Parallel Runtime Interface for Fortran (PRIF) Specification, Revision 0.5

Dan Bonachea[*], Katherine Rasmussen[†], Brad Richardson[‡], Damian Rouson[§]
Lawrence Berkeley National Laboratory, USA
fortran@lbl.gov - fortran.lbl.gov

Lawrence Berkeley National Laboratory Technical Report (LBNL-2001636)
doi:10.25344/S4CG6G

December 19, 2024

Abstract

This document specifies an interface to support the parallel features of Fortran, named the Parallel Runtime Interface for Fortran (PRIF). PRIF is a proposed solution in which the runtime library is primarily responsible for implementing coarray allocation, deallocation and accesses, image synchronization, atomic operations, events, teams and collective subroutines. In this interface, the compiler is responsible for transforming the invocation of Fortran-level parallel features into procedure calls to the necessary PRIF subroutines. The interface is designed for portability across shared- and distributed-memory machines, different operating systems, and multiple architectures. Implementations of this interface are intended as an augmentation for the compiler's own runtime library. With an implementation-agnostic interface, alternative parallel runtime libraries may be developed that support the same interface. One benefit of this approach is the ability to vary the communication substrate. A central aim of this document is to define a parallel runtime interface in standard Fortran syntax, which enables us to leverage Fortran to succinctly express various properties of the procedure interfaces, including argument attributes.

WORK IN PROGRESS This document is still a draft and may continue to evolve.
Feedback and questions should be directed to: fortran@lbl.gov

[*]https://orcid.org/0000-0002-0724-9349
[†]https://orcid.org/0000-0001-7974-1853
[‡]https://orcid.org/0000-0002-3205-2169
[§]https://orcid.org/0000-0002-2344-868X

Contents

1 Change Log

1.1 Revision 0.1

- Identify parallel features
- Sketch out high-level design
- Decide on compiler vs PRIF responsibilities

1.2 Revision 0.2 (Dec 2023)

- Change name to PRIF
- Fill out interfaces to all PRIF provided procedures
- Write descriptions, discussions and overviews of various features, arguments, etc.

1.3 Revision 0.3 (May 2024)

- `prif_(de)allocate` are renamed to `prif_(de)allocate_coarray`
- `prif_(de)allocate_non_symmetric` are renamed to `prif_(de)allocate`
- `prif_local_data_size` renamed to `prif_size_bytes` and add a client note about the procedure
- Update interface to `prif_base_pointer` by replacing three arguments, `coindices`, `team`, and `team_number`, with one argument `image_num`. Update the semantics of `prif_base_pointer`, as it is no longer responsible for resolving the coindices and team information into a number that represents the image on the initial team before returning the address. That is now expected to occur before the `prif_base_pointer` call and passed into the `image_num` argument.
- Add target attribute on `coarray_handles` argument to `prif_deallocate_coarray`
- Add pointer attribute on `handle` argument to `coarray_cleanup` callback for `prif_allocate_coarray`
- Add target attribute on `value` argument to `prif_put` and `prif_get`
- Add new PRIF-specific constant `PRIF_STAT_OUT_OF_MEMORY`
- Clarify that remote pointers passed to various procedures must reference storage allocated using `prif_allocate_coarray` or `prif_allocate`
- Clarify description of the `allocated_memory` argument for the procedures `prif_allocate_coarray` and `prif_allocate`
- Clarify descriptions of `event_var_ptr`, `lock_var_ptr`, and `notify_ptr`
- Clarify descriptions for `prif_stop`, `prif_put`, `prif_get`, intrinsic derived types, sections about `MOVE_ALLOC` and coarray accesses
- Replace the phrase “local completion” with the phrase “source completion”, and add the new phrase to the glossary
- Clarify that `prif_stop` should be used to initiate normal termination
- Describe the `operation` argument to `prif_co_reduce`
- Rename and clarify the cobounds arguments to `prif_alias_create`
- Clarify the descriptions of `source_image`/`result_image` arguments to collective calls
- Clarify completion semantics for atomic operations
- Rename `coindices` argument names to `cosubscripts` to more closely correspond with the terms used in the Fortran standard
- Rename `local_buffer` and `local_buffer_stride` argument names to `current_image_buffer` and `current_image_buffer_stride`
- Update `coindexed-object` references to *coindexed-named-object* to match the term change in the most recent Fortran 2023 standard
- Convert several explanatory sections to “Notes”
- Add implementation note about PRIF being defined in Fortran
- Add section “How to read the PRIF specification”
- Add section “Glossary”
- Improve description of the `final_func` argument to `prif_allocate_coarray` and move some of previous description to a client note.

1.4 Revision 0.4 (Jul 2024)

- Changes to Coarray Access (puts and gets):
  - Refactor to provide separate procedure interfaces for the various combinations of: direct vs indirect target location, puts with or without a *notify-variable*, direct vs indirect *notify-variable* location, and strided vs contiguous data access.
  - Add discussion of direct and indirect location accesses to the Design Decisions and Impact section
  - Rename `_raw_` procedures to `_indirect_`
  - Replace `cosubscripts`, `team`, and `team_number` arguments with `image_num`
  - Replace `first_element_addr` arguments with `offset`
  - Replace `type(*) value` arguments with `size` and `current_image_buffer`
  - Rename `remote_ptr_stride` arguments to `remote_stride`
  - Rename `current_image_buffer_stride` arguments to `current_image_stride`
  - Rename `size` arguments to `size_in_bytes`
- Other changes to PRIF procedure interfaces:
  - Establish a new uniform argument ordering across all non-collective communication procedures
  - Remove `prif_base_pointer`. Direct access procedures should be used instead.
  - Add direct versions of `prif_event_post`, `prif_lock`, and `prif_unlock` and rename previous versions to `..._indirect`
  - Convert `prif_num_images` into three different procedures with no optional arguments, in order to more closely align with the Fortran standard. Do the same with `prif_image_index`.
  - Correct the kind for atomic procedures from `atomic_int_kind` to `PRIF_ATOMIC_INT_KIND` and from `atomic_logical_kind` to `PRIF_ATOMIC_LOGICAL_KIND`
  - Remove target attribute from `coarray_handles` argument in `prif_deallocate_coarray`
  - Rename `element_length` argument in `prif_allocate_coarray` to `element_size`
  - Rename `image_index` argument in `prif_this_image_no_coarray` to `this_image`
  - Remove generic interfaces throughout
- Miscellaneous new features:
  - Allow multiple calls to `prif_init` from each process, and add `PRIF_STAT_ALREADY_INIT` constant
  - Add new PRIF-specific constants `PRIF_VERSION_MAJOR` and `PRIF_VERSION_MINOR`
- Narrative and editorial improvements:
  - Add/improve Common Arguments subsections and add links to them below procedure interfaces
  - Elide argument lists for all procedures and add prose explaining how the PRIF specification presents the procedure interfaces
  - Add client notes to subsections introducing PRIF Types, and permute subsection order
  - Add guidance to clients regarding coarray dummy arguments
  - Remove grammar non-terminals, including `coindexed-named-object`
  - Add several terms to the glossary
  - Numerous minor wording changes throughout

1.5 Revision 0.5 (Dec 2024)

- Convert all instances of `c_intmax_t` to `c_int64_t`
- Replace `lbounds`, `ubounds`, `element_size` arguments in `prif_allocate_coarray` with `size_in_bytes`
- Specify definition of `prif_coarray_handle` to be a derived type with one member that is a pointer
- Add `prif_register_stop_callback`
- Remove `contiguous` attribute from argument `a` in collective subroutines
- Update argument list to `prif_co_reduce` to use `operation_wrapper` and add `cdata` argument
- Add `prif_co_max_character` and `prif_co_min_character`
- Constrain argument type for argument `a` to `prif_co_max`, `prif_co_min`, and `prif_co_sum`
- Add client note indicating that `prif_co_reduce` may be used to support other collective calls where `a` is not an interoperable type
- Clarify the semantics of derived types passed to `prif_co_broadcast`
- Add `prif_local_data_pointer`
- Prohibit overlap between source and destination memory regions in coarray access
- Numerous minor editorial changes throughout

2 Problem Description

In order to be fully Fortran 2023 compliant, a Fortran compiler needs support for what is commonly referred to as Coarray Fortran, which includes features related to multi-image parallelism. These features include the following statements, subroutines, functions, types, and kind type parameters:

- **Statements:**
  - *Synchronization:* `SYNC ALL`, `SYNC IMAGES`, `SYNC MEMORY`, `SYNC TEAM`
  - *Events:* `EVENT POST`, `EVENT WAIT`
  - *Notify:* `NOTIFY WAIT`
  - *Error termination:* `ERROR STOP`
  - *Locks:* `LOCK`, `UNLOCK`
  - *Failed images:* `FAIL IMAGE`
  - *Teams:* `FORM TEAM`, `CHANGE TEAM`
  - *Critical sections:* `CRITICAL`, `END CRITICAL`
- **Intrinsic functions:**
  - *Image Queries:* `NUM_IMAGES`, `THIS_IMAGE`, `FAILED_IMAGES`, `STOPPED_IMAGES`, `IMAGE_STATUS`
  - *Coarray Queries:* `LCOBOUND`, `UCOBOUND`, `COSHAPE`, `IMAGE_INDEX`
  - *Teams:* `TEAM_NUMBER`, `GET_TEAM`
- **Intrinsic subroutines:**
  - *Collective subroutines:* `CO_SUM`, `CO_MAX`, `CO_MIN`, `CO_REDUCE`, `CO_BROADCAST`
  - *Atomic subroutines:* `ATOMIC_ADD`, `ATOMIC_AND`, `ATOMIC_CAS`, `ATOMIC_DEFINE`, `ATOMIC_FETCH_ADD`, `ATOMIC_FETCH_AND`, `ATOMIC_FETCH_OR`, `ATOMIC_FETCH_XOR`, `ATOMIC_OR`, `ATOMIC_REF`, `ATOMIC_XOR`
  - *Other subroutines:* `EVENT_QUERY`
- **Types, kind type parameters, and values:**
  - *Intrinsic derived types:* `EVENT_TYPE`, `TEAM_TYPE`, `LOCK_TYPE`, `NOTIFY_TYPE`
  - *Atomic kind type parameters:* `ATOMIC_INT_KIND`, `ATOMIC_LOGICAL_KIND`
  - *Values:* `STAT_FAILED_IMAGE`, `STAT_LOCKED`, `STAT_LOCKED_OTHER_IMAGE`, `STAT_STOPPED_IMAGE`, `STAT_UNLOCKED`, `STAT_UNLOCKED_FAILED_IMAGE`

In addition to supporting the above features, compliant Fortran compilers also need to be able to handle parallel execution concepts such as image control. The image control concept affects the behaviors of some statements that were introduced in Fortran expressly for supporting parallel programming, but image control also affects the behavior of some statements that pre-existed parallelism in standard Fortran:

- **Image control statements:**
  - *Pre-existing statements*: `ALLOCATE`, `DEALLOCATE`, `STOP`, `END`, `MOVE_ALLOC` on coarray
  - *New statements:* `SYNC ALL`, `SYNC IMAGES`, `SYNC MEMORY`, `SYNC TEAM`, `CHANGE TEAM`, `END TEAM`, `CRITICAL`, `END CRITICAL`, `EVENT POST`, `EVENT WAIT`, `FORM TEAM`, `LOCK`, `UNLOCK`, `NOTIFY WAIT`

One example consequence of these statements being categorized as image control statements is the need to restrict movement of surrounding code by optimizing compilers.

3 Proposed Solution

This specification proposes an interface to support the above features, named the Parallel Runtime Interface for Fortran (PRIF). By defining an implementation-agnostic interface, we envision facilitating the development of alternative parallel runtime libraries that support the same interface. One benefit of this approach is the ability to vary the communication substrate. A central aim of this document is to specify a parallel runtime interface in standard Fortran syntax, which enables us to leverage Fortran to succinctly express various properties of the procedure interfaces, including argument attributes. See Bonachea, Rasmussen, Richardson and Rouson (2024) for additional details.

3.1 Parallel Runtime Interface for Fortran (PRIF)

The Parallel Runtime Interface for Fortran is a proposed interface in which the PRIF implementation is responsible for coarray allocation, deallocation and accesses, image synchronization, atomic operations, events, teams, and collective subroutines. In this interface, the compiler is responsible for transforming the invocation of Fortran-level parallel features to add procedure calls to the necessary PRIF subroutines. Below is a table showing the delegation of tasks between the compiler and the PRIF implementation. The interface is designed for portability across shared- and distributed-memory machines, different operating systems, and multiple architectures.

