N_CHILDREN_SYNC(process, future)

IN	process	the process being queried
OUT	future	the number of children of the process

[C] `XPI_Err XPI_Process_get_n_children_sync(XPI_Addr process, size_t *n);`

This retrieves the number of children for a process.

XPI Interface (XPI_PROCESS_GET_CHILD) . . . . . get a process' child process

XPI_PROCESS_GET_CHILD_ACTION(i) CONTINUE(child)

IN	i	the index
CONT	child	the address of the *i*-th child

[C] `XPI_Err XPI_PROCESS_GET_CHILD_ACTION(size_t *i) /* CONT(XPI_Addr child) */;`

XPI_PROCESS_GET_CHILD(i, process, future)

IN	i	the index
IN	process	the process being queried
IN	future	a future representing the address of the *i*-th child

[C] `XPI_Err XPI_Process_get_child(XPI_Addr process, size_t i, XPI_Addr future);`

XPI_PROCESS_GET_CHILD_SYNC(i, process, child)

IN	i	the index
IN	process	the process being queried
OUT	child	the address of the *i*-th child

[C] `XPI_Err XPI_Process_get_child_sync(XPI_Addr process, size_t i, XPI_Addr *child);`

This retrieves the `i`-th child of a process. If `i` is out of the range of valid child indices, this will return XPI_NULL.

8.5 Memory Management

8.5.1 Allocation & Distribution

XPI defines a *C*-like, `malloc`, `free` interface for global memory allocation. The allocation interface provides hints to suggest allocation distributions to the runtime. Actual allocation distributions may not match the hint, and are not static.

XPI Interface (XPI_PROCESS_GLOBAL_MALLOC) . . . . . allocate global memory

XPI_PROCESS_GLOBAL_MALLOC_ACTION(size, count, distribution) CONTINUE(result)

IN	size	the number of bytes to allocate
IN	count	the number of array elements to allocate
IN	distribution	an initial distribution hint
CONT	result	the global address of the allocation

[C]
```
struct XPI_Global_Malloc_Descriptor {
  size_t count;
  size_t size;
  XPI_Distribution distribution;
};

XPI_Err XPI_PROCESS_GLOBAL_MALLOC_ACTION(struct XPI_Global_Malloc_Descriptor *arg)
   /* CONT(XPI_Addr address) */;
```

XPI_PROCESS_GLOBAL_MALLOC(size, count, distribution, process, future)

IN	size	the number of bytes to allocate
IN	count	the number of array elements to allocate
IN	distribution	an initial distribution hint
IN	process	the process which should allocate the memory
IN	future	future representing the global address of the allocation

[C]
```
XPI_Err XPI_Process_global_malloc(XPI_Addr process, size_t count, size_t size,
                          XPI_Distribution distribution,
                          XPI_Addr future);
```

XPI_PROCESS_GLOBAL_MALLOC_SYNC(size, count, distribution, process, future)

IN	size	the number of bytes to allocate
IN	count	the number of array elements to allocate
IN	distribution	an initial distribution hint
IN	process	the process which should allocate the memory
OUT	future	the global address of the allocation

[C]
```
XPI_Err XPI_Process_global_malloc_sync(XPI_Addr process, size_t count,
                               size_t size,
                               XPI_Distribution distribution,
                               XPI_Addr* address);
```

Allocates a *size*-byte region in global memory. The distribution parameter provides a hint to the implementation of how this allocation should be initially distributed. This operation is currently defined to be synchronous.

XPI Interface (XPI_PROCESS_GLOBAL_FREE) ..... free a global memory region

XPI_PROCESS_GLOBAL_FREE_ACTION(process, address)

IN	process	the process where the memory was allocated
IN	address	the global memory to free

[C]
```
XPI_Err XPI_PROCESS_GLOBAL_FREE_ACTION(XPI_Addr *address) /* CONT() */;
```

XPI_PROCESS_GLOBAL_FREE(process, address, future)

IN	process	the process where the memory was allocated
IN	address	the global memory to free

IN	future (optional)	a future to be used for ordering

[C]
```
XPI_Err XPI_Process_global_free(XPI_Addr process, XPI_Addr address,
                                XPI_Addr future);
```

XPI_PROCESS_GLOBAL_FREE_SYNC(process, address)

IN	process	the process where the memory was allocated
IN	address	the global memory to free

[C]
```
XPI_Err XPI_Process_global_free_sync(XPI_Addr process, XPI_Addr address);
```

Frees a region of globally allocated memory. The `address` must be the result of an XPI_PROCESS_GLOBAL_MALLOC_ACTION call. This call is asynchronous, however the `future` can be used to wait until the operation has completed globally. It is not an error to free XPI_NULL.

Rationale. XPI relies on relocatable data and execution to account for load imbalances that introduce latency or waiting. It's philosophy is that data distributions for exascale applications will be most effectively managed by the runtime. XPI acknowledges that, in many cases, the programmer will know what a reasonable distribution of the data will be in advance, and thus XPI provides the hint-based allocation scheme, to minimize overheads due to distribution warm up.