Implementations of PRIF are intended as an augmentation for the compiler's own runtime library. While the interface can support multiple implementations, we envision needing to build the PRIF implementation as part of installing the compiler. The procedures and types provided for direct invocation as part of the PRIF implementation shall be defined in a Fortran module with the name `prif`.

3.2 Delegation of tasks between the Fortran compiler and the PRIF implementation

The following table outlines which tasks will be the responsibility of the Fortran compiler and which tasks will be the responsibility of the PRIF implementation. A 'X' in the "Fortran compiler" column indicates that the compiler has the primary responsibility for that task, while a 'X' in the "PRIF implementation" column indicates that the compiler will invoke the PRIF implementation to perform the task and the PRIF implementation has primary responsibility for the task's implementation. See the Procedure descriptions for the list of PRIF implementation procedures that the compiler will invoke.

Tasks	Fortran compiler	PRIF implementation
Establish and initialize static coarrays prior to `main`	X	
Track corank of coarrays	X	
Track local coarrays for implicit deallocation when exiting a scope	X	
Initialize a coarray with `SOURCE=` as part of `ALLOCATE`	X	
Provide `prif_critical_type` coarrays for `CRITICAL`	X	
Provide final subroutine for each derived type appearing in a coarray that is finalizable or has allocatable components	X	
Track variable allocation status, including resulting from use of `MOVE_ALLOC`	X	
Intrinsics related to parallelism, eg. `NUM_IMAGES`, `COSHAPE`, `IMAGE_INDEX`		X
Allocate and deallocate a coarray		X
Reference a coindexed object		X
Team statements/constructs: `FORM TEAM`, `CHANGE TEAM`, `END TEAM`		X
Team stack abstraction		X
Track coarrays for implicit deallocation at `END TEAM`		X
Atomic subroutines, e.g. `ATOMIC_FETCH_ADD`		X
Collective subroutines, e.g. `CO_BROADCAST`, `CO_SUM`		X
Synchronization statements, e.g. `SYNC ALL`, `SYNC TEAM`		X
Events: `EVENT POST`, `EVENT WAIT`		X
Locks: `LOCK`, `UNLOCK`		X
`CRITICAL` construct		X
`NOTIFY WAIT` statement		X

NOTE: Caffeine - LBNL's Implementation of the Parallel Runtime Interface for Fortran

Implementations for much of the Parallel Runtime Interface for Fortran exist in Caffeine, a parallel runtime library supporting coarray Fortran compilers. Caffeine continues to be developed in order to fully implement PRIF. Caffeine targets the GASNet-EX exascale networking middleware, however PRIF is deliberately agnostic to details of the communication substrate. As such it should be possible to develop PRIF implementations targeting other substrates including the Message Passing Interface (MPI). See Rouson and Bonachea (2022) for additional details.

3.3 Design Decisions and Impact

As stated earlier, PRIF specifies a set of **Fortran** types, values, and procedure interfaces, all provided by the PRIF implementation in the `prif` Fortran module. This means that a compiler will typically need to transform Fortran code making use of the parallel features as though it had been written to use PRIF directly. Conceptually this could happen as a source-to-source transformation, but in practice it's expected to happen in later phases of processing. It is worth further noting that whilst each implementation of PRIF defines the contents of the PRIF types and the values of the named constants, because PRIF is a Fortran module, a compiler should have access to their definitions during code compilation in the same way as other Fortran modules. One notable consequence of the current design is that different PRIF implementations will likely not be ABI compatible.

The PRIF design delegates the responsibility of defining the coarray descriptor, a metadata abstraction which tracks multi-image properties of each coarray, to the PRIF implementation. The PRIF implementation provides the compiler with a pointer-like handle (`prif_coarray_handle`) that references the coarray descriptor. The compiler is then responsible for storing and passing the handle back to the implementation for operations involving a given coarray. For Fortran procedures with coarray dummy arguments, the compiler should ensure that the coarray handle corresponding to the actual argument is made available for use in coarray operations within the procedure. This could be achieved by passing the handle as an extra argument, or by including the handle in a compiler-managed variable descriptor.

Many of the PRIF procedures providing communication involving coindexed data have direct and indirect variants. The direct variants accept a coarray handle as an argument and can operate on data stored within the coarray, i.e. memory locations allocated using `prif_allocate_coarray`. The indirect variants accept a pointer instead, and are used for operating on data which is not necessarily stored directly within a coarray, i.e. the memory location was either allocated using `prif_allocate`, or is being accessed through a pointer component in a different coarray. Note that for `put` operations, the target location of the coindexed assignment and the notify variable to be modified upon completion can independently be direct or indirect. The pointer to an indirect location will typically be obtained using `prif_get*` to retrieve pointer information from the representation of an allocatable or pointer component of some derived type stored within a coarray.

The distinction between direct and indirect access is necessitated by the fact that coarrays are permitted to be of derived types with allocatable or pointer components. Unlike the coarray data, the target memory referenced by these components is generally allocated non-collectively, and those allocations can occur before or after the collective allocation of the coarray. Nevertheless, Fortran requires this target memory to be accessible to remote images. Consider the below program as an example.

```
program coarray_with_allocatable_component
  type :: my_type
    integer, allocatable :: component
  end type
  type(my_type) :: coarray[*]
  if (this_image() == 1) then
      allocate(coarray%component, source = 42)
  endif
  sync all
  print *, coarray[1]%component
end program
```

It is also valid for a pointer component in one coarray to reference data stored in another coarray. Consider the below program as an example.

```
program coarray_with_pointer_component
  type :: my_pointer
    integer, pointer :: val
  end type
  integer, target :: i[*]
  type(my_pointer) :: j[*]
  i = this_image()
  j%val => i
  sync all
  print *, j[1]%val
end program
```

3.4 How to read the PRIF specification

The PRIF types and procedures generally align with corresponding types and procedures from the Fortran standard. In many cases, the correspondence is clear from the identifiers. For example, the PRIF procedure `prif_num_images` corresponds to the intrinsic function `NUM_IMAGES` that is defined in the Fortran standard. In other cases, the correspondence may be less clear and is stated explicitly.

In order to avoid redundancy, some details are omitted from this document, because the corresponding descriptions in the Fortran standard contain the detailed specification of concepts and behavior required by the language. For example, this document references the term coarray multiple times, but does not define it since it is part of the language and the Fortran standard defines it. As such, in order to fully understand the PRIF specification, it is critical to read and reference the Fortran 2023 standard alongside it. Additionally, the descriptions in the PRIF specification use similar language to the language used in the Fortran standard, for example terms like "shall". Where PRIF uses terms not defined in the standard, their definitions may be found in the `Glossary`.

4 PRIF Types and Named Constants

The types and named constants described in this section shall be defined in the `prif` module and have the `public` attribute, unless otherwise specified.

4.1 Fortran Intrinsic Derived Types

These types will be defined by the PRIF implementation. The compiler should use these PRIF-provided implementation definitions for the corresponding types in the compiler's implementation of the `ISO_FORTRAN_ENV` module. This enables the internal structure of each given type to be tailored as needed for a given PRIF implementation.

Each type specified in this section is defined by the PRIF implementation as a derived type with only private components. Each is an extensible type with no type parameters. Each nonallocatable component is fully default-initialized.

CLIENT NOTE:

The components comprising the PRIF definitions of the Fortran Intrinsic Derived types are deliberately unspecified by PRIF, and to ensure portability the compiler should not hard-code reliance on those details. However note that at compile-time the detailed representation corresponding to a given PRIF implementation will be visible to the compiler in the interface declarations of the `prif` module.

4.1.1 `prif_team_type`

- implementation for `TEAM_TYPE` from `ISO_FORTRAN_ENV`

4.1.2 `prif_event_type`

- implementation for `EVENT_TYPE` from `ISO_FORTRAN_ENV`

4.1.3 `prif_lock_type`

- implementation for `LOCK_TYPE` from `ISO_FORTRAN_ENV`

4.1.4 `prif_notify_type`

- implementation for `NOTIFY_TYPE` from `ISO_FORTRAN_ENV`

4.2 PRIF-Specific Types

These derived types are defined by the PRIF implementation in the `prif` module. They don't correspond directly to types mandated by the Fortran specification, but rather are helper types used in PRIF to provide the parallel Fortran features.

Each type specified in this section is defined by the PRIF implementation as a derived type with only private components and with no type parameters.

4.2.1 `prif_coarray_handle`

- `prif_coarray_handle` is a derived type provided by the PRIF implementation. It represents a reference to a coarray descriptor and is passed back and forth across PRIF for coarray operations.
- The `prif_coarray_handle` derived type shall be defined exactly as shown in the code snippet below. The `prif_coarray_descriptor` derived type shall be a private derived type with private components that are implementation-dependent.

```
type, public :: prif_coarray_handle
  private
  type(prif_coarray_descriptor), pointer :: info
end type
```

- The `prif_coarray_handle` type shall only have intrinsic assignment.
- `prif_coarray_handle` values contain a pointer that is specific to a particular image. As such, handle values are not meaningful to other images and should not be directly passed between images.

4.2.2 `prif_critical_type`

- `prif_critical_type` is a derived type provided by the PRIF implementation that is used for implementing `critical` blocks. Each nonallocatable component is fully default-initialized.

CLIENT NOTE:

The components comprising the `prif_coarray_descriptor` and `prif_critical_type` types are deliberately unspecified by PRIF, and to ensure portability the compiler should not hard-code reliance on those details. However note that at compile-time the detailed representation corresponding to a given PRIF implementation will be visible to the compiler in the interface declarations of the `prif` module.

4.3 Named Constants in `ISO_FORTRAN_ENV`

These named constants will be defined in the PRIF implementation and it is proposed that the compiler will deploy a rename to use the PRIF implementation definitions for these values in the compiler's implementation of the `ISO_FORTRAN_ENV` module.

4.3.1 `PRIF_ATOMIC_INT_KIND`

This shall be set to an implementation-defined value from the compiler-provided `INTEGER_KINDS` array.

4.3.2 `PRIF_ATOMIC_LOGICAL_KIND`

This shall be set to an implementation-defined value from the compiler-provided `LOGICAL_KINDS` array.

4.3.3 `PRIF_CURRENT_TEAM`

This shall be a value of type `integer(c_int)` that is defined by the implementation. It shall be distinct from the values `PRIF_INITIAL_TEAM` and `PRIF_PARENT_TEAM`

4.3.4 `PRIF_INITIAL_TEAM`

This shall be a value of type `integer(c_int)` that is defined by the implementation. It shall be distinct from the values `PRIF_CURRENT_TEAM` and `PRIF_PARENT_TEAM`

4.3.5 `PRIF_PARENT_TEAM`

This shall be a value of type `integer(c_int)` that is defined by the implementation. It shall be distinct from the values `PRIF_CURRENT_TEAM` and `PRIF_INITIAL_TEAM`

4.3.6 `PRIF_STAT_FAILED_IMAGE`

This shall be a value of type `integer(c_int)` that is defined by the implementation to be negative if the implementation cannot detect failed images and positive otherwise. It shall be distinct from all other stat named constants defined by this specification.

4.3.7 `PRIF_STAT_LOCKED`

This shall be a value of type `integer(c_int)` that is defined by the implementation. It shall be distinct from all other stat named constants defined by this specification.

4.3.8 `PRIF_STAT_LOCKED_OTHER_IMAGE`

This shall be a value of type `integer(c_int)` that is defined by the implementation. It shall be distinct from all other stat named constants defined by this specification.

4.3.9 `PRIF_STAT_STOPPED_IMAGE`

This shall be a positive value of type `integer(c_int)` that is defined by the implementation. It shall be distinct from all other stat named constants defined by this specification.

4.3.10 `PRIF_STAT_UNLOCKED`

This shall be a value of type `integer(c_int)` that is defined by the implementation. It shall be distinct from all other stat named constants defined by this specification.

4.3.11 `PRIF_STAT_UNLOCKED_FAILED_IMAGE`

This shall be a value of type `integer(c_int)` that is defined by the implementation. It shall be distinct from all other stat named constants defined by this specification.

4.4 PRIF-Specific Named Constants

These named constants have no directly corresponding constants specified in the Fortran standard.

4.4.1 `PRIF_STAT_OUT_OF_MEMORY`

This shall be a value of type `integer(c_int)` that is defined by the implementation. It shall be distinct from all other stat named constants defined by this specification. It shall indicate a low-memory error condition and may be returned by `prif_allocate_coarray` or `prif_allocate`.

4.4.2 `PRIF_STAT_ALREADY_INIT`

This shall be a value of type `integer(c_int)` that is defined by the implementation. It shall be distinct from all other stat named constants defined by this specification. It shall indicate that `prif_init` has previously been called.

4.4.3 `PRIF_VERSION_MAJOR`

This shall be a named constant of type `integer(c_int)` that is defined by the implementation and represents the major revision number of the PRIF specification (i.e. this document) that the implementation supports.

4.4.4 `PRIF_VERSION_MINOR`

This shall be a named constant of type `integer(c_int)` that is defined by the implementation and represents the minor revision number of the PRIF specification (i.e. this document) that the implementation supports.

5 PRIF Procedures

PRIF provides implementations of parallel Fortran features, as specified in Fortran 2023. For any given `prif_*` procedure that corresponds to a Fortran procedure or statement of similar name, the constraints and semantics associated with each argument to the `prif_*` procedure match those of the

analogous argument to the parallel Fortran feature, except where this document explicitly specifies otherwise. Similarly, the constraints and semantics of the PRIF procedure itself match those of the analogous parallel Fortran feature, except where this document explicitly specifies otherwise. In particular, any image synchronization requirements specified for a parallel Fortran feature are performed by the corresponding PRIF procedure, unless otherwise specified.

This section specifies PRIF subroutine declarations, formatted as in this example:

```
subroutine prif_stop(...)
  logical(c_bool), intent(in) :: quiet
  integer(c_int), intent(in), optional :: stop_code_int
  character(len=*), intent(in), optional :: stop_code_char
end subroutine
```

Unless otherwise noted, each such subroutine declaration appearing in this document specifies a public `module subroutine` interface declaration that shall be provided by a compliant PRIF implementation in the `prif` Fortran module, along with an implementation. As shown in the first line of the declaration above, the *dummy-arg-list* is elided using `...` as a presentational short-hand. Subroutine dummy arguments are specified in-order on subsequent lines, and compliant module subroutines shall accept dummy arguments using those same names and ordering.

Where `optional` dummy arguments would be allowed to appear in the corresponding parallel Fortran feature, `optional` dummy arguments are used for the equivalent PRIF procedure. For most cases where a parallel feature provides different overloads with different lists of valid arguments, distinct corresponding procedure variants are specified in PRIF.

IMPLEMENTATION NOTE:

PRIF is defined as a set of Fortran procedures, types and named constants, and as such an implementation of PRIF cannot be expressed solely in C/C++. However C/C++ can be used to implement internal portions of PRIF procedures via calls to `BIND(C)` procedures.

CLIENT NOTE:

PRIF procedures, types and named constants are defined as Fortran entities, *without* the `BIND(C)` attribute (except where otherwise noted), and thus clients should use them as such.

5.1 Common Arguments

There are multiple Common Arguments sections throughout this specification that outline details of the arguments that are common for the following sections of procedure interfaces.

5.1.1 Integer and Pointer Arguments

There are several categories of arguments where the PRIF implementation will need pointers and/or integers. These fall broadly into the following categories:

1. `integer(c_intptr_t)`: Anything containing a pointer representation where the compiler might be expected to perform pointer arithmetic
2. `type(c_ptr)` and `type(c_funptr)`: Anything containing a pointer to an object/function where the compiler is expected only to pass it (back) to the PRIF implementation
3. `integer(c_size_t)`: Anything containing an object size, in units of bytes or elements, e.g. `size_in_bytes`, `offset`, shape, etc.
4. `integer(c_ptrdiff_t)`: strides between elements for non-contiguous coarray accesses
5. `integer(c_int)`: Integer arguments corresponding to image index and stat arguments. It is expected that the most common integer arguments appearing in Fortran code will be of default integer kind, it is expected that this will correspond with that kind, and there is no reason to expect these arguments to have values that would not be representable in this kind.
6. `integer(c_int64_t)`: Cobounds, indices, cosubscripts, and any other argument to an intrinsic procedure that accepts or returns an arbitrary integer.

The compiler is responsible for generating values and temporary variables as necessary to pass arguments of the correct type/size, and perform conversions when needed.

5.1.2 `stat` and `errmsg` Arguments

- **`stat`** : This argument is `intent(out)` and represents the presence and type of any error that occurs. A value of zero indicates no error occurred. It is of type `integer(c_int)`, to minimize the frequency that integer conversions will be needed. If the user program provides a different kind of integer as the argument, it is the compiler's responsibility to use an intermediate variable as the argument to the PRIF procedure and provide conversion to the actual argument.
- **`errmsg` or `errmsg_alloc`** : There are two optional `intent(out)` arguments for this, one which is allocatable and one which is not. It is the compiler's responsibility to ensure the appropriate optional argument is passed, and at most one shall be provided in any given call. If no error occurs, the definition status of the actual argument is unchanged.

5.2 Program Startup and Shutdown

For any program that uses parallel Fortran features, the compiler shall insert calls to `prif_init` and `prif_stop`. These procedures will initialize and terminate the parallel runtime. `prif_init` shall be called prior to any other calls to the PRIF implementation and shall be called at least once per process. Any second or subsequent call to `prif_init` by a given process is guaranteed to return immediately with no effect on system state, with `PRIF_STAT_ALREADY_INIT` assigned to the variable specified in the `stat` argument. `prif_stop` shall be called to initiate normal termination if the program reaches normal termination at the end of the main program.

5.2.1 `prif_init`

Description: This procedure will initialize the parallel environment.

```
subroutine prif_init(...)
  integer(c_int), intent(out) :: stat
end subroutine
```

Further argument descriptions:

- **`stat`**: a zero value indicates success, the named constant `PRIF_STAT_ALREADY_INIT` indicates previous initialization and any other non-zero value indicates an error occurred during initialization

5.2.2 `prif_stop`

Description: This procedure synchronizes all executing images, cleans up the parallel runtime environment, and terminates the program. Calls to this procedure do not return. This procedure supports both normal termination at the end of a program, as well as any `STOP` statements from the user source code.

```
subroutine prif_stop(...)
  logical(c_bool), intent(in) :: quiet
  integer(c_int), intent(in), optional :: stop_code_int
  character(len=*), intent(in), optional :: stop_code_char
end subroutine
```

Further argument descriptions: At most one of the arguments `stop_code_int` or `stop_code_char` shall be supplied.

- **`quiet`**: if this argument has the value `.true.`, no output of signaling exceptions or stop code will be produced. If a `STOP` statement does not contain this optional part, the compiler should provide the value `.false.`.
- **`stop_code_int`**: is used as the process exit code if it is provided. Otherwise, the process exit code is `0`.
- **`stop_code_char`**: is written to the unit identified by the named constant `OUTPUT_UNIT` from the intrinsic module `ISO_FORTRAN_ENV` if provided.

5.2.3 `prif_error_stop`

Description: This procedure terminates all executing images. Calls to this procedure do not return.

```
subroutine prif_error_stop(...)
  logical(c_bool), intent(in) :: quiet
  integer(c_int), intent(in), optional :: stop_code_int
  character(len=*), intent(in), optional :: stop_code_char
end subroutine
```

Further argument descriptions: At most one of the arguments `stop_code_int` or `stop_code_char` shall be supplied.

- `quiet`: if this argument has the value `.true.`, no output of signaling exceptions or stop code will be produced. If an `ERROR STOP` statement does not contain this optional part, the compiler should provide the value `.false.`.
- `stop_code_int`: is used as the process exit code if it is provided. Otherwise, the process exit code is a non-zero value.
- `stop_code_char`: is written to the unit identified by the named constant `ERROR_UNIT` from the intrinsic module `ISO_FORTRAN_ENV` if provided.

5.2.4 `prif_register_stop_callback`

Description: Register a procedure to be invoked when either `prif_stop` or `prif_error_stop` are invoked, before program termination. This procedure may be called multiple times to register multiple different callbacks that will all be invoked during the invocation of `prif_stop` or `prif_error_stop`, in the reverse order in which they were registered by the calling image. The registered procedure is permitted to invoke other `prif` procedures, so long as it does not cause subsequent invocation of either `prif_stop` or `prif_error_stop`. If an image invokes `prif_error_stop`, only that image will invoke the registered callback procedures. For a `prif_stop` invocation, the callback procedures will be invoked after all of the images have synchronized.

```
subroutine prif_register_stop_callback(...)
  procedure(prif_stop_callback_interface), pointer, intent(in) :: callback
end subroutine
```

Further argument descriptions:

- **callback**: shall be a pointer to a procedure with the following interface, where the `is_error_stop` argument indicates whether `prif_stop` or `prif_error_stop` was invoked, and the remaining arguments are the same as the arguments to the `prif_stop` or `prif_error_stop` procedure that led to this callback invocation. The abstract interface below shall be publicly defined in the `prif` module.

```
abstract interface
  subroutine prif_stop_callback_interface(...)
    logical(c_bool), intent(in) :: is_error_stop, quiet
    integer(c_int), intent(in), optional :: stop_code_int
    character(len=*), intent(in), optional :: stop_code_char
  end subroutine
end interface
```

5.2.5 `prif_fail_image`

Description: causes the executing image to cease participating in program execution without initiating termination. Calls to this procedure do not return.

```
subroutine prif_fail_image()
end subroutine
```

5.3 Image Queries

5.3.1 Common Arguments in Image Queries

- `team`: a value of type `prif_team_type` that identifies a current or ancestor team containing the calling image. When the `team` argument has the `optional` attribute and is absent, the team specified is the current team.

5.3.2 `prif_num_images`

Description: Query the number of images in the specified or current team.

```
subroutine prif_num_images(...)
  integer(c_int), intent(out) :: num_images
end subroutine

subroutine prif_num_images_with_team(...)
  type(prif_team_type), intent(in) :: team
  integer(c_int), intent(out) :: num_images
end subroutine

subroutine prif_num_images_with_team_number(...)
  integer(c_int64_t), intent(in) :: team_number
  integer(c_int), intent(out) :: num_images
end subroutine
```

Argument descriptions

Further argument descriptions:

- `team_number`: identifies the initial team or a sibling team of the current team

5.3.3 `prif_this_image`

Description: Determine the image index or cosubscripts with respect to an indicated coarray of the calling image, with respect to a given team or the current team.

```
subroutine prif_this_image_no_coarray(...)
  type(prif_team_type), intent(in), optional :: team
  integer(c_int), intent(out) :: this_image
end subroutine

subroutine prif_this_image_with_coarray(...)
  type(prif_coarray_handle), intent(in) :: coarray_handle
  type(prif_team_type), intent(in), optional :: team
  integer(c_int64_t), intent(out) :: cosubscripts(:)
end subroutine

subroutine prif_this_image_with_dim(...)
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_int), intent(in) :: dim
  type(prif_team_type), intent(in), optional :: team
  integer(c_int64_t), intent(out) :: cosubscript
end subroutine
```

Argument descriptions

Further argument descriptions:

- `coarray_handle`: handle for a descriptor of an established coarray
- `cosubscripts`: the cosubscripts that would identify the calling image in the specified team when used as cosubscripts for the specified coarray
- `dim`: identify which of the elements from `cosubscripts` should be returned as the `cosubscript` value
- `cosubscript`: the element identified by `dim` of the array `cosubscripts` that would have been returned without the `dim` argument present

5.3.4 `prif_failed_images`

Description: Determine the image indices of any images known to have failed. It is the compiler's responsibility to convert to a different kind if the `kind` argument to `FAILED_IMAGES` appears.

```
subroutine prif_failed_images(...)
  type(prif_team_type), intent(in), optional :: team
  integer(c_int), allocatable, intent(out) :: failed_images(:)
end subroutine
```

Argument descriptions

5.3.5 `prif_stopped_images`

Description: Determine the image indices of any images known to have initiated normal termination. It is the compiler's responsibility to convert to a different kind if the `kind` argument to `STOPPED_IMAGES` appears.

```
subroutine prif_stopped_images(...)
  type(prif_team_type), intent(in), optional :: team
  integer(c_int), allocatable, intent(out) :: stopped_images(:)
end subroutine
```

Argument descriptions

5.3.6 `prif_image_status`

Description: Determine the image execution state of an image

```
subroutine prif_image_status(...)
  integer(c_int), intent(in) :: image
  type(prif_team_type), intent(in), optional :: team
  integer(c_int), intent(out) :: image_status
end subroutine
```

Argument descriptions

Further argument descriptions:

- `image`: the image index of the image in the given or current team for which to return the execution status
- `image_status`: defined to the value `PRIF_STAT_FAILED_IMAGE` if the identified image is known to have failed, `PRIF_STAT_STOPPED_IMAGE` if the identified image is known to have initiated normal termination, otherwise zero.