Advice to users. Users may not assume that the requested distribution has been satisfied, nor should they assume that the distribution remains consistent throughout the execution of the code.

8.5.2 Global Virtual Memory Mapping

The global address space is a virtual address space, implying that there exists a virtual→physical address mapping layer in the hardware or runtime. It is active because this mapping can be modified dynamically at runtime by either the system automatically, or explicitly by the application. This is important because, unlike cache-based shared memory systems, we expect the physical address to encode location information.

XPI Interface (XPI_PROCESS_PIN) ..... pin a global memory region

XPI_PROCESS_PIN_ACTION(process, base, extent) CONTINUE(result)

IN	process	the process managing the memory region
IN	base	the base global address to pin
IN	extent	the extent to pin
CONT	result	the local virtual address corresponding to *base*

[C]
```
struct XPI_Process_Pin_Descriptor {
  XPI_Addr base;
  XPI_AddrDiff extent;
};

XPI_Err XPI_PROCESS_PIN_ACTION(struct XPI_Process_Pin_Descriptor *arg)
   /* CONT(void* address) */;
```

XPI_PROCESS_PIN(process, base, extent, future)

IN	process	the process managing the memory region
IN	base	the base global address to pin
IN	extent	the extent to pin
IN	future	a future representing the local virtual address corresponding to *base*

[C]
```
XPI_Err XPI_Process_pin(XPI_Addr process, XPI_Addr base, XPI_AddrDiff extent,
                        XPI_Addr address);
```

XPI_PROCESS_PIN_SYNC(process, base, extent)

IN	process	the process managing the memory region
IN	base	the base global address to pin
IN	extent	the extent to pin

[C]
```
XPI_Err XPI_Process_pin_sync(XPI_Addr process, XPI_Addr base,
                             XPI_AddrDiff extent, void **result);
```

Pins a global address range. The entire range should be part of a single allocation, and resident locally.

XPI Interface (XPI_PROCESS_UNPIN) ..... unpin a global memory region

XPI_PROCESS_UNPIN_ACTION(process, address)

IN	process	the process managing the memory region
IN	address	the address to unpin

[C]
```
XPI_Err XPI_PROCESS_UNPIN_ACTION(XPI_Addr *address) /* CONT() */;
```

XPI_PROCESS_UNPIN(process, address, future)

IN	process	the process managing the memory region
IN	address	the address to unpin
IN	future (optional)	a future to be used for ordering

[C]
```
XPI_Err XPI_Process_unpin(XPI_Addr process, XPI_Addr address, XPI_Addr future);
```

XPI_PROCESS_UNPIN_SYNC(process, address)

IN	process	the process managing the memory region
IN	address	the address to unpin

[C]
```
XPI_Err XPI_Process_unpin_sync(XPI_Addr process, XPI_Addr address);
```

This action releases a previously pinned region to the system. The *address* must correspond to the global base address of a previously pinned region.

8.5.3 Standard Library

XPI Interface (XPI_PROCESS_MEMCPY) ..... copy data within global memory

XPI_PROCESS_MEMCPY_ACTION(process, from, to, bytes) CONTINUE(from)

IN	process	the process responsible for the operation
IN	from	the global address to copy from
IN	to	the global address to copy from
IN	bytes	the number of bytes to copy
CONT	from	the *from* address

[C]
```
struct XPI_Process_Memcpy_Descriptor {
  XPI_Addr from;
  XPI_Addr to;
  size_t bytes;
};

XPI_Err XPI_PROCESS_MEMCPY_ACTION(struct XPI_Process_Memcpy_Descriptor *arg)
   /* CONT(XPI_Addr from) */;
```

XPI_PROCESS_MEMCPY(process, from, to, bytes, future)

IN	process	the process responsible for the operation
IN	from	the global address to copy from
IN	to	the global address to copy from
IN	bytes	the number of bytes to copy
IN	future (optional)	a future representing the *from* address

[C]
```
XPI_Err XPI_Process_memcpy(XPI_Addr process, XPI_Addr from, XPI_Addr to,
                    size_t bytes, XPI_Addr future);
```

XPI_PROCESS_MEMCPY_SYNC(process, from, to, bytes)

IN	process	the process responsible for the operation
IN	from	the global address to copy from
IN	to	the global address to copy from
IN	bytes	the number of bytes to copy

[C]
```
XPI_Err XPI_Process_memcpy_sync(XPI_Addr process, XPI_Addr from, XPI_Addr to,
                        size_t bytes, XPI_Addr *result);
```

Copies bytes in the global address space asynchronously. The from and to ranges may overlap. The future provides strong ordering if needed.