5.4 Storage Management

5.4.1 `prif_allocate_coarray`

Description: This procedure allocates a coarray and provides a handle referencing a corresponding coarray descriptor. This call is collective over the current team. Calls to `prif_allocate_coarray` will be inserted by the compiler when there is an explicit coarray allocation or at the beginning of a program to allocate space for statically declared coarrays in the source code. The PRIF implementation will store the coshape information in order to internally track it during the lifetime of the coarray.

```
subroutine prif_allocate_coarray(...)
  integer(c_int64_t), intent(in) :: lcobounds(:), ucobounds(:)
  integer(c_size_t), intent(in) :: size_in_bytes
  type(c_funptr), intent(in) :: final_func
  type(prif_coarray_handle), intent(out) :: coarray_handle
  type(c_ptr), intent(out) :: allocated_memory
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Further argument descriptions:

- `lcobounds` and `ucobounds`: Shall be the lower and upper bounds of the codimensions of the coarray being allocated. Shall be rank-one arrays with the same dimensions as each other. The specified cobounds shall be sufficient to have a unique index for every image in the current team. I.e. `product(ucobounds - lcobounds + 1) >= num_images()`.
- `size_in_bytes`: The size, in bytes, of the calling image's portion of the coarray element data. This argument shall have the same value on all images in corresponding calls.
- `final_func`: Shall be the C address of a procedure that is interoperable, or `C_NULL_FUNPTR`. If not null, this procedure will be invoked by the PRIF implementation once by each image at deallocation of this coarray, before the storage is released. The procedure's interface shall be equivalent to the following Fortran interface

  ```
  subroutine coarray_cleanup(handle, stat, errmsg) bind(C)
    type(prif_coarray_handle), pointer, intent(in) :: handle
    integer(c_int), intent(out) :: stat
    character(len=:), intent(out), allocatable :: errmsg
  end subroutine
  ```

  or to the following equivalent C prototype:

  ```
  void coarray_cleanup(
      CFI_cdesc_t* handle, int* stat, CFI_cdesc_t* errmsg)
  ```

- `coarray_handle`: A handle referencing the calling image's coarray descriptor of the allocated coarray. The descriptor and handle are created by the PRIF implementation, and the compiler uses the handle for subsequent operations involving the associated coarray and for deallocation of the associated coarray.
- `allocated_memory`: A pointer to the block of allocated but uninitialized memory that provides the storage for the calling image's coarray element data. The compiler is responsible for associating the Fortran-level coarray object with this storage, and initializing the storage if necessary. The returned pointer value may differ across corresponding calls from images in the team.

CLIENT NOTE:

`final_func` is used by the compiler to support various clean-up operations at coarray deallocation, whether it happens explicitly (i.e. via `prif_deallocate_coarray`) or implicitly (e.g. via `prif_end_team`). First, `final_func` may be used to support the user-defined final subroutine for derived types. Second, it may be necessary for the compiler to generate such a subroutine to clean up allocatable components, typically with calls to `prif_deallocate`. Third, it may also be necessary to modify the allocation status of an allocatable coarray variable, especially in the case that it was allocated through a dummy argument.
The coarray handle can be interrogated by the procedure callback using PRIF queries to determine the memory address and size of the data in order to orchestrate calling any necessary final subroutines or deallocation of any allocatable components, or the context data to orchestrate modifying the allocation status of a local variable portion of the coarray. The `pointer` attribute for the `handle` argument is required because `prif_coarray_handle` is not C interoperable.

5.4.2 `prif_allocate`

Description: This procedure is used to non-collectively allocate remotely accessible storage, such as needed for an allocatable component of a coarray.

```
subroutine prif_allocate(...)
  integer(c_size_t), intent(in) :: size_in_bytes
  type(c_ptr), intent(out) :: allocated_memory
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Further argument descriptions:

- `size_in_bytes`: The size, in bytes, of the object to be allocated.
- `allocated_memory`: A pointer to the block of allocated but uninitialized memory that provides the requested storage. The compiler is responsible for associating the Fortran object with this storage, and initializing the storage if necessary.

5.4.3 `prif_deallocate_coarray`

Description: This procedure releases memory previously allocated for all of the coarrays associated with the handles in `coarray_handles`. This means that any local objects associated with this memory become invalid. The compiler will insert calls to this procedure when exiting a local scope where implicit deallocation of a coarray is mandated by the standard and when a coarray is explicitly deallocated through a `DEALLOCATE` statement. This call is collective over the current team, and the provided array of handles must denote corresponding coarrays (in the same order on every image) that were allocated by the current team using `prif_allocate_coarray` and not yet deallocated. The implementation starts with a synchronization over the current team, and then the final subroutine for each coarray (if any) will be called. A synchronization will also occur before control is returned from this procedure, after all deallocation has been completed.

```
subroutine prif_deallocate_coarray(...)
  type(prif_coarray_handle), intent(in) :: coarray_handles(:)
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Further argument descriptions:

- `coarray_handles`: Is an array of all of the handles for the coarrays that shall be deallocated.

5.4.4 `prif_deallocate`

Description: This non-collective procedure releases memory previously allocated by a call to `prif_allocate` and not yet deallocated.

```
subroutine prif_deallocate(...)
  type(c_ptr), intent(in) :: mem
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Further argument descriptions:

- `mem`: Pointer to the block of memory to be released.

CLIENT NOTE:

Calls to `prif_allocate_coarray` and `prif_deallocate_coarray` are collective operations, while calls to `prif_allocate` and `prif_deallocate` are not. Note that a call to `MOVE_ALLOC` with coarray arguments is also a collective operation, as described in the section below.

CLIENT NOTE:

The compiler is responsible to generate code that collectively runs `prif_allocate_coarray` once for each static coarray and initializes them where applicable.

5.4.5 `prif_alias_create`

Description: Create a new coarray descriptor aliased to an existing coarray, with possibly altered corank and cobounds. This may be needed as part of `CHANGE TEAM` after `prif_change_team`, or to pass to a coarray dummy argument (especially in the case that the cobounds are different). This call does not alter data in the coarray.

```
subroutine prif_alias_create(...)
  type(prif_coarray_handle), intent(in) :: source_handle
  integer(c_int64_t), intent(in) :: alias_lcobounds(:)
  integer(c_int64_t), intent(in) :: alias_ucobounds(:)
  type(prif_coarray_handle), intent(out) :: alias_handle
end subroutine
```

Further argument descriptions:

- `source_handle`: a handle to an existing coarray descriptor (which may itself be an alias) for which a new alias descriptor is to be created. The original descriptor is not modified.
- `alias_lcobounds` and `alias_ucobounds`: the cobounds to be used for the new alias. Both arguments must have the same size, but it need not match the corank associated with `source_handle`
- `alias_handle`: a handle to a new coarray descriptor that aliases the data in an existing coarray

5.4.6 `prif_alias_destroy`

Description: Delete an alias descriptor for a coarray. Does not deallocate or alter the original coarray.

```
subroutine prif_alias_destroy(...)
  type(prif_coarray_handle), intent(in) :: alias_handle
end subroutine
```

Further argument descriptions:

- `alias_handle`: handle to the alias descriptor to be destroyed

5.4.7 `MOVE_ALLOC`

`MOVE_ALLOC` is not provided by PRIF because it depends on unspecified details of the compiler's `allocatable` representation. It is the compiler's responsibility to implement `MOVE_ALLOC` using PRIF-provided operations. For example, according to the Fortran standard, `MOVE_ALLOC` with coarray arguments is an image control statement that requires synchronization, so the compiler should likely insert call(s) to `prif_sync_all` as part of the implementation.

CLIENT NOTE:

It is envisioned that the use of `prif_set_context_data` and `prif_get_context_data` will allow for an efficient implementation of `MOVE_ALLOC` that maintains tracking of allocation status

5.5 Coarray Queries

5.5.1 Common Arguments in Coarray Queries

- `coarray_handle`: a handle for a descriptor of an established coarray to be accessed by this operation

5.5.2 `prif_image_index`

Description: This procedure returns the index of the image, on the identified team or the current team if no team is provided, identified by the cosubscripts provided in the `sub` argument applied to the indicated coarray descriptor

```
subroutine prif_image_index(...)
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_int64_t), intent(in) :: sub(:)
  integer(c_int), intent(out) :: image_index
end subroutine

subroutine prif_image_index_with_team(...)
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_int64_t), intent(in) :: sub(:)
  type(prif_team_type), intent(in) :: team
  integer(c_int), intent(out) :: image_index
end subroutine

subroutine prif_image_index_with_team_number(...)
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_int64_t), intent(in) :: sub(:)
  integer(c_int), intent(in) :: team_number
  integer(c_int), intent(out) :: image_index
end subroutine
```

Argument descriptions

Further argument descriptions:

- `team` and `team_number`: Specifies a team
- `sub`: A list of integers that identify a specific image in the identified or current team when interpreted as cosubscripts for the specified coarray descriptor.

5.5.3 `prif_lcobound`

Description: This procedure returns the lower cobound(s) associated with a coarray descriptor. It is the compiler's responsibility to convert to a different kind if the `kind` argument to `LCOBOUND` appears.

```
subroutine prif_lcobound_no_dim(...)
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_int64_t), intent(out) :: lcobounds(:)
end subroutine

subroutine prif_lcobound_with_dim(...)
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_int), intent(in) :: dim
  integer(c_int64_t), intent(out):: lcobound
end subroutine
```

Argument descriptions

Further argument descriptions:

- `lcobounds`: an array of the size of the corank of the coarray descriptor, returns the lower cobounds of the indicated coarray descriptor
- `dim`: which codimension of the coarray descriptor to report the lower cobound of
- `lcobound`: the lower cobound of the given dimension

5.5.4 `prif_ucobound`

Description: This procedure returns the upper cobound(s) associated with a coarray descriptor. It is the compiler's responsibility to convert to a different kind if the `kind` argument to `UCOBOUND` appears.

```
subroutine prif_ucobound_no_dim(...)
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_int64_t), intent(out) :: ucobounds(:)
end subroutine

subroutine prif_ucobound_with_dim(...)
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_int), intent(in) :: dim
  integer(c_int64_t), intent(out):: ucobound
end subroutine
```

Argument descriptions

Further argument descriptions:

- `ucobounds`: an array of the size of the corank of the coarray descriptor, returns the upper cobounds of the indicated coarray descriptor
- `dim`: which codimension of the coarray descriptor to report the upper cobound of
- `ucobound`: the upper cobound of the given dimension

5.5.5 `prif_coshape`

Description: This procedure returns the sizes of codimensions of a coarray descriptor. It is the compiler's responsibility to convert to a different kind if the `kind` argument to `COSHAPE` appears.

```
subroutine prif_coshape(...)
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_size_t), intent(out) :: sizes(:)
end subroutine
```

Argument descriptions

Further argument descriptions:

- `sizes`: an array whose size matches the corank of the indicated coarray descriptor, returns the difference between the upper and lower cobounds + 1

5.5.6 `prif_local_data_pointer`

Description: This procedure returns a `c_ptr` referencing the block of storage containing the calling image's element data in the coarray associated with the given handle. In the case of a `prif_coarray_handle` constructed by a call to `prif_allocate_coarray`, the `local_data` argument is defined to the same value as the `allocated_memory` argument provided during allocation of that coarray.

```
subroutine prif_local_data_pointer(...)
  type(prif_coarray_handle), intent(in) :: coarray_handle
  type(c_ptr), intent(out) :: local_data
end subroutine
```

Argument descriptions

5.5.7 `prif_size_bytes`

Description: This procedure returns the size of the coarray element data associated with each image. This will be equal to the `size_in_bytes` argument provided to `prif_allocate_coarray` at the time that the coarray was allocated.

```
subroutine prif_size_bytes(...)
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_size_t), intent(out) :: data_size
end subroutine
```

Argument descriptions

CLIENT NOTE:

`prif_size_bytes` can be used to calculate the number of elements in an array coarray given only the handle and element size

5.5.8 `prif_set_context_data`

Description: This procedure stores a `c_ptr` associated with a coarray for future retrieval. A typical usage would be to store a reference to the actual variable whose allocation status must be changed in the case that the coarray is deallocated.

```
subroutine prif_set_context_data(...)
  type(prif_coarray_handle), intent(in) :: coarray_handle
  type(c_ptr), intent(in) :: context_data
end subroutine
```

Argument descriptions

CLIENT NOTE:

Each coarray includes some "context data" on a per-image basis, which the compiler may use to support proper implementation of coarray arguments, especially with respect to `MOVE_ALLOC` operations on allocatable coarrays. This data is accessed using the procedures `prif_get_context_data` and `prif_set_context_data`. PRIF does not interpret the contents of this context data in any way, and it is only accessible on the current image. The context data is a property of the allocated coarray object, and is thus shared between all handles and aliased descriptors that refer to the same coarray allocation (i.e. those created from a call to `prif_alias_create`).

5.5.9 `prif_get_context_data`

Description: This procedure returns the `c_ptr` provided in the most recent call to `prif_set_context_data` with the same coarray (possibly via an aliased coarray descriptor).

```
subroutine prif_get_context_data(...)
  type(prif_coarray_handle), intent(in) :: coarray_handle
  type(c_ptr), intent(out) :: context_data
end subroutine
```

Argument descriptions

5.6 Contiguous Coarray Access

The memory consistency semantics of coarray accesses follow those defined by the Image Execution Control section of the Fortran standard. In particular, coarray accesses will maintain serial dependencies for the issuing image. Any data access ordering between images is defined only with respect to ordered segments. Note that for put operations, "source completion" means that the provided source locations are no longer needed (e.g. their memory can be freed once the procedure has returned).

5.6.1 Common Arguments in Contiguous Coarray Access

- `image_num`
  - an argument identifying the image to be communicated with
  - is permitted to identify the calling image
  - this image index is always relative to the initial team, regardless of the current team

- `coarray_handle`: handle for a descriptor of an established coarray to be accessed by this operation. `offset` and `size_in_bytes` must specify a range of storage entirely contained within the element data of the indicated coarray.
- `offset`: indicates an offset in bytes from the beginning of the elements in a remote coarray (indicated by `coarray_handle`) on a selected image (indicated by `image_num`)
- `remote_ptr`: pointer to where on the identified image the data begins. The referenced storage must have been allocated using `prif_allocate` or `prif_allocate_coarray`.
- `current_image_buffer`: pointer to contiguous memory on the calling image that either contains the source data to be copied (puts) or is the destination memory for the data to be retrieved (gets).
- `size_in_bytes`: how much data is to be transferred in bytes
- `notify_ptr`: pointer on the identified image to the notify variable that should be updated after completion of the put operation. The referenced variable shall be of type `prif_notify_type`, and the storage must have been allocated using `prif_allocate` or `prif_allocate_coarray`.
- `notify_coarray_handle, notify_offset`: a coarray handle and byte offset that identifies the location of a `prif_notify_type` variable to be updated after completion of the put operation. That variable must be entirely contained within the element data of the coarray indicated by `notify_coarray_handle`

5.6.2 `prif_get`

Description: This procedure fetches data in a coarray from a specified image, when the data to be copied are contiguous in linear memory on both sides. The compiler can use this to implement reads from a coindexed object. It need not call PRIF when a coarray reference is not a coindexed object. This procedure blocks until the requested data has been successfully assigned to the `current_image_buffer` argument. If `image_num` indicates the calling image and there is any overlap between the source and destination memory regions, then behavior is undefined. This procedure corresponds to a coindexed object reference that reads contiguous coarray data.

```
subroutine prif_get(...)
  integer(c_int), intent(in) :: image_num
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_size_t), intent(in) :: offset
  type(c_ptr), intent(in) :: current_image_buffer
  integer(c_size_t), intent(in) :: size_in_bytes
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Argument descriptions

5.6.3 `prif_get_indirect`

Description: This procedure implements the semantics of `prif_get` but fetches `size_in_bytes` number of contiguous bytes from the given image, starting at `remote_ptr` on the given image, copying into `current_image_buffer`.

```
subroutine prif_get_indirect(...)
  integer(c_int), intent(in) :: image_num
  integer(c_intptr_t), intent(in) :: remote_ptr
  type(c_ptr), intent(in) :: current_image_buffer
  integer(c_size_t), intent(in) :: size_in_bytes
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Argument descriptions

5.6.4 `prif_put`

Description: This procedure assigns to the element data of a coarray, when the data to be assigned are contiguous in linear memory on both sides. The compiler can use this to implement assignment to a coindexed object. It need not call PRIF when a coarray reference is not a coindexed object. This procedure blocks on source completion. If `image_num` indicates the calling image and there is any overlap between the source and destination memory regions, then behavior is undefined. This procedure corresponds to a contiguous coindexed object reference on the left hand side of an assignment statement.

```
subroutine prif_put(...)
  integer(c_int), intent(in) :: image_num
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_size_t), intent(in) :: offset
  type(c_ptr), intent(in) :: current_image_buffer
  integer(c_size_t), intent(in) :: size_in_bytes
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Argument descriptions

5.6.5 `prif_put_indirect`

Description: This procedure implements the semantics of `prif_put` but assigns to `size_in_bytes` number of contiguous bytes on the given image, starting at `remote_ptr` on the given image, copying from `current_image_buffer`.

```
subroutine prif_put_indirect(...)
  integer(c_int), intent(in) :: image_num
  integer(c_intptr_t), intent(in) :: remote_ptr
  type(c_ptr), intent(in) :: current_image_buffer
  integer(c_size_t), intent(in) :: size_in_bytes
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Argument descriptions

5.6.6 `prif_put_with_notify`

Description: This procedure implements the semantics of `prif_put` with the addition of support for the semantics of the `NOTIFY=` specifier through a coarray handle and an offset

```
subroutine prif_put_with_notify(...)
  integer(c_int), intent(in) :: image_num
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_size_t), intent(in) :: offset
  type(c_ptr), intent(in) :: current_image_buffer
  integer(c_size_t), intent(in) :: size_in_bytes
  type(prif_coarray_handle), intent(in) :: notify_coarray_handle
  integer(c_size_t), intent(in) :: notify_offset
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Argument descriptions

5.6.7 `prif_put_with_notify_indirect`

Description: This procedure implements the semantics of `prif_put` with the addition of support for the semantics of the `NOTIFY=` specifier through a pointer

```
subroutine prif_put_with_notify_indirect(...)
  integer(c_int), intent(in) :: image_num
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_size_t), intent(in) :: offset
  type(c_ptr), intent(in) :: current_image_buffer
  integer(c_size_t), intent(in) :: size_in_bytes
  integer(c_intptr_t), intent(in) :: notify_ptr
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Argument descriptions

5.6.8 `prif_put_indirect_with_notify`

Description: This procedure implements the semantics of `prif_put` but assigns to `size_in_bytes` number of contiguous bytes on the given image, starting at `remote_ptr` on the given image, copying from `current_image_buffer` and with support for the `NOTIFY=` specifier through a coarray handle and offset

```
subroutine prif_put_indirect_with_notify(...)
  integer(c_int), intent(in) :: image_num
  integer(c_intptr_t), intent(in) :: remote_ptr
  type(c_ptr), intent(in) :: current_image_buffer
  integer(c_size_t), intent(in) :: size_in_bytes
  type(prif_coarray_handle), intent(in) :: notify_coarray_handle
  integer(c_size_t), intent(in) :: notify_offset
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Argument descriptions

5.6.9 `prif_put_indirect_with_notify_indirect`

Description: This procedure implements the semantics of `prif_put` but assigns to `size_in_bytes` number of contiguous bytes on the given image, starting at `remote_ptr` on the given image, copying from `current_image_buffer` and with support for the `NOTIFY=` specifier through a pointer

```
subroutine prif_put_indirect_with_notify_indirect(...)
  integer(c_int), intent(in) :: image_num
  integer(c_intptr_t), intent(in) :: remote_ptr
  type(c_ptr), intent(in) :: current_image_buffer
  integer(c_size_t), intent(in) :: size_in_bytes
  integer(c_intptr_t), intent(in) :: notify_ptr
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Argument descriptions

5.7 Strided Coarray Access

5.7.1 Common Arguments in Strided Coarray Access

- `image_num`
  - an argument identifying the image to be communicated with
  - is permitted to identify the calling image
  - this image index is always relative to the initial team, regardless of the current team
- `coarray_handle`: handle for a descriptor of an established coarray to be accessed by this operation. The combination of arguments must specify a set of storage locations entirely contained within the element data of the indicated coarray.
- `offset`: indicates an offset in bytes from the beginning of the elements in a remote coarray (indicated by `coarray_handle`) on a selected image (indicated by `image_num`)
- `remote_ptr`: pointer to where on the identified image the data begins. The referenced storage must have been allocated using `prif_allocate` or `prif_allocate_coarray`.
- `remote_stride`: The stride (in units of bytes) between elements in each dimension on the specified image. Each component of stride may independently be positive or negative, but (together with `extent`) must specify a region of distinct (non-overlapping) locations. For the procedures that provide the `remote_ptr` argument, the striding starts at the `remote_ptr`. For the procedures that provide the `coarray_handle` and `offset` arguments, the striding starts at the location that resides at `offset` bytes past the beginning of the remote elements indicated by `coarray_handle`.
- `current_image_buffer`: pointer to memory on the calling image that either contains the source data to be copied (puts) or is the destination memory for the data to be retrieved (gets).
- `current_image_stride`: The stride (in units of bytes) between elements in each dimension in the calling image buffer. Each component of stride may independently be positive or negative, but (together with `extent`) must specify a region of distinct (non-overlapping) locations. The striding starts at the `current_image_buffer`.
- `element_size`: The size, in bytes, of each block of contiguous element data to be copied
- `extent`: How many elements in each dimension should be transferred. `remote_stride`, `current_image_stride` and `extent` must all have equal size.
- `notify_coarray_handle, notify_offset`: a coarray handle and byte offset that identifies the location of a `prif_notify_type` variable to be updated after completion of the put operation. That variable must be entirely contained within the element data of the coarray indicated by `notify_coarray_handle`
- `notify_ptr`: pointer on the identified image to the notify variable that should be updated after completion of the put operation. The referenced variable shall be of type `prif_notify_type`, and the storage must have been allocated using `prif_allocate` or `prif_allocate_coarray`.

5.7.2 `prif_get_strided`

Description: Copy data from the given image and indicated coarray, writing into `current_image_buffer`, progressing through `current_image_buffer` in `current_image_stride` increments and through remote memory in `remote_stride` increments, transferring `extent` number of elements in each dimension. This procedure blocks until the requested data has been successfully assigned to the destination locations on the calling image. If `image_num` indicates the calling image and any memory location is specified as both a source and destination location, then behavior is undefined.

```
subroutine prif_get_strided(...)
  integer(c_int), intent(in) :: image_num
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_size_t), intent(in) :: offset
  integer(c_ptrdiff_t), intent(in) :: remote_stride(:)
  type(c_ptr), intent(in) :: current_image_buffer
  integer(c_ptrdiff_t), intent(in) :: current_image_stride(:)
  integer(c_size_t), intent(in) :: element_size
  integer(c_size_t), intent(in) :: extent(:)
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Argument descriptions

5.7.3 `prif_get_strided_indirect`

Description: This procedure implements the semantics of `prif_get_strided` but starting at `remote_ptr` on the given image.

```
subroutine prif_get_strided_indirect(...)
  integer(c_int), intent(in) :: image_num
  integer(c_intptr_t), intent(in) :: remote_ptr
  integer(c_ptrdiff_t), intent(in) :: remote_stride(:)
  type(c_ptr), intent(in) :: current_image_buffer
  integer(c_ptrdiff_t), intent(in) :: current_image_stride(:)
  integer(c_size_t), intent(in) :: element_size
  integer(c_size_t), intent(in) :: extent(:)
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Argument descriptions

5.7.4 `prif_put_strided`

Description: Assign data to memory on the given image, starting at the location indicated by `coarray_handle` and `offset`, copying from `current_image_buffer`, progressing through `current_image_buffer` in `current_image_stride` increments and through remote memory in `remote_stride` increments, transferring `extent` number of elements in each dimension. This procedure blocks on source completion. If `image_num` indicates the calling image and any memory location is specified as both a source and destination location, then behavior is undefined.

```
subroutine prif_put_strided(...)
  integer(c_int), intent(in) :: image_num
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_size_t), intent(in) :: offset
  integer(c_ptrdiff_t), intent(in) :: remote_stride(:)
  type(c_ptr), intent(in) :: current_image_buffer
  integer(c_ptrdiff_t), intent(in) :: current_image_stride(:)
  integer(c_size_t), intent(in) :: element_size
  integer(c_size_t), intent(in) :: extent(:)
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Argument descriptions

5.7.5 `prif_put_strided_indirect`

Description: This procedure implements the semantics of `prif_put_strided` but starting at `remote_ptr` on the given image.