Appendices

Appendix A

XPI Declarations

Appendix B

XPI Error Codes

XPI_ERR_NO_MEM	12, 18
XPI_ERR_INV_PARCEL	14, 15, 16, 17, 18, 19, 20, 23, 24, 25, 26
XPI_ERR_INV_ADDR	20, 21, 27
XPI_ERR_OUT_OF_RANGE	32

Appendix C

Examples

Example C.1 *Fetch-and-add Via Compare-and-swap*

```
1  #include <assert.h>
2  #include <xpi.h>
3
4  XPI_Err action_post_load64(uint64_t*);
5  XPI_Err action_post_cas64(uint64_t*);
6
7  /** Register all of the local actions that we need to use. */
8  static __attribute__((constructor))
9  void
10 local_init() {
11    XPI_register_action((XPI_Action)action_post_load64);
12    XPI_register_action((XPI_Action)action_post_cas64);
13 }
14
15 /**
16  * post_cas64_env
17  *
18  * The environment for the action_post_cas64. It's free variables are the
19  * value we expected to see during the cas itself, and the value we're trying
20  * to add, for use when our cas fails and we need to do it again.
21  */
22 typedef struct {
23    uint64_t expected;
24    int64_t val;
25 } post_cas64_env;
26
27 /**
28  * do_continue_cas
29  *
30  * This local utility function sets up a single instance of the CAS/post-cas
31  * loop for the current thread using its continuation. It is used from
32  * post-load and post-cas.
33  */
34 static XPI_Err
35 do_continue_cas(post_cas64_env *env) {
36    XPI_Addr addr = XPI_Thread_get_addr();
37    XPI_Parcel c = XPI_Thread_get_cont();
38
39    XPI_Parcel_push(c);
40    XPI_Parcel_set_addr(c, addr);
41    XPI_Parcel_set_action(c, (XPI_Action)action_post_cas64);
```

```
42     XPI_Parcel_set_env(c, sizeof(post_cas64_env), env);
43
44     XPI_Parcel_push(c);
45     XPI_Parcel_set_addr(c, addr);
46     XPI_Parcel_set_action(c, (XPI_Action)XPI_AGAS_CAS_U64_ACTION);
47
48     uint64_t to = env->expected + env->val;
49
50     void* args[] = {
51        &env->expected,
52        &to
53     };
54
55     size_t sizes[] = {
56        sizeof(uint64_t),
57        sizeof(uint64_t)
58     };
59
60     XPI_continue(2, sizes, args);
61
62     return XPI_SUCCESS;
63  }
64
65  /**
66   * action_post_cas64
67   *
68   * After using the builtin AGAS_CAS, this post-cas continuation checks to see
69   * if the value was updated successfully (i.e., the actual value that we saw
70   * was the expected value). If it was, then we can continue the fetched value,
71   * otherwise we go through another CAS/post-cas loop.
72   */
73  XPI_Err
74  action_post_cas64(uint64_t *actual) {
75     post_cas64_env *env = (post_cas64_env*)XPI_Thread_get_env();
76
77     if (*actual == env->expected) {
78        XPI_continue1(sizeof(uint64_t), actual);
79        return XPI_SUCCESS;
80     }
81
82     env->expected = *actual;
83     return do_continue_cas(env);
84  }
85
86  /**
87   * action_post_load64
88   *
89   * The first time we try to do the CAS, we need to read the value from the
90   * location first. This post_load64 happens after a LOAD_U64 and just uses the
91   * do_continue_cas utility to perform a single instance of the CAS/post_cas
92   * loop.
93   */
94  XPI_Err
95  action_post_load64(uint64_t *from) {
96     int64_t val = *(int64_t*)XPI_Thread_get_env();
97     post_cas64_env env = { *from, val };
98     return do_continue_cas(&env);
99  }
100
```

```
101 /**
102  * fetch-and-add via compare-and-swap
103  *
104  * This is not the most efficient way to perform a fetch-and-add, but is how we
105  * would do it if the only atomic memory primitive we have is
106  * compare-and-swap. Essentially, the sender allocates a local future for the
107  * "fetched" value, and then starts a CAS/post-cas loop at the global address'
108  * locality, until the cas succeeds. It actually uses the LOAD/post-load
109  * combination to get the value for the first iteration.
110  */
111 uint64_t
112 fadd64_via_cas64_sync(XPI_Addr addr, int64_t val) {
113    /* Allocate a future that we use to receive the fetched value. */
114    XPI_Addr process;
115    XPI_Thread_get_process_sync(XPI_Thread_get_self(), &process);
116
117    XPI_Addr f;
118    XPI_Process_future_new_sync(process, 1, sizeof(int64_t), XPI_LOCAL, &f);
119
120    /* Create the async chain that triggers the future when complete. */
121    XPI_Parcel p;
122    XPI_Parcel_create(&p);
123    XPI_Parcel_set_addr(p, f);
124    XPI_Parcel_set_action(p, XPI_LCO_TRIGGER_ACTION);
125
126    XPI_Parcel_push(p);
127    XPI_Parcel_set_addr(p, addr);
128    XPI_Parcel_set_action(p, (XPI_Action)action_post_load64);
129    XPI_Parcel_set_env(p, sizeof(val), &val);
130
131    XPI_Parcel_push(p);
132    XPI_Parcel_set_addr(p, addr);
133    XPI_Parcel_set_action(p, XPI_AGAS_LOAD_U64_ACTION);
134
135    XPI_Parcel_send(p, XPI_NULL);
136    XPI_Parcel_free(p);
137
138    /* wait for the result, and return it */
139    uint64_t result;
140    XPI_Thread_wait(f, &result);
141    return result;
142 }
```