```
subroutine prif_put_strided_indirect(...)
  integer(c_int), intent(in) :: image_num
  integer(c_intptr_t), intent(in) :: remote_ptr
  integer(c_ptrdiff_t), intent(in) :: remote_stride(:)
  type(c_ptr), intent(in) :: current_image_buffer
  integer(c_ptrdiff_t), intent(in) :: current_image_stride(:)
  integer(c_size_t), intent(in) :: element_size
  integer(c_size_t), intent(in) :: extent(:)
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Argument descriptions

5.7.6 `prif_put_strided_with_notify`

Description: This procedure implements the semantics of `prif_put_strided` with support for the `NOTIFY=` specifier through a coarray handle and an offset.

```
subroutine prif_put_strided_with_notify(...)
  integer(c_int), intent(in) :: image_num
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_size_t), intent(in) :: offset
  integer(c_ptrdiff_t), intent(in) :: remote_stride(:)
  type(c_ptr), intent(in) :: current_image_buffer
  integer(c_ptrdiff_t), intent(in) :: current_image_stride(:)
  integer(c_size_t), intent(in) :: element_size
  integer(c_size_t), intent(in) :: extent(:)
  type(prif_coarray_handle), intent(in) :: notify_coarray_handle
  integer(c_size_t), intent(in) :: notify_offset
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Argument descriptions

5.7.7 `prif_put_strided_with_notify_indirect`

Description: This procedure implements the semantics of `prif_put_strided` with support for the `NOTIFY=` specifier through a pointer.

```
subroutine prif_put_strided_with_notify_indirect(...)
  integer(c_int), intent(in) :: image_num
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_size_t), intent(in) :: offset
  integer(c_ptrdiff_t), intent(in) :: remote_stride(:)
  type(c_ptr), intent(in) :: current_image_buffer
  integer(c_ptrdiff_t), intent(in) :: current_image_stride(:)
  integer(c_size_t), intent(in) :: element_size
  integer(c_size_t), intent(in) :: extent(:)
  integer(c_intptr_t), intent(in) :: notify_ptr
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Argument descriptions

5.7.8 `prif_put_strided_indirect_with_notify`

Description: This procedure implements the semantics of `prif_put_strided` but starting at `remote_ptr` on the given image and with support for the `NOTIFY=` specifier through a coarray handle and an offset.

```
subroutine prif_put_strided_indirect_with_notify(...)
  integer(c_int), intent(in) :: image_num
  integer(c_intptr_t), intent(in) :: remote_ptr
  integer(c_ptrdiff_t), intent(in) :: remote_stride(:)
  type(c_ptr), intent(in) :: current_image_buffer
  integer(c_ptrdiff_t), intent(in) :: current_image_stride(:)
  integer(c_size_t), intent(in) :: element_size
  integer(c_size_t), intent(in) :: extent(:)
  type(prif_coarray_handle), intent(in) :: notify_coarray_handle
  integer(c_size_t), intent(in) :: notify_offset
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Argument descriptions

5.7.9 `prif_put_strided_indirect_with_notify_indirect`

Description: This procedure implements the semantics of `prif_put_strided` but starting at `remote_ptr` on the given image and with support for the `NOTIFY=` specifier through a pointer.

```
subroutine prif_put_strided_indirect_with_notify_indirect(...)
  integer(c_int), intent(in) :: image_num
  integer(c_intptr_t), intent(in) :: remote_ptr
  integer(c_ptrdiff_t), intent(in) :: remote_stride(:)
  type(c_ptr), intent(in) :: current_image_buffer
  integer(c_ptrdiff_t), intent(in) :: current_image_stride(:)
  integer(c_size_t), intent(in) :: element_size
  integer(c_size_t), intent(in) :: extent(:)
  integer(c_intptr_t), intent(in) :: notify_ptr
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Argument descriptions

5.8 `SYNC` Statements

5.8.1 `prif_sync_memory`

Description: Ends one Fortran segment and begins another, waiting on any pending communication operations with other images.

```
subroutine prif_sync_memory(...)
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

5.8.2 `prif_sync_all`

Description: Performs a collective synchronization of all images in the current team.

```
subroutine prif_sync_all(...)
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

5.8.3 `prif_sync_team`

Description: Performs a collective synchronization with the images of the identified team.

```
subroutine prif_sync_team(...)
  type(prif_team_type), intent(in) :: team
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Further argument descriptions:

- `team`: Identifies the team to synchronize.

5.8.4 `prif_sync_images`

Description: Performs a collective synchronization with the listed images.

```
subroutine prif_sync_images(...)
  integer(c_int), intent(in), optional :: image_set(:)
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Further argument descriptions:

- `image_set`: The image indices of the images in the current team with which to synchronize. Image indices are relative to the current team. Given a scalar argument to SYNC IMAGES, the compiler should pass its value in an array of size 1. Given an asterisk (`*`) argument to SYNC IMAGES, the compiler should omit the `image_set` argument.

5.9 Locks

5.9.1 Common Arguments in Locks

- `image_num`
  - an argument identifying the image to be communicated with
  - is permitted to identify the calling image
  - this image index is always relative to the initial team, regardless of the current team
- `coarray_handle`: handle for a descriptor of an established coarray to be accessed by this operation. Together with `offset` must identify the location of a `prif_lock_type` variable entirely contained within the element data of the indicated coarray.
- `offset`: indicates an offset in bytes from the beginning of the elements in a remote coarray (indicated by `coarray_handle`) on a selected image (indicated by `image_num`)
- `lock_var_ptr`: a pointer to the base address of the lock variable to be locked or unlocked on the identified image. The referenced variable shall be of type `prif_lock_type`, and the referenced storage must have been allocated using `prif_allocate` or `prif_allocate_coarray`.
- `acquired_lock`: when present, the argument is defined to `.true.` if the lock has become locked by the calling image, otherwise is defined to `.false.`

5.9.2 `prif_lock`

Description: Waits until the identified lock variable is unlocked and then locks it if the `acquired_lock` argument is not present. Otherwise it sets the `acquired_lock` argument to `.false.` if the identified lock variable was already locked by another image, or locks the identified lock variable and sets the `acquired_lock` argument to `.true.`. If the identified lock variable was already locked by the calling image, then an error condition occurs.

```
subroutine prif_lock(...)
  integer(c_int), intent(in) :: image_num
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_size_t), intent(in) :: offset
  logical(c_bool), intent(out), optional :: acquired_lock
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Argument descriptions

5.9.3 `prif_lock_indirect`

Description: This procedure implements the semantics of `prif_lock`, but with the lock variable identified by `lock_var_ptr`.

```
subroutine prif_lock_indirect(...)
  integer(c_int), intent(in) :: image_num
  integer(c_intptr_t), intent(in) :: lock_var_ptr
  logical(c_bool), intent(out), optional :: acquired_lock
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Argument descriptions

5.9.4 `prif_unlock`

Description: Unlocks the identified lock variable. If the identified lock variable was not locked by the calling image, then an error condition occurs.

```
subroutine prif_unlock(...)
  integer(c_int), intent(in) :: image_num
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_size_t), intent(in) :: offset
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Argument descriptions

5.9.5 `prif_unlock_indirect`

Description: This procedure implements the semantics of `prif_unlock`, but with the lock variable identified by `lock_var_ptr`.

```
subroutine prif_unlock_indirect(...)
  integer(c_int), intent(in) :: image_num
  integer(c_intptr_t), intent(in) :: lock_var_ptr
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Argument descriptions

5.10 Critical

The compiler shall define a coarray, and establish (allocate) it in the initial team, that shall only be used to begin and end critical blocks. An efficient compiler may allocate one such coarray for each critical block. The coarray shall be a scalar coarray of type `prif_critical_type`.

5.10.1 Common Arguments in Critical

- `critical_coarray`: the handle for the `prif_critical_type` coarray associated with the selected critical construct

5.10.2 `prif_critical`

Description: This procedure waits until any other image which has executed this procedure with a corresponding coarray has subsequently executed `prif_end_critical` with the same coarray an identical number of times.

```
subroutine prif_critical(...)
  type(prif_coarray_handle), intent(in) :: critical_coarray
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Argument descriptions

5.10.3 `prif_end_critical`

Description: Completes execution of the critical construct associated with the provided coarray handle.

```
subroutine prif_end_critical(...)
  type(prif_coarray_handle), intent(in) :: critical_coarray
end subroutine
```

Argument descriptions

5.11 Events and Notifications

5.11.1 Common Arguments in Events and Notifications

- `image_num`
  - an argument identifying the image to be communicated with
  - is permitted to identify the calling image
  - this image index is always relative to the initial team, regardless of the current team

5.11.2 `prif_event_post`

Description: Atomically increment the count of the event variable by one.

```
subroutine prif_event_post(...)
  integer(c_int), intent(in) :: image_num
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_size_t), intent(in) :: offset
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Argument descriptions

Further argument descriptions:

- `coarray_handle`: handle for a descriptor of an established coarray to be accessed by this operation. Together with `offset` must identify the location of a `prif_event_type` variable entirely contained within the element data of the indicated coarray.

- `offset`: indicates an offset in bytes from the beginning of the elements in a remote coarray (indicated by `coarray_handle`) on a selected image (indicated by `image_num`)

5.11.3 `prif_event_post_indirect`

Description: Atomically increment the count of the event variable by one.

```
subroutine prif_event_post_indirect(...)
  integer(c_int), intent(in) :: image_num
  integer(c_intptr_t), intent(in) :: event_var_ptr
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Argument descriptions

Further argument descriptions:

- `event_var_ptr`: a pointer to the base address of the event variable to be incremented on the identified image. The referenced variable shall be of type `prif_event_type`, and the referenced storage must have been allocated using `prif_allocate` or `prif_allocate_coarray`.

5.11.4 `prif_event_wait`

Description: Wait until the count of the provided event variable on the calling image is greater than or equal to `until_count`, and then atomically decrement the count by that value.

```
subroutine prif_event_wait(...)
  type(c_ptr), intent(in) :: event_var_ptr
  integer(c_int64_t), intent(in), optional :: until_count
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Further argument descriptions:

- `event_var_ptr`: a pointer to the event variable on the calling image to be waited on. The referenced variable shall be of type `prif_event_type`, and the referenced storage must have been allocated using `prif_allocate_coarray` or `prif_allocate`.
- `until_count`: the count of the given event variable to be waited for. Treated as the value 1 if not provided.

5.11.5 `prif_event_query`

Description: Query the count of an event variable on the calling image.

```
subroutine prif_event_query(...)
  type(c_ptr), intent(in) :: event_var_ptr
  integer(c_int64_t), intent(out) :: count
  integer(c_int), intent(out), optional :: stat
end subroutine
```

Further argument descriptions:

- `event_var_ptr`: a pointer to the event variable on the calling image to be queried. The referenced variable shall be of type `prif_event_type`, and the referenced storage must have been allocated using `prif_allocate_coarray` or `prif_allocate`.
- `count`: the current count of the given event variable.

5.11.6 `prif_notify_wait`

Description: Wait on notification of an incoming put operation

```
subroutine prif_notify_wait(...)
  type(c_ptr), intent(in) :: notify_var_ptr
  integer(c_int64_t), intent(in), optional :: until_count
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Further argument descriptions:

- `notify_var_ptr`: a pointer to the notify variable on the calling image to be waited on. The referenced variable shall be of type `prif_notify_type`, and the referenced storage must have been allocated using `prif_allocate_coarray` or `prif_allocate`.
- `until_count`: the count of the given notify variable to be waited for. Treated as the value 1 if not provided.

5.12 Teams

Team creation forms a hierarchical abstraction, where any given team may create zero or more child teams. The initial team is created by the `prif_init` procedure. Each subsequently created team's parent is the then-current team. Team membership is thus strictly hierarchical, and there is a unique path from any given team, along the ancestry tree delineated by team creation, back to the initial team.

5.12.1 `prif_form_team`

Description: Create teams. Each image receives a team value denoting the newly created team containing all images in the current team which specify the same value for `team_number`.

```
subroutine prif_form_team(...)
  integer(c_int64_t), intent(in) :: team_number
  type(prif_team_type), intent(out) :: team
  integer(c_int), intent(in), optional :: new_index
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Further argument descriptions:

- `new_index`: the index that the calling image will have in its new team

5.12.2 `prif_get_team`

Description: Get the team value for the current or an ancestor team. It returns the current team if `level` is not present or has the value `PRIF_CURRENT_TEAM`, the parent team if `level` is present with the value `PRIF_PARENT_TEAM`, or the initial team if `level` is present with the value `PRIF_INITIAL_TEAM`

```
subroutine prif_get_team(...)
  integer(c_int), intent(in), optional :: level
  type(prif_team_type), intent(out) :: team
end subroutine
```

Further argument descriptions:

- `level`: identify which team value to be returned

5.12.3 `prif_team_number`

Description: This procedure returns the `team_number` that was specified in the call to `prif_form_team` for the specified team, or -1 if the team is the initial team. If `team` is not present, the current team is indicated.

```
subroutine prif_team_number(...)
  type(prif_team_type), intent(in), optional :: team
  integer(c_int64_t), intent(out) :: team_number
end subroutine
```

5.12.4 `prif_change_team`

Description: changes the current team to the specified team. For any associate names specified in the `CHANGE TEAM` statement the compiler should follow a call to this procedure with calls to `prif_alias_create` to create an aliased coarray descriptor, and associate any non-coindexed references to the associate name within the `CHANGE TEAM` construct with the selector.

```
subroutine prif_change_team(...)
  type(prif_team_type), intent(in) :: team
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

5.12.5 `prif_end_team`

Description: Changes the current team to the parent team. During the execution of `prif_end_team`, the PRIF implementation will deallocate any coarrays that became allocated during the change team construct. Prior to invoking `prif_end_team`, the compiler is responsible for invoking `prif_alias_destroy` to delete any aliased coarray descriptors created as part of the `CHANGE TEAM` construct.

```
subroutine prif_end_team(...)
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

5.13 Collective Subroutines

5.13.1 Common Arguments in Collective Subroutines

- `source_image` or `result_image`
  - Identifies the image in the current team that is the root of the collective operation.
  - If `result_image` is omitted, then all participating images receive the resulting value.

5.13.2 `prif_co_broadcast`

Description: Broadcast value to images

```
subroutine prif_co_broadcast(...)
  type(*), intent(inout), target :: a(..)
  integer(c_int), intent(in) :: source_image
  integer(c_int), optional, intent(out) :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Argument descriptions

Further argument descriptions:

- `a`
  - shall have the same shape, type, and type parameter values, in corresponding references.
  - shall not be polymorphic
  - the value of the `a` argument on the `source_image` is assigned to the `a` argument on all other images via byte-level copy (as if a `TRANSFER` was applied to the source variable)
  - If `a` contains any allocatable or pointer sub-objects, such members are given indeterminate value and allocation status on images other than `source_image`. The caller is responsible for reallocating and populating allocatable subobjects on receiving images.

5.13.3 `prif_co_max`

Description: Compute maximum value across images.

```
subroutine prif_co_max(...)
  type(*), intent(inout), target :: a(..)
  integer(c_int), intent(in), optional :: result_image
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Argument descriptions

Further argument descriptions:

- `a`
  - shall be of type integer with an interoperable kind or of type real with an interoperable kind

5.13.4 `prif_co_max_character`

Description: Compute maximum value across images.

```
subroutine prif_co_max_character(...)
  character(len=*,kind=c_char), intent(inout), target :: a(..)
  integer(c_int), intent(in), optional :: result_image
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Argument descriptions

Further argument descriptions:

- `a`
  - shall be of type character with kind `c_char`

5.13.5 `prif_co_min`

Description: Compute minimum value across images.

```
subroutine prif_co_min(...)
  type(*), intent(inout), target :: a(..)
  integer(c_int), intent(in), optional :: result_image
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Argument descriptions

Further argument descriptions:

- `a`
  - shall be of type integer with an interoperable kind or of type real with an interoperable kind

5.13.6 `prif_co_min_character`

Description: Compute minimum value across images.

```
subroutine prif_co_min_character(...)
  character(len=*,kind=c_char), intent(inout), target :: a(..)
  integer(c_int), intent(in), optional :: result_image
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Argument descriptions

Further argument descriptions:

- `a`
  - shall be of type character with kind `c_char`

5.13.7 `prif_co_sum`

Description: Compute sum across images

```
subroutine prif_co_sum(...)
  type(*), intent(inout), target :: a(..)
  integer(c_int), intent(in), optional :: result_image
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Argument descriptions

Further argument descriptions:

- `a`
  - shall have any numeric type with an interoperable kind

5.13.8 `prif_co_reduce`

Description: Generalized reduction across images.

CLIENT NOTE:

In addition to supporting `CO_REDUCE`, `prif_co_reduce` may be invoked to support any user code `CO_MIN`, `CO_MAX`, `CO_SUM` call where the `a` argument is not an interoperable type.

```
subroutine prif_co_reduce(...)
  type(*), intent(inout), target :: a(..)
  procedure(prif_operation_wrapper_interface), pointer, intent(in) :: &
    operation_wrapper
  type(c_ptr), intent(in), value :: cdata
  integer(c_int), intent(in), optional :: result_image
  integer(c_int), intent(out), optional :: stat
  character(len=*), intent(inout), optional :: errmsg
  character(len=:), intent(inout), allocatable, optional :: errmsg_alloc
end subroutine
```

Argument descriptions

Further argument descriptions:

- `a`
  - shall have the same shape, type, and type parameter values, in corresponding references
  - shall not be of a type with an ultimate component that is allocatable or a pointer
  - shall not be polymorphic
- `operation_wrapper`: pointer to a client-provided thread-safe reduction subroutine. This may for example reference a compiler-generated subroutine wrapper around a user-provided OPERATION argument to `CO_REDUCE`. Calls to the wrapper may not invoke any PRIF subroutines that incur communication. The operation is assumed to be mathematically associative and commutative, otherwise the resulting values computed may be indeterminate and/or (in the case `result_image` is omitted) may vary across images. The operation may use the `cdata` argument to receive information (such as the operation or data type) not provided by the other arguments. Shall apply the desired operation element-wise to each of the `count` pairs of operands, storing the result in the location from which the second (right-hand) operand is retrieved. `arg1` or `arg2_and_out` are permitted to alias elements of `a`.

`operation_wrapper` shall have the following interface, which is publicly defined in the `prif` module:

```
abstract interface
  subroutine prif_operation_wrapper_interface(...) bind(C)
    type(c_ptr), intent(in), value :: arg1
    type(c_ptr), intent(in), value :: arg2_and_out
    integer(c_size_t), intent(in), value :: count
    type(c_ptr), intent(in), value :: cdata
  end subroutine
end interface
```

`operation_wrapper` argument descriptions:

- `arg1`: Pointer to a contiguous array of left operands. These may be caller-provided input values or intermediate results. The reduction operation is not permitted to write to this memory.
- `arg2_and_out`: Pointer to a contiguous array of right operands and result. These may be caller-provided input values or intermediate results. The reduction operation must write the result(s) to this memory, as described below.
- `count`: Number of elements in `arg1` and `arg2_and_out` arrays. Shall be a non-negative value, which need *not* be equal to `product(shape(a))`; specifically, this wrapper might be invoked to reduce elements corresponding to less or more than one pair of image contributions to the reduction operation. If `count` is equal to zero, the `operation_wrapper` shall have no side effects.
- `cdata`: Client/user data that is the value that the calling image provided when it invoked the corresponding `prif_co_reduce`. It is intended to assist in implementing more than a single data type and/or operation with a common reduction subroutine, or to pass in any dynamic information needed by the reduction (e.g. runtime type parameters).

Example: As an example, consider a call to `CO_REDUCE` with non-interoperable arguments such as the following:

```
type(my_type(len1=*, len2=*)), intent(in) :: a(:)
...
call CO_REDUCE(a, user_func)
...
```

Where `my_type` has a definition like the following:

```
type :: my_type(len1, len2)
  integer, len :: len1, len2
  integer :: ints(len1)
  real :: reals(len2)
end type
```

The `CO_REDUCE` call could be transformed into something like the following:

```
call prif_co_reduce(a, wrapper, c_null_ptr)
contains
  subroutine wrapper(arg1, arg2_and_out, count, cdata) bind(C)
    type(c_ptr), intent(in), value :: arg1          !! "Left" operands
    type(c_ptr), intent(in), value :: arg2_and_out !! "Right" operands and result
    integer(c_size_t), intent(in), value :: count  !! Operand count
    type(c_ptr), intent(in), value ::  cdata       !! Client data, unused here

    type(my_type(len1=a%len1, len2=a%len2)), pointer :: &
      lhs(:)=>null(), rhs_and_result(:)=>null()
    integer(c_size_t) :: i

    if (count == 0) return
    call c_f_pointer(arg1, lhs, [count])
    call c_f_pointer(arg2_and_out, rhs_and_result, [count])
    do i=1, count
      rhs_and_result(i) = user_func(lhs(i), rhs_and_result(i))
    end do
  end subroutine
```

An alternative transformation for this `CO_REDUCE` call that uses `cdata` to avoid introduction of a subprogram by instead passing a pointer to a generated derived type:

```
type reduction_context_data
   type(c_funptr) :: user_op
   integer :: len1, len2
end type
```

with a call like the following:

```
 type(reduction_context_data), target :: cdata
 cdata = reduction_context_data(c_funloc(user_func), a%len1, a%len2)
 call prif_co_reduce(a, operation_wrapper, c_loc(cdata))
```

where the `operation_wrapper` is defined as follows:

```
subroutine operation_wrapper(arg1, arg2_and_out, count, cdata) bind(C)
  type(c_ptr), intent(in), value :: arg1         !! "Left" operands
  type(c_ptr), intent(in), value :: arg2_and_out !! "Right" operands and result
  integer(c_size_t), intent(in), value :: count  !! Operand count
  type(c_ptr), intent(in), value ::  cdata       !! Client data

  type(reduction_context_data), pointer :: stuff=>null()

  if (count == 0) return
  call c_f_pointer(cdata, stuff)
  block
    abstract interface
       pure function user_op(lhs, rhs) result(res)
         import stuff
         type(my_type(len1=stuff%len1, len2=stuff%len2)), intent(in) :: &
           lhs, rhs
         type(my_type(len1=stuff%len1, len2=stuff%len2)) :: res
       end function
    end interface

    procedure(user_op), pointer :: op_ptr
    type(my_type(len1=stuff%len1, len2=stuff%len2)), pointer :: &
      lhs(:)=>null(), rhs_and_result(:)=>null()

    call c_f_procpointer(stuff%user_op, op_ptr)
    call c_f_pointer(arg1, lhs, [count])
    call c_f_pointer(arg2_and_out, rhs_and_result, [count])

    block
      integer(c_size_t) i

      do i=1, count
        rhs_and_result(i) = op_ptr(lhs(i), rhs_and_result(i))
      end do
    end block
  end block

end subroutine
```

5.14 Atomic Memory Operations

All atomic operations are fully blocking operations, meaning they do not return to the caller until after all semantics involving the atomic variable are fully committed with respect to all images.

5.14.1 Common Arguments in Atomic Memory Operations

- `image_num`
  - an argument identifying the image to be communicated with
  - is permitted to identify the calling image
  - this image index is always relative to the initial team, regardless of the current team
- `coarray_handle`: handle for a descriptor of an established coarray to be accessed by this operation. In combination with `offset`, must reference storage entirely contained within the element data of the indicated coarray.
- `offset`: indicates an offset in bytes from the beginning of the elements in a remote coarray (indicated by `coarray_handle`) on a selected image (indicated by `image_num`)
- `atom_remote_ptr`: Is the location of the atomic variable on the identified image to be operated on. The referenced storage must have been allocated using `prif_allocate` or `prif_allocate_coarray`.
- `value`: value to perform the operation with (non-fetching and fetching operations) or value to which the variable shall be set, or retrieved from the variable (atomic access procedures)
- `old`: is set to the initial value of the atomic variable

5.14.2 Non-Fetching Atomic Operations

Description: Each of the following procedures atomically performs the specified operation on a coindexed object.

5.14.2.1 `prif_atomic_add`, Addition

```
subroutine prif_atomic_add(...)
  integer(c_int), intent(in) :: image_num
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_size_t), intent(in) :: offset
  integer(PRIF_ATOMIC_INT_KIND), intent(in) :: value
  integer(c_int), intent(out), optional :: stat
end subroutine

subroutine prif_atomic_add_indirect(...)
  integer(c_int), intent(in) :: image_num
  integer(c_intptr_t), intent(in) :: atom_remote_ptr
  integer(PRIF_ATOMIC_INT_KIND), intent(in) :: value
  integer(c_int), intent(out), optional :: stat
end subroutine
```

Argument descriptions

5.14.2.2 `prif_atomic_and`, Bitwise And

```
subroutine prif_atomic_and(...)
  integer(c_int), intent(in) :: image_num
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_size_t), intent(in) :: offset
  integer(PRIF_ATOMIC_INT_KIND), intent(in) :: value
  integer(c_int), intent(out), optional :: stat
end subroutine

subroutine prif_atomic_and_indirect(...)
  integer(c_int), intent(in) :: image_num
  integer(c_intptr_t), intent(in) :: atom_remote_ptr
  integer(PRIF_ATOMIC_INT_KIND), intent(in) :: value
  integer(c_int), intent(out), optional :: stat
end subroutine
```

Argument descriptions

5.14.2.3 `prif_atomic_or`, Bitwise Or

```
subroutine prif_atomic_or(...)
  integer(c_int), intent(in) :: image_num
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_size_t), intent(in) :: offset
  integer(PRIF_ATOMIC_INT_KIND), intent(in) :: value
  integer(c_int), intent(out), optional :: stat
end subroutine

subroutine prif_atomic_or_indirect(...)
  integer(c_int), intent(in) :: image_num
  integer(c_intptr_t), intent(in) :: atom_remote_ptr
  integer(PRIF_ATOMIC_INT_KIND), intent(in) :: value
  integer(c_int), intent(out), optional :: stat
end subroutine
```

Argument descriptions

5.14.2.4 `prif_atomic_xor`, Bitwise Xor

```
subroutine prif_atomic_xor(...)
  integer(c_int), intent(in) :: image_num
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_size_t), intent(in) :: offset
  integer(PRIF_ATOMIC_INT_KIND), intent(in) :: value
  integer(c_int), intent(out), optional :: stat
end subroutine

subroutine prif_atomic_xor_indirect(...)
  integer(c_int), intent(in) :: image_num
  integer(c_intptr_t), intent(in) :: atom_remote_ptr
  integer(PRIF_ATOMIC_INT_KIND), intent(in) :: value
  integer(c_int), intent(out), optional :: stat
end subroutine
```

Argument descriptions

5.14.3 Fetching Atomic Operations

Description: Each of the following procedures atomically performs the specified operation on a coindexed object, and retrieves the original value.

5.14.3.1 `prif_atomic_fetch_add`, Addition

```
subroutine prif_atomic_fetch_add(...)
  integer(c_int), intent(in) :: image_num
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_size_t), intent(in) :: offset
  integer(PRIF_ATOMIC_INT_KIND), intent(in) :: value
  integer(PRIF_ATOMIC_INT_KIND), intent(out) :: old
  integer(c_int), intent(out), optional :: stat
end subroutine

subroutine prif_atomic_fetch_add_indirect(...)
  integer(c_int), intent(in) :: image_num
  integer(c_intptr_t), intent(in) :: atom_remote_ptr
  integer(PRIF_ATOMIC_INT_KIND), intent(in) :: value
  integer(PRIF_ATOMIC_INT_KIND), intent(out) :: old
  integer(c_int), intent(out), optional :: stat
end subroutine
```

Argument descriptions

5.14.3.2 `prif_atomic_fetch_and`, Bitwise And

```
subroutine prif_atomic_fetch_and(...)
  integer(c_int), intent(in) :: image_num
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_size_t), intent(in) :: offset
  integer(PRIF_ATOMIC_INT_KIND), intent(in) :: value
  integer(PRIF_ATOMIC_INT_KIND), intent(out) :: old
  integer(c_int), intent(out), optional :: stat
end subroutine

subroutine prif_atomic_fetch_and_indirect(...)
  integer(c_int), intent(in) :: image_num
  integer(c_intptr_t), intent(in) :: atom_remote_ptr
  integer(PRIF_ATOMIC_INT_KIND), intent(in) :: value
  integer(PRIF_ATOMIC_INT_KIND), intent(out) :: old
  integer(c_int), intent(out), optional :: stat
end subroutine
```

Argument descriptions

5.14.3.3 `prif_atomic_fetch_or`, Bitwise Or

```
subroutine prif_atomic_fetch_or(...)
  integer(c_int), intent(in) :: image_num
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_size_t), intent(in) :: offset
  integer(PRIF_ATOMIC_INT_KIND), intent(in) :: value
  integer(PRIF_ATOMIC_INT_KIND), intent(out) :: old
  integer(c_int), intent(out), optional :: stat
end subroutine
```

```
subroutine prif_atomic_fetch_or_indirect(...)
  integer(c_int), intent(in) :: image_num
  integer(c_intptr_t), intent(in) :: atom_remote_ptr
  integer(PRIF_ATOMIC_INT_KIND), intent(in) :: value
  integer(PRIF_ATOMIC_INT_KIND), intent(out) :: old
  integer(c_int), intent(out), optional :: stat
end subroutine
```

Argument descriptions

5.14.3.4 `prif_atomic_fetch_xor`, Bitwise Xor

```
subroutine prif_atomic_fetch_xor(...)
  integer(c_int), intent(in) :: image_num
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_size_t), intent(in) :: offset
  integer(PRIF_ATOMIC_INT_KIND), intent(in) :: value
  integer(PRIF_ATOMIC_INT_KIND), intent(out) :: old
  integer(c_int), intent(out), optional :: stat
end subroutine

subroutine prif_atomic_fetch_xor_indirect(...)
  integer(c_int), intent(in) :: image_num
  integer(c_intptr_t), intent(in) :: atom_remote_ptr
  integer(PRIF_ATOMIC_INT_KIND), intent(in) :: value
  integer(PRIF_ATOMIC_INT_KIND), intent(out) :: old
  integer(c_int), intent(out), optional :: stat
end subroutine
```

Argument descriptions

5.14.4 Atomic Access

Description: The following procedures atomically set or retrieve the value of a coindexed object.

5.14.4.1 `prif_atomic_define`, set variable's value

```
subroutine prif_atomic_define_int(...)
  integer(c_int), intent(in) :: image_num
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_size_t), intent(in) :: offset
  integer(PRIF_ATOMIC_INT_KIND), intent(in) :: value
  integer(c_int), intent(out), optional :: stat
end subroutine

subroutine prif_atomic_define_logical(...)
  integer(c_int), intent(in) :: image_num
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_size_t), intent(in) :: offset
  logical(PRIF_ATOMIC_LOGICAL_KIND), intent(in) :: value
  integer(c_int), intent(out), optional :: stat
end subroutine

subroutine prif_atomic_define_int_indirect(...)
  integer(c_int), intent(in) :: image_num
  integer(c_intptr_t), intent(in) :: atom_remote_ptr
  integer(PRIF_ATOMIC_INT_KIND), intent(in) :: value
  integer(c_int), intent(out), optional :: stat
end subroutine
```

```
subroutine prif_atomic_define_logical_indirect(...)
  integer(c_int), intent(in) :: image_num
  integer(c_intptr_t), intent(in) :: atom_remote_ptr
  logical(PRIF_ATOMIC_LOGICAL_KIND), intent(in) :: value
  integer(c_int), intent(out), optional :: stat
end subroutine
```

Argument descriptions

5.14.4.2 `prif_atomic_ref`, retrieve variable's value

```
subroutine prif_atomic_ref_int(...)
  integer(c_int), intent(in) :: image_num
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_size_t), intent(in) :: offset
  integer(PRIF_ATOMIC_INT_KIND), intent(out) :: value
  integer(c_int), intent(out), optional :: stat
end subroutine

subroutine prif_atomic_ref_logical(...)
  integer(c_int), intent(in) :: image_num
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_size_t), intent(in) :: offset
  logical(PRIF_ATOMIC_LOGICAL_KIND), intent(out) :: value
  integer(c_int), intent(out), optional :: stat
end subroutine

subroutine prif_atomic_ref_int_indirect(...)
  integer(c_int), intent(in) :: image_num
  integer(c_intptr_t), intent(in) :: atom_remote_ptr
  integer(PRIF_ATOMIC_INT_KIND), intent(out) :: value
  integer(c_int), intent(out), optional :: stat
end subroutine

subroutine prif_atomic_ref_logical_indirect(...)
  integer(c_int), intent(in) :: image_num
  integer(c_intptr_t), intent(in) :: atom_remote_ptr
  logical(PRIF_ATOMIC_LOGICAL_KIND), intent(out) :: value
  integer(c_int), intent(out), optional :: stat
end subroutine
```

Argument descriptions

5.14.5 Atomic Compare-and-Swap

Description: Performs an atomic compare-and-swap operation. If the value of the atomic variable is equal to the value of the `compare` argument, set it to the value of the `new` argument. The `old` argument is set to the initial value of the atomic variable.

```
subroutine prif_atomic_cas_int(...)
  integer(c_int), intent(in) :: image_num
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_size_t), intent(in) :: offset
  integer(PRIF_ATOMIC_INT_KIND), intent(out) :: old
  integer(PRIF_ATOMIC_INT_KIND), intent(in) :: compare
  integer(PRIF_ATOMIC_INT_KIND), intent(in) :: new
  integer(c_int), intent(out), optional :: stat
end subroutine
```

```
subroutine prif_atomic_cas_logical(...)
  integer(c_int), intent(in) :: image_num
  type(prif_coarray_handle), intent(in) :: coarray_handle
  integer(c_size_t), intent(in) :: offset
  logical(PRIF_ATOMIC_LOGICAL_KIND), intent(out) :: old
  logical(PRIF_ATOMIC_LOGICAL_KIND), intent(in) :: compare
  logical(PRIF_ATOMIC_LOGICAL_KIND), intent(in) :: new
  integer(c_int), intent(out), optional :: stat
end subroutine

subroutine prif_atomic_cas_int_indirect(...)
  integer(c_int), intent(in) :: image_num
  integer(c_intptr_t), intent(in) :: atom_remote_ptr
  integer(PRIF_ATOMIC_INT_KIND), intent(out) :: old
  integer(PRIF_ATOMIC_INT_KIND), intent(in) :: compare
  integer(PRIF_ATOMIC_INT_KIND), intent(in) :: new
  integer(c_int), intent(out), optional :: stat
end subroutine

subroutine prif_atomic_cas_logical_indirect(...)
  integer(c_int), intent(in) :: image_num
  integer(c_intptr_t), intent(in) :: atom_remote_ptr
  logical(PRIF_ATOMIC_LOGICAL_KIND), intent(out) :: old
  logical(PRIF_ATOMIC_LOGICAL_KIND), intent(in) :: compare
  logical(PRIF_ATOMIC_LOGICAL_KIND), intent(in) :: new
  integer(c_int), intent(out), optional :: stat
end subroutine
```

Argument descriptions

Further argument descriptions:

- `compare`: the value with which to compare the atomic variable
- `new`: the value to assign into the atomic variable, if it is initially equal to the `compare` argument

6 Glossary

- **Client Note**: a note that is relevant information for compiler developers who are clients of the PRIF interface
- **Implementation Note**: a note that is relevant information for runtime library developers who are implementing the PRIF interface
- **Source Completion**: The source-side resources provided to a communication operation by this image are no longer in use by the PRIF implementation, and the client is now permitted to modify or reclaim them.
- **coindexed object**: A coindexed object is a named coarray variable followed by an image selector (an expression including square brackets).
- **direct location**: A memory location that was allocated using `prif_allocate_coarray`, and can be accessed by remote images using the coarray handle returned from that allocation.
- **indirect location**: A memory location that was not allocated by the same call to `prif_allocate_coarray` that allocated a given coarray, but which is accessible by remote images through that coarray as an allocatable or pointer component. This memory must have been allocated by either `prif_allocate` or `prif_allocate_coarray`. See Design Decisions for additional information.

7 Future Work

At present all communication operations are semantically blocking on at least source completion. We acknowledge that this prohibits certain types of static optimization, namely the explicit overlap of communication with computation. In the future we intend to develop split-phased/asynchronous versions of various communication operations to enable more opportunities for static optimization of communication.

At present PRIF does not expose a capability for an image to directly access memory on another image. We acknowledge that in some cases an image may be co-located with the image whose coarray data it wants to access, but we don't currently expose this capability to PRIF clients. In the future we intend to expose shared-memory bypass for coarray access to PRIF clients.

8 Acknowledgments

This research is supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231

The authors would like to thank Etienne Renault and Jean-Didier Pailleux of SiPearl and Jeff Hammond of NVIDIA for providing helpful comments and suggestions regarding an earlier revision of this specification.

9 Copyright

10 Legal Disclaimer